Язык программирования C++

Лекции и упражнения

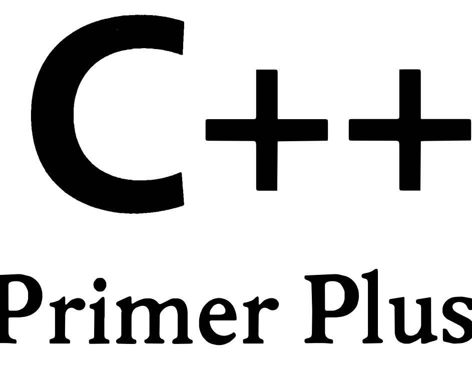
5-е издание

Стивен Прата

Язык программирования

Лекции и упражнения

5-е издание



Primer Plus

Fifth Edition

Stephen Prata

800 East 96th St., Indianapolis, Indiana, 46240 USA

Язык программирования

Лекции и упражнения

5-е издание

Стивен Прата

Москва • Санкт-Петербург • Киев
2007

ББК 32.973.26-018.2.75
П70
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского *Д.Я. Иваненко, А.Ю. Маркушиной, Н.А. Мухина*

Под редакцией *Ю.Н. Артеменко*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com
127055 , Москва, а/я 783; 03150, Киев, а/я 152

Прата, Стивен.

П70 Язык программирования C++. Лекции и упражнения, 5-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2007. — 1184 с. : ил. — Парал. тит. англ.

ISBN 5-8459-1127-3 (рус.)

Книга известного специалиста и лектора в области компьютерных технологий посвящена последнему стандарту одного из наиболее мощных языков объектно-ориентированного программирования — C++, который завоевал многомиллионную армию поклонников во всем мире. Книгу отличает простой и доступный стиль изложения, изобилие примеров и множество рекомендаций по написанию высококачественных программ. Подробно рассматриваются такие вопросы, как представление данных, операции и операторы, управляющие структуры и функции. Немалое внимание уделяется работе с классами, шаблонами и пространствами имен, а также генерации и обработке исключений. Исчерпывающие сведения о концепциях объектно-ориентированного программирования дадут возможность максимально успешно и эффективно создавать живучий программный код. Приводимые в конце каждой главы вопросы для самоконтроля и упражнения для самостоятельной проработки позволят надежно закрепить полученные знания.

Книга рассчитана на программистов разной квалификации, а также будет полезна для студентов и преподавателей дисциплин, связанных с программированием.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Sams Publishing.

Authorized translation from the English language edition published by Sams Publishing, Copyright © 2005.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein.

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized.

Russian language edition is published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2007.

ISBN 5-8459-1127-3 (рус.)
ISBN 0-672-32697-3 (англ.)

© Издательский дом "Вильямс", 2007
© by Sams Publishing, 2005

Оглавление

Введение 23

Глава 1. С чего начать? 33

Глава 2. Приступаем к изучению C++ 53

Глава 3. Работа с данными 91

Глава 4. Составные типы 137

Глава 5. Циклы и выражения отношений 205

Глава 6. Операторы ветвления и логические операции 259

Глава 7. Функции: программные модули C++ 307

Глава 8. Дополнительные сведения о функциях 365

Глава 9. Модели памяти и пространства имен 421

Глава 10. Объекты и классы 475

Глава 11. Работа с классами 529

Глава 12. Классы и динамическое распределение памяти 587

Глава 13. Наследование классов 661

Глава 14. Повторное использование кода в C++ 731

Глава 15. Дружественность, исключения и другие понятия 813

Глава 16. Класс `string` и стандартная библиотека шаблонов 879

Глава 17. Ввод, вывод и файлы 971

Приложение А. Основания систем счисления 1057

Приложение Б. Зарезервированные слова языка C++ 1061

Приложение В. Набор символов ASCII 1065

Приложение Г. Приоритеты операций 1069

Приложение Д. Другие операции 1073

Приложение Е. Шаблонный класс `string` 1085

Приложение Ж. Методы и функции библиотеки STL 1103

Приложение З. Рекомендуемая литература и ресурсы в Internet 1139

Приложение И. Переход к стандарту ANSI/ISO C++ 1143

Приложение К. Ответы на вопросы для самоконтроля 1151

Предметный указатель 1175

Содержание

Об авторе 19
Посвящается 19
Благодарности 20
От издательства 21

Введение 23

Предисловие к пятому изданию 23
Особенности данного учебника 23
Примеры кода, приведенные в этой книге 24
Об этой книге 24
Примечание для преподавателей 29
Соглашения, используемые в этой книге 30
Системы, использованные при разработке примеров для данной книги 31

Глава 1. С чего начать? 33

Изучение языка C++: с чем вы будете иметь дело 34
Истоки языка C++: немного истории 34
Язык программирования C 35
Философия программирования на языке C 36
Переход к C++: объектно-ориентированное программирование 37
C++ и обобщенное программирование 39
Происхождение языка программирования C++ 39
Переносимость и стандарты 40
Порядок создания программы 43
Создание файла исходного кода 44
Компиляция и компоновка 46
Компиляция и связывание в Unix 46
Компиляция и связывание в Linux 47
Компиляторы командной строки для MS-DOS 48
Компиляторы для Windows 48
C++ в компьютерах Macintosh 51
Резюме 51

Глава 2. Приступаем к изучению C++ 53

Первые шаги в C++ 53
Функция `main()` 55
Заголовок функции как интерфейс 56
Почему именно `main()`? 58
Комментарии в языке C++ 58
Препроцессор C++ и файл `iostream` 59
Имена заголовочных файлов 60

Пространства имен 61
Вывод в C++ с помощью cout 63
Манипулятор endl 64
Символ новой строки 65
Форматирование исходного кода C++ 65
Лексемы и обобщенный пробел 66
Стиль написания исходного кода C++ 66
Операторы в языке C++ 67
Операторы объявления и переменные 68
Операторы присваивания 70
Новый трюк с объектом cout 70
Другие операторы C++ 71
Использование cin 72
Конкатенация с помощью cout 73
cin и cout: признак класса 73
Функции 75
Использование функции, имеющей возвращаемое значение 75
Разновидности функций 79
Функции, определяемые пользователем 80
Использование определяемых пользователем функций, имеющих возвращаемое значение 83
Местоположение директивы using в программах со множеством функций 85
Резюме 87
Вопросы для самоконтроля 88
Упражнения по программированию 89

Глава 3. Работа с данными 91

Простые переменные 92
Имена, присваиваемые переменным 92
Целочисленные типы 94
Целочисленные типы short, int и long 95
Типы без знаков 100
Выбор целочисленного типа 102
Целочисленные константы 103
Как компилятор C++ определяет тип константы 105
Тип char: символы и короткие целые числа 106
Тип bool 114
Квалификатор const 115
Числа с плавающей точкой 116
Запись чисел с плавающей точкой 117
Типы чисел с плавающей точкой 118
Константы с плавающей точкой 121
Преимущества и недостатки чисел с плавающей точкой 121
Арифметические операции в языке C++ 122
Порядок выполнения операций: приоритеты операций и ассоциативность 124
Операция нахождения остатка целочисленного деления 127
Преобразования типов 128

Резюме 133
Вопросы для самоконтроля 134
Упражнения по программированию 135

Глава 4. Составные типы 137

Введение в массивы 138
Замечания по программе 140
Правила инициализации массивов 141
Строки 142
Конкатенация строковых констант 144
Использование строк в массивах 144
Риск, связанный с вводом строк 146
Построчное чтение ввода 147
Смешивание строкового и числового ввода 151
Введение в класс string 153
Присваивание, конкатенация и добавление 154
Дополнительные сведения об операциях над классом string 155
Дополнительные сведения о строковом вводе-выводе 157
Введение в структуры 159
Использование структур в программах 161
Может ли структура содержать член типа string? 163
Прочие свойства структур 164
Массивы структур 165
Битовые поля в структурах 167
Объединения 167
Перечисления 169
Установка значений перечислителей 171
Диапазоны значений перечислителей 171
Указатели и свободное хранилище 172
Объявление и инициализация указателей 175
Опасность указателей 177
Указатели и числа 178
Выделение памяти операцией new 178
Освобождение памяти операцией delete 181
Использование new для создания динамических массивов 182
Указатели, массивы и арифметика указателей 185
Замечания по программе 186
Указатели и строки 190
Использование new для создания динамических структур 195
Автоматическое, статическое и динамическое хранилища 198
Резюме 200
Вопросы для самоконтроля 201
Упражнения по программированию 202

Глава 5. Циклы и выражения отношений 205

Введение в цикл for 205
Части цикла for 207

Вернемся к циклу for 213
Изменение шага цикла 215
Внутрь строки с помощью цикла for 216
Операции инкремента и декремента 217
Побочные эффекты и последовательные точки 218
Сравнение префиксной и постфиксной форм 219
Операции инкремента и декремента и указатели 220
Комбинация операций присваивания 221
Составные операторы, или блоки 221
Операция запятой (или еще о синтаксических трюках) 223
Выражения отношений 226
Вероятные ошибки 227
Сравнение строк в стиле C 229
Сравнение строк — объектов класса string 232
Цикл while 233
Замечания по программе 235
Сравнение циклов for и while 235
Подождем минуточку — построение цикла задержки 237
Цикл do while 239
Циклы и текстовый ввод 242
Применение для ввода простого cin 242
Спасение в виде cin.get(char) 243
Который из cin.get()? 244
Условие конца файла 245
Еще одна версия cin.get() 248
Вложенные циклы и двумерные массивы 252
Инициализация двумерного массива 253
Резюме 255
Вопросы для самоконтроля 256
Упражнения по программированию 257

Глава 6. Операторы ветвления и логические операции 259

Оператор if 259
Оператор if else 261
Форматирование операторов if else 263
Конструкция if else if else 264
Логические выражения 266
Логическая операция ИЛИ: || 266
Логическая операция И: && 267
Логическая операция НЕ: ! 272
Факты о логических операциях 273
Альтернативные представления 275
Библиотека символьных функций cctype 275
Операция ?: 277
Оператор switch 279
Использование перечислителей в качестве меток 282
switch и if else 283

Операторы `break` и `continue` 284
Замечания по программе 285
Циклы для чтения чисел 286
Замечания по программе 289
Простой файловый ввод-вывод 290
Текстовый ввод-вывод и текстовые файлы 290
Запись текстового файла 292
Чтение текстового файла 295
Резюме 301
Вопросы для самоконтроля 301
Упражнения по программированию 303

Глава 7. Функции: программные модули C++ 307

Обзор функций 307
Определение функции 308
Прототипирование и вызов функции 311
Аргументы функций и передача по значению 314
Множественные аргументы 315
Еще одна функция с двумя аргументами 318
Функции и массивы 320
Как указатели позволяют функциям обрабатывать массивы 322
Последствия использования массивов в качестве аргументов 323
Дополнительные примеры функций с массивами 325
Функции, работающие с диапазонами массивов 331
Указатели и `const` 332
Функции и двумерные массивы 336
Функции и строки в стиле C 338
Функции с аргументами — строками в стиле C 338
Функции, возвращающие строки в стиле C 340
Функции и структуры 341
Передача и возврат структур 342
Еще один пример использования функций со структурами 344
Передача адресов структур 348
Функции и объекты класса `string` 350
Рекурсия 352
Рекурсия с одиночным рекурсивным вызовом 352
Рекурсия с множественными рекурсивными вызовами 353
Указатели на функции 355
Основы указателей на функции 355
Пример с указателем на функцию 358
Резюме 359
Вопросы для самоконтроля 360
Упражнения по программированию 361

Глава 8. Дополнительные сведения о функциях 365

Встроенные функции C++ 365
Ссылочные переменные 368

Создание ссылочных переменных 369
Ссылки в роли параметров функций 372
Свойства и особенности ссылок 375
Временные переменные, ссылочные аргументы и спецификатор const 377
Использование ссылок при работе со структурами 379
Использование ссылок на объект класса 383
Еще один урок ООП: объекты, наследование и ссылки 387
Когда целесообразно использовать ссылочные аргументы 390
Аргументы, определяемые по умолчанию 391
Перегрузка функций 394
Пример перегрузки 396
Когда целесообразно использовать перегрузку функций 399
Шаблоны функций 400
Перегруженные шаблоны 403
Явные специализации 405
Создание экземпляров и специализация 409
Какую версию функции выбирает компилятор? 411
Резюме 417
Вопросы для самоконтроля 418
Упражнения по программированию 419

Глава 9. Модели памяти и пространства имен 421

Раздельная компиляция 421
Продолжительность существования области хранения, область видимости и компоновка 427
Область видимости и связывание 428
Автоматическая продолжительность хранения 428
Статическая продолжительность хранения 434
Спецификаторы и классификаторы 445
Функции и связывание 447
Языковое связывание 448
Схемы хранения и динамическое распределение памяти 449
Операция new с адресацией 450
Пространства имен 454
Традиционные пространства имен C++ 454
Новые свойства пространства имен 456
Пример пространства имен 463
Будущее пространств имен 467
Резюме 468
Вопросы для самоконтроля 469
Упражнения по программированию 471

Глава 10. Объекты и классы 475

Процедурное и объектно-ориентированное программирование 476
Абстракции и классы 477
Что такое тип? 477
Классы в C++ 478

Реализация функций-членов класса 483
Использование классов 488
Обзор 492
Конструкторы и деструкторы классов 493
Усовершенствование класса `Stock` 499
Обзор конструкторов и деструкторов 506
Изучение объектов: указатель `this` 507
Массив объектов 513
Возврат к интерфейсу и реализации 516
Область видимости класса 517
Константы области видимости класса 518
Абстрактные типы данных 519
Резюме 524
Вопросы для самоконтроля 525
Упражнения по программированию 526

Глава 11. Работа с классами 529

Перегрузка операций 530
Время в наших руках: разработка примера перегрузки операции 531
Добавление операции сложения 534
Ограничения перегрузки 537
Дополнительные сведения о перегруженных операциях 539
Что такое друзья? 541
Создание друзей 543
Общий вид друга: перегрузка операции << 544
Перегруженные операции: сравнение функций-членов и функций-не-членов 550
Дополнительные сведения о перегрузке: класс `Vector` 551
Использование члена состояния 559
Перегрузка арифметических операций для класса `Vector` 561
Комментарии к реализации 563
Использование класса `Vector` в программе "Случайная прогулка" 564
Автоматическое преобразование и приведение типов в классах 567
Преобразования и друзья 578
Резюме 582
Вопросы для самоконтроля 583
Упражнения по программированию 584

Глава 12. Классы и динамическое распределение памяти 587

Динамическая память и классы 588
Простой пример и статические члены класса 588
Неявные функции-члены 597
Новый усовершенствованный класс `String` 606
О чем следует помнить при использовании операции `new` в конструкторах 617
Замечания о возвращаемых объектах 619
Использование указателей на объекты 623
Обзор технических приемов 633

Моделирование очереди 635
Класс Queue 635
Класс Customer 646
Моделирование 649
Резюме 654
Вопросы для самоконтроля 655
Упражнения по программированию 657

Глава 13. Наследование классов 661

Начало работы с простым базовым классом 662
Порождение класса 664
Конструкторы: анализ доступа 666
Использование производного класса 669
Специальные отношения между производным и базовым классами 671
Наследование: отношение is-a 673
Полиморфное общедоступное наследование 675
Разработка классов Brass и BrassPlus 676
Статическое и динамическое связывание 688
Совместимость типов указателя и ссылки 688
Виртуальные методы и динамическое связывание 689
Что следует знать о виртуальных методах 693
Управление доступом: protected 696
Абстрактные базовые классы 698
Использование концепции АБК 701
Философия АБК 705
Наследование и динамическое распределение памяти 705
Случай 1: производный класс не использует операцию new 706
Случай 2: производный класс не использует операцию new 707
Пример наследования с динамическим распределением памяти и друзьями 709
Обзор проекта класса 714
Функции-члены, которые генерирует компилятор 714
Анализ других методов класса 716
Анализ общедоступного наследования 719
Сводка функций классов 724
Резюме 725
Вопросы для самоконтроля 726
Упражнения по программированию 727

Глава 14. Повторное использование кода в C++ 731

Классы с членами-объектами 732
Класс valarray: краткий обзор 732
Проект класса Student 733
Пример класса Student 735
Инициализация включенных объектов 737
Использование интерфейса для включенного объекта 738
Использование нового класса Student 740
Приватное наследование 742

Пример класса Student (новая версия) 743
Инициализация компонентов базового класса 743
Множественное наследование 753
Сколько всего сотрудников? 758
Какой использовать метод? 761
Краткий обзор множественного наследования 771
Шаблоны класса 772
Определение шаблона класса 772
Использование шаблонного класса 776
Более пристальный взгляд на шаблонные классы 778
Пример шаблона массива и нетипизированные аргументы 784
Универсальность шаблонов 786
Рекурсивное использование шаблонов 786
Специализация шаблона 789
Члены-шаблоны 792
Шаблоны как параметры 795
Шаблонные классы и друзья 797
Резюме 804
Вопросы для самоконтроля 806
Упражнения по программированию 808

Глава 15. Дружественность, исключения и другие понятия 813

Друзья 813
Дружественные классы 814
Дружественные функции-члены 818
Другие дружественные отношения 821
Вложенные классы 823
Вложенные классы и доступ 825
Вложение в шаблон 827
Исключения 830
Вызов abort() 831
Возврат кода ошибки 832
Механизм исключений 834
Использование объектов в качестве исключений 837
Раскручивание стека 841
Дополнительные свойства исключений 847
Класс exception 849
Исключения, классы и наследование 853
Потеря исключений 858
Предостережения при использовании исключений 861
RTTI 863
Для чего нужен RTTI 863
Как работает RTTI? 864
Операция dynamic_cast 864
Операции приведения типов 872
Резюме 875
Вопросы для самоконтроля 876
Упражнения по программированию 877

Глава 16. Класс string и стандартная библиотека шаблонов 879

Класс string 879
    Создание объекта string 880
    Ввод для класса string 884
    Работа со строками 886
    Дополнительные возможности класса string 891
Класс auto_ptr 894
    Использование auto_ptr 895
    Соображения по использованию auto_ptr 897
Стандартная библиотека шаблонов (STL) 899
    Шаблон vector 899
    Что еще можно делать с помощью векторов 901
    Дополнительные возможности векторов 906
Обобщенное программирование 910
    Зачем нужны итераторы? 910
    Виды итераторов 915
    Иерархия итераторов 917
    Концепции, уточнения и модели 919
    Виды контейнеров 926
    Ассоциативные контейнеры 936
Функциональные объекты (также известные как функторы) 942
    Концепции функторов 943
    Предопределенные функторы 946
    Адаптируемые функторы и функциональные адаптеры 948
Алгоритмы 950
    Группы алгоритмов 950
    Основные свойства алгоритмов 951
    STL и класс string 952
    Сравнение функций и методов контейнеров 954
    Использование STL 955
Другие библиотеки 959
    vector и valarray 959
Резюме 965
Вопросы для самоконтроля 967
Упражнения по программированию 968

Глава 17. Ввод, вывод и файлы 971

Обзор ввода и вывода в C++ 972
    Потоки и буферы 972
    Потоки, буферы и файл iostream 975
    Перенаправление 977
Вывод с помощью cout 978
    Перегруженная операция << 979
    Другие методы ostream 982
    Сброс содержимого выходного буфера 984
    Форматирование с помощью cout 985
Ввод с помощью cin 1001

Как cin >> воспринимает ввод 1003
Состояния потока 1005
Другие методы класса istream 1010
Другие методы istream 1017
Файловый ввод и вывод 1021
Простой файловый ввод-вывод 1021
Проверка потока и is_open() 1024
Открытие нескольких файлов 1025
Обработка командной строки 1026
Режимы файла 1028
Произвольный доступ 1039
Внутреннее форматирование 1046
Что теперь? 1049
Резюме 1050
Вопросы для самоконтроля 1051
Упражнения по программированию 1052

Приложение А. Основания систем счисления 1057

Десятичные числа (основание 10) 1057
Восьмеричные целые числа (основание 8) 1057
Шестнадцатеричные числа (основание 16) 1058
Двоичные числа (основание 2) 1058
Двоичная и шестнадцатеричная формы записи 1059

Приложение Б. Зарезервированные слова языка C++ 1061

Служебные слова C++ 1061
Альтернативные лексемы 1062
Зарезервированные имена библиотек C++ 1062

Приложение В. Набор символов ASCII 1065

Приложение Г. Приоритеты операций 1069

Приложение Д. Другие операции 1073

Битовые операции 1073
Операции сдвига 1073
Логические битовые операции 1075
Альтернативные варианты представления битовых операций 1077
Примеры использования битовых операций 1078
Операции разыменования членов 1079

Приложение Е. Шаблонный класс string 1085

Тринадцать типов и константа 1085
Информация о данных, конструкторы и вспомогательные элементы 1087
Конструкторы по умолчанию 1089

Конструкторы, использующие массивы 1089
Конструкторы, использующие часть массива 1090
Конструкторы копирования 1090
Конструкторы, использующие n копий символа 1091
Конструкторы, использующие диапазон 1092
Различные операции с памятью 1092
Доступ к строке 1093
Основные варианты присваивания 1094
Поиск строки 1094
Семейство find() 1094
Семейство rfind() 1095
Семейство find_first_of() 1095
Семейство find_last_of() 1096
Семейство find_first_not_of() 1096
Семейство find_last_not_of() 1097
Методы и функции сравнения 1097
Модификаторы строк 1098
Методы присоединения и добавления 1098
Дополнительные методы присваивания 1099
Методы вставки 1100
Методы удаления 1100
Методы замены 1101
Другие методы модифицирования: copy() и swap() 1102
Вывод и ввод 1102

Приложение Ж. Методы и функции библиотеки STL 1103

Члены, общие для всех контейнеров 1103
Дополнительные члены для векторов, списков и двусторонних очередей 1106
Дополнительные члены для множеств и таблиц 1108
Функции библиотеки STL 1110
Операции, не изменяющие последовательности 1110
Операции, видоизменяющие последовательности 1115
Операции сортировки и связанные с ними операции 1123
Сортировка 1125
Бинарный поиск 1127
Слияние 1128
Работа с множествами 1129
Работа с частично упорядоченными полными бинарными деревьями 1131
Поиск максимального и минимального значений 1132
Работа с перестановками 1134
Числовые операции 1135

Приложение З. Рекомендуемая литература и ресурсы в Internet 1139

Рекомендуемая литература 1139
Ресурсы в Internet 1141

Приложение И. Переход к стандарту ANSI/ISO C++ 1143

Используйте альтернативные варианты для некоторых директив препроцессора 1143
Используйте const вместо #define для определения констант 1143
Используйте inline вместо #define для определения коротких функций 1145
Используйте прототипы функций 1146
Используйте приведение типов 1146
Знакомьтесь с функциональными особенностями C++ 1147
Используйте новую организацию заголовков 1147
Используйте пространства имен 1148
Используйте шаблон autoptr 1149
Используйте класс string 1149
Используйте библиотеку STL 1150

Приложение К. Ответы на вопросы для самоконтроля 1151

Ответы на вопросы для самоконтроля из главы 2 1151
Ответы на вопросы для самоконтроля из главы 3 1152
Ответы на вопросы для самоконтроля из главы 4 1153
Ответы на вопросы для самоконтроля из главы 5 1154
Ответы на вопросы для самоконтроля из главы 6 1155
Ответы на вопросы для самоконтроля из главы 7 1156
Ответы на вопросы для самоконтроля из главы 8 1158
Ответы на вопросы для самоконтроля из главы 9 1160
Ответы на вопросы для самоконтроля из главы 10 1161
Ответы на вопросы для самоконтроля из главы 11 1163
Ответы на вопросы для самоконтроля из главы 12 1164
Ответы на вопросы для самоконтроля из главы 13 1167
Ответы на вопросы для самоконтроля из главы 14 1168
Ответы на вопросы для самоконтроля из главы 15 1170
Ответы на вопросы для самоконтроля из главы 16 1171
Ответы на вопросы для самоконтроля из главы 17 1172

Предметный указатель 1175

Об авторе

Стивен Прата (Stephen Prata) — преподаватель астрономии, физики и вычислительной техники в морском колледже города Кентфилд, штат Калифорния. Диплом бакалавра он получил в Калифорнийском технологическом институте, а степень доктора философии — в Калифорнийском университете в Беркли. Стивен — автор и соавтор более десятка книг, включая *C Primer Plus* (*Язык программирования C. Лекции и упражнения, 5-е издание*, Издательский дом "Вильямс", 2006) и *Unix Primer Plus*.

Посвящается

Моим коллегам и студентам из морского колледжа, с которыми очень приятно работать.

Стивен Прата

Благодарности

Благодарности к пятому изданию

Я хочу выразить благодарность Лоретте Йатс (Loretta Yates) и Сонглин Кию (Songlin Qiu) из издательства Sams Publishing за руководство этим проектом. Я признателен своему коллеге Фреду Шмитту (Fred Schmitt), который дал мне несколько полезных советов. Спасибо Рону Личти (Ron Liechty) из компании Metrowerks за любезно предложенную помощь.

Благодарности к четвертому изданию

Организация и поддержка этого проекта была бы невозможной, если бы не участие редакторов из издательств Pearson и Sams. Я хочу выразить благодарность Линде Шарп (Linda Sharp), Карен Вачс (Karen Wachs) и Лори Мак-Гуайр (Laurie McGuire). Спасибо Майклу Мэддоксу (Michael Maddox), Биллу Крауну (Bill Craun), Крису Маундеру (Chris Maunder) и Филиппу Бруно (Phillipe Bruno) за технический обзор и редактирование. Я еще раз благодарю Майкла Мэддокса (Michael Maddox) и Билла Крауна (Bill Craun) за то, что они предоставили материалы по Real World Notes. В завершение я хочу выразить благодарность Рону Личти (Ron Liechty) из компании Metrowerks и Грегу Камю (Greg Comeau) из Comeau Computing за их помощь в работе с компиляторами C++.

Благодарности к третьему изданию

Я хочу поблагодарить редакторов из издательств Macmillian и The Waite Group за их участие в работе над этой книгой: Трейси Данкелбергер (Tracy Dunkelberger), Сьюзен Уолтон (Susan Walton) и Андреа Розенберг (Andrea Rosenberg). Спасибо Рассу Джейкобсу (Russ Jacobs) за техническое редактирование и редактирование содержания книги. Я хочу поблагодарить Дейва Марка (Dave Mark), Алекса Харпера (Alex Harper) и, в особенности, Рона Личти (Ron Liechty) за помощь и сотрудничество в подготовке этой книги.

Благодарности ко второму изданию

Я хочу поблагодарить Митчелла Уэйта (Mitchell Waite) и Скотта Каламара (Scott Calamar) за помощь в подготовке второго издания, а также Джоэла Фугаццотто (Joel Fugazzotto) и Джоанну Миллер (Joanne Miller) за руководство данным проектом вплоть до полного его завершения. Спасибо Майклу Маркотти (Michael Marcotty) из компании Metrowerks за то, что он не оставил без внимания ни один мой вопрос по третьей версии компилятора CodeWarrior. Я хочу выразить благодарность инструкторам, которые потратили свое время на организацию обратной связи по первому изданию: Джеффу Бакуолтеру (Jeff Buckwalter), Эрлу Бриннеру (Earl Brynner), Майку Холланду (Mike Holland), Энди Яо (Andy Yao), Ларри Сандерсу (Larry Sanders), Шахину Момтази (Shahin Momtazi) и Дону Стивенсу (Don Stephens). Напоследок я хочу поблагодарить Хейди Брамбо (Heidi Brumbaugh) за помощь в редактировании нового и исправленного материала книги.

Благодарности к первому изданию

В работе над этой книгой принимали участие многие люди. В частности, я хочу поблагодарить Митча Уэйта (Mitch Waite) за его работу по составлению проекта, редактирование, а также за рецензию рукописи. Я признателен Гарри Хендерсону (Harry Henderson) за рецензирование нескольких последних глав и тестирование программ с использованием компилятора Zortech C++. Я благодарен Дэвиду Геррольду (David Gerrold) за рецензию всей рукописи, а также за его настойчивое требование учитывать интересы читателей, имеющих незначительный опыт в программировании. Я благодарен Хэнку Шиффману (Hank Shiffman) за тестирование программ с использованием Sun C++ и Кенту Уильямсу (Kent Williams) за тестирование программ с помощью AT&T cfront и G++. Спасибо Нэну Борресону (Nan Borreson) из компании Borland International, за то, что он так любезно и энергично помог разобраться с Turbo C++ и Borland C++. Спасибо вам, Рут Майерс (Ruth Myers) и Кристин Буш (Christine Bush), за обработку непрекращающегося потока корреспонденции, связанной с этим проектом. В завершение я хочу поблагодарить Скотта Каламара (Scott Calamar) за то, что работа над этой книгой велась надлежащим образом.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: info@williamspublishing.com
WWW: http://www.williamspublishing.com

Информация для писем из:

России: 115419, Москва, а/я 783
Украины: 03150, Киев, а/я 152

Введение

Предисловие к пятому изданию

Процесс изучения языка программирования C++ чем-то напоминает эпопею первооткрывателя, в частности потому, что этот язык охватывает несколько парадигм программирования: объектно-ориентированного программирования (ООП), обобщенного программирования и традиционного процедурного программирования. По мере добавления новых возможностей C++ напоминал "движущуюся мишень", однако сейчас, после того как в 2003 году была принята вторая версия стандарта ISO/ANSI C++, язык стал стабильным. Современные компиляторы поддерживают большинство, а порой и все функциональные особенности, описанные в этом стандарте, поэтому программистам необходимо потратить немало времени, чтобы изучить и применять их в своих программах. В пятом издании этой книги отражены правила, изложенные в стандарте ISO/ANSI, и описывается именно эта обновленная версия языка C++.

В книге рассматривается базовый язык программирования C и функциональные возможности языка C++, что характеризует эту книгу как самодостаточную. Изучение языка программирования сопровождается небольшими программами, которые можно без труда копировать и пробовать выполнять. В этой книге вы ознакомитесь с операциями ввода-вывода, научитесь составлять программы так, чтобы они выполняли повторяющиеся задачи и на основании анализа выбирали определенное действие, узнаете о многочисленных способах обработки данных и об использовании функций. В этой книге будут раскрыты многие функциональные особенности языка программирования C++, дополняющие язык C, к числу которых относятся следующие:

- Классы и объекты.
- Наследование.
- Полиморфизм, виртуальные функции и динамическая идентификация типов (RunTime Type Identification, RTTI).
- Перегрузка функций.
- Ссылочные переменные.
- Обобщенное, или независимое от типа, программирование на основе шаблонов и стандартной библиотеки шаблонов (Standard Template Library — STL).
- Механизм исключений, обрабатывающий программные ошибки.
- Пространства имен, позволяющие управлять именованием функций, классов и переменных.

Особенности данного учебника

Изложение материала в книге характеризуется некоторыми особенностями. Учебник написан в духе традиций, которые сложились почти двадцать лет назад с момента выхода книги *Язык программирования C. Лекции и Упражнения*, и в процессе работы над ним автор руководствовался следующими требованиями:

- Учебник должен быть легко читаемым и удобным в качестве руководства.
- Предполагается, что читатель не знаком с принципами программирования, имеющими отношение к теме учебника.
- Изложение материала сопровождается небольшими простыми примерами, которые читатель может выполнить самостоятельно. Примеры способствуют пониманию изложенного материала.
- На страницах учебника представлено достаточное количество иллюстраций.
- Учебник должен содержать вопросы и упражнения, которые позволят читателю закрепить пройденный материал. Благодаря ним, учебник можно будет с успехом использовать для коллективного обучения.

Книга, написанная в соответствии с этими правилами, поможет вам разобраться во всех тонкостях языка программирования C++ и научит использовать его для решения конкретных задач. Например:

- В учебнике содержится понятийное руководство по использованию конкретных функциональных особенностей, таких, например, как общедоступное наследование для моделирования отношений *is-a.*
- В учебнике показаны примеры распространенных стилей и приемов программирования на языке C++.
- В нем имеется большое количество врезок, в которых вы найдете советы, предостережения, памятки, замечания по совместимости и примеры из практики.

Автор и редакторы этого учебника приложили максимум усилий, чтобы он получился простым, материал был понятным, и вы остались довольными прочитанным. Мы стремились к тому, чтобы вы, изучив предложенный материал, смогли самостоятельно писать надежные и эффективные программы и получать удовольствие от этого занятия.

Примеры кода, приведенные в этой книге

Учебник изобилует примерами кода, большинство из которых представляют собой готовые программы. Как и в предыдущих изданиях, примеры в этой книге составлены на обобщенном языке C++, поэтому ни один из них не "привязан" к какому-нибудь определенному типу компьютеров, операционной системе или компилятору. Каждый пример тестировался в системах Windows XP, Macintosh OS X и Linux. Лишь в некоторых программах возникли проблемы, связанные с несоответствием компилятора. С момента выхода предыдущего издания этой книги степень соответствия компилятора стандарту C++ стала более высокой.

Готовые программы, предложенные в книге в качестве примеров, можно найти на Web-сайте издательства.

Об этой книге

Эта книга состоит из 17 глав и 10 приложений, которые вкратце описаны далее.

Глава 1. С чего начать?

В главе 1 рассказывается о том, как Бьерн Страуструп (Bjarne Stroustrup) создал язык программирования C++, реализовав в языке C принципы объектно-ориентированного программирования. Вы узнаете, чем отличаются процедурные языки программирования, примером которых является C, и языки программирования с концепцией объектного ориентирования, примером которых является C++. Вы узнаете, как комитетами ANSI/ISO был разработан и утвержден стандарт для языка C++. Здесь рассматривается порядок создания программы на C++ с учетом особенностей некоторых современных компиляторов C++. В конце главы приведены соглашения, используемые в этой книге.

Глава 2. Приступаем к изучению C++

В главе 2 рассматривается процесс написания простых программ на языке C++. Вы узнаете о роли функции `main()` и о некоторых разновидностях операторов, используемых в программах на C++. Для операций ввода-вывода в программах вы будете использовать предопределенные объекты `cout` и `cin`, а еще вы научитесь создавать и использовать переменные. В конце главы вы познакомитесь с функциями — программными модулями языка C++.

Глава 3. Работа с данными

Для хранения двух разновидностей данных — целых чисел (чисел без дробной части) и чисел с плавающей точкой (числа с дробной частью) — в языке программирования C++ предусмотрены встроенные типы. Чтобы удовлетворить разнообразные требования программистов, язык C++ предлагает по несколько типов в каждой категории данных. Об этих типах, а также о создании переменных и написании констант различных типов, речь пойдет в главе 3. Вы узнаете также о том, как в языке C++ осуществляется явное и неявное преобразование одного типа в другой.

Глава 4. Составные типы

На основе базовых встроенных типов данных в языке программирования C++ можно создавать более совершенные типы. Наиболее совершенной формой являются классы, о которых пойдет речь в главах 9–13. В главе 4 будут рассмотрены другие формы, включая массивы, содержащие несколько однотипных значений; структуры, хранящие несколько разнотипных значений; указатели, которые идентифицируют ячейки памяти. Вы узнаете о том, как создаются и хранятся текстовые строки и как выполняется обработка операций текстового ввода-вывода за счет использования символьных массивов, присущих языку C, и класса `string` языка C++. В последней части главы вы узнаете о том, как в языке C++ осуществляется распределение памяти, включая операции `new` и `delete` явного управления памятью.

Глава 5. Циклы и выражения отношений

Часто бывает так, что в программе необходимо выполнить некоторые повторяющиеся действия, и для этих целей в языке C++ предусмотрены три циклических структуры: цикл `for`, цикл `while` и цикл `do while`. Завершение цикла осуществляется по заранее определенному условию, поэтому для создания циклических структур и их управления в языке C++ используются операции отношения. В главе 5 вы узнаете, как создавать циклы, в которых входные данные считываются и обрабатываются по-

символьно. В последней части главы вы научитесь создавать двумерные массивы и использовать вложенные циклы для их обработки.

Глава 6. Операторы ветвления и логические операции

Поведение программы будет "интеллектуальным", если она сможет адаптироваться к различным ситуациям. В главе 6 рассказывается об управлении ходом выполнения программ с помощью операторов `if`, `if else` и `switch` и условные операции. Вы узнаете о том, как с помощью условных операций можно выразить проверку для принятия решения. Кроме этого, вы познакомитесь с библиотекой функций `cctype`, которая используется для оценки символьных отношений (например, для того, чтобы проверить, является ли данный символ цифрой или же это непечатаемый символ). В конце этой главы будет дан вводный обзор процессов файлового ввода-вывода.

Глава 7. Функции: программные модули C++

Функции являются основными стандартными программными блоками в языке C++. Главное внимание в этой главе уделено возможностям, характерным как для функций языка C++, так и для функций языка C. В частности, здесь будет представлен общий формат описания функций и рассказано о том, как с помощью прототипов функций можно повысить надежность создаваемых программ. Вы научитесь создавать функции для обработки массивов данных, символьных строк и структур. В этой главе будет рассмотрена рекурсия, возникающая при вызове функцией самой себя; вы узнаете о том, как с помощью функции можно реализовать стратегию "разделяй и властвуй". В конце главы вы познакомитесь с указателями на функции, благодаря которым одна функция может с помощью аргумента вызывать другую функцию.

Глава 8. Дополнительные сведения о функциях

В главе 8 будет рассказано о новых возможностях функций в C++. Мы рассмотрим подставляемые функции, с помощью которых можно ускорить выполнение программ за счет увеличения их размера. Вы будете работать со ссылочными переменными, которые предлагают альтернативный способ передачи информации в функции. Благодаря аргументам, используемым по умолчанию, функции могут задавать значения автоматически тем аргументам функций, которые не указываются при вызове. Перегрузка функций позволяет создавать функции с одинаковыми именами, но принимающие разные списки аргументов. Все эти особенности часто применяются при создании классов. Вы узнаете также о шаблонах функций, благодаря которым можно создавать целые семейства связанных функций.

Глава 9. Модели памяти и пространства имен

В главе 9 рассмотрены вопросы объединения программ, состоящих из множества файлов. В ней будут представлены варианты распределения памяти, рассмотрены способы управления памятью; вы узнаете, что такое область видимости, компоновка и пространства имен, определяющие, в каких частях программы будут известны те или иные переменные.

Глава 10. Объекты и классы

Класс представляет собой тип, определенный пользователем, а объект (например, переменная) является экземпляром класса. В главе 10 вы узнаете о том, что такое объектно-ориентированное программирование, и как создаются классы. В объявлении

класса указывается информация, хранящаяся в объекте класса, и операции (методы класса), разрешенные над объектами класса. Некоторые части объекта будут видимыми для внешнего мира (общедоступная часть), а другие будут скрытыми (приватная часть). В момент создания и уничтожения объектов инициируются специальные методы класса, называемые конструкторами и деструкторами. Здесь вы узнаете об этих и других особенностях классов и получите представление о вариантах использования классов для реализации абстрактных типов данных, таких как стек.

Глава 11. Работа с классами

В главе 11 продолжается знакомство с классами. Прежде всего, будет рассмотрен механизм перегрузки операций, посредством которой программист определяет, как операция (например, +) будет работать с объектами класса. Мы поговорим о дружественных функциях, позволяющих обращаться к данным класса, которые в общем случае являются недоступными. Вы узнаете о том, как некоторые конструкторы и функции-члены перегруженных операций могут быть использованы для управления преобразованием одного типа класса в другой.

Глава 12. Классы и динамическое распределение памяти

Зачастую бывает полезным применение указателя на член класса для динамического распределения памяти. Если в конструкторе класса вы будете использовать операцию `new` для динамического распределения памяти, то на этот случай вы должны предусмотреть соответствующий деструктор, явный конструктор копирования и явную операцию присваивания. В главе 12 мы покажем, как это можно сделать, и расскажем о поведении функций-членов, которые генерируются неявным образом, если явные описания не предусмотрены. Опыт работы с классами вы закрепите на примерах использования указателей на объекты и моделировании очереди.

Глава 13. Наследование классов

Одной из самых замечательных особенностей объектно-ориентированного программирования является наследование, благодаря которому производный класс наследует особенности базового класса, что открывает возможности для повторного использования кода базового класса. В главе 13 обсуждается общедоступное наследование, моделирующее отношения *is-a*, при которых производный объект рассматривается как частный случай базового объекта. Здесь в качестве примера можно упомянуть физика, который является частным случаем ученого. Некоторые отношения при наследовании являются полиморфными, что означает возможность писать код, используя несколько связанных между собой классов, для которых метод с одним и тем же именем может реализовывать поведение, зависящее от типа объекта. Чтобы такое поведение было возможным, необходима особая функция-член, называемая виртуальной функцией. Нередко для отношений наследования лучше всего использовать абстрактные базовые классы. В этой главе будут рассмотрены все эти вопросы с описанием ситуаций, в которых общедоступное наследование будет иметь смысл.

Глава 14. Повторное использование кода в C++

Общедоступное наследование — это всего лишь один из способов повторного использования написанного вами кода. Глава 14 посвящена рассмотрению других вариантов повторного использования кода. Ситуация, при которой члены одного класса являются одновременно объектами другого класса, называется включением.

Включение можно использовать для моделирования отношений *has-a*, при которых один класс имеет компоненты, принадлежащие другому классу (например, автомобиль и его двигатель). Для моделирования таких отношений можно использовать приватное и защищенное наследование. Мы поговорим о каждом способе и рассмотрим характерные для них особенности. Здесь вы узнаете о шаблонах классов, которые позволяют определить класс посредством некоторого неопределенного обобщенного типа, а затем использовать шаблон для создания определенных классов на основе определенных типов. Например, с помощью шаблона стека можно создать стек целых чисел и стек строк. В конце главы вы узнаете о множественном общедоступном наследовании, при котором один класс может быть произведен на основе нескольких классов.

Глава 15. Дружественность, исключения и другие понятия

В этой главе речь пойдет о так называемых "друзьях", а именно — о дружественных классах и дружественных функциях-членах. Здесь вы найдете сведения о некоторых нововведениях в языке C++, начиная с исключений, которые предлагают механизм выхода из необычных ситуаций, возникающих при выполнении программы (например, при возникновении несоответствующих значений аргументов функций или при нехватке памяти). Из этой главы вы узнаете, что собой представляет RTTI (механизм динамической идентификации типов объектов). В заключительной части главы будет рассказано о безопасных альтернативных вариантах неограниченного приведения типов.

Глава 16. Класс `string` и стандартная библиотека шаблонов

В главе 16 рассказывается о некоторых полезных библиотеках классов, с недавнего времени используемых в языке C++. Класс `string` является удобным и мощным альтернативным вариантом традиционных строк в языке C. Класс `auto_ptr` помогает управлять динамически распределяемой памятью. Библиотека STL (Standard Template Library — стандартная библиотека шаблонов) содержит несколько обобщенных контейнеров, включая шаблонные представления массивов, очередей, списков, множеств и карт. В нее также включено большое число обобщенных алгоритмов, которые можно использовать как для контейнеров STL, так и для обычных массивов. Шаблонный класс `valarray` позволяет работать с числовыми массивами.

Глава 17. Ввод, вывод и файлы

В главе 17 рассматриваются вопросы ввода-вывода в языке C++ и обсуждается форматирование вывода. Здесь вы узнаете, как с помощью методов классов можно определять состояние потоков ввода или вывода и проверить, например, что во входных данных была допущена ошибка несоответствия типов или достигнут конец файла. Чтобы произвести классы для управления файловым вводом-выводом, в языке C++ используется наследование. Из этой главы вы узнаете о том, как открывать файлы для ввода и вывода, как добавлять данные в файл, использовать бинарные файлы, как организовать случайный доступ к файлу. Напоследок вы узнаете о том, как можно применять методы стандартного ввода-вывода для чтения и записи строк.

Приложение А. Основания систем счисления

В приложении А представлены примеры восьмеричных, шестнадцатеричных и двоичных чисел.

Приложение Б. Зарезервированные слова языка C++

В этом приложении приводится список зарезервированных слов в языке C++.

Приложение В. Набор символов ASCII

В приложении В приведен набор символов ASCII с указанием представлений в десятеричной, восьмеричной, шестнадцатеричной и двоичной системах счисления.

Приложение Г. Приоритеты операций

В приложении Г представлен список операций в языке C++, упорядоченных по приоритету их выполнения.

Приложение Д. Другие операции

В приложении Д перечислены операции языка C++, которые не были рассмотрены в основном тексте (например, поразрядные операции).

Приложение Е. Шаблонный класс `string`

В этом приложении приводятся сведения о методах и функциях класса `string`.

Приложение Ж. Методы и функции библиотеки STL

В приложении Ж перечислены методы контейнеров и общие функции алгоритмов из библиотеки STL.

Приложение З. Рекомендуемая литература и ресурсы в Internet

В этом приложении предложен список книг, которые вы можете использовать для дальнейшего изучения языка C++.

Приложение И. Переход к стандарту ANSI/ISO C++

В этом приложении вы найдете правила преобразования кода на языке C и старых версий C++ в код на языке C++ стандарта ANSI/ISO.

Приложение К. Ответы на вопросы для самоконтроля

В этом приложении содержатся ответы на вопросы, поставленные в конце каждой главы этой книги.

Примечание для преподавателей

Перед этим изданием настоящей книги была поставлена задача написать ее таким образом, чтобы ее можно было использовать или в качестве руководства, или в качестве самоучителя. Ниже перечислены некоторые особенности, характерные для использования пятого издания этой книги в качестве руководства:

- В этой книге описывается обобщенный язык программирования C++, не зависящий от конкретной реализации.
- Предложенный материал соответствует стандарту ANSI/ISO для языка C++; материал включает обсуждение шаблонов, библиотеки STL, класса `string`, исключений, RTTI и пространств имен.

- Предполагается, что читатель не знаком с языком С, поэтому книгу могут использовать люди, не знающие язык С (хотя, конечно, некоторые навыки в программировании не будут лишними).
- Темы книги упорядочены таким образом, что первые главы могут быть изучены довольно быстро как обзорные главы на курсах, слушатели которых уже имеют представление о языке С.
- В конце каждой главы предложены вопросы для самоконтроля и упражнения по программированию. В приложении К даны ответы на вопросы для самоконтроля. Решения некоторых примеров можно найти на Web-сайте издательства.
- В книге предложены темы, которые можно использовать в курсах по вычислительной технике. К ним можно отнести рассмотрение абстрактных типов данных (abstract data type — ADT), стеков, очередей, простых списков, обобщенного программирования и использования рекурсии для реализации стратегии "разделяй и властвуй".
- Большинство глав имеют такой объем, чтобы их можно было изучить в течение одной недели или даже быстрее.
- В книге вы найдете ответы на вопросы о том, *когда* можно использовать определенные функциональные особенности, и *каким образом* их использовать. Например, общедоступное наследование связывается с отношениями *is-a*, композиция и приватное наследование — с отношениями *has-a*, и рассматриваются условия использования виртуальных функций.

Соглашения, используемые в этой книге

Для выделения различных частей текста в книге используются следующие типографские соглашения:

- Строки кода, команды, операторы, переменные, имена файлов и вывод программ будут выделены `моноширинным шрифтом`:

```
#include <iostream>
int main()
{
   using namespace std;
   cout << "What's up, Doc!\n";
   return 0;
}
```

- Входные данные для программ, которые вы должны вводить самостоятельно, представлены `моноширинным шрифтом с полужирным начертанием`:

```
Please enter your name:
Plato
```

- Заполнители в синтаксических описаниях выделены `моноширинным шрифтом с курсивным начертанием`. Каждый такой заполнитель вы должны будете заменить реальным именем файла, параметром или любым другим элементом, который он представляет.
- *Курсив* используется для выделения новых терминов.

Для выделения специфических вопросов в этой книге используются следующие врезки:

Замечание по совместимости

Большинство компиляторов пока что не соответствуют в полной мере стандарту ISO/ANSI, поэтому здесь мы будем предупреждать о возможных несоответствиях.

На память!

Здесь будет содержаться информация по текущему вопросу, которую необходимо запомнить.

Совет

Здесь вы найдете краткие полезные советы и рекомендации по программированию.

Внимание!

Здесь вы встретите предупреждения о возможных ошибках.

На заметку!

Нечто вроде вместилища разнообразных комментариев, которые не подпадают ни под одну из указанных выше категорий.

Пример из практики

Профессиональные программисты делятся с вами своими замечаниями, исходя из своего опыта.

Врезка

Врезка содержит более детальное обсуждение или дополнительные сведения по текущему вопросу.

Системы, использованные при разработке примеров для данной книги

Примеры, приведенные в этой книге, были разработаны с использованием Microsoft Visual C++ 7.1 (версия, поставляемая вместе с Microsoft Visual Studio .NET 2003) и Metrowerks CodeWarrior Development Studio 9 на компьютере Pentium с жестким диском, работающем под управлением операционной системы (ОС) Windows XP Professional. Большинство программ было протестировано с применением компилятора командной строки Borland C++ 5.5 и GNU gpp 3.3.3 в той же системе, компилятора Comeau 4.3.3 и GNU g++ 3.3.1 на IBM-совместимом компьютере Pentium под управлением ОС SuSE 9.0 Linux и с использованием компилятора Metrowerks Development Studio 9 на компьютере Macintosh G4, функционирующем под управлением OS 10.3. Расхождения, которые являются результатом несоответствия стандарту, в этой книге упоминаются в общем виде, например: "в более старых реализациях используется `ios::fixed` вместо `ios_base::fixed`". В книге также упоминаются некоторые известные ошибки и индивидуальные отличительные особенности старых компиляторов, которые могут приводить к затруднениям или путанице; в последующих версиях эти недочеты устранены.

Для любого программиста язык C++ откроет целый мир возможностей; изучайте его и наслаждайтесь своей работой!

ГЛАВА 1

С чего начать?

В этой главе:

- История и философия языков С и С++
- Сравнение процедурного и объектно-ориентированного программирования
- Язык С++: принципы объектно-ориентированного программирования, реализованные в языке С
- Язык С++: принципы обобщенного программирования, реализованные в языке С
- Стандарты языка программирования
- Порядок создания программы

Добро пожаловать в С++! Этот удивительный язык, сочетающий в себе функциональные возможности языка С и принципы объектно-ориентированного программирования, в девяностых годах прошлого столетия занял лидирующую позицию среди существующих языков программирования, и продолжает удерживать ее и по сей день. Своим происхождением он обязан языку программирования С, поэтому ему свойственны такие характеристики, как эффективность, компактность, быстродействие и переносимость. Благодаря принципам объектной ориентации, язык программирования С++ предлагает оригинальную методологию программирования, которая позволяет решать современные задачи программирования, степень сложности которых постоянно растет. Благодаря возможности использования шаблонов, язык С++ предлагает еще одну новую методологию программирования: обобщенное программирование. Во всем этом есть свои положительные и отрицательные стороны. Характеристики языка С++ свидетельствуют не только о том, что это очень мощный язык программирования, но также и том, что на его изучение нужно потратить довольно много времени.

Мы начнем с небольшого экскурса в историю происхождения языка С++, а затем перейдем к основным правилам создания программ на С++. Оставшаяся часть книги будет посвящена тому, чтобы научить читателя программированию на языке С++: вначале мы рассмотрим простые основы языка, после чего приступим к изучению объектно-ориентированного программирования (ООП), освоим весь его жаргон — объекты, классы, инкапсуляцию, сокрытие данных, полиморфизм и наследование — и напоследок перейдем к изучению обобщенного программирования. (Естественно, по мере изучения С++ эти профессиональные термины перейдут из разряда модных словечек в лексикон образованного человека.)

Изучение языка C++: с чем вы будете иметь дело

В языке программирования C++ объединены три самостоятельных традиции в программировании: традиция процедурного языка, примером которого является C; традиция объектно-ориентированного языка, представленного классами, которые язык C++ добавляет в C; традиция обобщенного программирования, поддерживаемого шаблонами C++. В этой главе как раз и рассматриваются упомянутые традиции. Однако прежде всего давайте разберемся с тем, как эти традиции влияют на изучение C++. Одна из причин использования языка C++ заключается в том, чтобы воспользоваться возможностями объектной ориентации. Для этого вы должны обладать навыками работы со стандартным языком C, который предлагает для C++ базовые типы, операции, управляющие структуры и синтаксические правила. Поэтому, зная язык программирования C, вы можете смело приступать к изучению языка C++. Это, однако, совсем не означает, что вам достаточно будет изучить одни лишь ключевые слова и конструкции. Переход от C к C++ будет стоить вам стольких же усилий, сколько изучение языка C с самого начала. Кроме этого, если вы знакомы с языком C, то при переходе к языку C++ вы должны будет отвыкнуть от привычного вам способа написания программ. Если вы не знаете C, то для изучения языка C++ вам придется освоить компоненты языка C, компоненты ООП и обобщенные компоненты, и по крайней мере, вам не придется отвыкать от привычного способа написания программ. Если у вас начинает складываться впечатление, что для изучения языка программирования C++ придется напрячь свои умственные способности, вы не ошибаетесь. Эта книга будет служить вам хорошим помощником в процессе обучения, поэтому вы сможете сэкономить свои силы.

Настоящая книга построена таким образом, что в ней рассматриваются как элементы языка C, лежащего в основе C++, так и новые компоненты C++, поэтому она будет полезна даже для неподготовленного читателя. Процесс обучения начинается с рассмотрения функциональных возможностей, общих для каждого языка программирования. Даже если вы и знакомы с языком C, эта часть книги все равно окажется очень полезной. В ней уделено внимание концепциям, значение которых будет раскрыто позже, и показаны различия между языками C++ и C. После того как вы усвоите основы языка C, мы перейдем к знакомству с суперструктурой C++. С этого момента мы займемся рассмотрением объектов и классов и вы узнаете, как они реализованы в C++. Затем мы поговорим о шаблонах.

Эта книга не является полным справочником по языку C++, и в ней не рассматриваются все его тонкости. Однако у вас есть возможность изучить все основные особенности языка программирования, в том числе шаблоны, исключения и пространства имен, которые были добавлены совсем недавно.

А теперь давайте рассмотрим вкратце историю происхождения языка C++.

Истоки языка C++: немного истории

Развитие компьютерных технологий в течение последних нескольких десятков лет происходило удивительно быстрыми темпами. Современные ноутбуки могут производить вычисления гораздо быстрее и хранить намного больше информации, чем

вычислительные машины 60-х годов прошлого столетия. (Очень немногие программисты могут вспомнить, как им приходилось возиться с колодами перфокарт, чтобы загрузить их в огромную компьютерную систему, занимающую отдельную комнату и имеющую немыслимый по тем временам объем памяти 100 Кбайт — сейчас этой памяти не достаточно, чтобы запустить хорошую игру для ПК.) В ногу со временем развивались и языки программирования. Их развитие не было столь впечатляющим, однако имело огромное значение. Для больших и мощных компьютеров требовались более сложные программы, для которых, в свою очередь, нужно было решать новые задачи программного управления и сопровождения.

В семидесятых годах прошлого века такие языки программирования, как C и Pascal, способствовали зарождению эры структурного программирования, привнесшего столь необходимые на то время порядок и дисциплину. Кроме того, что язык C предлагал инструментальные средства для структурного программирования, с его помощью можно было создавать компактные быстро выполняющиеся программы, а также справляться с аппаратными задачами, например управлением портами связи и дисковыми накопителями. Благодаря этим характеристикам, в восьмидесятых годах язык C стал доминирующим языком программирования. Наряду с ростом популярности языка C зарождалась и новая парадигма программирования: объектно-ориентированное программирование, или ООП, которое было реализовано в таких языках, как SmallTalk и C++. Давайте остановимся на рассмотрении языка C и ООП.

Язык программирования C

В начале семидесятых годов прошлого столетия Деннис Ритчи (Dennis Ritchie), сотрудник фирмы Bell Laboratories, участвовал в проекте по разработке операционной системы Unix. (*Операционная система*, или ОС, представляет собой совокупность программ, предназначенных для управления ресурсами компьютера и обслуживания взаимодействия пользователя и компьютера. Так, например, именно по команде ОС на экран монитора выводится системное приглашение, и именно ОС запускает для вас программы на выполнение.) Для своей работы Ритчи нуждался в языке программирования, который был бы лаконичным, с помощью которого можно было бы создавать компактные и быстро выполняющиеся программы, и посредством которого можно было бы эффективно управлять аппаратными средствами. По сложившейся на то время традиции программисты использовали для этих задач язык ассемблера, тесно связанный с внутренним машинным языком. Однако язык ассемблера является языком *низкого уровня*, что свидетельствует о прямой его зависимости от конкретного процессора компьютера. Поэтому если вы хотите, чтобы ассемблерная программа, написанная для одного компьютера, могла работать на компьютере другого типа, то, возможно, вам придется полностью переписать программу на другом языке ассемблера. Если, например, вы решили приобрести себе новый автомобиль и обнаружили, что конструкторы изменили расположение элементов системы управления и их назначение, и вам теперь нужно заново научиться водить машину, то один раз в жизни это можно пережить. Однако ОС Unix предназначалась для работы на самых разнообразных типах компьютеров (или платформах), поэтому о написании одной и той же программы для каждого компьютера не могло быть и речи. Таким образом, нужно было создать язык программирования *высокого уровня*, целью которого являлось бы решение задач, а не обслуживание определенного аппаратного устройства.

Специальные программы, называемые *компиляторами*, переводят язык программирования высокого уровня на внутренний язык определенного компьютера. Поэтому одну и ту же программу, написанную на языке программирования высокого уровня, можно запускать на различных платформах, используя для этого различные компиляторы. Ритчи нужен был язык, который сочетал бы в себе эффективность языка низкого уровня и возможность доступа к аппаратным средствам с универсализмом и переносимостью языка высокого уровня. Поэтому, будучи знакомым со старыми языками программирования, он создал язык С.

Философия программирования на языке С

Поскольку С++ прививает языку С новые принципы программирования, мы должны сначала изучить принципы программирования на языке С. В общем случае компьютерные языки программирования имеют дело с двумя концепциями — данные и алгоритмы. *Данные* — это та информация, которую использует и обрабатывает программа. *Алгоритмы* — это методы, используемые программой (рис. 1.1). Как и большинство распространенных в то время языков программирования, язык С был *процедурным* языком программирования. Это означает, что в этом языке акцент ставился на обработку данных с помощью алгоритма. Суть процедурного программирования заключается в том, чтобы запланировать действия, которые должен предпринять компьютер, и с помощью языка программирования их реализовать. В программе предварительно описывается некоторое количество процедур, которые должен будет выполнить компьютер, чтобы получить определенный результат. Здесь в качестве примера можно упомянуть кулинарный рецепт, в котором записан порядок действий, выполнив которые кондитер сможет испечь пирог.

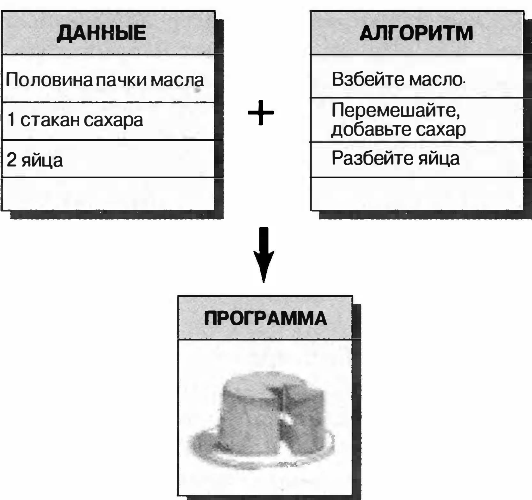


Рис. 1.1. Данные + алгоритмы = программа

В первых процедурных языках программирования, к которым относятся FORTRAN и BASIC, с увеличением размера программ возникали организационные проблемы. Что это значит? В программах очень часто используются операторы ветвления, ко-

торые передают выполнение тому или иному набору инструкций в зависимости от результата проверки. В большинстве старых программ было так много слишком запутанных переходов (на профессиональном жаргоне такой подход называется "спагетти-подобным программированием"), что при прочтении программу совершенно невозможно было понять, и попытка модифицировать такую программу сулила одни неприятности. Чтобы выйти из ситуации такого рода, специалисты по вычислительной технике разработали более дисциплинированный стиль программирования, называемый *структурным программированием.* Язык программирования С обладает всеми возможностями для реализации этого подхода. Например, в структурном программировании ветвление (выбор следующей инструкции для выполнения) осуществляется с помощью небольшого набора удобных и гибких конструкций. В языке С эти конструкции (циклы `for`, `while`, `do while` и оператор `if else`) включены в его собственный словарь.

Другим новшеством было так называемое *нисходящее проектирование*, или *проектирование сверху вниз.* В языке С эта идея состояла в том, чтобы разбить большую программу на небольшие, более управляемые задачи. Если после разбиения одна из задач все равно остается большой, вы продолжаете этот процесс до тех пор, пока программа не будет разделена на небольшие модули, с которыми будет просто работать. (Организуйте свой процесс обучения. Ах, нет! Приведите в порядок свой рабочий стол, картотеку, и наведите порядок на книжных полках. Ах, нет! Начните со стола, наведите порядок в каждом выдвижном ящике, начните со среднего. Да, судя по всему, я могу справиться с этой задачей.) Этот подход вполне осуществим в языке С, так как он позволяет разрабатывать программные модули, называемые *функциями*, которые отвечают за выполнение конкретной задачи. Как вы могли заметить, в структурном программировании отображено процедурное программирования, в том смысле, что программа представляется в виде действий, которые она должна выполнить.

Переход к С++: объектно-ориентированное программирование

Невзирая на то, что благодаря принципам структурного программирования сопровождение программ стало более четким, надежным и простым, написать большую программу все еще было непросто. Новый подход к решению этой задачи был воплощен в объектно-ориентированном программировании (ООП). В отличие от процедурного программирования, в котором акцентируется алгоритм, в ООП акцентируются данные. Вместо того чтобы пытаться решать задачу, приспосабливая ее к процедурному подходу в языке программирования, в ООП язык программирования приспосабливается к решению задачи. Суть заключается в том, чтобы создать такие формы данных, которые могли бы выразить особенности решаемой задачи.

В языке программирования С++ существуют такие понятия, как *класс*, который представляет собой спецификацию, описывающую такую новую форму данных, и *объект*, который представляет собой конкретную структуру данных, созданную в соответствии с этой спецификацией. Например, класс может описывать общие характеристики руководителя компании (фамилия и имя, занимаемая должность, годовой доход, необычные способности и тому подобное), а объект может представлять конкретно человека (Гилфорд Шипблат, вице-президент компании, годовой доход

составляет $325 000, знает способ восстановления реестра Windows). Вообще, класс описывает, какие данные используются для отображения объекта *и* какие операции могут быть выполнены над этими данными. Можно, например, задать класс для описания треугольника. В некоторой части данных может быть указано расположение вершин треугольника, высота и ширина, цвет и стиль линии контура, цвет шаблона для закраски треугольника. В операционной части этого описания могут быть описаны методы перемещения треугольника, изменения его размеров, вращения, изменения цвета и шаблонов, копирования треугольника для переноса в другое местоположение. Если позже вы попробуете воспользоваться своей программой, чтобы нарисовать треугольник, она сможет создать объект в соответствии с его описанием. Объект будет содержать все значения, описывающие треугольник, и для его изменения вы можете воспользоваться методами класса. Если вы нарисуете два треугольника, программа создаст два объекта, по одному для каждого треугольника.

Подход к проектированию программ, применяемый в ООП, заключается в следующем. Вначале вы разрабатываете классы, в точности представляющие все те элементы, с которыми будет работать программа. Например, программа рисования может работать с классами, представляющими треугольники, линии, окружности, кисти, перья и тому подобное. В описаниях классов, как вы теперь уже знаете, указываются разрешенные операции для каждого класса, такие как перемещение по кругу или вращение вокруг оси. Затем можно продолжить процесс разработки программы, уже с помощью объектов для этих классов. Процесс перехода с нижнего уровня организации, например с классов, до верхнего уровня — проектирования программы, называется *восходящим* программированием.

Компоновка данных и методов в описании класса — не единственный прием, который можно реализовать с помощью ООП. В нем открываются возможности для повторного использования кода, что позволяет сократить большой объем работы. Сокрытие информации позволяет предотвратить несанкционированный доступ к данным. Благодаря полиморфизму можно создавать множество описаний для операций и функций с программным контекстом, уточняющим, какое описание используется. Благодаря наследованию можно порождать новые классы на основе уже существующих классов. Как видите, ООП предлагает множество новых идей и, в отличие от процедурного программирования, другой принцип программирования. Вместо того чтобы сосредоточиваться на задачах, вы сосредоточиваетесь на концепциях представления. В некоторых случаях вместо нисходящего программирования можно применять принцип восходящего программирования. Все эти вопросы, сопровождаемые простыми и понятными примерами, будут рассмотрены в этой книге.

Придумать полезный и надежный класс не так-то просто. К счастью, в языках программирования, в которых реализованы принципы ООП, этот процесс может быть упрощен за счет заимствования существующих классов. Производители программного обеспечения предлагают большое количество полезных библиотек классов, включая такие библиотеки, использование которых упрощает написание программ для сред вроде Windows или Macintosh. Одно из замечательных преимуществ языка C++ заключается в том, что он позволяет без особых сложностей использовать повторно и адаптировать существующий надежный код.

C++ и обобщенное программирование

Еще одной парадигмой программирования, реализованной в языке C++, является обобщенное программирование. Как и ООП, обобщенное программирование направлено на упрощение повторного использования кода и на абстрагирование общих концепций. Однако в ООП акцент программирования ставится на данные, а в обобщенном программировании — на алгоритмы. И его цель тоже другая. ООП — это инструмент для управления большими проектами, тогда как обобщенное программирование предлагает инструменты для выполнения простых задач, например сортировку данных или слияние списков. Термин *обобщенный* приписывается коду, тип которого является независимым. Данные в языке программирования C++ могут быть представлены разными способами: в виде целых чисел, чисел с дробной частью, в виде символов, строк символов и в виде определяемых пользователем сложных структур нескольких типов. Например, если вам необходимо сортировать данные различных типов, то в общем случае для каждого типа потребовалось бы создавать отдельную функцию сортировки. В обобщенном программировании язык расширен, поэтому вы можете один раз написать функцию для обобщенного (то есть неопределенного) типа и использовать ее для множества существующих типов. Для этого в языке C++ предусмотрены шаблоны.

Происхождение языка программирования C++

Как и язык C, C++ был создан в начале восьмидесятых годов прошлого столетия в Bell Laboratories, где работал Бьерн Страуструп (Bjarne Stroustrup). Вот что об этом говорит сам Страуструп: "C++ был создан главным образом потому, что мои друзья, да и я сам, не имели никакого желания писать программы на ассемблере, C или какомлибо языке программирования высокого уровня, существовавшем в то время. Задача заключалась в том, чтобы сделать процесс написания хороших программ простым и более приятным для каждого программиста".

Пример из практики: домашняя страница Бьерна Страуструпа

Бьерн Страуструп, создатель языка программирования C++, является автором нескольких широко известных справочных руководств. Его персональный Web-сайт, размещенный в AT&T Labs Research, должен стать вашей стартовой страницей:

`www.research.att.com/~bs`

На этом сайте можно найти интересные исторические предпосылки возникновения C++, биографические материалы Бьерна Страуструпа и часто задаваемые вопросы. Занимательно, что наиболее часто Страуструпа спрашивают о том, как правильно произносится его имя и фамилия. Загрузите файл `.WAV`, чтобы услышать это самому!

Страуструп стремился прежде всего к тому, чтобы сделать язык C++ полезным, а не внедрить какой-нибудь новый принцип или стиль программирования. Функциональные особенности C++ в первую очередь должны удовлетворять требования программиста, а не быть воплощением красивой теории. За основу C++ Страуструп взял язык C, известный своей лаконичностью, пригодностью для систем-

ного программирования, широкой доступностью и тесной взаимосвязью с ОС Unix. Принципы ООП были позаимствованы в языке моделирования Simula67. Страуструпу удалось реализовать в языке С принципы ООП и поддержку обобщенного программирования без существенного изменения языка. Поэтому язык программирования С++ в первом приближении можно рассматривать как расширенную версию языка С; в пользу этого свидетельствует тот факт, что любая рабочая программа на языке С — это также и программа на языке С++. Существуют, правда, некоторые незначительные отличия, но не более того. Программы, написанные на С++, могут использовать существующие библиотеки программного обеспечения для языка С. *Библиотека* представляет собой совокупность программных модулей, которые могут быть вызваны из программы. В этих библиотеках предлагаются надежные и проверенные решения ко многим наиболее часто встречающимся задачам программирования, позволяя тем самым экономить ваши усилия и время. Распространение языка С++ стало возможным во многом благодаря именно библиотекам.

Название языка С++ происходит от операции приращения (инкремента) ++ в языке С, при которой значение переменной увеличивается на единицу. Таким образом, имя С++ в точности отображает расширенную версию языка С.

Компьютерная программа переводит практическую задачу в последовательность действий, которые должен выполнить компьютер. Благодаря принципам ООП, язык С++ может устанавливать связь с концепциями, составляющими задачу, а благодаря возможностям языка С, он может работать с аппаратными средствами (рис. 1.2). Такое сочетание возможностей способствовало росту популярности языка С++. Процесс переключения с одного аспекта программы на другой можно представить себе как процесс переключения передач в автомобиле. (В действительности, некоторые приверженцы ООП считают, что реализация принципов ООП в языке С — это то же самое, что попытка научить летать поросенка.) Кроме этого, поскольку С++ реализует принципы ООП в языке С, их можно просто проигнорировать. Однако в этом случае вы утратите массу возможностей.

После того, как С++ действительно стал популярным языком программирования, Страуструп ввел шаблоны, открыв возможности для обобщенного программирования. И только после этого стало ясно, насколько удачным было использование шаблонов — их значение оказалось еще большим, чем реализация ООП, хотя кто-то может и не согласиться с этим. То обстоятельство, что в языке С++ объединены ООП, обобщенное программирование и более традиционный процедурный подход свидетельствует о том, что в С++ утилитарный подход господствует над идеологическим, и это одна из причин популярности данного языка.

Переносимость и стандарты

Представьте, что у себя на работе вы написали удобную программу на языке С++ для старенького ПК Pentium, и вдруг начальник вашего отдела решил заменить ваш компьютер на Macintosh G5 — компьютер с другим процессором и другой операционной системой. Сможете ли вы запустить свою программу на новой платформе? Естественно, вам придется повторно компилировать программу, используя компилятор С++ для новой платформы. А нужно ли что-нибудь изменять в написанном вами коде? Если вы, ничего не изменяя в своей программе, сможете ее повторно компилировать и без помех запускать, то такую программу мы называем *переносимой*.

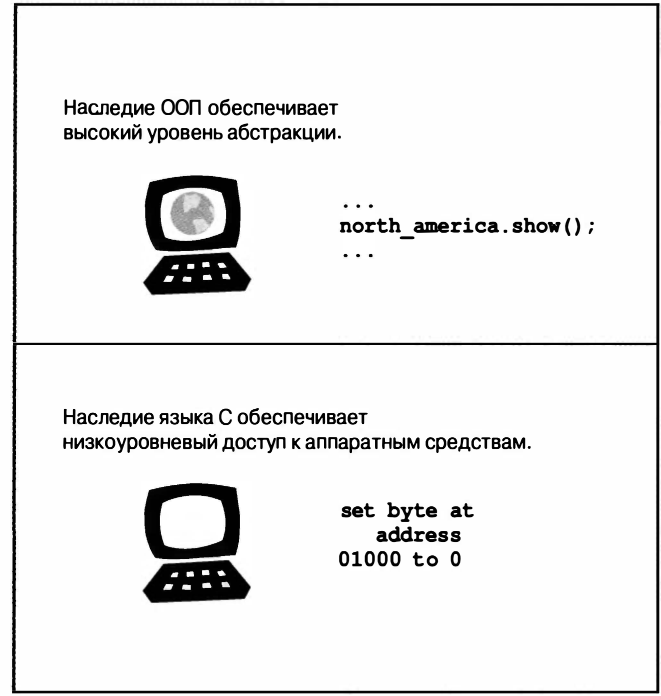


Рис. 1.2. Двойственность языка С++

Чтобы сделать программу переносимой, нужно справиться с двумя проблемами. Первая — это аппаратное обеспечение. Программа, настроенная на работу с конкретным аппаратным обеспечением, вряд ли будет переносимой программой. Если программа напрямую управляет платой видеоадаптера IBM PC, то на платформе Sun, например, он выдаст сплошную тарабарщину. (Проблему переносимости можно свести к минимуму, если локализовать части программы, зависящие от аппаратного обеспечения, в модулях функций; затем эти модули можно будет просто переписать.) В этой книге мы не будем рассматривать такой способ программирования.

Второй проблемой является несоответствие языков программирования. Вы согласитесь с тем, что в реальном мире тоже существует проблема с разговорными языками? Жители Бруклина, например, могут не понять сводку новостей на диалекте Йоркшира, и это притом, что все они разговаривают на английском языке. Такая же ситуация и в мире компьютеров: среди языков программирования можно различить характерные "диалекты". Будет ли реализация С++ на Windows XP такой же, как и на платформах Red Hat Linux или Macintosh OS X? Несмотря на желание большинства конструкторов добиться совместимости их собственных версий С++ с другими версиями, сделать это практически невозможно. Выход один — нужно утвердить какой-нибудь один общий стандарт, который бы описывал работу языка программирования. В Национальном институте стандартизации США (American National Standards Institute — ANSI) в 1990 году был сформирован комитет (ANSI X3J16), задача которого заключалась в разработке стандарта для языка программирования С++. (Для языка С в институте ANSI уже был подготовлен соответствующий стандарт.) Вскоре

к этому процессу подключилась и Международная организация по стандартизации (International Organization for Standardization — ISO), у которой на тот момент был сформирован свой собственный комитет (ISO-WG-21). В результате усилия комитетов ANSI и ISO были объединены и работа по созданию стандарта для языка C++ велась совместно. Три раза в году эти комитеты проводили совместные обсуждения, поэтому для отчетности об их работе мы будем говорить как об одном комитете ANSI/ISO. В решении, принятом комитетом ANSI/ISO относительно создания стандарта, было сказано, что C++ стал важным и широко распространенным языком программирования. В нем также сказано, что язык C++ достиг некоторого уровня зрелости и совершенности, поэтому нет смысла разрабатывать стандарты для быстро развивающегося языка программирования. Тем не менее, с момента работы комитета ANSI/ISO язык C++ претерпел значительные изменения.

Работа по утверждению стандарта для языка C++ в комитете ANSI/ISO началась в 1990 году. В последующие годы комитет огласил некоторые предварительные рабочие доклады. В апреле 1995 года для открытого обсуждения был представлен проект комитета (Committee Draft — CD), а в декабре 1996 года вышел второй проект (CD2). В этих документах были не только уточнены существующие особенности языка C++, но и описаны расширения языка: исключения, динамическая идентификация типов (Runtime Type Identification — RTTI), шаблоны и стандартная библиотека шаблонов (Standard Template Library — STL). Окончательный международный стандарт (International Standard, ISO/IEC 14882:1998) был утвержден в 1998 году тремя организациями: ISO, Международной электротехнической комиссией (International Electrotechnical Commission — IEC) и ANSI. В 2003 году было опубликовано второе издание стандарта C++ (IOS/IEC 14882:2003); новое издание представляет собой технически исправленную и переработанную версию. В нем были исправлены ошибки первой редакции (отмечены опечатки, устранены неясности и тому подобное), при этом особенности языка программирования не были изменены. Наша книга написана в соответствии с этим стандартом.

В стандарте ANSI/ISO для языка C++ использованы правила стандарта ANSI для языка C, поскольку считается, что язык C++ является суперструктурой языка C. Это означает, что любая действительная программа на языке C в идеале должна быть действительной программой на языке C++. Между ANSI C и соответствующими правилами для языка C++ имеются некоторые различия, однако все они несущественные. Так, например, ANSI C включает некоторые возможности, которые были впервые представлены в C++: определение прототипа функции и спецификатор типа `const`.

До выхода ANSI C в сообществе программистов C в качестве стандарта использовался стандарт, основанный на книге *Язык программирования C* (Издательский дом "Вильямс", 2005 год), написанной Брайаном Керниганом и Деннисом Ритчи. Этот стандарт часто назывался как K&R C; после появления ANSI C более простой стандарт K&R C в наше время называется *классическим C*.

Стандарт ANSI C описывает не только язык программирования C, но и стандартную библиотеку C, которая должна поддерживаться в реализациях ANSI C. Эта библиотека используется также и в C++; в книге мы называем ее *стандартной библиотекой C* или просто *стандартной библиотекой*. Кроме этого, C++ стандарта ANSI/ISO предлагает стандартную библиотеку классов C++.

Совсем недавно стандарт C был пересмотрен; новый стандарт, часто именуемый C99, был принят ISO в 1999 году и ANSI в 2000 году. В этот стандарт включены новые

возможности языка C, например, новый целочисленный тип, который поддерживается некоторыми компиляторами C++. Несмотря на то что эти особенности не описаны в нынешнем стандарте C++, они могут быть включены уже в следующий стандарт.

До того как комитет ANSI/ISO C++ начал разработку стандарта, многие программисты ориентировались на самую последнюю версию C++, разработанную Bell Laboratories. Например, компилятор может быть совместимым с версиями C++ 2.0 и 3.0.

Язык C++ продолжает совершенствоваться, и уже ведется разработка следующей версии стандарта. Новая версия имеет рабочее название C++0X, поскольку ее появление ожидается в конце текущего десятилетия, примерно в 2009 году.

Эта книга написана на основе второго издания стандарта ISO/ANSI для C++ (ISO/IEC 14882:2003), поэтому примеры, представленные в книге, будут работать в любой реализации C++, совместимой с этим стандартом. (По крайней мере, именно так и должно быть с точки зрения переносимости, и мы на это надеемся.) Однако стандарт C++ Standard по-прежнему остается новым, и вы можете найти в компиляторах несколько расхождений с ним. Например, к моменту написания данной книги в некоторых компиляторах C++ отсутствовали пространства имен и новейшие возможности работы с шаблонами, а поддержка стандартной библиотеки шаблонов, описываемой в главе 16, была неполной. В некоторых старых Unix-системах используется предварительный транслятор, который передает транслированный код компилятору C, не полностью совместимому со стандартом ANSI, поэтому некоторые элементы языка остаются нереализованными и не поддерживаются некоторые функции стандартной библиотеки ANSI и заголовочные файлы. Даже если компилятор будет соответствовать стандарту, некоторые характеристики, такие как количество байтов, используемых для хранения целого числа, будут зависеть от реализации.

Прежде чем вплотную заняться изучением языка C++, давайте поговорим о том, каким является порядок создания программ.

Порядок создания программы

Предположим, что вы написали программу на C++. Как вы собираетесь ее запускать? Действия, которые вы должны предпринять, зависят от вашей компьютерной среды и от используемого компилятора C++, однако в общем случае они должны быть примерно следующими (рис. 1.3):

1. С помощью текстового редактора или подобной ему программы напишите свою программу и сохраните ее в файле. Это будет *исходный код* вашей программы.
2. Выполните компиляцию исходного кода. Для этого необходимо запустить программу, которая выполняет трансляцию (перевод) исходного кода на внутренний язык, называемый *машинным языком* главного компьютера. Файл, содержащий транслированную программу, является *объектным кодом* (или конечной программой) вашей программы.
3. Свяжите свой объектный код с дополнительным кодом. Например, программы на языке C++ обычно используют *библиотеки*. В библиотеке C++ содержится объектный код для набора компьютерных подпрограмм, называемых функциями, которые выполняют такие задачи, как отображение информации на экране монитора, нахождение квадратного корня числа и так далее. В процессе связывания ваш объектный код комбинируется с объектным кодом для используемых

вами функций и с некоторым стандартным кодом запуска для формирования исполняемой программы, или программы времени выполнения. Файл, содержащий этот готовый продукт, называется *исполняемым кодом.*

Термин *исходный код* вы будете постоянно встречать на страницах этой книги, поэтому постарайтесь запомнить его.

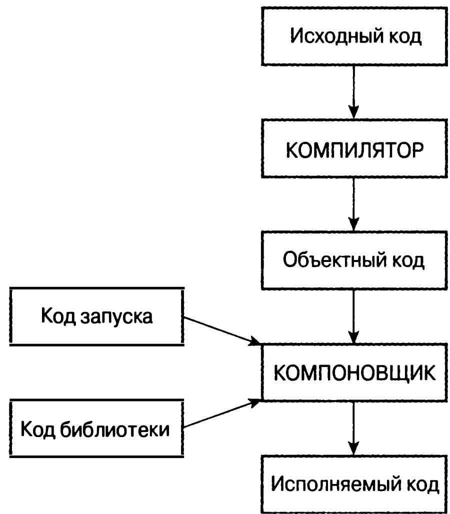


Рис. 1.3. Этапы выполнения программы

Программы в этой книге являются обобщенными программами и должны запускаться в любой системе, в которой поддерживается современная версия языка C++. (Однако вам может понадобиться одна из самых последних версий для работы с пространствами имен и новейшими особенностями шаблонов.) Этапы сборки программы могут быть разными. Давайте остановимся на рассмотрении этих вопросов.

Создание файла исходного кода

В оставшейся части этой книги мы будет говорить обо всем, что связано с файлом исходного кода; в этом разделе речь пойдет о механизме создания этого файла. В некоторых реализациях языка C++, таких как Microsoft Visual C++, Borland C++ (разнообразные версии), Watcom C++, Digital Mars C++ и Metrowerks CodeWarrior предлагается так называемая *интегрированная среда разработки* (Integrated Development Environment — IDE), которая позволяет управлять всеми этапами разработки программы, включая редактирование, из одной главной программы. В других реализациях языка программирования C++, таких как AT&T C++ или GNU C++ на платформах Unix и Linux, а также в свободно распространяемых версиях компиляторов Borland и Digital Mars, обрабатываются только этапы компиляции и компоновки, и все команды вы должны вводить в системной командной строке. В этих случаях для создания и изменения исходного кода можно использовать любой доступный текстовый редактор. В системе Unix можно применять редакторы `vi`, `ed` или `emacs`. В системе DOS можно работать в `edlin` либо `edit`, или в любом другом доступном редакторе

текстов программ. Можно даже использовать текстовый процессор при условии, что искомый файл вы будете сохранять в формате текстового файла DOS ASCII, а не в специальном формате текстового процессора.

Присваивая файлу исходного кода имя, следует использовать соответствующий суффикс, посредством которого файл обозначается как файл C++. Благодаря этому суффиксу не только вы, но и компилятор будет знать, что данный файл является исходным кодом C++. (Если Unix-компилятор пожалуется на неверное магическое число ("bad magic number"), следует иметь в виду, что это его излюбленный способ указывать на неправильный суффикс.) Суффикс начинается с точки, за которой следует символ или группа символов, называемых *расширением* (рис. 1.4).

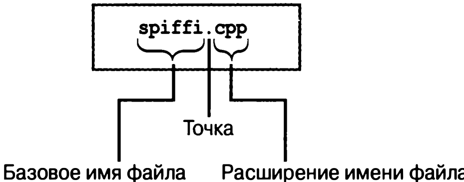


Рис. 1.4. Элементы имени файла исходного кода

Выбор расширения будет зависеть от реализации C++. В табл. 1.1 перечислены некоторые распространенные варианты. Например, `spiffy.C` — это допустимое имя Unix-файла исходного кода. Обратите внимание, что в Unix важно соблюдать регистр символов: символ `C` должен быть записан в верхнем регистре. Следует отметить, что расширения, записанные в нижнем регистре, тоже допускаются, однако в стандартной версии языка C используется именно такой формат расширения. Поэтому чтобы избежать путаницы в Unix-системах, следует применять `c` для программ на языке C и `C` для программ на языке C++. Можно также использовать один или два дополнительных символа: в некоторых Unix-системах, например, можно использовать расширения `cc` и `cxx`. В DOS, более простой системе по сравнению с Unix, нет разницы в том, в каком регистре будет введен символ, поэтому для различия программ на C и C++ в DOS-реализациях применяются дополнительные символы (табл. 1.1).

Таблица 1.1. Расширения исходного кода

Реализация C++	Расширение исходного кода
Unix	C, cc, cxx, c
GNU C++	C, cc, cxx, cpp, c++
Digital Mars	cpp, cxx
Borland C++	cpp
Watcom	cpp
Microsoft Visual C++	cpp, cxx, cc
Metrowerks CodeWarrior	cpp, cp, cc, cxx, c++

Компиляция и компоновка

Первоначально Страуструп реализовал C++ с программой компилятора C++ в C, и не стал разрабатывать прямой компилятор C++ для объектного кода. Эта программа, называемая `cfront` (сокращение от *C front end*), транслировала исходный код C++ в исходный код C, который затем можно было компилировать с помощью стандартного компилятора C. Этот подход способствовал росту популярности C++ в среде программистов C. В других реализациях этот подход использовался для переноса C++ на другие платформы. По мере совершенствования языка C++ и роста его популярности все большее число конструкторов предпочитали создавать компиляторы C++, которые генерировали объектный код непосредственно из исходного кода C++. Такой прямой подход ускорял процесс компиляции и обособлял язык C++.

Пользователи часто даже не замечают разницу между транслятором `cfront` и компилятором. Например, в Unix-системе команда `CC` может сначала передать вашу программу транслятору `cfront`, а затем автоматически передать выходные данные транслятора компилятору C, называемому `cc`. С этого момента мы будем употреблять термин *компилятор* для обозначения комбинаций трансляции и компиляции. Способ компиляции зависит от реализации, и в следующих разделах мы рассмотрим некоторые варианты. В этих разделах будут описаны только основные этапы, однако это вовсе не означает, что вам не следует изучать документацию по своей системе.

Если у вас есть транслятор `cfront` и вы знаете язык C, можно испытать трансляции ваших программ на C++, чтобы на деле проверить реализацию некоторых элементов языка C++.

Компиляция и связывание в Unix

Вызов обычного компилятора C++ в Unix-системе осуществляется посредством команды `CC`. Однако современные компьютеры на платформе Unix могут вообще не иметь компилятора, могут иметь собственный компилятор или сторонний компилятор, будь-то коммерческий или свободно распространяемый вроде GNU `g++`. Во многих этих случаях (но не тогда, когда компилятора вообще нет!) команда `CC` будет работать, вызывая компилятор, используемый в данной системе. Для простоты можно полагать, что команда `CC` будет доступна, но в следующих разделах она может быть заменена другой командой.

Команда `CC` используется для компиляции ваших программ. Имя команды вводится в верхнем регистре для того, чтобы отличить его от обычного компилятора C в Unix — `cc`. Компилятор `CC` является компилятором командной строки, что означает ввод команд компиляции в командной строке Unix.

Например, чтобы выполнить компиляцию файла `spiffy.C` исходного кода C++, в строке приглашения Unix можно ввести следующую команду:

```
CC spiffy.C
```

Если, благодаря опыту, посвящению или удаче в вашей программе не окажется ошибок, компилятор сгенерирует файл объектного кода с расширением `o`. В нашем случае компилятор создаст файл с именем

```
spiffy.o
```

Затем компилятор автоматически передает файл объектного кода системному компоновщику (редактору связей) — программе, которая комбинирует ваш код с кодом библиотеки с целью формирования исполняемого файла. По умолчанию исполняемому файлу присваивается имя a.out. Если вы используете только один файл исходного кода, тогда компоновщик удалит файл spiffy.o, поскольку в нем больше нет необходимости. Чтобы запустить программу, достаточно ввести имя исполняемого файла:

```
a.out
```

Обратите внимание, что если вы компилируете новую программу, то новый исполняемый файл a.out заменит предыдущий файл a.out. (Это происходит потому, что исполняемые файлы занимают достаточно много места, поэтому замена старых исполняемых файлов позволяет сократить объем занимаемой памяти.) Если вы разрабатываете исполняемую программу, которую необходимо сохранить на будущее, используйте Unix-команду mv, чтобы изменить имя исполняемого файла.

В языке C++, как и в C, одна программа может занимать несколько файлов. (Многие программы, рассматриваемые нами в этой книге в главах 8–16, также занимают несколько файлов.) В этом случае можно компилировать программу, перечислив все эти файлы в командной строке, как показано далее:

```
CC my.C precious.C
```

Если у вас имеется несколько файлов исходного кода, то компилятор не будет удалять файлы объектного кода. Так, если вы просто измените файл my.C, то повторную компиляцию программы можно будет выполнить с помощью следующей команды:

```
CC my.C precious.o
```

В результате будет повторно компилирован файл my.C и связан с ранее компилированным файлом precious.o.

Иногда возникает необходимость явным образом указывать некоторые библиотеки. Например, чтобы обратиться к функциям, определенным в библиотеке математических операций, вам, возможно, придется добавить флаг -lm в командной строке:

```
CC usingmath.C -lm
```

Компиляция и связывание в Linux

В системах Linux чаще всего используется компилятор g++ — компилятор GNU C++, разработанный фондом свободного программного обеспечения Free Software Foundation. Компилятор входит в состав большинства дистрибутивов Linux, однако может устанавливаться и отдельно. Компилятор g++ работает подобно стандартному компилятору Unix. Например, в результате выполнения команды

```
g++ spiffy.cxx
```

будет создан исполняемый файл с именем a.out.

В некоторых версиях компилятора необходимо привязываться к библиотеке C++:

```
g++ spiffy.cxx -lg++
```

Чтобы компилировать несколько файлов исходного кода, достаточно перечислить их в командной строке:

```
g++ my.cxx precious.cxx
```

В результате будет создан исполняемый файл с именем `a.out` и два файла объектного кода, `my.o` и `precious.o`. Если впоследствии вы измените только один файл исходного кода, например `my.cxx`, то повторную компиляцию можно будет выполнить, используя `my.cxx` и `precious.o`:

```
g++ my.cxx precious.o
```

Существует еще один вариант компилятора C++ — Comeau (`www.comeaucomputing.com`). Для его работы необходимо наличие компилятора GNU. Однако компилятор Comeau предлагает наиболее полную и точную реализацию стандарта C++.

Компилятор GNU может работать на различных платформах, в том числе в режиме командной строки на ПК под управлением Windows и на различных платформах в Unix-системах.

Компиляторы командной строки для MS-DOS

Наиболее дешевый вариант компиляции программ на C++ в системах на базе Windows заключается в том, чтобы загрузить свободно распространяемый компилятор командной строки, работающий в окне MS-DOS в системе Windows. Версия компилятора G++ для MS-DOS называется `gpp`; этот компилятор можно найти на сайте `www.borland.com/bcppbuilder/freecompiler`. Более свежая версия этого компилятора поставляется с относительно недорогой персональной версией Borland C++Builder X. На сайте `www.digitalmars.com` можно найти бесплатный компилятор командной строки, разработанный компанией Digital Mars. Устанавливается C++Builder X автоматически. Что же касается всего остального, то вам необходимо внимательно читать инструкции по установке, поскольку некоторые процессы установки не автоматизированы.

Чтобы использовать компилятор `gpp`, вы должны сначала открыть окно MS-DOS. Чтобы компилировать файл исходного кода с именем `great.cpp`, в командной строке введите следующую команду:

```
gpp great.cpp
```

В случае успешной компиляции будет сформирован исполняемый файл `a.exe`.

Чтобы использовать компилятор Borland, работающий в онлайновом режиме, сначала необходимо открыть окно MS-DOS. Затем, чтобы компилировать файл исходного кода с именем `great.cpp`, в командной строке введите следующую команду:

```
bcc32 great.cpp
```

В случае успешной компиляции будет сформирован исполняемый файл `great.exe`.

Компиляторы для Windows

Компиляторов для Windows столь много, и так часто появляются их модификации, что нет смысла рассматривать их все по отдельности. Среди наиболее популярных производителей можно выделить Microsoft, Borland, Metrowerks и Digital Mars. Несмотря на разный дизайн и выполняемые задачи, все они характеризуются одинаковыми особенностями.

Как правило, нужно создать проект программы и добавить в него один или несколько файлов, составляющих программу. Каждый производитель предлагает IDE-среду с набором меню и, возможно, с программой автоматизированного помощника,

которую удобно использовать в процессе создания проекта. Очень важно определиться с тем, какой тип будет иметь создаваемая вами программа. Обычно компилятор предлагает несколько вариантов, среди которых приложение для Windows, приложение Windows MFC (Microsoft Foundation Classes — библиотека базовых классов Microsoft), динамически подключаемая библиотека, элемент управления ActiveX, программа, исполняемая в режиме DOS или в режиме командной строки, статическая библиотека или консольное приложение. Некоторые из них могут иметь 16- и 32-разрядные версии.

Поскольку программы в этой книге являются обобщенными, вам необходимо будет избегать вариантов, для которых требуется код для определенной платформы, например, приложение для Windows. Вместо этого нужно запускать программу в режиме командной строки. Выбор режима зависит от компилятора. Для Microsoft Visual C++ используется вариант Win32 Console Application. (Если вы используете Visual Studio .NET, можно выбрать опцию Empty Project (Пустой проект) в окне Application Settings (Настройки приложения).) Компиляторы Metrowerks предлагают вариант Win32 Console C++ App и Win32 WinSIOUX C++ App, каждый из которых будет работать. (Первый режим запускает компилированную программу в окне DOS; второй — в стандартном окне Windows.) В некоторых версиях Borland предлагается режим EasyWin, при котором эмулируется сеанс DOS; в других версиях предлагается консольный вариант (Console). В общем случае необходимо искать в меню такие варианты, как Console, character-mode, DOS executable и попытаться воспользоваться ими.

После того как ваш проект будет готов, вам нужно будет выполнить компиляцию и компоновку своей программы. В IDE-среде обычно предлагается несколько вариантов, такие как Compile, Build, Make, Build All, Link, Execute и Run (но не обязательно все варианты сразу в одной IDE-среде!):

- Compile обычно означает компиляцию кода в открытом в настоящий момент файле.
- Build или Make обычно означает компиляцию кода для всех файлов исходного кода, входящих в состав данного проекта. Часто этот процесс является инкрементным. То есть, если в проекте было три файла, и вы изменили только один из них, то повторно компилирован будет только этот файл.
- Build All обычно означает компиляцию всех файлов исходного кода с самого начала.
- Как уже было сказано ранее, Link означает объединение компилированного исходного кода с необходимым библиотечным кодом.
- Run или Execute означает запуск программы. Обычно если вы еще не завершили выполнение предыдущих этапов, команда Run выполнит их за вас, прежде чем будет запущена программа.

Если вы нарушите правила языка, компилятор выдаст сообщение об ошибке и укажет на строку кода, в которой она была найдена. К сожалению, если вы еще недостаточно хорошо знаете язык программирования, то понять смысл этого сообщения порой бывает достаточно трудно. В некоторых случаях ошибка может вкрасться перед указанной строкой, а бывает так, что одна ошибка порождает целую серию сообщений об ошибках.

Совет

Исправляя ошибки, начните с самой первой. Если вы не можете ее найти в указанной строке, проверьте предыдущую строку кода.

Имейте в виду, что принятие программы определенным компилятором вовсе не означает, что эта программа является допустимой программой на C++. И то, что определенный компилятор отвергает программу, не обязательно означает, что эта программа не является допустимой программой C++. Современные компиляторы в большей степени соответствуют принятому стандарту, нежели их предшественники. В наши дни компилятор Comeau (и другие пользователи интерфейса Edison Design Group) наиболее всего соответствуют стандарту.

Совет

Время от времени компиляторы, не закончив процесс создания программы, генерируют бессмысленные сообщения об ошибках, которые невозможно устранить. В подобных ситуациях следует воспользоваться командой Build All, чтобы начать процесс компиляции с самого начала. К сожалению, отличить эту ситуацию от другой, более распространенной, когда сообщение об ошибке только кажется бессмысленным, очень тяжело.

Обычно IDE-среда позволяет запускать программу во вспомогательном окне. В некоторых IDE-средах это окно закрывается после завершения выполнения программы, а в некоторых оно остается открытым. Если ваш компилятор закрывает окно, вы не успеете увидеть вывод программы, если только не умеете очень быстро читать и не обладаете фотографической памятью. Чтобы увидеть результат выполнения, в конце программы необходимо ввести следующий код:

```
    cin.get(); // добавьте этот оператор
    cin.get(); // и, возможно, этот тоже
    return 0;
}
```

Оператор `cin.get()` считывает следующее нажатие клавиши, поэтому программа будет находиться в режим ожидания до тех пор, пока вы не нажмете клавишу <Enter>. (Если не нажать на нее, программа не получит информацию ни от одной нажатой клавиши, поэтому не стоит нажимать другую клавишу.) Второй оператор необходим на случай, если программа оставит необработанным нажатие клавиши после предыдущего ввода информации в программе. Например, при вводе числа вы нажимаете соответствующую клавишу, а затем клавишу <Enter>. Программа может считать число, но оставить необработанным нажатие клавиши <Enter>, и затем считывает его в первом операторе `cin.get()`. Компилятор Borland C++Builder несколько отличается от более традиционных компиляторов. Его основная стихия — программирование для Windows. Чтобы использовать старые версии для обобщенных программ, выберите в меню File (Файл) команду New (Создать). Затем выберите Console App (Консольное приложение), в результате чего откроется окно, в котором будет выведен макет функции `main()`. В макете необходимо оставить две следующих нестандартных строки:

```
#include <vcl\condefs.h>
#pragma hdrstop
```

Для C++Builder X выберите в меню File (Файл) команду New⇨New Console (Создать⇨Новая консоль). Вы не получите основу функции `main()`. Вместо этого выберите в меню File (Файл) команду New File (Создать файл) и добавьте новый файл `.cpp` в свой проект.

C++ в компьютерах Macintosh

Основным компилятором C++ для компьютеров Macintosh является Metrowerks CodeWarrior. Он предлагает IDE-среды на базе проектов, которые, в основном, подобны компиляторам для Windows. Работа начинается с создания нового проекта. Выберите в меню File (Файл) команду New Project (Создать проект). Вам будет предложено выбрать один из типов проектов. Для CodeWarrior выберите MacOS: C/C++:ANSI C++ Console в старых версиях, MacOS:C/C++:Standard Console:Std C++ Console в недавних версиях или же MacOS C++ Stationery:Mac OS Carbon:Standard Console:C++ Console Carbon в последних версиях. (Можно выбрать и другие допустимые варианты; например, выбрать Classic вместо Carbon или C++ Console Carbon Altivec вместо простого C++ Console Carbon.)

Компилятор CodeWarrior создает небольшой файл исходного кода, который является частью исходного проекта. Можно попытаться выполнить компиляцию и запустить эту программу, чтобы проверить правильность настройки вашей системы. Однако после того как вы предложите свой собственный код, этот файл нужно будет удалить из своего проекта. Для этого выделите файл в окне проекта и выберите в меню Project (Проект) команду Remove (Удалить).

Затем необходимо добавить в проект свой исходный код. Для этого можно выбрать в меню File (Файл) команду New (Создать), чтобы создать новый файл, или Open (Открыть), чтобы открыть существующий файл. Вы должны использовать подходящий суффикс, например `.cp` или `.cpp`. Добавить этот файл в проект можно через меню Project (Проект). Для некоторых программ, представленных в этой книге, необходимо добавлять несколько файлов исходного кода. Когда все будет готово, выберите в меню Project (Проект) команду Run (Выполнить).

Совет

Чтобы сэкономить время, можно использовать только один проект для всех примеров программ. В этом случае следует удалить предыдущий пример файла исходного кода из списка проекта и добавить новый исходный код. В результате будет сэкономлено дисковое пространство.

В состав компилятора входит отладчик, который помогает обнаруживать причины возникновения ошибок во время выполнения.

Резюме

Компьютеры стали более мощными, компьютерные программы стали большими и сложными. Как результат, компьютерные языки стали более развитыми и теперь они упрощают управление процессом написания программ. Язык С наделен такими элементами, как управляющие структуры и функции, с помощью которых можно расширить возможности управления ходом выполнения программы и реализовать более структурированный, модульный способ написания программ. Для этих инструментальных средств в C++ поддерживается объектно-ориентированное программирование (ООП), а также обобщенное программирование. Благодаря этому открываются возможности для еще более высокой степени модульности и повторного использования кода, что позволяет сократить время на разработку и повысить надежность создаваемых программ.

Популярность языка программирования C++ привела к появлению большого количества реализаций для множества компьютерных платформ; стандарт ISO/ANSI для C++ обеспечивает основу для взаимной совместимости этих реализаций. В стандарте указано, какими возможностями должен обладать язык, как он себя должен вести, какими должны быть библиотеки функций, классы и шаблоны. Стандарт поддерживает идею переносимого языка, программы на котором могут работать на множестве различных вычислительных платформ и в различных реализациях языка.

Чтобы создать программу на языке C++, вы создаете один или несколько файлов исходного кода программы, выраженного посредством языка C++. Эти файлы представляют собой текстовые файлы, которые необходимо компилировать и компоновать для формирования файлов на машинном языке, содержащих исполняемые программы. Эти задачи часто выполняются в IDE-среде, которая предлагает текстовой редактор для подготовки файлов исходного кода, компилятор и компоновщик для формирования исполняемых файлов, а также другие ресурсы наподобие средств управления проектом и процессом отладки. Однако одни и те же задачи могут быть выполнены также и в командной строке, в которой соответствующие инструменты вызываются по отдельности.

ГЛАВА 2

Приступаем к изучению C++

В этой главе:

- Как создать программу на C++
- Общий формат программы на C++
- Директива `#include`
- Функция `main()`
- Как использовать объект `cout` для вывода
- Как помещать комментарии в программу на C++
- Как и в каких случаях использовать манипулятор `endl`
- Как объявлять и использовать переменные
- Как использовать объект `cin` для ввода
- Как определять и использовать простые функции

Приступая к сооружению простого дома, вы начинаете возводить его фундамент и каркас. Если с самого начала работы вы не построите цельную структуру, то впоследствии вы не сможете вставить окна или дверные коробки, нельзя будет соорудить купол обсерватории или выстроить паркетный танцевальный зал. Это же относится и к изучению компьютерного языка программирования: прежде всего вам необходимо освоить базовую структуру программы. И только после этого можно будет перейти к изучению деталей, например, циклов и объектов. В этой главе рассматривается базовая структура программы на C++ и некоторые дополнительные вопросы. Часть материала будет посвящена функциям и классам, которым в последующих главах будет уделено особое внимание. (Идея заключается в том, чтобы постепенно, по ходу изложения материала, предлагать вам основные положения, а затем уже рассматривать каждый вопрос детально.)

Первые шаги в C++

Давайте попробуем написать простую программу на языке C++, которая будет выводить текст некоторого сообщения на экран монитора. В листинге 2.1 для вывода символа применяется объект `cout`. Исходный код содержит строки комментариев для читателя, которые отмечаются парой символов `//`; эти символы будут игнорироваться компилятором. Язык программирования C++ *чувствителен к регистру*; это означает, что символы в верхнем и нижнем регистре отличаются друг от друга. Поэтому необходимо использовать те символы, которые продемонстрированы в примерах. Например, в приведенной далее программе используется `cout`, поэтому если вы введете `Cout` или `COUT`, компилятор отвергнет эту запись и сообщит об использовании неизвестных идентификаторов. (Компилятор чувствителен также и к орфографии,

поэтому не пытайтесь использовать kout, coot или другие варианты.) Расширение cpp имени файла чаще всего используется для обозначения программы на C++; встречаются также и другие расширения, о чем было сказано в главе 1.

Листинг 2.1. myfirst.cpp

```
// myfirst.cpp -- выводит сообщение на экран
#include <iostream>                          // директива препроцессора
int main()                                   // заголовок функции
{  // начало тела функции
   using namespace std;                      // описания становятся видимыми
   cout << "Come up and C++ me some time.";  // сообщение
   cout << endl;                             // начало новой строки
   cout << "You won't regret it!" << endl;   // дополнительное сообщение
   return 0;                                 // завершение функции main()
}                                            // конец тела функции
```

Замечание по совместимости

Если вы пользуетесь устаревшим компилятором, возможно, вам необходимо будет использовать #include <iostream.h> вместо #include <iostream>; в этом случае нужно также опустить строку using namespace std;. То есть, вам следует заменить

```
#include <iostream>       // так будет в будущем
```

на

```
#include <iostream.h>     // в случае, если будущее еще не наступило
```

и исключить полностью следующую строку:

```
using namespace std;      // это тоже будет в будущем
```

(В некоторых очень старых компиляторах вместо #include <iostream.h> используется #include <stream.h>; если вы работаете со старым компилятором, то тогда лучше обновить компилятор или приобрести старую книгу.) Использовать iostream.h вместо iostream стали совсем недавно, поэтому некоторым компиляторам об этом ничего не известно.

В некоторых средах с оконным интерфейсом выполнение программы осуществляется в отдельном окне, которое после завершения выполнения программы автоматически закрывается. Как мы уже говорили в главе 1, можно сделать так, чтобы окно оставалось открытым до тех пор, пока пользователь не нажмет клавишу; для этого необходимо ввести следующую строку кода перед оператором возврата:

```
cin.get();
```

В некоторых программах необходимо добавлять две такие строки. Благодаря этому коду программа будет находиться в режиме ожидания, пока пользователь не нажмет клавишу. Этот код мы рассмотрим более подробно в главе 4.

Подгонка кода программы

Не исключено, что для того чтобы в своей системе вы могли выполнить примеры из этой книги, их нужно будет изменить. Чаще всего потребуется пара изменений, о чем мы говорили в первой врезке замечания по совместимости. Одно из них необходимо для того, чтобы установить соответствие со стандартами языка программирования; если вы применяете устаревший компилятор, тогда вместо iostream.h необходимо использовать iostream и исключить строку namespace. Второе изменение вызвано условиями среды программирования. Чтобы выходные данные программы оставались видимыми на экране монитора, вам нужно будет добавить один или два оператора cin.get(). Эти корректировки справедливы для каждого примера из этой книги, поэтому вам не следует забывать о них! В следующих врезках замечаний по совместимости будут отмечены другие возможные изменения, которые вам, возможно, необходимо будет осуществить в своей системе.

После того как вы выберете подходящий редактор, скопируйте эту программу (или воспользуйтесь файлами исходного кода, размещенными на Web-сайте издательства). Затем с помощью компилятора C++ можно будет создать исполняемый код, о чем говорилось в главе 1. Далее показан результат выполнения компилированной программы из листинга 2.1:

```
Come up and C++ me some time.
You won't regret it!
```

Ввод и вывод в языке C

Если у вас есть опыт программирования на языке C, то использование объекта `cout` вместо функции `printf()` может показаться странным. На самом деле, в C++ можно применять как `printf()`, так и `scan()` и другие стандартные функции ввода-вывода языка C, при том условии, что вы будете использовать обычный для языка C файл `stdio.h`. Однако книга, которую вы держите в руках, посвящена C++, поэтому мы применяем средства вывода C++, которые являются более полезными, нежели средства C.

Программы на C++ создают на основе стандартных блоков, называемых *функциями*. Обычно программу организовывают так, чтобы она выполняла главные задачи, а затем разрабатывают отдельные функции для их обработки. Пример, представленный в листинге 2.1, настолько простой, что состоит всего лишь из одной функции `main()`. Пример `myfirst.cpp` содержит следующие элементы:

- Комментарии, обозначаемые префиксом `//`.
- Директива препроцессора `#include`.
- Заголовок функции: `int main()`.
- Директива `using namespace`.
- Тело функции, ограниченное символами `{` и `}`.
- Операторы, которые используют объект C++ `cout` для отображения сообщения.
- Оператор возврата для прекращения выполнения функции `main()`.

Давайте обсудим все эти элементы более подробно. Лучше всего начать с функции `main()`, поскольку некоторые предшествующие ей элементы, такие как директива препроцессора, будет проще понять после рассмотрения функции `main()`.

Функция `main()`

Пример программы из листинга 2.1 имеет следующую фундаментальную структуру:

```
int main()
{
   операторы
   return 0;
}
```

В этих строках говорится о том, что существует функция с именем `main()`, и они описывают ее поведение. Все вместе эти строки слагают *описание*, или *определение функции*. Описание функции состоит из двух частей: первой строки, `int main()`, которая называется *заголовком функции*, и частью, заключенной в скобки ({ и }), которая называется *телом функции*. На рис. 2.1 приведена графическая иллюстрация функции `main()`. В заголовке функции вкратце описан ее интерфейс с остальной частью программы, а в теле функции содержатся инструкции компилятору, в которых описываются действия данной функции. В языке C++ каждая полная инструкция называется *оператором*. Каждый оператор должен завершаться символом точки с запятой, поэтому не забывайте ставить ее при наборе примеров.

Рис. 2.1. Функция `main()`

Последний оператор функции `main()` называется *оператором возврата*. Его назначение — завершить выполнение функции. В этой главе мы еще вернемся к нему.

Операторы и точки с запятыми

Оператор представляет полную инструкцию компилятору. Чтобы понять ваш исходный код, компилятор должен знать, где заканчивается один оператор и начинается другой. В некоторых языках программирования используется разделительный знак операторов. В языке программирования FORTRAN, например, для разделения операторов друг от друга применяется символ конца строки. В языке программирования Pascal для разделения операторов используется точка с запятой. В языке Pascal точку с запятой можно в некоторых случаях опускать, например, после оператора и перед `END`, когда на самом деле не происходит разделение двух операторов. (Прагматики и минималисты могут не согласиться с тем, что *можно* — это значит *нужно*.) Однако в языке C++, в отличие от C, точка с запятой больше используется как *терминатор*, или признак завершения, а не как разделительный знак. Разница заключается в том, что точка с запятой, действующая как терминатор, является не маркером *между* операторами, а *частью* оператора. На практике в языке C++ точку с запятой никогда нельзя опускать.

Заголовок функции как интерфейс

Сейчас самое главное усвоить следующее правило: в соответствии с синтаксисом языка программирования C++ описание функции `main()` должно начинаться с заголовка `int main()`. Синтаксис заголовка функции мы детально рассмотрим чуть поз-

же, в разделе "Функции", а пока что для тех, кому просто не терпится удовлетворить свой интерес, мы вкратце расскажем о сути этого заголовка.

Функция в языке C++ активизируется, или *вызывается* другой функцией (иногда говорят, что функция обращается к другой функции), и в заголовке функции описывается интерфейс между функцией и вызывающей ее функцией. Та часть, которая предшествует имени функции, называется *типом возвращаемого значения функции*; в ней описывается информационный поток, поступающий из функции обратно к той функции, которая произвела к ней обращение. Та часть, которая заключена в скобки после имени функции, называется *списком аргументов* или *списком параметров*; в ней описывается информационный поток, который поступает из функции, производящей обращение, к вызываемой функции. Применительно к функции `main()` этот формат будет немного отличаться, поскольку она, как правило, не вызывается из других частей программы. Обычно к функции `main()` обращается код запуска, добавляемый компилятором в вашу программу — она нужна в качестве связующего звена между программой и ОС (Unix, Windows XP и так далее). По сути, в заголовке функции описывается интерфейс между функцией `main()` и ОС.

Давайте рассмотрим интерфейс функции `main()`, начиная с части `int`. Функция C++, к которой производит обращение другая функция, возвращает значение для активизации (вызова) функции. Это значение называется *возвращаемым значением*. В нашем случае функция `main()` может возвращать целочисленное значение, что можно понять по ключевому слову `int`. Далее, обратите внимание на пустые скобки. В общем случае, функция в языке C++ может передавать какую-нибудь информацию другой функции при обращении к ней. Эту информацию описывает часть заголовка функции, заключенная в скобки. В нашем случае пустые скобки означают, что функция `main()` не принимает никакой информации, или, если обратиться к общепринятой терминологии, функция `main()` не принимает аргументов. (Если сказать, что функция `main()` не принимает аргументов, это не будет означать, что функция `main()` не имеет смысла. Напротив, под *аргументом* программисты подразумевают информацию, которая передается из одной функции в другую.)

Итак, заголовок

```
int main()
```

говорит о том, что функция `main()` возвращает целочисленное значение функции, которая произвела к ней обращение, и что функция `main()` не принимает никакой информации от функции, производящей к ней обращение.

Во многих существующих программах используется классическая форма записи заголовка в языке C:

```
main()          // оригинальный стиль языка C
```

В классическом C игнорирование возвращаемого типа равнозначно тому, что функция имеет тип `int`. Однако в языке C++ от такого подхода отказались.

Можно использовать и такой вариант:

```
int main(void) // самый подробный стиль
```

Показать, что данная функция не принимает аргументы, проще всего можно с помощью ключевого слова `void` в круглых скобках. В языке C++ (но не в C) принято, что пустые скобки равнозначны использованию ключевого слова `void`. (В языке C пустые скобки означают, что вы ничего не сообщаете о наличии аргументов.)

Некоторые программисты используют следующий заголовок и опускают оператор возврата:

```
void main()
```

Эта запись имеет логический смысл, поскольку возвращаемый тип void означает, что функция не возвращает значения. Этот вариант не принят в стандарте C++, хотя и работает в некоторых системах. Лучше всего не использовать эту формы записи, а использовать форму, присущую стандартному C++; излишнее усердие не всегда оправдано.

И, наконец, стандарт ANSI/ISO для C++ идет на уступки тем программистам, которым не очень по душе необходимость ставить оператор возврата в конце функции main(). Если компилятор достиг завершения функции main(), не встретив при этом оператора возврата, результат будет точно таким же, как если бы функция main() заканчивалась следующим оператором:

```
return 0;
```

Неявный возврат возможен только для функции main(), но не для какой-либо другой функции.

Почему именно main()?

Относительно того, что в программе myfirst.cpp функции присваивается имя main(), имеется неопровержимый довод: вы просто обязаны использовать именно эту функцию. Как правило, в любой программе на языке C++ должна присутствовать функция main(). (И, между прочим, только эта функция, а не Main(), MAIN() или даже mane(). Вы еще не забыли о чувствительности к регистру и орфографии?) Поскольку в программе myfirst.cpp имеется только одна функция, то она и должна соответствовать функции main(). Запуск программы на C++ начинается с функции main(). Таким образом, если в вашей программе эта функция будет отсутствовать, ваша программа окажется неполной и компилятор выдаст сообщение об отсутствии определенной функции main().

Существуют, правда и исключения. Так, например, при программировании для Windows вы можете написать модуль динамически подключаемой библиотеки (Dynamic Link Library — DLL). Эта библиотека представляет собой код, который могут использовать другие Windows-программы. Поскольку модуль DLL не является автономной программой, для него функция main() просто не нужна. Для программ, предназначенных для работы в специальных средах (например, программа для микросхемы контроллера в автоматическом устройстве), функцию main() применять не обязательно. А вот в вашей обычной автономной программе эту функцию необходимо использовать; в этой книге рассматриваются именно такие программы.

Комментарии в языке C++

В языке C++ комментарий обозначается двумя косыми чертами (//). *Комментарий* — это примечание, написанное программистом для пользователя программы, которое обычно идентифицирует раздел программы или содержит пояснения к определенному коду. Компилятор игнорирует комментарии. Он знает C++ не хуже вас, но комментарии читать не может. Поэтому для него листинг 2.1 будет выглядеть следующим образом:

```
#include <iostream>
int main()
{
   using namespace std;
   cout << "Come up and C++ me some time.";
   cout << endl;
   cout << "You won't regret it!" << endl;
   return 0;
}
```

Комментарий в C++ начинается с символов // и заканчивается в конце строки. Комментарий может занимать одну строку целиком, а может быть размещен в той же строке, что и код. Кстати, обратите внимание на первую строку в листинге 2.1:

```
// myfirst.cpp -- выводит сообщение на экран
```

В этой книге каждая программа начинается с комментария, в котором указывается имя файла исходного кода и краткое описание программы. Как уже было сказано в главе 1, расширение имени файла исходного кода зависит от реализации C++. В разных реализациях могут использоваться варианты вроде `myfirst.C` или `myfirst.cxx`.

Совет

Используйте комментарии для документирования своих программ. Чем сложнее программа, тем более ценными будут ваши комментарии. Они помогут не только другим пользователям разобраться с вашим кодом, но и вы сами сможете понять его, особенно когда вы не работали с программой некоторое время.

Комментарии в стиле С

Язык C++ распознает комментарии, написанные в стиле С, которые ограничены символами /* и */:

```
#include <iostream> /* комментарий в стиле С */
```

Поскольку комментарий в С завершается символами */, а не в конце строки, его можно продолжить и на другой строке. В своей программе вы можете использовать как один, так и оба стиля обозначения комментариев. Мы рекомендуем стиль, принятый в C++. Поскольку в нем не нужно придерживаться порядка чередования конечного и начального символа, вероятность допустить ошибку при вводе обозначения будет невелика. В стандарте C99 для языка С добавлен комментарий //.

Препроцессор C++ и файл iostream

Для начала мы расскажем вкратце о том, что вы должны знать обязательно. Если ваша программа предназначена для использования обычных средств ввода и вывода в C++, вы вводите следующие две строки:

```
#include <iostream>
using namespace std;
```

Если вашему компилятору не понравятся эти строки (например, если он сообщит, что не может обнаружить файл `iostream`), тогда попробуйте ввести такую строку:

```
#include <iostream.h > // совместима со старыми компиляторами
```

Вот и все, что вам нужно знать, чтобы ваша программа заработала. А теперь давайте обо всем по порядку.

В языке C++, как и в C, используется *препроцессор*. Препроцессор — это программа, которая выполняет обработку файла исходного кода до начала компиляции. (В некоторых реализациях языка C++, как упоминалось в главе 1, для преобразования программы на C++ в C используется программа транслятора. Хотя транслятор тоже является некоторым подобием препроцессора, мы его не рассматриваем; мы говорим о препроцессоре, обрабатывающем директивы, чье имя начинается с символа #.) Для вызова препроцессора ничего особенного делать не нужно. Он запускается автоматически во время компиляции программы.

В листинге 2.1 использовалась директива `#include`:

```
#include <iostream> // директива препроцессора
```

В соответствии с этой директивой препроцессор добавляет содержимое файла `iostream` в вашу программу. Это обычное действие препроцессора: добавление или замена текста в исходном коде перед его компиляцией.

В связи с этим возникает вопрос: зачем нужно добавлять содержимое файла `iostream` в вашу программу? Это объясняется тем, что между программой и внешним миром существует связь. Сокращение `io` в `iostream` расшифровывается как *input* (ввод), что обозначает входные данные программы, и *output* (вывод), что обозначает информацию, передаваемую из программы. Схема ввода-вывода в языке программирования C++ включает несколько определений, которые можно найти в файле `iostream`. Для вашей первой программы эти определения необходимы для того, чтобы использовать объект `cout` для вывода сообщения на экран монитора. В соответствии с директивой `#include`, содержимое файла `iostream` будет отправлено компилятору вместе с содержимым вашего файла. На самом деле содержимое файла `iostream` заменяет строку `#include <iostream>` в программе. Ваш исходный файл не изменяется, а составной файл и файл `iostream` переходят к следующему этапу компиляции.

На память!

Программы, в которых для ввода и вывода используются объекты `cin` и `cout`, должны включать файл `iostream` (или, в некоторых системах, `iostream.h`).

Имена заголовочных файлов

Файлы вроде `iostream` называются *включаемыми файлами* (поскольку они включаются в другие файлы) или *заголовочными файлами* (так как они помещаются в самом начале файла). Компиляторы C++ работают со множеством заголовочных файлов, каждый из которых поддерживает отдельное семейство элементов. В языке C для заголовочных файлов использовалось расширение h, благодаря которому можно было просто распознать тип файла по его имени. Например, заголовочный файл `math.h` в языке C поддерживает различные математические функции языка C. Первоначально так было и в языке C++. Например, заголовочный файл, поддерживающий ввод и вывод, имел имя `iostream.h`. Однако позже язык C++ претерпел некоторые изменения. Сейчас расширение `h` зарезервировано для старых заголовочных файлов языка C (которые пока что могут использоваться программами на C++), тогда как заголовочные файлы в C++ не имеют расширений. Существуют также заголовочные файлы языка C, которые можно преобразовать в заголовочные файлы языка C++. Имена

этих файлов были изменены: было исключено расширение h (стиль языка C++) и добавлен префикс c (файл создан на языке C). Например, версией math.h в языке C++ является заголовочный файл cmath. Иногда версии заголовочных файлов C в языках C и C++ одинаковы, а в других случаях новая версия может несколько отличаться. Для чистых заголовочных файлов C++, например iostream, отсутствие h является не просто косметическим изменением, но свидетельством использования пространства имен, о чем мы будем говорить в следующем разделе. В табл. 2.1 приводятся соглашения по именованию для заголовочных файлов.

Таблица 2.1. Соглашения по именованию для заголовочных файлов

Тип заголовка	Правило	Пример	Комментарии
Старый стиль C++	Заканчивается на .h	iostream.h	Используется в программах на C++.
Старый стиль C	Заканчивается на .h	math.h	Используется в программах на C и C++.
Новый стиль C++	Без расширения	iostream	Используется в программах на C++, использует namespace std.
Преобразованный C	Префикс c, без расширения	cmath	Используется в программах на C++, может использовать особенности, не характерные для C, например, namespace std.

Учитывая традицию применения в C различных расширений имен файлов для идентификации различных типов файлов, представляется целесообразным использовать для обозначения заголовочных файлов в C++ некоторое специальное расширение, например .hx или .hxx. Так считают и в комитете ANSI/ISO. Однако договориться об этом очень сложно, поэтому, в конечном счете, все останется по-прежнему.

Пространства имен

Если вместо iostream.h вы применяете iostream, тогда необходимо использовать следующую директиву пространства имен, чтобы определения в iostream были доступны вашей программе:

```
using namespace std;
```

Это *директива* using. Сейчас самое главное просто запомнить ее. Мы еще вернемся к этому вопросу, в том числе и в главе 9, а пока что давайте вкратце поговорим о пространствах имен.

Поддержка пространства имен — это довольно новая особенность C++, позволяющая упростить написание программ, в которых комбинируется существующий код из нескольких источников. Единственная проблема заключается в том, что вы можете использовать два предварительно упакованных продукта, в каждом из которых присутствует, например, функция wanda(). Если вы попытаетесь использовать функцию wanda(), компилятору не будет известно, какая версия функции вам необходима. С помощью средства пространства имен производитель может запаковывать свои продукты в модуль, называемый *пространством имен*. Вы, в свою очередь, можете использовать имя пространства имен, чтобы определить, продукт какого произ-

водителя вам необходим. Так, например, Microflop Industries может поместить свои описания в пространство имен Microflop. Для его функции wanda() можно использовать Microflop::wanda() в качестве полного имени функции. Piscine::wanda() может означать версию функции wanda() от Piscine Corporation. Таким образом, для отличия различных версий в своей программе вы можете использовать пространства имен:

```
Microflop::wanda("go dancing?");        // использует версию пространства
                                        // имен Microflop
Piscine::wanda("a fish named Desire");  // использует версию пространства
                                        // имен Piscine namespace
```

Смысл в том, что классы, функции и переменные, которые являются стандартными компонентами компиляторов C++, теперь находятся в пространстве имен std. Это относится к заголовочным файлам без .h. Это означает, например, что переменная cout, используемая для вывода и определенная в iostream, на самом деле называется std::cout, а endl на самом деле — это std::endl. Таким образом, директиву using можно опустить и написать код следующим образом:

```
std::cout << "Come up and C++ me some time.";
std::cout << std::endl;
```

Однако большинство пользователей стараются не преобразовывать код до пространства имен, применяющий iostream.h и cout, в код пространства имен, который использует iostream и std::cout, если только это не требует особых усилий. Вот здесь и приходит на помощь директива using. Следующая строка означает, что вы можете применять имена, определенные в пространстве имен std, без префикса std::

```
using namespace std;
```

Теперь, благодаря директиве using, будут доступны все имена в пространстве имен std. В настоящее время это рассматривается как проявление лени. Лучше всего сделать доступными только те имена, которые вам необходимы, с помощью объявления using:

```
using std::cout;       // делает доступным cout
using std::endl;       // делает доступным endl
using std::cin;        // делает доступным cin
```

Если эти директивы применить вместо следующих:

```
using namespace std;   // способ для ленивых; доступны все имена
```

то cin и cout можно использовать без std::. Однако если вам нужны будут другие имена из iostream, тогда их придется добавлять в список using самостоятельно. В этой книге используется "ленивый" подход, что объясняется двумя причинами. Во-первых, для простых программ нет особой разницы, какой использовать метод управления пространством имен. Во-вторых, лучше акцентировать внимание на более важных аспектах C++. Позднее, в книге вы встретите разные методы пространств имен.

Вывод в C++ с помощью cout

Давайте разберемся, как вывести сообщение на экран. В программе myfirst.cpp имеется следующий оператор C++:

```
cout << "Come up and C++ me some time.";
```

Та часть, которая заключена в двойные кавычки, является сообщением, которое необходимо вывести на экран. В языке программирования C++ любая последовательность символов, заключенных в двойные кавычки, называется *символьной строкой*, вероятно потому, что она состоит из нескольких символов, собранных в одну большую конструкцию. Запись << означает, что оператор отправляет строку в cout; символы указывают на направление передачи информации. А что такое cout? Это предопределенный объект, который знает о том, как отображать разнообразные элементы, включая строки, цифры и индивидуальные символы. (Как вы помните из главы 1, *объект* представляет собой экземпляр класса, а *класс* определяет способ хранения и использования данных.)

Вероятно, так быстро использовать объекты еще рано, потому что к их рассмотрению мы перейдем еще не скоро. На самом деле, в приведенном примере показана одна из сильных сторон объектов. Вы не обязаны знать внутреннюю структуру объекта, чтобы иметь возможность его использовать. Все, что вам необходимо знать — это его интерфейс, или способ его использования. Интерфейс объекта cout довольно прост. Если string представляет строку, то для вывода строки на экран монитора можно сделать следующее:

```
cout << string;
```

Вот и все, что вам нужно знать, чтобы отобразить строку на экране. А сейчас давайте посмотрим, как этот процесс можно описать с точки зрения концептуального представления C++. В этом представлении вывод рассматривается как поток, то есть набор символов, передаваемых из программы. Этот поток представляет объект cout, свойства которого определены в файле iostream. Свойства объекта cout включают операцию вставки (<<), которая добавляет в поток данные, указанные в правой части. Рассмотрим следующий оператор (обратите внимание на завершающую точку с запятой):

```
cout << "Come up and C++ me some time.";
```

В выходной поток будет помещена строка "Come up and C++ me some time.". Таким образом, вы можете сказать, что ваша программа не выводит на экран сообщение, а помещает строку в поток вывода. Это звучит более выразительно. (Посмотрите на рис. 2.2.)

Несколько слов о перегрузке операций

Если вы начали изучать язык программирования C++, имея опыт программирования на C, то, вероятно, заметили, что операция вставки (<<) выглядит практически так же, как и поразрядная операция сдвига влево (<<). Это пример *перегрузки операций*, когда один и тот же символ операции может трактоваться по-разному. Чтобы выяснить назначение выполняемой операции, компилятор обращается к контексту. В языке программирования C тоже есть примеры перегрузки операций. Например, символ & отвечает и за адресную операцию, и за поразрядную операцию AND (поразрядное И).

Символ * соответствует умножению и разыменованию указателя. Здесь важно понять, что имеется в виду: один и тот же символ трактуется по-разному, и компилятор определяет соответствующее назначение исходя из контекста. (В повседневной жизни мы тоже нередко обращаемся к контексту, чтобы правильно понять сказанную фразу.) В языке C++ понятие перегрузки операций расширено, благодаря чему можно переопределять назначение операции для классов — типов, определяемых пользователем.

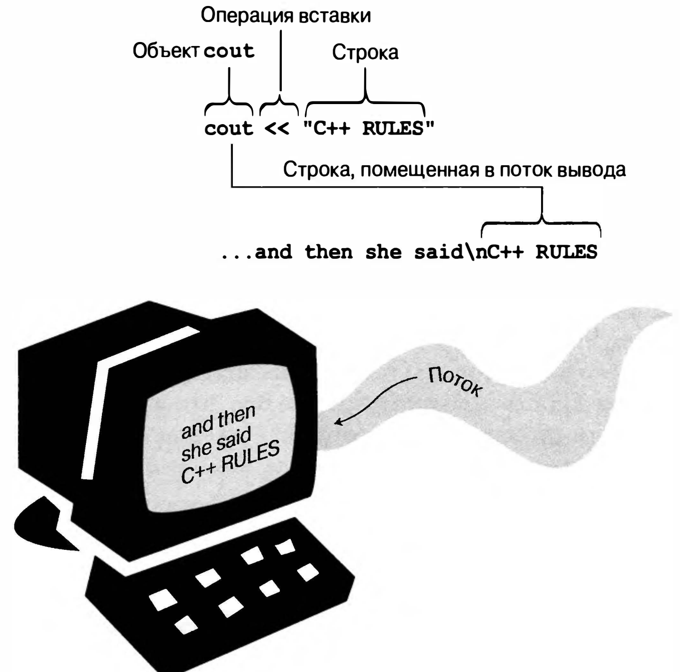


Рис. 2.2. Отображение строки с использованием `cout`

Манипулятор endl

Теперь давайте поговорим об еще одной строке из листинга 2.1:

```
cout << endl;
```

`endl` — специальный условный знак в языке C++, который представляет важное понятие начала новой строки. Помещение `endl` в поток вывода означает, что курсор на экране будет перемещен в начало следующей строки. Специальные условные знаки наподобие `endl`, имеющие определенное значение для объекта `cout`, называются *манипуляторами*. Как и `cout`, манипулятор `endl` определен в заголовочном файле `iostream` и является частью пространства имен `std`.

Обратите внимание, что в процессе печати строки символов `cout` не переходит автоматически на следующую строку, поэтому первый оператор `cout` из листинга 2.1 оставляет курсор в позиции после точки в конце строки вывода. Вывод для каждого оператора `cout` начинается с той позиции, где был закончен последний вывод, поэтому если опустить манипулятор `endl`, результат будет таким:

```
Come up and C++ me some time.You won't regret it!
```

Заметьте, что Y следует сразу за точкой. Давайте рассмотрим еще один пример. Допустим, что вы ввели следующий код:

```
cout << "The Good, the";
cout << "Bad, ";
cout << "and the Ukulele";
cout << endl;
```

В результате его выполнения будет выведена следующая строка:

```
The Good, theBad, and the Ukulele
```

И здесь легко заметить, что одна из строк начинается сразу после окончания предыдущей строки. Чтобы поставить между двумя строками пробел, необходимо включить его в одну из строк. (Не забывайте, что эти примеры должны быть помещены в завершенную программу, с заголовком функции main() и открывающей и закрывающей фигурными скобками, иначе их выполнение будет невозможным.)

Символ новой строки

Обозначить новую строку в выводе в C++ можно и старым способом — посредством обозначения \n, принятого в языке C:

```
cout << "What's next?\n"; // \n обозначает начало новой строки
```

Комбинация \n рассматривается как один символ, называемый символом *новой строки.*

Если вы отображаете строку, вам нужно будет вводить меньше символов, чем при использовании манипулятора endl:

```
cout << "Jupiter is a large planet.\n";      // отображает фразу, переходит
                                             // на следующую строку
cout << "Jupiter is a large planet." << endl;// отображает фразу, переходит
                                             // на следующую строку
```

С другой стороны, если вы хотите сгенерировать отдельную новую строку, то в каждом из случаев придется вводить одинаковое количество символов; большинство программистов считают, что набирать endl гораздо удобнее:

```
cout << "\n";  // начинает новую строку
cout << endl;  // начинает новую строку
```

В этой книге используется встроенный символ новой строки (\n) при отображении строк в кавычках, а во всех других случаях — манипулятор endl.

Символ новой строки является примером специальных комбинаций клавиш, называемых "управляющими последовательностями"; речь о них пойдет в главе 3.

Форматирование исходного кода C++

Код в некоторых языках программирования, таких как FORTRAN, является построчным, то есть в одной строке вводится один оператор. В этих языках для разделения операторов служит возврат каретки. В языке C++ для обозначения завершения каждого оператора служит символ точки с запятой. Поэтому в C++ возврат каретки можно обрабатывать точно так же, как пробел или табуляцию. Это означает, что в

C++ можно использовать пробел там, где можно было бы использовать возврат каретки и наоборот. Поэтому вы могли бы записывать один оператор в нескольких строках или ставить несколько операторов в одной строке. Например, код myfirst.cpp можно было бы переформатировать следующим образом:

```
#include <iostream>
      int
main
() {  using
   namespace
         std; cout
            <<
"Come up and C++ me some time."
;  cout <<
endl; cout <<
"You won't regret it!" <<
endl;return 0; }
```

Этот код смотрится не ахти как, хотя и является действительным. При вводе кода необходимо соблюдать некоторые правила. В частности, в языках C и C++ нельзя ставить пробел, табуляцию или возврат каретки в середине элемента, например в имени, и нельзя помещать возврат каретки в середину строки. Далее показаны примеры неправильного ввода:

```
int ma in()     // НЕПРАВИЛЬНО — пробел в имени
re
turn 0;         // НЕПРАВИЛЬНО — возврат каретки в слове
cout << "Behold the Beans
of Beauty!";    // НЕПРАВИЛЬНО — возврат каретки в строке
```

Лексемы и обобщенный пробел

Лексемами называются неделимые элементы в строке кода (рис. 2.3). Как правило, для разделения лексем друг от друга используется пробел, табуляция или возврат каретки, которые все вместе называются *обобщенным пробелом* или *пустым пространством.* Некоторые одиночные символы, такие как круглые скобки и запятые, являются лексемами, которые не нужно отделять обобщенным пробелом. Далее показаны примеры того, где используется и где не используется обобщенный пробел:

```
return0;        // НЕПРАВИЛЬНО, должно быть return 0;
return(0);      // ПРАВИЛЬНО, обобщенный пробел опущен
return (0);     // ПРАВИЛЬНО, используется обобщенный пробел
intmain();      // НЕПРАВИЛЬНО, обобщенный пробел опущен
int main()      // ПРАВИЛЬНО, обобщенный пробел опущен в скобках
int main ( )    // ТОЖЕ ПРАВИЛЬНО, обобщенный пробел используется в скобках
```

Стиль написания исходного кода C++

Хотя в C++ вы можете форматировать свой код так, как вам больше нравится, ваши программы будут легче читаться, если придерживаться осмысленного стиля их написания. Код, пусть даже и действительный, но написанный беспорядочным образом, не принесет вам никакого удовольствия от работы.

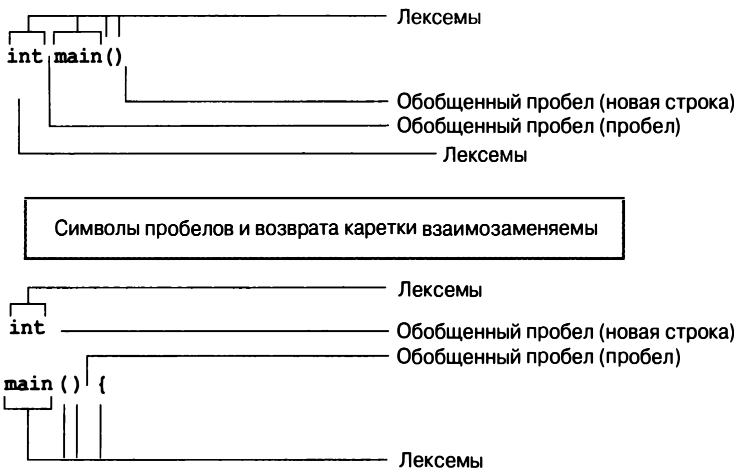


Рис. 2.3. Лексемы и обобщенный пробел

Большинство программистов придерживаются стиля, продемонстрированного в листинге 2.1. Для него характерны следующие правила:

- Один оператор в одной строке.
- Открывающая и закрывающая фигурные скобки для функции, каждая из которых занимает свою собственную строку.
- Операторы и фигурные скобки в функциях отделены абзацем.
- Вокруг круглых скобок, связанных с именем функции, не должно быть обобщенного пробела.

Первые три правила предназначены для того, чтобы сделать код чистым и легко читаемым. Четвертое правило помогает отличать функции от встроенных структур в С++, таких как циклы, для которых также используются круглые скобки. Далее в книге мы еще не раз будем напоминать вам о других правилах.

Операторы в языке С++

Программа, написанная на языке С++, представляет собой набор функций. Каждая из них, в свою очередь, представляет собой набор операторов. В языке программирования С++ имеется несколько разновидностей операторов, поэтому давайте рассмотрим некоторые из них. В листинге 2.2 можно увидеть операторы двух видов. Первый, *оператор объявления*, создает переменную. Второй, *оператор присваивания*, присваивает этой переменной определенное значение. Кроме этого, в программе продемонстрирована новая возможность объекта cout.

Листинг 2.2. carrots.cpp

```
// carrots.cpp -- программа по технологии производства пищевых продуктов
// использует и отображает переменную
#include <iostream>
int main()
```

```
{
   using namespace std;
   int carrots;            // объявление переменной целочисленного типа
   carrots = 25;           // присваивание значения переменной
   cout << "I have ";
   cout << carrots;        // отображение значения переменной
   cout << " carrots.";
   cout << endl;
   carrots = carrots - 1;  // изменение переменной
   cout << "Crunch, crunch. Now I have " << carrots << " carrots." << endl;
   return 0;
}
```

Раздел объявления отделен от остальной части программы с помощью пустой строки. Этот прием часто можно было встретить в программах на С, а в программах на С++ это редкость. Далее показан результат работы программы, представленной в листинге 2.2:

```
I have 25 carrots.
Crunch, crunch. Now I have 24 carrots.
```

Следующие несколько страниц мы посвятим рассмотрению этой программы.

Операторы объявления и переменные

Компьютер — это точная и организованная машина. Для того чтобы сохранить элемент информации в компьютере, вы должны идентифицировать как ячейку памяти, так и количество памяти, требуемое для хранения данной информации. В языке С++ проще всего это можно сделать с помощью *оператора объявления*, который идентифицирует тип памяти и присваивает метку ячейке. Например, в программе, представленной в листинге 2.2, имеется следующий оператор объявления (обратите внимание на наличие точки с запятой):

```
int carrots;
```

Этот оператор объявляет, что программа использует достаточно памяти для хранения целого числа, для ссылки на которое С++ использует метку `int`. Компилятор анализирует подробные данные о размещении и присваивании метки ячейке памяти. С++ может обрабатывать несколько разновидностей, или типов, данных, и среди них тип `int` является одним из самых распространенных. Он соответствует целому числу (integer) — числу без дробной части. В языке С++ тип `int` может принимать как положительные, так и отрицательные значения, а размер диапазона зависит от реализации языка. В главе 3 мы будет говорить подробно как о типе `int`, так и о других основных типах данных.

Кроме определения типа, оператор объявления говорит, что с этого момента программа будет использовать имя `carrots` для обозначения значения, хранящегося в этой ячейке памяти. `Carrots` — это *переменная*, поскольку вы можете изменять ее значение. В языке С++ необходимо объявлять каждую переменную. Если вы забудете объявить переменную в `carrots.cpp`, компилятор выдаст сообщение об ошибке, когда программа попытается использовать переменную `carrots`. (На самом деле, можно умышленно не объявить переменную просто для того, чтобы проверить, как на это

будет реагировать компилятор. Затем, если вы убедитесь в его реакции, вы сможете проверять свои программы на предмет необъявленных переменных.)

Для чего необходимо объявлять переменные?

В некоторых языках программирования, особенно в языке BASIC, новую переменную можно создавать в том месте, где вы впервые используете новое имя, без явного объявления. На первый взгляд, это может показаться удобным, однако у этого способа есть большой минус. Дело в том, что если вы допустите орфографическую ошибку в имени переменной, вы непреднамеренно создадите новую переменную, не реализуя ее. То есть, в языке BASIC допускается следующая ситуация:

```
CastleDark = 34
...
CastleDank = CastleDark + MoreGhosts
...
PRINT CastleDark
```

Поскольку `CastleDank` написано с ошибкой (вместо `r` напечатано `n`), то изменения, которые вы произведете в этой переменной, оставят неизмененной переменную `CastleDark`. Ошибки подобного рода порой довольно трудно отыскать, поскольку они не нарушают ни одного правила в языке BASIC. И наоборот, в языке C++ может быть объявлена переменная `CastleDark`, а написанная с ошибкой переменная `CastleDank` может быть не объявлена. Таким образом, эквивалентный код на C++ нарушает правило объявления переменной и компилятор выдаст сообщение об ошибке.

В общем случае, в объявлении указывается тип сохраняемых данных и имя программы, которая будет использовать данные этой переменной. В нашем случае программа создает переменную с именем `carrots`, в которой хранится целое число (рис. 2.4).

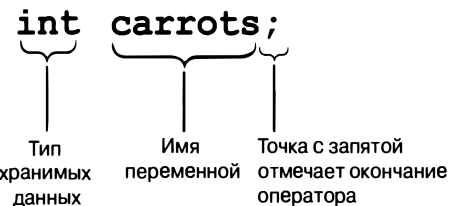


Рис. 2.4. Объявление переменной

Оператор объявления в программе называется оператором *определяющего объявления*, или, говоря проще, *определением*. Его присутствие означает, что компилятор выделит пространство памяти для хранения значений переменной. В более сложных ситуациях можно применять *ссылочное объявление*. В этом случае компьютер будет использовать переменную, которая уже где-то была определена. Вообще, объявление не должно быть определением, однако в нашем примере это так.

Если вы знакомы с языками программирования C или Pascal, вы должны знать о том, как объявляются переменные. В языке C++ принят другой способ объявления переменной. Вы знаете, что в языках C и Pascal все объявления находятся в самом начале функции или процедуры. А в языке C++ такого требования нет, поэтому объявлять переменную можно перед ее первым использованием. Таким образом, чтобы узнать о типе переменной вам не придется возвращаться в начало программы. Пример такого объявления вы увидите далее в этой главе. Такая форма объявления

имеет свой недостаток — имена всех переменных не собраны в одном месте, поэтому сразу определить, какие переменные использует функция проблематично. (Между прочим, в стандарте C99 для языка C правила объявления переменных такие же, как и в C++.)

Совет

В языке C++ принято объявлять переменную как можно ближе к той строке, в которой она впервые используется.

Операторы присваивания

Оператор присваивания присваивает значение ячейке памяти. Например, оператор

```
carrots = 25;
```

присваивает целое значение 25 ячейке памяти, обозначаемой переменной carrots. Символ = называется *операцией присваивания*. Одной из интересных особенностей языка C++ (как и C) является то, что вы можете использовать операцию присваивания несколько раз подряд. Например, приведенный ниже код является допустимым:

```
int steinway;
int baldwin;
int yamaha;
yamaha = baldwin = steinway = 88;
```

Операция присваивания выполняется поочередно, справа налево. Сначала значение 88 присваивается переменной steinway; затем это же значение присваивается переменной baldwin и, наконец, переменной yamaha. (В языке C++, как и в C, допускается написание такого причудливого кода.)

Второй оператор присваивания из листинга 2.2 показывает, что вы можете изменить значение переменной:

```
carrots = carrots - 1; // изменяет значение переменной
```

Выражение в правой части оператора присваивания (carrots - 1) является примером из арифметики. Компьютер вычитает 1 из 25, значения переменной carrots, в результате чего получает 24. Затем оператор присваивания сохраняет новое значение в ячейке carrots.

Новый трюк с объектом cout

До настоящего момента в примерах, приведенных в этой главе, объект cout принимал строки для последующего их отображения. В листинге 2.2 объект cout принимает переменную целочисленного типа:

```
cout << carrots;
```

Программа не печатает слово carrots; наоборот, она печатает целое значение 25, присвоенное переменной carrots. Здесь можно отметить два трюка. Во-первых, объект cout присваивает переменной carrots свое значение — 25. Во-вторых, объект cout преобразовывает значение в соответствующие символы вывода.

Как видите, объект cout работает и со строками, и с целыми числами. Этот факт может показаться вам не очень-то существенным, однако учтите, что целое число 25 все-таки отличается от строки "25". Строка содержит символы, посредством кото-

рых вы записываете число (то есть, символ 2 и символ 5). Программа хранит код для символа 2 и символа 5. Чтобы напечатать строку, cout печатает каждый символ. Однако целое число 25 хранится в виде цифрового значения. Вместо того чтобы хранить каждую цифру по отдельности, компьютер хранит 25 в виде двоичного числа (см. приложение А). Здесь смысл заключается в том, что объект cout должен преобразовать целое число в символ, прежде чем он будет выведен на экран. К тому же, объект cout сам определяет, что переменная carrots содержит целочисленное значение, требующее преобразования.

Возможно, посредством сравнения можно будет понять, насколько выгодно отличается использование cout от возможностей языка С. Чтобы напечатать строку "25" и целое число 25 в языке С, можно воспользоваться многоцелевой функцией вывода printf():

```
printf("Printing a string: %s\n", "25");
printf("Printing an integer: %d\n", 25);
```

Не вдаваясь в тонкости функции printf(), скажем, что вы должны использовать специальные коды (%s и %d), чтобы указать, что вы намерены отобразить — строку или целое число. Если вы хотите, чтобы функция printf() напечатала строку, и по ошибке передаете ей целое число, то printf() не заметит здесь никакой ошибки. Она продолжит работу и выведет на экран бессмысленный набор символов.

Что касается объекта cout, то его интеллектуальное поведение возможно благодаря особенностям объектно-ориентированного программирования языка C++. По сути, операция вставки (<<) в языке C++ изменяет свое поведение в зависимости от того, с данными какого типа она будет работать. Это не что иное, как пример перегрузки операций. В следующих главах, когда мы займемся рассмотрением перегрузки функций и перегрузки операций, вы узнаете, как можно самостоятельно реализовывать подобные функции.

cout и printf()

Если вы привыкли программировать на С и работали с функцией printf(), работа объекта cout может показаться вам странной. Конечно, можно по-прежнему использовать функцию printf(), особенно если у вас уже накоплен опыт работы с ней. Однако на самом деле поведение cout ничуть не хуже, чем поведение функции printf(), со всеми его спецификациями преобразованиями. Наоборот, cout обладает значительными преимуществами по сравнению с этой функцией. Тот факт, что объект cout может распознавать типы данных, свидетельствует о его “интеллекте”. Кроме этого, он является *расширяемым*. Это означает, что вы можете переопределять операцию <<, с тем чтобы объект cout мог распознать и отобразить новые типы данных. А если вам по душе широкие возможности управления, которые предлагает printf(), вы можете добиться того же результата и с помощью объекта cout (см. главу 17).

Другие операторы C++

Давайте рассмотрим еще пару примеров операторов. Программа, представленная в листинге 2.3, продолжает предыдущий пример, позволяя вам вводить значение во время выполнения программы. Для этого в ней используется объект cin — точная копия объекта cout, который предназначен для ввода данных в программу. Кроме этого, в программе продемонстрирован еще один способ использования универсального объекта cout.

Листинг 2.3. getinfo.cpp

```
// getinfo.cpp -- ввод и вывод
#include <iostream>
int main()
{
   using namespace std;
   int carrots;
   cout << "How many carrots do you have?" << endl;
   cin >> carrots; // ввод данных в C++
   cout << "Here are two more. ";
   carrots = carrots + 2;
// следующая строка соединяет вывод
   cout << "Now you have" << carrots << " carrots." << endl;
   return 0;
}
```

Ниже показаны результаты выполнения этой программы:

```
How many carrots do you have?
12
Here are two more. Now you have 14 carrots.
```

У этой программы две новые особенности: использование cin для чтения вводимых данных с клавиатуры и комбинирование четырех операторов вывода в одном. Давайте поговорим об этом подробнее.

Использование cin

Как видно из листинга 2.3, значение, введенное с клавиатуры (12), в конечном итоге присваивается переменной carrots. Это осуществляется посредством следующего оператора:

```
cin << carrots;
```

Взглянув на него, вы можете определить информацию, передаваемую из объекта cin переменной carrots. Конечно, существует более формальное описание этого процесса. Подобно тому, как C++ рассматривает вывод как поток символов, выходящих из программы, ввод рассматривается как поток символов, входящих в программу. Файл iostream определяет cin как объект, представляющий поток. Для вывода операция << помещает символы в поток вывода. Для ввода объект cin использует операцию >> с целью извлечения символов из потока ввода. Обычно в правой части операции указывается переменная, которой будут присваиваться извлеченные данные. (Символы << и >> были выбраны для визуальной подсказки направления потока информации.)

Как и cout, объект cin является интеллектуальным объектом. Он преобразовывает входные данные, представляющие собой серию символов, введенных с клавиатуры, в форму, приемлемую для переменной, которая получает эту информацию. В нашем случае программа объявляет carrots как целочисленную переменную, поэтому входные данные преобразуются в цифровую форму, используемую компьютером для хранения целых чисел.

Конкатенация с помощью cout

Еще одной новой возможностью getinfo.cpp является комбинирование четырех операторов вывода в одном. В файле iostream определена операция <<, поэтому вы можете объединить (или сцепить) вывод следующим образом:

```
cout << "Now you have " << carrots << " carrots." << endl;
```

Такая запись позволяет объединить вывод строки и вывод целого числа в одном операторе. В результате выполнения кода вывод будет таким же, как если бы использовалась следующая запись:

```
cout << "Now you have ";
cout << carrots;
cout << " carrots";
cout << endl;
```

Если хотите, вот еще один совет по использованию объекта cout: можно переза-писать объединенную версию, используя один оператор в четырех строках. Это выглядит следующим образом:

```
cout << "Now you have "
     << carrots
     << " carrots."
     << endl;
```

Такая запись возможна благодаря тому, что в языке C++ новые строки и пробелы между лексемами рассматриваются как взаимозаменяемые. Последний способ удобно применять, когда длина строки получается слишком большой.

Обратите внимание на еще одну деталь. Предложение

```
Now you have 14 carrots.
```

выводится в той же строке, что и предложение

```
Here are two more.
```

Это происходит потому, что, как было сказано выше, вывод одного оператора cout следует непосредственно за выводом предыдущего оператора cout. Это справедливо даже в том случае, если между ними будут находиться другие операторы.

cin и cout: признак класса

Вы уже знаете об объектах cin и cout достаточно много. В этом разделе мы поговорим о том, что собой представляет класс. Как было сказано в главе 1, класс является одним из основных понятий объектно-ориентированного программирования (ООП) в языке C++.

Класс представляет собой тип данных, определяемый пользователем. Чтобы определить класс, вы описываете, какую разновидность информации он может хранить, и какие действия вы можете выполнять на основе этих данных. Класс имеет такое же отношение к объекту, какое имеется между типом и переменной. То есть, в определении класса описывается форма данных и способы их использования, а объект — это сущность, созданная в соответствии со спецификацией формы данных. Здесь можно привести такую аналогию: если класс можно рассматривать как категорию, напри-

мер, известные актеры, то объект можно рассматривать как конкретного представителя этой категории, например, мультипликационный герой Лягушонок Кермит. Продолжая эту аналогию далее, можно сказать, что представление актеров в классе может включать описания действий, связанных с классом: чтение роли, выражение печали, демонстрация угрозы, получение награды и тому подобное. Если перейти к терминологии ООП, то можно сказать, что класс в языке C++ соответствует тому, что в некоторых языках программирования называется *объектным типом*, а объект в языке C++ соответствует экземпляру объекта или переменной экземпляра.

Теперь давайте перейдем к деталям. Вспомните такое объявление переменной:

```
int carrots;
```

В этом объявлении создается конкретная переменная (carrots), обладающая свойствами типа int. Другими словами, переменная carrots может хранить целое число и может использоваться в различных целях, например, для вычитания или сложения чисел. Что же касается cout, то про него можно сказать, что это объект, созданный для того, чтобы хранить свойства класса ostream. В определении класса ostream (еще один наследник файла iostream) описывается разновидность данных объекта ostream, а также операции, которые вы можете выполнять на основе этих данных, вроде помещения числа или строки в поток вывода. Точно так же и cin представляет собой объект, созданный со свойствами класса istream, который тоже определен в iostream.

На память!

Класс описывает все свойства типа данных, а объект является сущностью, созданной в соответствии с этим описанием.

Вы уже знаете, что классы представляют собой определяемые пользователями типы. Тем не менее, вы как пользователь не создаете классы ostream и istream. Подобно функциям, которые могут быть включены в библиотеки функций, классы могут быть помещены в библиотеки классов. Этот относится и к классам ostream и istream. Технически они не являются встроенными классами C++; наоборот, они представляют собой примеры классов, которые идут отдельно от языка. Определения классов находятся в файле iostream и не встроены в компилятор. Эти определения можно даже изменять, хотя это и не рекомендуется делать. (Если выразиться точнее, это абсолютно неверная затея.) Семейство классов iostream и связанное с ним семейство fstream (файловый ввод-вывод) являются единственными наборами определений классов в первых реализациях языка C++. Однако по решению комитета ANSI/ISO в стандарт было включено несколько дополнительных библиотек классов. Также во многих реализациях предлагаются дополнительные определения классов, входящие в состав пакета. Действительно, привлекательной чертой C++ является наличие содержательных и полезных библиотек классов, которые позволяют разрабатывать программы для платформ Unix, Macintosh и Windows.

В описании класса определяются все действия, которые могут быть выполнены над объектами этого класса. Чтобы выполнить такое действие над определенным объектом, вы посылаете этому объекту сообщение. Например, если вы хотите, чтобы объект cout отобразил строку, вы посылаете сообщение, которое, фактически, говорит: "Эй, объект! Отобрази-ка это.". Послать сообщение в языке C++ можно двумя способами. В одном из них используется метод класса, что, по сути, представляет

собой вызов функции. Другой способ, который используется применительно к объектам cin и cout, заключается в повторном определении операции. Так, оператор

```
cout << "I am not a crook."
```

использует повторно определенную операцию << для отправки команды "отобразить сообщение" объекту cout. В этом случае в сообщении содержится аргумент, представляющий собой строку, которую необходимо отобразить на экране. (Посмотрите на рис. 2.5.)

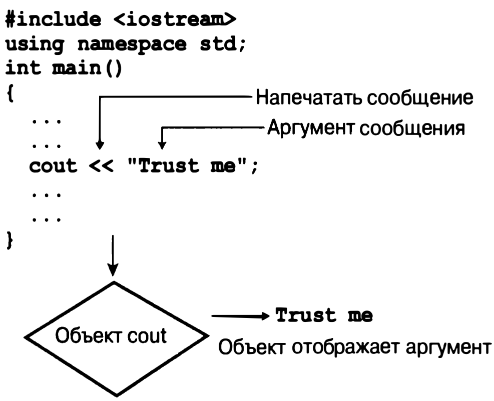


Рис. 2.5. Отправка сообщения объекту

Функции

Поскольку функции представляют собой модули, из которых строятся программы на языке C++, и поскольку они необходимы для определений ООП в языке C++, вам нужно хорошенько с ними познакомиться. Часть вопросов, связанных с функциями, требует серьезного рассмотрения, поэтому основной разговор о функциях мы поведем несколько позже, в главах 7 и 8. Тем не менее, если сейчас мы рассмотрим некоторые основополагающие характеристики функций, то при дальнейшем ознакомлении с функциями вы уже будете иметь о них некоторое представление. Итак, в оставшейся части этой главы мы будет говорить исключительно о фундаментальных характеристиках функций.

Функции в C++ можно условно разбить на две категории: функции, которые возвращают значения, и функции, не возвращающие значения. По каждому типу функции можно найти примеры в стандартной библиотеке функций C++. Кроме этого, вы можете создавать свои собственные функции каждого типа. Давайте сейчас рассмотрим библиотеку функций, имеющих возвращаемое значение, и потренируемся создавать свои собственные простые функции.

Использование функции, имеющей возвращаемое значение

Функция, имеющая возвращаемое значение, генерирует значение, которое вы можете присвоить переменной. Например, стандартная библиотека C/C++ содержит

функцию sqrt(), которая возвращает корень квадратный из числа. Допустим, нам необходимо вычислить корень квадратный из 6.25 и полученное значение присвоить переменной x. В своей программе вы можете использовать следующий оператор:

```
x = sqrt(6.25); // возвращает значение 2.5 и присваивает его переменной x
```

Выражение sqrt(6.25) активизирует, или *вызывает*, функцию sqrt(). Выражение sqrt(6.25) называется *вызовом функции*, активизируемая функция называется *вызываемой функцией*, а функция, содержащая вызов функции, называется *вызывающей функцией* (рис. 2.6).

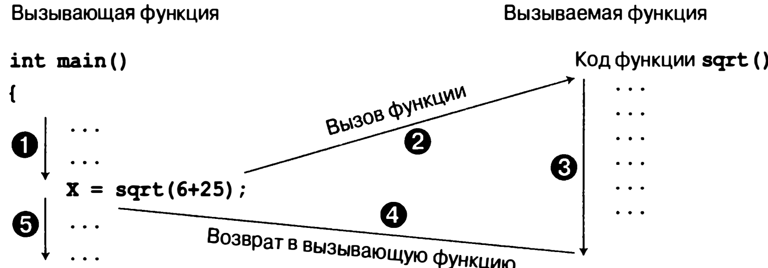


Рис. 2.6. Вызов функции

Информацией, которая отправляется функции, является значение в круглых скобках (в нашем случае — 6.25); говорят, что это значение *передается* функции. Значение, которое отсылается функции подобным образом, называется *аргументом* или *параметром* (рис. 2.7). В результате выполнения функции sqrt() получается результат 2.5. Он будет отправлен обратно вызывающей функции; полученное значение называется *возвращаемым значением* функции. Возвращаемое значение можно представить себе как значение, которое заменяет вызов функции в операторе после завершения выполнения функции. Так, в нашем примере возвращаемое значение присваивается переменной x. Говоря кратко, аргументом является та информация, которая отправляется функции, а возвращаемое значение является значением, которое отправляется обратно из функции.

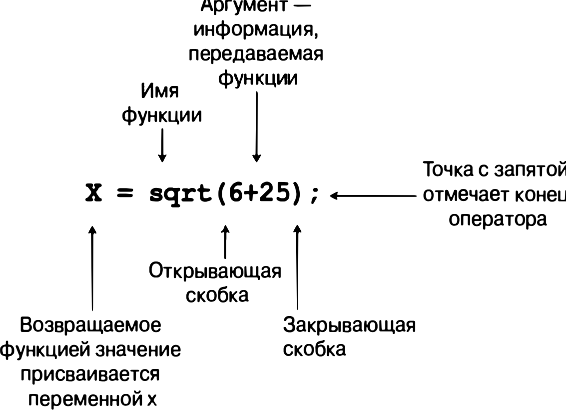


Рис. 2.7. Синтаксис вызова функции

Это практически все, за исключением того, что прежде чем компилятор C++ будет использовать функцию, он должен узнать, какого типа аргументы использует функция, и какого типа является возвращаемое ею значение. То есть, возвращает ли функция целочисленное значение? Символьное? Число с дробной частью? Признание виновности? Что-то другое? Если компилятор этого знать не будет, он не сможет интерпретировать возвращаемое значение. Передача информации такого рода в языке C++ осуществляется посредством оператора прототипа функции.

На память!

Программа на C++ должна содержать прототип каждой функции, используемой в программе.

Прототип выполняет ту же роль, что и объявление для переменных: в нем перечисляются используемые типы. Например, в библиотеке C++ функция `sqrt()` определена как функция, которая принимает число с (возможно) дробной частью (например, `6.25`) в качестве аргумента и возвращает число этого же типа. В некоторых языках программирования такие числа называются *вещественными числами*, а в C++ для них принято имя `double`. (В главе 3 мы поговорим о них подробно.) Прототип функции `sqrt()` имеет следующий вид:

```
double sqrt(double);   // прототип функции
```

Первое `double` означает, что функция `sqrt()` возвращает значение, имеющее тип `double`. `double` в круглых скобках означает, что для функции `sqrt()` необходим аргумент `double`. Таким образом, этот прототип описывает функцию `sqrt()` так, как она используется в следующем фрагменте кода:

```
double x;              // объявляет x как переменную, имеющую тип double
x = sqrt(6.25);
```

Завершающая точка с запятой в прототипе идентифицирует его в качестве оператора и поэтому присваивает ему статус прототипа, а не заголовка функции. Если точку с запятой опустить, то компилятор будет интерпретировать строку как заголовок функции, и будет считать, что дальше должно следовать тело функции, определяющее ее.

Если в программе используется функция `sqrt()`, значит, в ней должен быть и прототип функции. Для этого существуют два варианта:

- Можно ввести самостоятельно прототип функции в файл исходного кода.
- Можно воспользоваться заголовочным файлом `cmath` (в старых системах — файл `math.h`), который имеет в себе прототип.

Второй вариант более предпочтителен, поскольку заголовочный файл всегда имеет соответствующий прототип функции. Каждая функция в библиотеке C++ имеет свой прототип в одном или нескольких заголовочных файлах. Достаточно просмотреть описание функции в вашем руководстве или в оперативной справочной системе, где должно быть сказано, какой заголовочный файл необходимо использовать. Например, в описании функции `sqrt()` сказано о необходимости применения заголовочного файла `cmath`. (И в этом случае вам, возможно, нужно будет использовать старый заголовочный файл `math.h`, который работает для программ на C и на C++.)

Не следует путать прототип функции с определением функции. Прототип, как вы могли видеть, описывает только интерфейс функции. Другими словами, он описывает информацию, которая получает функция, и информацию, которую она отправля-

ет обратно. А в определении помещен код для работы функции – например, код для вычисления корня квадратного числа. В языках С и С++ в отношении библиотечных функций существует различие между прототипом и определением. Библиотечные файлы содержат компилированный код для функций, а в заголовочных файлах находятся прототипы функций.

Прототип функции должен быть помещен перед первым использованием функции. Чаще всего прототипы помещают непосредственно перед описанием функции `main()`. В листинге 2.4 показан пример использования библиотечной функции `sqrt()`; для получения прототипа используется файл `cmath`.

Листинг 2.4. sqrt.cpp

```
// sqrt.cpp -- использование функции sqrt()
#include <iostream>
#include <cmath> // или math.h
int main()
{
   using namespace std;
   double area;
   cout << "Enter the floor area, in square feet, of your home: ";
   cin >> area;
   double side;
   side = sqrt(area);
   cout << "That's the equivalent of a square " << side
        << " feet to the side." << endl;
   cout << "How fascinating!" << endl;
   return 0;
}
```

Замечание по совместимости

Если вы работаете со старым компилятором, то в листинге 2.4, возможно, необходимо будет использовать `#include <math.h>` вместо `#include <cmath>`.

Использование библиотечных функций

Библиотечные функции С++ хранятся в библиотечных файлах. Когда компилятор компилирует программу, он должен находить библиотечные файлы для используемых функций. Компиляторы отличаются между собой тем, какие библиотечные файлы они ищут автоматически. Если вы попытаетесь выполнить код из листинга 2.4 и получите сообщение о том, что `_sqrt` является неопределенной внешней функцией (звучит так, будто бы необходимо отменить ее использование), значит, дело в том, что компилятор не выполняет автоматический поиск библиотеки математических функций. (Компиляторы обычно добавляют к именам функций символ подчеркивания в качестве префикса — это еще один намек о том, что они оставляют за собой последнее слово о вашей программе.) Если вы получили такое сообщение, просмотрите документацию по компилятору: в ней должно быть сказано, как заставить компилятор искать соответствующую библиотеку. Например, в обычных реализациях Unix необходимо, чтобы в конце командной строки была указана опция `-lm` (то есть, ***l**ibrary **m**ath*):

```
CC sqrt.C -lm
```

Похожим способом используется и компилятор Gnu в среде Linux:

```
g++ sqrt.C -lm
```

Если просто включить заголовочный файл `cmath`, то прототип функции будет предоставлен, однако компилятор не обязательно будет производить поиск соответствующего библиотечного файла.

Ниже показан пример выполнения программы, представленной в листинге 2.4:

```
Enter the floor area, in square feet, of your home: 1536
That's the equivalent of a square 39.1918 feet to the side.
How fascinating!
```

Так как функция `sqrt()` работает со значениями, имеющими тип `double`, то в этом примере переменные тоже имеют тип `double`. Обратите внимание, что объявление переменной, имеющей тип `double`, выполнено в той же форме, или синтаксисе, как если бы вы объявляли переменную типа `int`:

```
имя-типа имя-переменной;
```

Тип `double` позволяет переменным `area` и `side` хранить значения с десятичными дробями, например `536.0` и `39.1918`. В переменной, имеющей тип `double`, явное целое число, такое как `536`, хранится как вещественное число с десятичной дробью `.0`. В главе 3 вы увидите, что тип `double` предлагает более широкий диапазон значений, нежели `int`.

В языке программирования C++ новую переменную можно объявлять в любом месте программы, поэтому в файле `sqrt.cpp` переменная `side` объявлена непосредственно перед ее использованием. В языке C++ присваивать значение переменной можно и в момент ее создания, поэтому допускается следующая форма записи:

```
double side = sqrt(area);
```

Такая форма называется *инициализацией*; мы поговорим о ней в главе 3.

Следует отметить, что объект `cin` знает о том, как преобразовать информацию из потока ввода к типу `double`, а объект `cout` знает, как поместить тип `double` в поток вывода. Как мы уже говорили ранее, эти объекты являются интеллектуальными.

Разновидности функций

Для некоторых функций требуется более одного элемента информации. Такие функции используют несколько аргументов, которые разделяются запятыми. Например, математическая функция `pow()` принимает два аргумента и возвращает значение, равное первому аргументу, возведенному в степень, показателем которой является второй аргумент. Прототип этой функции выглядит так:

```
double pow(double, double); // прототип функции с двумя аргументами
```

Если, например, вы хотите вычислить 5^8 (5 в восьмой степени), тогда можно использовать такую функцию:

```
answer = pow(5.0, 8.0);// вызов функции с использованием списка аргументов
```

Другие функции вообще не принимают никакие аргументы. Например, одна из библиотек C (библиотека, связанная с заголовочным файлом `cstdlib` или `stdlib.h`) включает функцию `rand()`, не принимающую аргументов и возвращающую случайное целое число. Ее прототип выглядит так:

```
int rand(void);        // прототип функции, не принимающей аргументов
```

Ключевое слово `void` явным образом указывает на то, что функция не принимает аргументов. Если опустить это слово и оставить скобки пустыми, C++ интерпрети-

рует это как неявное объявление об отсутствии аргументов. Эту функцию можно использовать следующим образом:

```
myGuess = rand(); // вызов функции без аргументов
```

Обратите внимание, что в отличие от некоторых языков программирования, в C++ необходимо использовать круглые скобки в вызове функции даже в том случае, если аргументы отсутствуют.

Существуют также функции, которые не имеют возвращаемого значения. Например, предположим, что вы написали функцию, которая отображает число в денежном формате (доллары и центы). Вы можете передать ей аргумент, например, `23.5`, и на экране будет отображена сумма `$23.50`. Поскольку эта функция выводит значение на экран, а не вызывает программу, она не возвращает значения. Эту особенность функции вы указываете в ее прототипе с помощью ключевого слова `void` для возвращаемого типа:

```
void bucks(double);//прототип для функции, не имеющей возвращаемого значения
```

Поскольку функция `bucks()` не возвращает значения, вы не сможете использовать эту функцию как часть оператора присваивания или какого-либо другого выражения. Наоборот, вы имеете дело с чистым оператором вызова функции:

```
bucks(1234.56); // вызов функции, возвращаемое значение отсутствует
```

В некоторых языках программирования термин *функция* используется только для тех функций, которые возвращают значения, а термин *процедура* или *подпрограмма* — для тех функций, которые не значения возвращают. В языке C++, в отличие от C, термин *функция* используется для обеих разновидностей функций.

Функции, определяемые пользователем

В стандартной библиотеке C предусмотрено свыше 140 предварительно определенных функций. Если какая-нибудь из них подходит для решения вашей задачи, то, конечно, ее можно использовать. Однако нередко приходится писать собственные функции, особенно при разработке классов. В любом случае, придумывать свои собственные функции очень интересно, поэтому давайте попробуем научиться этому ремеслу. Вы уже использовали некоторые предварительно определенные функции, и все они имели имя `main()`. Каждая программа на C++ обязана иметь функцию `main()`, которую должен определить пользователь. Предположим, вам нужно добавить вторую функцию, определенную пользователем. Как и в случае с библиотечной функцией, к функции, определенной пользователем, можно обратиться по имени. И, как и для библиотечной функции, вы должны поместить прототип функции прежде, чем будет использована сама функция; прототип помещается перед определением функции `main()`. Новым элементом здесь является исходный код, который вы должны предоставить для новой функции. Самый простой способ сделать это — поместить этот код в этом же файле после кода функции `main()`. В листинге 2.5 вы найдете все эти элементы.

Листинг 2.5. oufrunc.cpp

```
// ourfunc.cpp -- определение вашей собственной функции
#include <iostream>
void simon(int); // прототип функции для simon()
```

```
int main()
{
   using namespace std;
   simon(3); // вызов функции simon()
   cout << "Pick an integer: ";
   int count;
   cin >> count;
   simon(count); // вызываем ее еще раз
   cout << "Done!" << endl;
   return 0;
}
void simon(int n) // определяем функцию simon()
{
   using namespace std;
   cout << "Simon says touch your toes " << n << " times." << endl;
} // функции void не требуют возвращающих операторов
```

Функция `main()` вызывает функцию `simon()` два раза: один раз с аргументом 3, а другой раз — с аргументом-переменной `count`. Между этими вызовами пользователь вводит целое число, которое присваивается переменной `count`. В этом примере в приглашении на ввод значения `count` не используется символ новой строки. В результате пользователь будет вводить значение в той же строке, что и приглашение. Далее показан результат выполнения этой программы.

```
Simon says touch your toes 3 times.
Pick an integer: 512
Simon says touch your toes 512 times.
Done!
```

Форма функции

Определение функции `simon()` в листинге 2.5 имеет тот же вид, что и определение функции `main()`. Во-первых, присутствует заголовок функции. Далее в фигурных скобках следует тело функции. Обобщить форму определения функции можно следующим образом:

```
тип имя_функции(список_аргументов)
{
   операторы
}
```

Обратите внимание, что в исходном коде, определяющем функцию `simon()`, присутствует закрывающая фигурная скобка функции `main()`. Как и в C, и в отличие от языка Pascal, в языке C++ не разрешается помещать одно определение функции внутри другого. Каждое определение функции располагается отдельно от остальных определений; все функции создаются одинаково (рис. 2.8.).

Заголовки функций

Функция `simon()` из листинга 2.5 имеет следующий заголовок:

```
void simon(int n)
```

void означает, что функция simon() не имеет возвращаемого значения. Поэтому при вызове этой функции не генерируется значение, которое вы могли бы присвоить переменной main(). Поэтому первый вызов функции имеет вид:

```
simon(3);            // справедливо для функций void
```

Поскольку в чистом виде функция simon() не возвращает значения, вы можете использовать ее следующим образом:

```
simple = simon(3);  // нельзя использовать для функций void
```

```
#include <iostream>
using namespace std;

void simon(int);
double taxes(double);

int main()
{
  ...
  return 0;
}

void simon(int n);
{
  ...
}

double taxes(double t)
{
  ...
  return 2 * t;
}
```


Рис. 2.8. Чередование определений функций в файле

int n в скобках означает, что вы будете использовать функцию simon() с одним аргументом, имеющим тип int. n — это новая переменная, которой было присвоено значение, переданное во время вызова функции. Поэтому при вызове функции

```
simon(3);
```

переменной n, определенной в заголовке функции simon(), присваивается значение 3. Когда оператор cout в теле функции использует переменную n, он использует значение, переданное в вызове функции. Поэтому simon(3) отображает в своем выводе значение 3. В результате вызова simon(count) на экране будет отображено значение 512, поскольку это значение присвоено переменной count. Короче говоря, заголовок для simon() говорит о том, что эта функция принимает один аргумент, имеющий тип int, и что она не имеет возвращаемого значения.

Давайте взглянем еще раз на заголовок функции main():

```
int main()
```

int означает, что функция main() возвращает целочисленное значение. Пустые скобки (которые не обязательно могут содержать void) означают, что эта функция не принимает аргументов. Функции, которые имеют возвращаемые значения, должны использовать ключевое слово return для передачи возвращаемого значения и

завершения выполнения функции. Поэтому в конце функции `main()` используется следующий оператор:

```
return 0;
```

Логически все верно: функция `main()` вроде бы должна возвращать значение, имеющее тип `int`, и таковым значением является ноль. А для чего вы возвращаете значение? Все-таки ни в одной программе вы не увидите, чтобы какая-нибудь функция вызывала функцию `main()`:

```
squeeze = main(); // такого нет в наших программах
```

Дело в том, что здесь предполагается, что вашу программу может вызывать операционная система (например, Unix или DOS). Поэтому возвращаемое функцией `main()` значение возвращается не в другую часть программы, а в ОС. Во многих ОС могут использоваться значения, возвращаемые программами. Например, сценарии оболочек Unix и командные файлы DOS могут быть созданы для запуска программ и проверки возвращаемых ими значений, обычно называемых *значениями выхода*. Обычно нулевое значение на выходе означает, что программа была выполнена успешно, в то время как ненулевое значение свидетельствует об ошибке. Поэтому вы можете написать программу на C++ для возврата ненулевого значения, если, скажем, она не смогла открыть файл. Затем можно было бы разработать сценарий оболочки или командный файл для запуска такой программы и принятия альтернативного действия в случае сбоя программы.

Ключевые слова

Ключевые слова — это словарь компьютерного языка. В этой главе мы использовали четыре ключевых слова: `int`, `void`, `return` и `double`. Поскольку эти ключевые слова зарезервированы исключительно для нужд языка C++, вы не можете использовать их для других целей. То есть, нельзя использовать `return` в качестве имени переменной или `double` в качестве имени функции. Однако их можно применять как часть имени, например `painter` (скрытое `int`) или `return_aces`. В приложении Б вы найдете полный перечень ключевых слов языка C++. Между прочим, `main` не является ключевым словом, поскольку не является частью языка программирования. Напротив, это имя обязательной функции. `main` можно использовать в качестве имени переменной. (Что может привести к возникновению проблем, рассказывать о которых в этой книге мы не имеем возможности; лучше всего этого не делать.) Точно так же и другие имена функций и объектов не являются ключевыми словами. Однако использование одного и того же имени, например `cout`, для объекта и для переменной в программе приведет к ошибке в работе компилятора. Другими словами, можно указать `cout` в качестве имени переменной в функции, которая не использует объект `cout` для вывода, и нельзя использовать `cout` одновременно в качестве имени переменной и в качестве объекта вывода.

Использование определяемых пользователем функций, имеющих возвращаемое значение

Мы продолжаем рассмотрение функций и переходим к тем из них, в которых используется оператор возврата. На примере функции `main()` уже был продемонстрирован проект функции, имеющей возвращаемое значение: определите возвращаемый тип в заголовке функции и укажите `return` в конце тела функции. Эту форму можно использовать, чтобы решить непростую задачу для иностранцев, приезжающих в Великобританию. В Великобритании используются весы, мерой которых являются

стоуны (*stone* переводится как *камень*; 1 стоун равен 14 фунтам, или приблизительно 6,35 кг), а в США используются фунты или килограммы. В этом контексте мера *stone* может быть выражена как в килограммах, так и в фунтах. (В английском языке нет такого внутреннего согласования, как в языке C++.) Один стоун равен 14 фунтам, и в программе, представленной в листинге 2.6, для этого преобразования используется функция.

Листинг 2.6. convert.cpp

```
// convert.cpp -- преобразовывает стоуны в фунты
#include <iostream>
int stonetolb(int); // прототип функции
int main()
{
   using namespace std;
   int stone;
   cout << "Enter the weight in stone: ";
   cin >> stone;
   int pounds = stonetolb(stone);
   cout << stone << " stone = ";
   cout << pounds << " pounds." << endl;
   return 0;
}
int stonetolb(int sts)
{
   return 14 * sts;
}
```

Далее показан пример выполнения этой программы:

```
Enter the weight in stone: 14
14 stone = 196 pounds.
```

В функции `main()` программа использует объект `cin` для передачи значения переменной целочисленного типа `stone`. Это значение передается функции `stonetolb()` в качестве аргумента и присваивается переменной `sts` в этой функции. Функция `stonetolb()` использует ключевое слово `return` для возврата значения `14*sts` функции `main()`. Это пример того, что после оператора `return` не обязательно должно следовать простое число. В этом случае с помощью сложного выражения вы избавляетесь от необходимости создавать новую переменную, которой должно быть присвоено значение, прежде чем оно может быть возвращено. Программа вычисляет значение этого выражения (в нашем примере — `196`) и возвращает его. Для возврата значения можно использовать и более громоздкую форму записи:

```
int stonetolb(int sts)
{
   int pounds = 14 * sts;
   return pounds;
}
```

Оба варианта дают один и тот же результат, однако для второго варианта требуется больше кода.

В общем случае, функцию с возвращаемым значением можно использовать там, где можно использовать простую константу того же типа, что и функция. Например,

функция stonetolb() возвращает значение, имеющее тип int. Значит, функцию можно использовать следующим образом:

```
int aunt = stonetolb(20);
int aunts = aunt + stonetolb(10);
cout << "Ferdie weighs " << stonetolb(16) << " pounds." << endl;
```

В каждом из случаев программа находит возвращаемое значение и затем использует его в этих операторах.

Как можно было видеть в приведенных примерах, прототип функции описывает ее интерфейс, то есть способ взаимодействия функции с остальной частью программы. Список аргументов показывает, какой тип информации передается функции, а тип функции указывает на тип возвращаемого ею значения. Программисты иногда характеризуют функции как *черные ящики* (термин, позаимствованный из электроники), для которых известны только входящий и исходящий потоки информации. Это определение лучше всего характеризует прототипы функций (рис. 2.9).

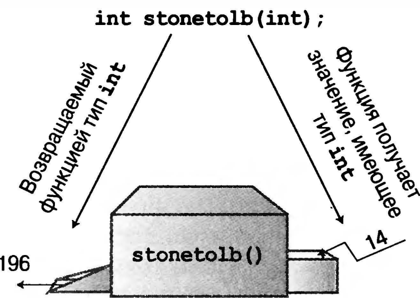


Рис. 2.9. Прототип функции и функция как черный ящик

Несмотря на свою краткость и простоту, stonetolb() обладает полным набором функциональных возможностей:

- Имеет заголовок и тело.
- Принимает аргумент.
- Возвращает значение.
- Нуждается в прототипе.

Функцию stonetolb() можно брать за основу при проектировании других функций. В главах 7 и 8 мы продолжим изучение функций. А материал в этой главе должен позволить вам понять работу функций в языке C++.

Местоположение директивы using в программах со множеством функций

Обратите внимание на то, что в листинге 2.5 директива using присутствует в каждой из двух функций:

```
using namespace std;
```

Это объясняется тем, что каждая функция использует объект cout и поэтому должна иметь доступ к определению cout в пространстве имен std.

Еще один способ сделать пространство имен std доступным для обеих функций в листинге 2.5 состоит в том, чтобы поместить директиву за пределами функций и перед ними:

```
// ourfuncl.cpp -- изменение местоположения директивы using
#include <iostream>
using namespace std; // влияет на все определения функций в этом файле
void simon(int);
int main()
{
   simon(3);
   cout << "Pick an integer: ";
   int count;
   cin >> count;
   simon(count);
   cout << "Done!" << endl;
   return 0;
}
void simon(int n)
{
   cout << "Simon says touch your toes " << n << " times." << endl;
}
```

Предпочтительнее делать так, чтобы доступ к пространству имен std был предоставлен только тем функциям, которым он действительно необходим. Например, в листинге 2.6 объект cout используется только функцией main(), поэтому нет необходимости делать доступным пространство имен std для функции stonetolb(). Таким образом, директива using помещается внутри только функции main(), предоставляя доступ к пространству имен std только этой функции.

Итак, сделать доступным пространство имен std для программы можно несколькими способами. Перечислим их:

- Можно поместить

  `using std namespace;`

  перед определением функции в файле, в результате чего все содержимое пространства имен std будет доступно для каждой функции в файле.

- Можно поместить

  `using std namespace;`

  в определении функции, в результате чего все содержимое пространства имен std будет доступно для этой функции.

- Вместо того чтобы использовать

  `using std namespace;`

  можно поместить директивы

  `using std::cout;`

  в определении функции и сделать доступным для каждой функции конкретный элемент, например, cout.

- Можно опустить директивы using полностью и использовать префикс std:: всякий раз при использовании элементов из пространства имен std:

  `std::cout << "I'm using cout and endl from the std namespace" << std::endl;`

Пример из практики: соглашения по именованию

У программистов C++ есть хорошая (возможно, это и не так) возможность использовать различные варианты присвоения имен функциям, классам и переменным. Среди программистов бытуют строгие и разнообразные мнения относительно стиля написания кода, что нередко служит поводом для настоящих словесных баталий на открытых форумах. Если рассмотреть возможные варианты присвоения имен функциям, то программист может использовать один из следующих:

```
MyFunction()
myfunction()
myFunction()
my_function()
my_funct()
```

Выбор зависит от команды разработчиков, индивидуальных особенностей используемых методов или библиотек, а также от предпочтений конкретного программиста. Существует мнение, что в языке C++ уместен любой допустимый стиль, поэтому любое решение будет зависеть лично от вас.

Если отстраниться от правил языка программирования, следует отметить, что выработка личного стиля именования, способствующего логичности и точности записи, заслуживает уважения. Если программист выработает аккуратный и узнаваемый стиль именования, то это не только будет служить признаком того, что программы, написанные им, являются качественными, но и может помочь ему в программистской карьере.

Резюме

Программа на языке C++ состоит из одного или нескольких модулей, называемых функциями. Выполнение программы начинается с функции `main()` (все символы — в нижнем регистре), поэтому ваша программа обязательно должна включать эту функцию. Функция состоит из заголовка и тела. В заголовке функции указывается, каким является возвращаемое значение (если таковое существует), генерируемое функцией, и какую информацию принимает функция в виде аргументов. Тело функции состоит из последовательности операторов языка C++, заключенных в фигурные скобки (`{}`).

В языке программирования C++ выделяют следующие типы операторов:

- **Оператор объявления**. В операторе объявления указывается имя и тип переменной, которая используется в функции.
- **Оператор присваивания**. Этот оператор использует операцию присваивания (=) для присваивания значения переменной.
- **Оператор сообщений**. Оператор сообщений посылает сообщение объекту, инициируя некоторое действие.
- **Вызов функции**. Вызов функции активизирует ее работу. Когда вызываемая функция завершает свою работу, программа возвращается к оператору в вызывающей функции, следующим за вызовом функции.
- **Прототип функции**. В прототипе функции объявляется тип возвращаемого функцией значения, а также количество и тип аргументов, передаваемых функции.
- **Оператор возврата**. Оператор возврата посылает значение из вызываемой функции обратно вызывающей функции.

Класс представляет собой определяемую пользователем спецификацию типа данных. В ней подробно описывается способ представления информации и действия, которые могут выполняться над этими данными. Объект — это сущность, созданная в соответствии с предписанием класса, а простая переменная является сущностью, созданной в соответствии с описанием типа данных.

В языке C++ имеются два предварительно определенных объекта (cin и cout) для обработки ввода и вывода. Они являются примерами классов istream и ostream, которые определены в файле iostream. Эти классы рассматривают ввод и вывод в виде потоков символов. Операция вставки (<<), которая определена для класса ostream, позволяет помещать данные в поток вывода, а операция извлечения (>>), которая определена для класса istream, позволяет извлекать информацию из потока ввода. Как cin, так и cout являются интеллектуальными объектами, способными автоматически преобразовывать информацию из одной формы в другую в соответствии с контекстом программы.

Язык программирования C++ может использовать обширный набор библиотечных функций языка C. Чтобы использовать библиотечную функцию, необходимо включить заголовочный файл, который предоставляет для функции ее прототип.

Теперь, когда вы уже знаете, что собой представляют простые программы на языке C++, можно приступить к изучению следующей главы, вникая во все более широкий круг вопросов.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля по каждой главе можно найти в приложении К.

1. Что такое модули программы на языке C++?
2. Что выполняет следующая директива препроцессора?
   ```
   #include <iostream>
   ```
3. Что выполняет следующий оператор?
   ```
   using namespace std;
   ```
4. Какой оператор вы могли бы использовать, чтобы напечатать фразу "Hello, world" и перейти на новую строку?
5. Какой оператор вы могли бы использовать для создания переменной целочисленного типа с именем cheeses?
6. Какой оператор вы могли бы использовать, чтобы присвоить переменной cheeses значение 32?
7. Какой оператор вы могли бы использовать, чтобы прочитать значение, введенное с клавиатуры, и присвоить его переменной cheeses?
8. Какой оператор вы могли бы использовать, чтобы напечатать фразу "We have X varieties of cheese", в которой текущее значение переменной cheese заменяет X?
9. Что говорят вам о функциях следующие их прототипы?
   ```
   int froop(double t);
   void rattle(int n);
   int prune(void);
   ```
10. В каких случаях не используется ключевое слово return при определении функции?

Упражнения по программированию

1. Напишите программу на C++, которая отобразит вашу фамилию и почтовый адрес на экране монитора.
2. Напишите программу на C++, которая выдает запрос на ввод расстояния в фарлонгах и преобразовывает его в ярды. (Один фарлонг равен 220 ярдам.)
3. Напишите программу на C++, которая использует три определяемых пользователем функции (включая `main()`), и результатом ее выполнения является следующий вывод:

   ```
   Three blind mice
   Three blind mice
   See how they run
   See how they run
   ```

   Одна функция, вызываемая два раза, должна генерировать первые две строки, а вторая функция, также вызываемая два раза, должна генерировать оставшиеся строки.
4. Напишите программу, в которой функция `main()` вызывает определяемую пользователем функцию, которая в качестве аргумента принимает значение температуры по Цельсию и возвращает эквивалентное значение температуры по Фаренгейту. Программа должна выдать запрос на ввод значения по Цельсию и отобразить следующий результат:

   ```
   Please enter a Celsius value: 20
   20 degrees Celsius is 68 degrees Fahrenheit.
   ```

   Для справки, формула для выполнения этого преобразования:

   Температура в градусах по Фаренгейту = 1,8 * Температура в градусах по Цельсию + 32
5. Напишите программу, в которой функция `main()` вызывает определяемую пользователем функцию, которая в качестве аргумента принимает расстояние в световых годах и возвращает расстояние в астрономических единицах. Программа должна выдать запрос на ввод значения светового года и отобразить следующий результат:

   ```
   Enter the number of light years: 4.2
   4.2 light years = 265608 astronomical units.
   ```

   Астрономическая единица равна среднему расстоянию Земли от Солнца (около 150 000 000 км, или 93 000 000 миль), а световой год соответствует расстоянию, пройденному лучом света за один земной год (примерно 10 триллионов километров, или 6 триллионов миль). (Ближайшая звезда после Солнца находится на расстоянии 4.2 световых года.) Используйте тип `double` (как в листинге 2.4) и следующий коэффициент преобразования:

   1 световой год = 63 240 астрономических единиц
6. Напишите программу, которая выдает запрос на ввод значения часов и значения минут. Функция `main()` должна передать эти два значения функции, имеющей тип `void`, которая отображает эти два значения в следующем виде:

   ```
   Enter the number of hours: 9
   Enter the number of minutes: 28
   Time: 9:28
   ```

ГЛАВА 3

Работа с данными

В этой главе:

- Правила присваивания имен переменным в языке C++
- Встроенные целочисленные типы данных в языке C++: `unsigned long`, `long`, `unsigned int`, `int`, `unsigned short`, `short`, `char`, `unsigned char`, `signed char`, `bool`
- Файл `climits`, в котором определены ограничения системы в отношении различных целочисленных типов данных
- Числовые константы различных целочисленных типов
- Использование квалификатора `const` для создания символьных констант
- Встроенные типы данных с плавающей точкой в языке C++: `float`, `double`, `long double`
- Файл `cfloat`, в котором определены ограничения системы в отношении различных типов данных с плавающей точкой
- Числовые константы различных типов данных с плавающей точкой
- Арифметические операции в языке C++
- Автоматическое преобразование типов
- Принудительное преобразование типов (приведение типов)

Смысл объектно-ориентированного программирования (ООП) заключается в том, чтобы программист имел возможность разрабатывать и распространять свои собственные типы данных. Разработанный вами тип должен наиболее точно соответствовать тем данным, с которыми вы будете иметь дело. Если вам удастся справиться с этой задачей, то в дальнейшем работать с данными будет намного удобнее. Однако прежде чем вы сможете создавать свои собственные типы данных, вы должны изучить встроенные типы данных в языке C++, поскольку именно они будут лежать в основе разрабатываемых типов.

Встроенные типы данных в языке C++ делятся на две группы: основные типы и составные типы. В этой главе речь пойдет об основных типах, которые представляют целые числа и числа с плавающей точкой. На первый взгляд может показаться, что для представления этих чисел можно обойтись всего двумя типами. Однако это не так: в языке C++ считается, что просто целочисленный тип и тип с плавающей точкой не в состоянии удовлетворить все требования программистов, поэтому каждый из этих типов предлагает несколько подтипов данных. В главе 4 мы рассмотрим некоторые типы, входящие в состав базовых типов; забегая наперед, можно сказать, что к ним относятся массивы, строки, указатели и структуры.

Естественно, программа должна иметь способ распознавания хранящихся данных. Как раз о таком способе — использовании переменных — мы и поговорим в этой главе. Затем мы рассмотрим арифметические действия в языке C++, а в последней части главы речь пойдет о преобразовании значений из одного типа в другой.

Простые переменные

Как правило, программа должна хранить разнообразную информацию. Это может быть и текущая цена на акции компании IBM, и величина средней влажности в Нью-Йорке в августе месяце, и самая часто встречающаяся буква в тексте Конституции США и частота ее употребления, и количество современных пародистов Элвиса Пресли и многое-многое другое. Чтобы сохранить элемент информации в памяти компьютера программа должна знать три основных свойства:

- Где будет храниться информация.
- Какое значение будет храниться.
- Какого рода эта информация.

В примерах, приведенных в предыдущих главах, использовалась стратегия объявления переменной. Тип, указываемый в разделе объявления, описывает разновидность информации, а имя переменной является символическим обозначением значения. Предположим, что Игорь, ассистент заведующего лабораторией, использует следующие операторы:

```
int braincount;
braincount = 5;
```

Эти операторы сообщают программе, что она хранит целое число, и что имя `braincount` представляет целочисленное значение, в данном случае 5. По сути, программа выделяет некоторую область памяти, способную уместить целое число, отмечает ее, присваивает ей метку `braincount` и копирует в нее значение 5. Эти операторы не говорят вам (или Игорю) о том, где именно в памяти хранится это значение, а сама программа отслеживает эту информацию. Конечно, вы можете использовать операцию `&`, чтобы получить адрес ячейки, в которой хранится значение переменной `braincount`. Об этой операции мы поговорим в следующей главе, где будет рассмотрена еще одна стратегия распознавания данных — использование указателей.

Имена, присваиваемые переменным

В языке программирования C++ приветствуется присваивание переменным выразительных имен. Если переменная представляет стоимость поездки, то ей следует присвоить такое имя, например, как `cost_of_trip` или `costOfTrip`, но не `x` или `cot`. В языке C++ необходимо придерживаться следующих правил присваивания имен:

- Допускается использование только тех символов, которые представляют алфавит и цифры, а также символ подчеркивания (_).
- Первым символом не должна быть цифра.
- Символы в верхнем регистре отличаются от символов в нижнем регистре.

- В качестве имени нельзя использовать ключевое слово C++.
- Имена, начинающиеся с двойного подчеркивания или с одного подчеркивания и следующим за ним символом в верхнем регистре, зарезервированы для реализаций C++, то есть их используют компиляторы и ресурсы. Имена, начинающиеся с одного символа подчеркивания, зарезервированы для использования в качестве глобальных идентификаторов в реализациях C++.
- Имя может иметь произвольную длину, и все символы в имени являются значимыми.

Предпоследнее правило немного отличается от предыдущих, поскольку использование такого имени, как `__time_stop` или `_Donut`, не приведет к возникновению ошибки компилятора; наоборот, результатом будет непредсказуемое поведение программы. Другими словами, в этом случае невозможно определить, каким будет конечный результат. Дело в том, что отсутствие в данном случае ошибки компилятора объясняется тем, что эти имена не являются недопустимыми, а зарезервированными для реализаций C++. Вопрос о глобальных именах связан с тем, в каком месте программы объявлены эти имена; об этом речь пойдет в главе 4.

Последнее правило отличает C++ от стандарта ANSI для языка C (C99), в котором значимыми являются только первые 63 символа в имени. (Пара имен, у которых первые 63 символа одинаковые, а 64-тые символы разные, в стандарте ANSI для языка C считаются одинаковыми именами.)

Ниже показаны примеры допустимых и недопустимых имен в языке C++:

```
int poodle;      // допустимое имя
int Poodle;      // допустимое имя, отличается от poodle
int POODLE;      // допустимое имя, отличается от двух предыдущих
Int terrier;     // недопустимое имя — нужно использовать int, а не Int
int my_stars3    // допустимое имя
int _Mystars3;   // допустимое имя, но зарезервированное —
                 // начинается с подчеркивания
int 4ever;       // недопустимое имя, потому что начинается с цифры
int double;      //недопустимое имя — double является ключевым словом C++
int begin;       // допустимое имя — begin является ключевым словом,
                 // но в языке Pascal
int __fools;     // допустимое имя, но зарезервированное —
                 // начинается с двойного подчеркивания
int the_very_best_variable_i_can_be_version_112; // допустимое имя
int honky-tonk; //недопустимое имя — запрещается использование дефиса
```

Обычно, если переменной присваивается имя, состоящее из двух или более слов, то для разделения слов используется символ подчеркивания, как в имени `my_onions`, или же первую букву каждого слова, кроме первого, записывают в верхнем регистре, как в имени `myEyeTooth`. (Ветераны языка C обычно стараются использовать символ подчеркивания, а приверженцы Pascal предпочитают применять заглавные буквы.) В любом случае, каждая форма записи упрощает визуальное восприятие отдельных слов и помогает отличить имя, `carDrip` от, скажем, имени `cardRip`, или `boat_sport` от `boats_port`.

Пример из практики: имена переменных

Вокруг схем именования переменных, как и схем именования функций, возникают горячие дискуссии, и мнения программистов по этому поводу очень расходятся. Как и в случае с именами функций, компилятору C++ все равно, какие имена вы присваиваете своим переменным — лишь бы они были записаны в соответствии с правилами именования. Однако вы во многом выиграете, если вам удастся выработать свой логичный и аккуратный стиль именования.

Как и в случае присваивания имен функциям, ключевым моментом в присваивании имен переменным является использование заглавных букв (см. врезку "Соглашения по именованию" в главе 2). Многие программисты, однако, включают в имя переменной дополнительный уровень информации — префикс, который характе́ризует тип переменной или ее содержимое. Например, переменную `myWeight`, имеющую целочисленный тип, можно назвать `nMyWeight`; здесь префикс `n` используется для представления целого числа — это бывает полезным при просмотре кода, когда описание переменной находится слишком далеко от данной строки. Как вариант, эта переменная может иметь имя `intMyWeight`, которое описывает ее более точным образом, хотя и содержит большее количество символов (истинное мучение для определенного круга программистов). Подобным образом применяются и другие префиксы: `str` или `sz` можно использовать для представления строки символов с завершающим нулем, `b` может представлять логическое (булевское, b от слова "Boolean") значение, `p` — указатель (от слова pointer), `c` — один символ (от слова "character").

По мере изучения языка C++ вы будете встречать множество примеров применения префиксов в именах переменных (включая "изящный" префикс `m_lpctstr` — значение члена класса, которое содержит длинный указатель на неизменяемую строку символов с завершающим нулем), а также другие, немного причудливые и, возможно, алогичные стили, которыми при желании можно будет пользоваться. Как и во всех других стилистических, субъективных сторонах C++, лучше всего придерживаться логичности и точности. Старайтесь присваивать такие имена, которые отвечали бы вашим требованиям, предпочтениям и собственному стилю. (Или, в случае необходимости, подбирайте имена, которые более всего предпочтительны для вашего работодателя.)

Целочисленные типы

Целыми являются числа без дробной части, например 2, 98, –5286 и 0. Существует множество целых чисел, и если предположить, что множество — это бесконечное количество, то ни один конечный объем памяти компьютера не в состоянии представить все возможные целые числа. Таким образом, компьютерный язык программирования может представить только некоторое подмножество целых чисел. Некоторые языки программирования, такие как Pascal, предлагают только один целочисленный тип (один тип, который охватывает все возможные целые числа!), а язык C++ предлагает несколько вариантов целых чисел. Благодаря этому вы можете выбрать такой целочисленный тип, который будет лучше всего отвечать вашим требованиям. Здесь имеется в виду соответствие типа данным, при котором прогнозируются разработанные типы данных ООП.

Целочисленные типы в языке C++ отличаются друг от друга объемом памяти, выделяемой для хранения данных. Чем больше объем памяти, тем шире может быть диапазон представляемых целых чисел. Кроме этого, одни типы (со знаком) могут представлять положительные и отрицательные числа, а другие (без знака) не могут представлять отрицательные числа. Обычно для описания объема памяти, используемого для хранения целого числа, применяется термин *ширина* или *разрядность*. Чем больше памяти необходимо для хранения значения, тем выше разрядность. В языке C++ базовыми целочисленными типами, в порядке увеличения разрядности, являются `char`, `short`, `int` и `long`. Каждый из этих типов имеет варианты со знаком и без

такового. Таким образом, на ваш выбор предлагается восемь различных целочисленных типов! Давайте рассмотрим эти типы более детально. Поскольку тип char обладает некоторыми специальными свойствами (чаще всего он используется для представления символов, а не чисел), то сначала мы поговорим об остальных типах.

Целочисленные типы short, int и long

Память компьютера организована в виде элементов, называемых *битами* (см. врезку "Биты и байты" далее в этой главе). Используя различное количество битов для хранения значений, типы short, int и long в языке C++ могут представлять до трех различных целочисленных разрядностей. Было бы удобно, если бы каждый тип в каждой системе всегда имел некоторую постоянную разрядность — например, если бы short всегда имел 16 битов, int имел 32 бита и так далее. Однако в реальной жизни не все так просто. Тем не менее, ни один из вариантов не может быть подходящим для каждого разработчика. Язык C++ предлагает гибкий стандарт, позаимствованный у C, с некоторыми гарантированными минимальными размерами типов. Перечислим их:

- Целочисленный тип short представлен не менее чем 16 битами.
- Целочисленный тип int как минимум такой же, как и тип short.
- Целочисленный тип long представлен не менее чем 32 битами, и как минимум такой же, как тип int.

Биты и байты

Основной единицей в памяти компьютера является *бит*. Его можно представить как электронный переключатель, который может быть установлен в два положения: "включен" и "выключен". Положение "выключен" равносильно значению 0, а положение "включен" — значению 1. Порция памяти, состоящая из 8 битов, может представлять одну из 256 различных комбинаций. Число 256 получено исходя из того, что каждый бит может представлять ноль или единицу, что в конечном итоге дает общую сумму комбинаций для 8 бит: 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2, или 256. Таким образом, 8-битная структура может представить число от 0 до 255, или число от –128 до 127. Каждый дополнительный бит удваивает количество комбинаций. Это означает, что вы можете использовать 16-битную порцию памяти для представления числа от 0 до 65 535 и 32-битную порцию памяти для представления числа от 0 до 4 294 672 295.

1 *байт* обычно означает 8-битную порцию памяти. В этом смысле байт представляет собой единицу измерения, которая описывает количество памяти в компьютере, при этом 1 Кбайт равен 1024 байтам, а 1 Мбайт составляет 1024 Кбайт. Однако в языке C++ понятие *байт* имеет два значения. В языке C++ байт состоит из как минимум достаточного количества смежных битов, которые могут вместить базовый набор символов для реализации. Другими словами, количество возможных значений должно быть равно или превышать количество индивидуальных символов. В США базовыми наборами символов обычно являются наборы ASCII или EBCDIC, каждый из которых может занимать 8 битов, поэтому 1 байт в языке C++ обычно равен 8 битам в системах, использующих эти наборы символов. Однако в интернациональном программировании могут использоваться большие наборы символов, такие как Unicode, поэтому в некоторых реализациях могут использоваться 16-битные байты или даже 32-битные.

В настоящее время во многих системах используется минимальная гарантия: тип short имеет 16 битов, а тип long — 32 бита. В итоге остается еще несколько свободных вариантов для типа int. Он может иметь 16, 24 или 32 бита и при этом соответствовать стандарту. Тип int обычно имеет 16 битов (столько же, сколько и тип

short) в старых реализациях для IBM-совместимых ПК и 32 бита (столько же, сколько и тип long) для Windows 98, Windows NT, Windows XP, Macintosh OS X, VAX и многих других реализаций для миникомпьютеров. В некоторых реализациях можно выбирать способ обработки типа int. (Что использует ваша реализация? Следующий пример показывает, как можно узнать об ограничении для вашей системы, не обращаясь к справочному руководству.) Различия в разрядностях типов между реализациями могут привести к возникновению ошибок при переносе программы из одной среды в другую. Однако эту проблему совсем несложно свести к минимуму, о чем будет сказано далее в этой главе.

Следующие имена типов вы будете использовать для объявления переменных, точно так же, как вы могли бы использовать int:

```
short score;   // создает целочисленную переменную, имеющую тип short
int temperature; // создает целочисленную переменную, имеющую тип int
long position; // создает целочисленную переменную, имеющую тип long
```

В действительности, тип short имеет более короткий диапазон значений, чем тип short int, а тип long имеет более короткий диапазон значений, чем тип long int, однако вряд ли какой-нибудь программист использует формы с более широким диапазоном значений.

Три типа, int, short и long, являются типами со знаком — то есть, они могут разбивать свой диапазон почти пополам между положительными и отрицательными значениями. Например, диапазоном значений для 16-битного типа int является диапазон от –32768 до 32767.

Если вы хотите узнать, какой размер отведен для целых чисел в вашей системе, можете воспользоваться средствами C++. Во-первых, можно использовать операцию sizeof, которая возвращает размер типа или переменной в байтах. (*Операция* представляет собой действие, выполняемое над одним или несколькими элементами для получения значения. Например, операция сложения, обозначаемая знаком +, суммирует два числа.) Обратите внимание, что значение байта зависит от реализации, поэтому 2-байтное значение int может составлять 16 битов в одной системе и 32 бита в другой. Во-вторых, заголовочный файл climits (или заголовочный файл limits.h в старых реализациях) содержит информацию об ограничениях системы в отношении целого типа. В частности, в нем определены символические имена для представления различных ограничений. Например, в нем определено имя INT_MAX как наибольшее из возможных значений int и CHAR_BIT как количество битов в байте. В листинге 3.1 показан пример использования этих средств и пример *инициализации*, которая представляет собой использование оператора объявления для присваивания значения переменной.

Листинг 3.1. limits.cpp

```
// limits.cpp -- некоторые ограничения целых чисел
#include <iostream>
#include <climits> //использование заголовочного файла limits.h для старых систем
int main()
{
   using namespace std;
   int n_int = INT_MAX;// инициализация, или присваивание переменной n_int
                       // максимального значения int
   short n_short = SHRT_MAX; // символы, определенные в файле limits.h
   long n_long = LONG_MAX;
```

```
    // операция sizeof выдает размер типа или переменной
    cout << "int is " << sizeof (int) << " bytes." << endl;
    cout << "short is " << sizeof n_short << " bytes." << endl;
    cout << "long is " << sizeof n_long << " bytes." << endl << endl;
    cout << "Maximum values:" << endl;
    cout << "int: " << n_int << endl;
    cout << "short: " << n_short << endl;
    cout << "long: " << n_long << endl << endl;
    cout << "Minimum int value = " << INT_MIN << endl;
    cout << "Bits per byte = " << CHAR_BIT << endl;
    return 0;
}
```

Замечание по совместимости

В языке C++ заголовочный файл `climits` является версией заголовочного файла `climits.h` в стандарте ANSI для языка С. Некоторые самые ранние версии C++ вообще не используют заголовочные файлы. Если вы работаете именно с такой версией, то с этим примером экспериментировать не стоит..

Ниже показан результат выполнения программы из листинга 3.1 с использованием Microsoft Visual C++ 7.1:

```
int is 4 bytes.
short is 2 bytes.
long is 4 bytes.

Maximum values:
int: 2147483647
short: 32767
long: 2147483647

Minimum int value = -2147483648
Bits per byte = 8
```

Далее приведен результат выполнения программы во второй системе — Borland C++ 3.1 для DOS:

```
int is 2 bytes.
short is 2 bytes.
long is 4 bytes.

Maximum values:
int: 32767
short: 32767
long: 2147483647

Minimum int value = -32768
Bits per byte = 8
```

Замечания по программам

В следующих разделах мы рассмотрим наиболее важные особенности этой программы.

Операция sizeof и заголовочный файл climits

Результат операции sizeof сообщает о том, что тип int имеет 4 байта в базовой системе, в которой используются 8-битные байты. Операцию sizeof можно выполнить либо над именем типа, либо над именем переменной. Если эта операция будет выполнена над именем типа, например, над int, то это имя должно быть заключено в круглые скобки. Для имен переменных вроде n_short, скобки можно опускать:

```
cout << "int is " << sizeof (int) << " bytes.\n";
cout << "short is " << sizeof n_short << " bytes.\n";
```

Заголовочный файл climits определяет символические константы (см. врезку "Символические константы как средство препроцессора" далее в этой главе) для представления ограничений типов. Как уже было сказано, INT_MAX представляет наибольшее значение, которое может хранить тип int; для нашей системы DOS таким значением является 32767. Производитель компилятора предоставляет файл climits, в котором отражены значения, соответствующие данному компилятору. Например, файл climits для Windows XP, в которой используются 32-битные значения int, определяет, что INT_MAX равно 2 147 483 647. В табл. 3.1 приводятся символические константы, определенные в файле climits; некоторые из них относятся к типам, которые нам еще предстоит рассмотреть.

Таблица 3.1. Символические константы, определенные в файле climits

Символическая константа	Ее представление
CHAR_BIT	Количество битов в char
CHAR_MAX	Максимальное значение char
CHAR_MIN	Минимальное значение char
SCHAR_MAX	Максимальное значение signed char
SCHAR_MIN	Минимальное значение signed char
UCHAR_MAX	Максимальное значение unsigned char
SHRT_MAX	Максимальное значение short
SHRT_MIN	Минимальное значение short
USHRT_MAX	Максимальное значение unsigned short
INT_MAX	Максимальное значение int
INT_MIN	Минимальное значение int
UINT_MAX	Максимальное значение unsigned int
LONG_MAX	Максимальное значение long
LONG_MIN	Минимальное значение long
ULONG_MAX	Максимальное значение unsigned long

Инициализация

Под *инициализацией* подразумевается комбинирование объявления переменной и присваивания ей начального значения. Например, оператор

```
int n_int = INT_MAX;
```

объявляет переменную n_int и присваивает ей наибольшее из возможных значений int. Для инициализации значений можно использовать и обычные константы.

Можно инициализировать переменную другой переменной, при условии, что эта другая переменная уже была объявлена. Можно даже инициализировать переменную выражением, при условии, что все значения в выражении будут известны на тот момент, когда программа достигнет объявления:

```
int uncles = 5;         // инициализация переменной uncles:
                        // присваивание значения 5
int aunts = uncles;     // инициализация переменной aunts:
                        // присваивание значения 5
int chairs = aunts + uncles + 4; // инициализация переменной chairs:
                                 // присваивание значения 14
```

Если объявление переменной uncles переместить в конец этого списка операторов, то две других инициализации окажутся недействительными, поскольку в тот момент, когда программа попытается инициализировать остальные переменные, значение переменной uncles не будет известно.

Синтаксис инициализации, показанный в предыдущем примере, заимствован из языка С; в языке С++ используется еще один синтаксис инициализации, не совместимый с С:

```
int owls = 101;  // традиционная инициализация в языке С
int wrens(432);  // альтернативный синтаксис C++; переменной wrens
                 // присваивается значение 432
```

На память!

Если вы не инициализируете переменную, определяемую внутри функции, то значение этой переменной будет *неопределенным*. Это означает, что еще до создания переменной в этой ячейке памяти уже будет храниться значение.

Если вы знаете, каким должно быть исходное значение переменной, инициализируйте ее. Правда, при разделении объявления переменной и присваивания значения может возникнуть кратковременная неопределенность:

```
short year;      // Что бы это могло быть?
year = 1492;     // А вот что.
```

Однако если вы инициализируете переменную во время ее объявления, это избавит вас от необходимости присваивать ей значение позже.

Символические константы как средство препроцессора

В файле climits содержатся строки наподобие следующей:

```
#define INT_MAX 32767
```

Мы уже говорили, что в процессе компиляции исходный код передается сначала препроцессору. Здесь #define, как и #include, является директивой препроцессора. Эта директива сообщает препроцессору следующее: найти в программе экземпляры символической константы INT_MAX и, если таковые существуют, подставить вместо них значение 32767. Таким образом, директива #define работает подобно команде "найти и заменить" текстового редактора или текстового процессора. После произведенных замен происходит компиляция измененной программы. Препроцессор производит поиск независимых лексем (отдельных слов) и пропускает вложенные слова. То есть, препроцессор не заменяет PINT_MAXIM на P32767IM. Кроме этого, директиву #define можно использовать, чтобы определить свои собственные символические константы (используя ключевое слово const, которое мы рассмотрим чуть позже), поэтому применять директиву #define вы будете редко. Однако она будет встречаться в некоторых заголовочных файлах, особенно в тех, которые предназначены для использования в С и С++.

Типы без знаков

Каждый из трех рассмотренных нами целочисленных типов включает беззнаковые типы, которые не могут хранить отрицательные значения. Преимущество этих типов очевидно, поскольку за их счет можно увеличить самое большое значение, которое способна хранить переменная. Например, если тип short представляет диапазон значений от –32768 до 32767, то беззнаковый вариант этого типа будет представлять диапазон от 0 до 65535. Естественно, типы без знаков следует использовать только для тех величин, которые никогда не будут отрицательными, например, подсчет населения, количество нутовых зерен или количество участников манифестации. Создать варианты базовых целых типов без знака можно с помощью ключевого слова unsigned:

```
unsigned short change;  // тип short без знака
unsigned int rovert;    // тип int без знака
unsigned quarterback;   // тоже тип int без знака
unsigned long gone;     // тип long без знака
```

Обратите внимание, что сам по себе unsigned короче, чем unsigned short.

В листинге 3.2 показан пример использования типов без знака. Вы увидите, что может произойти, если программа попытается превысить пределы, отведенные для целочисленных типов. В этом листинге можно увидеть еще один пример использования оператора #define.

Листинг 3.2. exceed.cpp

```
// exceed.cpp -- превышение пределов некоторых целочисленных типов
#include <iostream>
#define ZERO 0      // создает символ ZERO для значения 0
#include <climits>  // определяет INT_MAX как наибольшее значение int
int main()
{
   using namespace std;
   short sam = SHRT_MAX;//инициализирует переменную по максимальному значению
   unsigned short sue = sam; // инициализация действительна, если
                             // переменная sam уже была определена
   cout << "Sam has " << sam << " dollars and Sue has " << sue;
   cout << " dollars deposited." << endl
        << "Add $1 to each account." << endl << "Now ";
   sam = sam + 1;
   sue = sue + 1;
   cout << "Sam has " << sam << " dollars and Sue has " << sue;
   cout << " dollars deposited.\nPoor Sam!" << endl;
   sam = ZERO;
   sue = ZERO;
   cout << "Sam has " << sam << " dollars and Sue has " << sue;
   cout << " dollars deposited." << endl;
   cout << "Take $1 from each account." << endl << "Now ";
   sam = sam - 1;
   sue = sue - 1;
   cout << "Sam has " << sam << " dollars and Sue has " << sue;
   cout << " dollars deposited." << endl << "Lucky Sue!" << endl;
   return 0;
}
```

Замечание по совместимости

В листинге 3.2, как и в листинге 3.1, используется файл `climits`; скорее всего для старых компиляторов нужно использовать файл `climits.h`, а некоторые наиболее старые компиляторы вообще не работают ни с одним из этих файлов.

Ниже показан результат выполнения программы из листинга 3.2:

```
Sam has 32767 dollars and Sue has 32767 dollars deposited.
Add $1 to each account.
Now Sam has -32768 dollars and Sue has 32768 dollars deposited.
Poor Sam!
Sam has 0 dollars and Sue has 0 dollars deposited.
Take $1 from each account.
Now Sam has -1 dollars and Sue has 65535 dollars deposited.
Lucky Sue!
```

В этой программе переменной `sam`, имеющей тип `short`, и переменной `sue`, имеющей тип `unsigned short`, присваивается максимальное значение `short`, которое в нашей системе составляет 32767. Затем к значению каждой переменной прибавляется единица. С переменной `sue` все проходит гладко, поскольку новое значение меньше максимального значения для целочисленного беззнакового типа. А переменная `sam` вместо 32767 получает значение –32768! Аналогично и при вычитании единицы от нуля для переменной `sam` все пройдет гладко, а вот беззнаковая переменная `sue` получит значение 65535. Как видите, эти целые числа ведут себя почти так же, как и счетчик пройденного пути. Если предельное значение будет превышено, то отсчет начнется с противоположного конца диапазона (рис. 3.1).

Рис. 3.1. Типичное поведение при переполнении для целочисленных типов

Язык C++ гарантирует, что беззнаковые типы ведут себя именно таким образом. Однако язык C++ не гарантирует, что в целочисленных типах со знаком могут быть превышены пределы (переполнение и потеря значимости) без сообщения об ошибке; с другой стороны, это самое распространенное поведение в современных реализациях.

По ту сторону типа `long`

В стандарте C99 добавлено два новых типа, которые войдут в следующую редакцию стандарта C++. В действительности, многие компиляторы C++ уже поддерживают их. Это типы `long long` и `unsigned long long`. Каждый из них гарантированно имеет 64 бита и имеет размер как минимум такой же, как у типов `long` и `unsigned long`.

Выбор целочисленного типа

Какой из целочисленных типов следует использовать, учитывая все богатство выбора в языке C++? В общем случае, тип `int` имеет наиболее "естественный" размер целого числа для искомого компьютера. Под *естественным размером* подразумевается целочисленная форма, которую компьютер может обработать наиболее эффективным образом. Если у вас нет веской причины для выбора другого типа, то лучше всего использовать тип `int`.

Теперь давайте узнаем, в каких случаях будет выгоднее использовать другой тип. Если переменная представляет какую-то величину, которая никогда не будет отрицательной, например, количество слов в текстовом документе, то в этом случае можно использовать тип без знака; таким образом, переменная сможет представлять более высокие значения.

Если вы знаете, что переменная будет содержать целочисленные значения, слишком большие для 16-битного целого типа, то используйте тип `long`. Это справедливо даже в том случае, если в вашей системе тип `int` будет иметь 32 бита. В результате, если вы перенесете свою программу в систему, которая работает с 16-битными значениями `int`, то в работе программы не произойдет сбой (рис. 3.2).

Использование типа `short` позволит сократить потребление памяти, если `short` меньше чем `int`. Чаще всего это бывает важным только при работе с большим массивом целых чисел. (*Массив* представляет собой структуру данных, которая хранит несколько однотипных значений последовательно в памяти компьютера.) Если перед вами действительно встает вопрос экономии памяти, то вместо типа `int` предпочтительнее использовать тип `short`, даже если оба имеют одинаковый размер. Предположим, например, что вы переносите свою программу из ПК под управлением DOS с использованием 16-битного типа `int` в систему Windows XP. В результате этого будет удвоен объем памяти, необходимый для хранения массива чисел `int`, не изменяя при этом требований к массиву `short`. Всегда помните о том, что пользу приносит каждый сэкономленный бит информации.

Если вам необходим только один байт информации, вы можете применять тип `char`. Вскоре мы поговорим об этом типе.

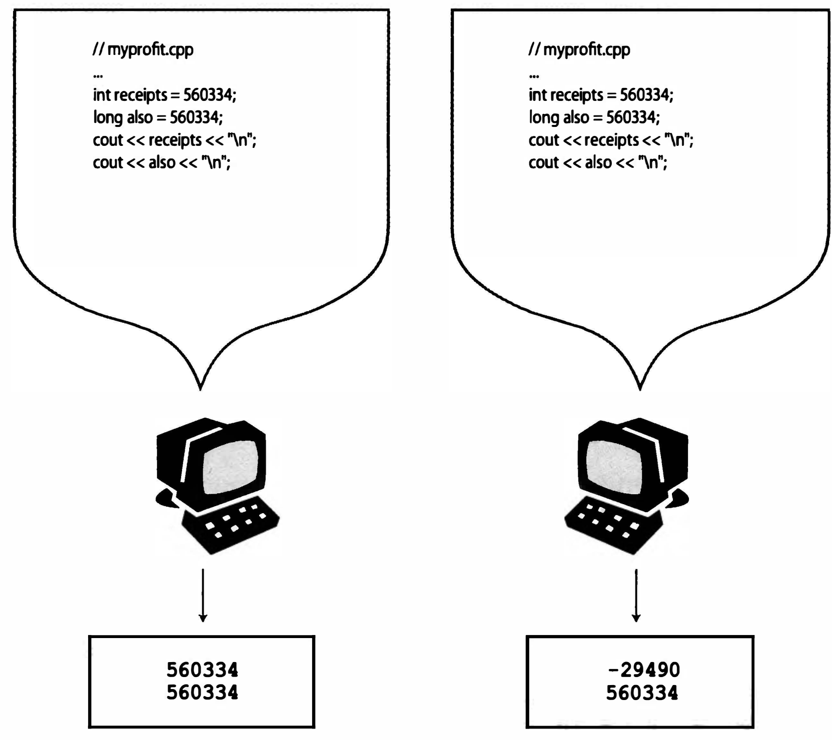


Рис. 3.2. Чтобы обеспечить переносимость программы, используйте для больших целых чисел тип `long`

Целочисленные константы

Целочисленная константа представляет собой число, записываемое явным образом, например, 212 или 1776. В языке C++, как и в C, целые числа можно записывать в трех различных системах счисления: с основанием 10 (наиболее распространенная форма), с основанием 8 (старая запись в системах семейства Unix) и с основанием 16 (излюбленная форма компьютерных хакеров). Описание этих систем можно найти в приложении А, а сейчас мы остановимся на рассмотрении представлений в C++. В языке C++ для обозначения основания постоянного числа используется первая или две первые цифры. Если первая цифра находится в диапазоне 1–9, то это число десятичное (с основанием 10); поэтому основанием числа 93 является 10. Если первой цифрой является ноль, а вторая цифра находится в диапазоне 1–7, то это число восьмеричное (основание 8); таким образом, 042 — это восьмеричное значение, соответствующее десятичному числу 32. Если первыми двумя символами являются 0x или 0X, то это шестнадцатеричное значение (основание 16); поэтому 0x42 — это шестнадцатеричное значение, соответствующее десятичному числу 66. В представлении десятичных значений символы a–f и A–F представляют шестнадцатеричные цифры, соответствующие значениям 10–15. 0xF — это 15, а 0xA5 — это 165 (10 раз по шестнадцать плюс 5). В листинге 3.3 показан пример этих представлений.

Листинг 3.3. hexoct1.cpp

```
// hexoct1.cpp -- показывает шестнадцатеричные и восьмеричные константы
#include <iostream>
int main()
{
   using namespace std;
   int chest = 42;      // десятичная целочисленная константа
   int waist = 0x42;    // шестнадцатеричная целочисленная константа
   int inseam = 042;    // восьмеричная целочисленная константа
   cout << "Monsieur cuts a striking figure!\n";
   cout << "chest = " << chest << "\n";
   cout << "waist = " << waist << "\n";
   cout << "inseam = " << inseam << "\n";
   return 0;
}
```

По умолчанию cout отображает целые числа в десятичной форме, независимо от того, как они были записаны в программе. Подтверждением этого является результат выполнения программы из листинга 3.3:

```
Monsieur cuts a striking figure!
chest = 42 (42 in decimal)
waist = 66 (0x42 in hex)
inseam = 34 (042 in octal)
```

Имейте в виду, что эти обозначения используются просто для удобства. Например, если вы прочитаете, что сегментом видеопамяти CGA является B000 в шестнадцатеричном представлении, то вам не придется переводить его в значение 45056 десятичного формата, прежде чем применять его в своей программе. Наоборот, вы можете просто использовать 0xB000. Вне зависимости от того, как вы запишете число десять — как 10, 012 или 0xA — в памяти компьютера оно будет храниться как двоичное число (с основанием 2).

Между прочим, если вам нужно будет отобразить значение в шестнадцатеричной или восьмеричной форме, то для этого можно воспользоваться возможностями объекта cout. Вспомните, что заголовочный файл iostream предлагает манипулятор endl, который сигнализирует объекту cout о начале новой строки. Кроме этого манипулятора существуют манипуляторы dec, hex и oct, которые сигнализируют объекту cout о форматах отображения целых чисел: десятичном, шестнадцатеричном и восьмеричном, соответственно. В листинге 3.4 манипуляторы hex и oct применяются для отображения десятичного значения 42 в трех формах. (Десятичная форма используется по умолчанию, и каждая форма записи остается в силе до тех пор, пока вы не измените ее.)

Листинг 3.4. hexoct2.cpp

```
//hexoct2.cpp--отображает значения в шестнадцатеричном и восьмеричном формате
#include <iostream>
using namespace std;
int main()
{
   using namespace std;
   int chest = 42;
```

```
    int waist = 42;
    int inseam = 42;
    cout << "Monsieur cuts a striking figure!" << endl;
    cout << "chest = " << chest << " (decimal)" << endl;
    cout << hex;  // манипулятор для изменения основания системы счисления
    cout << "waist = " << waist << " hexadecimal" << endl;
    cout << oct;  // манипулятор для изменения основания системы счисления
    cout << "inseam = " << inseam << " (octal)" << endl;
    return 0;
}
```

Далее показан результат выполнения этой программы:

```
Monsieur cuts a striking figure!
chest = 42 (decimal)
waist = 2a hexadecimal
inseam = 52 (octal)
```

Обратите внимание, что код, подобный

```
cout << hex;
```

ничего не отображает на экране монитора. Наоборот, он изменяет способ отображения целых чисел. Поэтому манипулятор hex на самом деле является сообщением для cout, на основании которого определяется дальнейшее поведение объекта cout. Обратите внимание также на то, что поскольку идентификатор hex является частью пространства имен std, используемого в этой программе, то программа не может применять hex в качестве имени переменной. Однако если опустить директиву using, и вместо нее использовать std::cout, std::endl, std::hex и std::oct, тогда hex можно будет использовать для именования переменных.

Как компилятор C++ определяет тип константы

О типе каждой отдельной переменной целочисленного типа компилятор C++ узнает из объявлений в программах. А как быть с константами? Предположим, что в своей программе вы представляете число посредством константы:

```
cout << "Year = " << 1492 << "\n";
```

В каком формате программа хранит значение 1492: в формате int, long или в формате другого целочисленного типа? Ответ таков: C++ хранит целочисленные константы в формате int, если только нет причины использовать другой тип. Таких причин может быть две: вы используете специальный суффикс, чтобы указать конкретный тип; данное значение слишком большое, чтобы его можно было хранить в виде int.

Во-первых, обратимся к суффиксам. Они представляют собой буквы, которые помещаются в конце числовой константы для обозначения типа константы. Суффикс l или L целого числа означает, что это целое число является константой long, суффикс u или U означает константу unsigned int, а суффикс ul (в любой комбинации символов и в любом регистре) означает константу unsigned long. (Поскольку начертание буквы l в нижнем регистре очень похоже на начертание цифры 1, рекомендуется использовать верхний регистр.) Например, в системе, использующей 16-битный

тип int и 32-битный тип long, число 22022 хранится в 16 битах как int, а число 22022L хранится в 32 битах как long. Точно также 22022LU и 22022UL имеют тип unsigned long.

Теперь давайте оценим размер. В языке C++ правила для десятичных целых чисел несколько отличаются от шестнадцатеричных и восьмеричных целых чисел. (Здесь под десятичными подразумеваются числа с основанием 10, а шестнадцатеричные — с основанием 16; термин *десятичный* не обязательно подразумевает десятичную точку.) Десятичное целое число без суффикса представляется наименьшим из следующих типов, которые могут его хранить: int, long или unsigned long. В компьютерной системе, использующей 16-битный тип int и 32-битный тип long, число 20000 представляется как int, число 40000 представляется как long, а 3000000000 представляется как unsigned long. Шестнадцатеричное или восьмеричное целое число без суффикса представляется наименьшим из следующих типов, которые могут его хранить: int, unsigned int, long или unsigned long. В той же компьютерной системе, в которой число 40000 представляется как long, его шестнадцатеричный эквивалент 0x9C40 представляется как unsigned int. Это объясняется тем, что шестнадцатеричная форма часто используется для выражения адресов памяти, которые сами по себе не могут быть отрицательными. Поэтому тип unsigned int является более подходящим, чем long для 16-битных адресов.

Тип char: символы и короткие целые числа

Пришло время рассмотреть последний целочисленный тип: char. Исходя из названия (сокращ. от *character* — символ), он предназначен для хранения символов, таких как буквы и цифры. Хранение чисел в памяти компьютера не представляет сложности, а вот хранение букв связано с рядом проблем. В языках программирования найден простой выход из положения: для цифр и букв используются коды. Поэтому тип char является еще одним целочисленным типом. Для целевой компьютерной системы он гарантирует представление всего диапазона базовых символов — буквы, цифры, знаки препинания и т.п. На практике в большинстве систем поддерживается менее 256 видов символов, поэтому один байт может представить весь диапазон. Поэтому, несмотря на то, что тип char чаще всего используется для хранения символов, вы можете его применять как целочисленный тип, который обычно меньше, чем short.

Самым распространенным набором символов в США является ASCII, который описан в приложении В. Каждый символ в этом наборе представлен числовым кодом (кодом ASCII). Например, символу A соответствует код 65, символу M соответствует код 77 и так далее. Для удобства в примерах из этой книги принята схема кодирования ASCII. Однако реализация C++ использует такой код, который используется в операционной системе, например, EBCDIC (расширенный двоично-десятичный код) в компьютерах IBM. Ни ASCII, ни EBCDIC не в состоянии в полной мере представить все интернациональные символы, поэтому в языке C++ поддерживается расширенный символьный тип, который способен хранить более широкий диапазон значений (которые, например, используются в международном наборе символов Unicode). Этот тип имеет обозначение wchar_t, и мы еще вернемся к нему в этой главе.

В листинге 3.5 показан пример использования типа char.

Листинг 3.5. chartype.cpp

```
// chartype.cpp -- тип char
#include <iostream>
int main( )
{
   using namespace std;
   char ch; // объявление переменной char
   cout << "Enter a character: " << endl;
   cin >> ch;
   cout << "Holla! ";
   cout << "Thank you for the " << ch << " character." << endl;
   return 0;
}
```

Ниже показан пример выполнения этой программы:

```
Enter a character:
M
Holla! Thank you for the M character.
```

Интересным является то, что вы вводите символ M, а не соответствующий код символа — 77, а программа выводит на экран символ M, а не код 77. Кроме этого, мы уже говорили, что 77 — это значение, которое хранится в переменной ch. Это объясняется не свойствами типа char, а работой объектов cin и cout. Они выполняют все необходимые преобразования по вашей команде. Во входных данных объект cin преобразует нажатие клавиши <M> в значение 77. На выходе объект cout преобразует значение 77 для отображения символа M; поведение объектов cin и cout зависит от типа переменной. Если переменной, имеющей тип int, присвоить значение 77, то объект cout отобразит 77 (другими словами, объект cout отображает два символа 7). Эта схема продемонстрирована в листинге 3.6. В нем также показан способ написания символьной константы в языке C++: символ заключается в одинарные кавычки, как в 'M'. (Заметьте, что в этом примере для строки не используются двойные кавычки. Объект cout может обрабатывать как одинарные, так и двойные кавычки, однако, как будет сказано в главе 4, между ними есть существенное различие.) Также в программе показан пример работы функции cout.put() объекта cout, которая отображает одиночный символ.

Листинг 3.6. morechar.cpp

```
// morechar.cpp -- сравнение типов char и int
#include <iostream>
int main()
{
   using namespace std;
   char ch = 'M';       // присваивает переменной ch код ASCII символа M
   int i = ch;          // сохраняет этот же код в int
   cout << "The ASCII code for " << ch << " is " << i << endl;
   cout << "Add one to the character code:" << endl;
   ch = ch + 1;         // изменяет код символа в переменной ch
   i = ch;              // сохраняет код нового символа в переменной i
   cout << "The ASCII code for " << ch << " is " << i << endl;
```

```
    // использование члена функции cout.put() для отображения символа
    cout << "Displaying char ch using cout.put(ch): ";
    cout.put(ch);
    // использование cout.put() для отображения символьной константы
    cout.put('!');
    cout << endl << "Done" << endl;
    return 0;
}
```

Ниже показан результат выполнения этой программы:

```
The ASCII code for M is 77
Add one to the character code:
The ASCII code for N is 78
Displaying char ch using cout.put(ch): N!
Done
```

Замечания по программе

В программе из листинга 3.6 обозначение `'M'` представляет числовой код символа M, поэтому при инициализации переменной ch, имеющей тип char, по значению `'M'` ей присваивается значение 77. Затем программа присваивает такое же значение переменной i, которая имеет тип int, поэтому обе переменные имеют значение 77. Далее объект cout отображает переменную ch как M, и переменную i как 77. Как уже было сказано ранее, в зависимости от типа значения объект cout выбирает способ отображения значения, что является еще одним доказательством интеллектуальных возможностей объектов ввода-вывода.

Поскольку переменная ch на самом деле является целочисленной, вы можете выполнять над ней целочисленные операции. Поэтому к ее значению была прибавлена единица, в результате чего было получено новое значение — 78. Это новое значение затем было присвоено переменной i. (Точно так же к значению переменной i можно просто прибавить единицу.) И в этом случае объект cout отображает символьный вариант (char) этого значения в виде буквы и целочисленный вариант (int) в виде цифры.

То, что в C++ символы могут быть представлены в виде целых чисел, является очень удобной возможностью, поскольку она облегчает работу с символьными значениями — ведь вам не нужно использовать неудобные функции для преобразования символов в ASCII и обратно.

Функция `cout.put()` используется в программе для отображения как переменной ch, так и символьной константы.

Функция-член: `cout.put()`

Что же такое функция `cout.put()`, и почему в ее имени присутствует точка? Функция `cout.put()` является нашим первым примером важного понятия ООП — это *функция-член*. Если вы помните, класс определяет способ представления данных и действия, которые можно над ними выполнять. Функция-член принадлежит к классу и описывает способ манипулирования данными класса. Класс ostream, например, имеет функцию-член `put()`, которая предназначена для вывода символов. Функцию-член можно использовать только для определенного объекта из этого класса, как в

данном случае для объекта cout. Чтобы использовать функцию-член класса для такого объекта как cout, между именем объекта (cout) и именем функции (put()) ставится точка. Точка в данном случае называется *операцией принадлежности* или *операцией членства*. В обозначении cout.put() подразумевается использование функции-члена класса put() и объекта класса cout. Более детально об этом мы поговорим в главе 10. А пока что единственными классами, с которыми мы имеем дело, являются istream и ostream, поэтому вы можете экспериментировать с их функциями-членами, чтобы получше ознакомиться с их работой.

Функция-член cout.put() предлагает альтернативный способ использования операции << для отображения символа. В общем случае, необходимость в этой функции может вызвать у вас удивление. Если обратиться к истории, то мы увидим, что до выхода версии 2.0 языка C++ объект cout мог использовать символьные *переменные* как символы, а символьные *константы* (например, 'M' и 'N') отображать как цифры. Дело в том, что в ранних версиях C++, как и в C, символьные константы хранились в целочисленной форме int. То есть, код 77 для 'M' мог быть сохранен в 16-битной и 32-битной форме. Тем временем переменные, имеющие тип char, обычно занимали 8 битов. Оператор, такой как

```
char c = 'M';
```

копировал 8 битов (важные 8 битов) из константы 'M' в переменную c. К сожалению, это означает, что для объекта cout константа 'M' и переменная c выглядят поразному, даже если каждый из них содержит одно и то же значение. Поэтому оператор вроде

```
cout << '$';
```

печатал код ASCII для символа $, а не символ $. А функция

```
cout.put('$');
```

печатала требуемый символ. Теперь же, после выхода версии 2.0, односимвольные константы сохраняются в виде char, а не int, поэтому объект cout правильно обрабатывает символьные константы.

Объект cin считывает символы из ввода двумя способами. Эти способы построены на использовании циклов, поэтому мы будем говорить о них в главе 5.

Константы char

Символьные константы в языке C++ можно записать несколькими способами. Обычные символы, такие как буквы, знаки препинания и цифры, проще всего заключать в одиночные кавычки. Такая форма записи будет символизировать числовой код символа. Например, в системе ASCII установлены следующие соответствия:

'A'	соответствует 65, коду ASCII для символа A
'a'	соответствует 97, коду ASCII для символа a
'5'	соответствует 53, коду ASCII для цифры 5
	соответствует 32, коду ASCII для символа пробела
'!'	соответствует 33, коду ASCII для символа восклицательного знака

Применять такое обозначение лучше, чем числовые коды, поскольку его форма понятна и не требует знания конкретного кода. Если в системе используется схема EBCDIC, то код 65 не будет соответствовать букве A, в то время как 'A' будет представлять символ.

Некоторые символы невозможно ввести в программу прямо с клавиатуры. Например, вы не сможете сделать символ новой строки частью строки, если нажмете клавишу <Enter>; наоборот, редактор программы будет интерпретировать нажатие клавиши как запрос на начало новой строки в вашем файле исходного кода. Использованию других символов препятствует особое значение, придаваемое им в языке C++. Например, символы двойной кавычки ограничивают строки, поэтому в середине строки их вводить нельзя. Для некоторых таких символов в языке C++ используются специальные обозначения, называемые *управляющими последовательностями* (табл. 3.2). Например, последовательность \a представляет символ предупреждения, по которому динамик издает сигнал или звонок телефона. Последовательность \n представляет новую строку. А последовательность \" представляет двойную кавычку как обычный символ, а не разделитель строки. Эти обозначения можно применять в строках или символьных константах, как показано ниже:

```
char alarm = '\a';
cout << alarm << "Don't do that again!\a\n";
cout << "Ben \"Buggsie\" Hacker\nwas here!\n";
```

Таблица 3.2. Коды управляющих последовательностей в C++

Название символа	Символ ASCII	Код C++	Десятичный код ASCII	Шестнадцатеричный код ASCII
Новая строка	NL (LF)	\n	10	0xA
Горизонтальная табуляция	HT	\t	9	0x9
Вертикальная табуляция	VT	\v	11	0xB
Возврат на одну позицию (обратное перемещение)	BS	\b	8	0x8
Возврат каретки	CR	\r	13	0xD
Предупреждение	BEL	\a	7	0x7
Обратная косая черта	\	\\	92	0x5C
Знак вопроса	?	\?	63	0x3F
Одинарная кавычка	'	\'	39	0x27
Двойная кавычка	"	\"	34	0x22

Последняя строка выдает такой результат:

```
Ben "Buggsie" Hacker
was here!
```

Обратите внимание, что управляющую последовательность, например, \n, вы интерпретируете как обычный символ, например Q. То есть вы заключаете ее в одинарные кавычки, чтобы создать символьную константу и опускаете кавычки, когда включаете ее как часть строки.

Символ новой строки является альтернативным вариантом манипулятора endl для помещения новых строк в вывод. Его можно использовать в обозначении символьной константы ('\n') или в качестве символа в строке ("\n"). Каждый из представленных далее вариантов перемещает курсор на экране на начало следующей строки:

```
cout << endl;          // использование манипулятора endl
cout << '\n';          // использование символьной константы
cout << "\n";          // использование строки
```

Символ новой строки можно включать в длинную строку текста; чаще всего этот вариант является более удобным, чем использование манипулятора endl. Например, следующие два оператора cout дают одинаковый результат:

```
cout << endl << endl << "What next?" << endl << "Enter a number:" << endl;
cout << "\n\nWhat next?\nEnter a number:\n";
```

Для отображения числа легче ввести манипулятор endl, чем "\n" или '\n', а для отображения строки проще использовать символ новой строки:

```
cout << x << endl;     // проще, чем cout << x << "\n";
cout << "Dr. X.\n";    // проще, чем cout << "Dr. X." << endl;
```

Наконец, можно использовать управляющие последовательности на основе восьмеричных или шестнадцатеричных кодов символа. Например, комбинация клавиш <Ctrl+Z> имеет ASCII-код 26, соответствующий 032 в восьмеричной системе и 0x1a в шестнадцатеричной. Этот символ можно отобразить посредством одной из управляющих последовательностей: \032 или \x1a. Из них можно составлять символьные константы, заключая их в одинарные кавычки, например '\032', и можно использовать в виде части строки, например, "hi\x1a there".

Совет

Если у вас есть возможность выбора между числовыми и символьными управляющими последовательностями, например, между \0x8 и \b, то лучше всего использовать символьный код. Числовое представление привязано к определенному коду, например ASCII, а символьное представление работает со всеми кодами и более удобно для чтения.

В листинге 3.7 приводится пример некоторых управляющих последовательностей. В нем используется символ предупреждения, чтобы привлечь ваше внимание, символ новой строки для перемещения курсора, а также символ возврата на одну позицию (забой) для возврата курсора на одну позицию влево. (Гудини (Houdini) однажды нарисовал картину с изображением реки Гудзон с помощью одних управляющих последовательностей; вне всяких сомнений — это был великий мастер своего дела.)

Листинг 3.7. bondini.cpp

```
// bondini.cpp -- использование управляющих последовательностей
#include <iostream>
int main()
{
    using namespace std;
    cout << "\aOperation \"HyperHype\" is now activated!\n";
    cout << "Enter your agent code:________\b\b\b\b\b\b\b\b";
    long code;
```

```
    cin >> code;
    cout << "\aYou entered " << code << "...\n";
    cout << "\aCode verified! Proceed with Plan Z3!\n";
    return 0;
}
```

Замечание по совместимости

Некоторые системы C++, основанные на компиляторах C, разработанных еще до выхода стандарта ANSI, не распознают последовательность \a. В системах, в которых используется кодировка символов ASCII, эту последовательность можно заменить на \007. Некоторые системы могут вести себя по-разному, отображая \b в виде небольшого треугольника, вместо того чтобы возвратить курсор, или даже стирая предыдущую позицию, или игнорируя \a.

Если выполнить программу из листинга 3.7, то на экране появится следующий текст:

```
Operation "HyperHype" is now activated!
Enter your agent code:________
```

После отображения символов подчеркивания программа использует символ возврата для возврата курсора к первому символу подчеркивания. После этого вы можете ввести секретный код, и выполнение программы продолжится дальше. Вот полный результат ее выполнения:

```
Operation "HyperHype" is now activated!
Enter your agent code:42007007
You entered 42007007...
Code verified! Proceed with Plan Z3!
```

Универсальные символьные имена

В реализациях C++ поддерживается основной базовый набор символов — то есть, символов, которые вы можете использовать для написания исходного кода. В нем представлены буквы (в верхнем и нижнем регистрах) и цифры стандартной клавиатуры США, символы, такие как { и =, используемые в языке C, и другие разнообразные символы, такие как символы новой строки и пробела. Существует также и базовый исполняемый набор символов (то есть символов, которые могут быть представлены при выполнении программы), добавляющий несколько дополнительных символов, например, символ возврата курсора и символ предупреждения. Стандарт C++ разрешает также реализацию для работы с расширенными исходными наборами символов и расширенными исполняемыми наборами символов. Более того, те дополнительные символы, которые квалифицируются как буквы, могут использоваться в качестве имени идентификатора. Так, в немецкой (German) реализации допускается использование умляутов, а во французской (French) — гласных с ударением. В языке C++ имеется механизм представления таких интернациональных символов, которые не зависят от конкретной клавиатуры: использование *универсальных имен символов.*

Универсальные имена символов подобны управляющим последовательностям. Универсальное имя символа начинается с последовательности \u или \U. За последовательностью \u следуют 8 шестнадцатеричных цифр, а за последовательностью \U — 16 шестнадцатеричных цифр. Эти цифры представляют код ISO 10646 для символа. (ISO 10646 является международным стандартом, разработка которого еще не

закончена; он предлагает числовые коды для широкого диапазона символов. См. врезку "Unicode и ISO 10646" далее в этой главе.)

Если в ваших реализациях поддерживается расширенный набор символов, то универсальные имена символов можно применять в идентификаторах (например, в символьных константах) и в строках. Взгляните на следующий фрагмент кода:

```
int k\u00F6rper;
cout << "Let them eat g\u00E2teau.\n";
```

Кодом ISO 10646 для символа ö является 00F6, а для символа â — 00E2. Поэтому в этом фрагменте переменной будет присвоено имя körper и отображено следующее:

```
Let them eat gâteau.
```

Если ваша система не поддерживает ISO 10646, она может отобразить какой-нибудь другой символ вместо â или, возможно, слово gu00E2teau.

Unicode и ISO 10646

Набор символов Unicode предлагает решение для представления различных наборов символов через стандартные числовые коды букв и символов, сгруппированных по типам. Например, кодировка ASCII включается как одна из подборок кодов Unicode, поэтому буквы американского латинского алфавита (вроде `A` и `Z`) имеют одинаковое представление в обеих системах. Unicode включает также и другие символы латинского алфавита, которые употребляются в европейских языках, буквы из других алфавитов, включая греческий, кириллицу, иврит, арабский, тайский и бенгальский, а также иероглифы (китайская и японская системы). В настоящий момент Unicode охватывает более 96 000 символов и 49 рукописных шрифтов, и в настоящий момент работа над этим кодом продолжается. Если вы желаете узнать больше, рекомендуется посетить Web-сайт `www.unicode.org`.

В Международной организации по стандартизации (ISO) была сформирована рабочая группа по разработке стандарта ISO 10646, который также является стандартом для кодировки многоязыковых текстов. Группа ISO 10646 и группа Unicode ведут совместную работу с 1991 года в целях согласования этих стандартов.

`signed char` и `unsigned char`

В отличие от `int`, тип `char` по умолчанию не имеет знака. Кроме этого, он по умолчанию не является также и беззнаковым типом. Выбор необходимого варианта оставлен за реализацией C++, что позволяет разработчикам компиляторов подбирать наиболее подходящий тип для аппаратных средств. Если для вас крайне важно, чтобы тип `char` имел определенное поведение, вы можете использовать `signed char` и `unsigned char` явным образом как отдельный тип:

```
char fodo;            // может иметь знак, может быть без знака
unsigned char bar;    // явное указание беззнакового типа
signed char snark;    // явное указание типа со знаком
```

Эти отличия будут особенно важными, если тип `char` использовать в качестве числового типа. Тип `unsigned char` обычно представляет диапазон значений от 0 до 255, а `signed char` обычно представляет диапазон значений от –128 до 127. Предположим, что вы хотите использовать переменную `char` для хранения значений больше 200. В одних системах этот вариант будет работать, а в других — нет. Однако

если для этого случая использовать тип unsigned char, тогда все будет в порядке. С другой стороны, если переменную char использовать для хранения стандартного символа ASCII, то не будет никакой разницы, может ли char представлять отрицательные значения, поэтому вы можете просто применять char.

На случай, если требуется больше: wchar_t

Иногда программа должна обрабатывать наборы символов, которые не вписываются в 8 битов (пример — система японских иероглифических писем). На этот случай в языке C++ имеется пара способов. Во-первых, если большой набор символов является базовым набором символов для реализации, то производитель компилятора может определить char как 16-битовый тип, или даже больше. Во-вторых, реализация может поддерживать как малый базовый набор символов, так и расширенный набор. Традиционный 8-битовый тип char может представлять базовый набор символов, а другой тип, называемый wchar_t (от *wide character type* — расширенный тип символов), — расширенный набор символов. Тип wchar_t является целочисленным типом, имеющим достаточно памяти для представления самого большого расширенного набора символов в системе. Этот тип имеет такой же размер и знак, как и один из остальных целочисленных типов, называемый *основным* типом. Выбор основного типа зависит от реализации, поэтому в одной системе это может быть unsigned short, а в другой — int.

Объекты cin и cout рассматривают ввод и вывод как потоки символов chars, поэтому для работы с типом wchar_t они не подходят. Самая последняя версия заголовочного файла iostream предлагает аналогичные им объекты wcin и wcout, предназначенные для обработки потоков wchar_t. Кроме этого, вы можете указать символьную константу в расширенном алфавите или строку, ставя перед ней L. В следующем фрагменте кода переменной bob присваивается версия wchar_t буквы P и отображается версия wchar_t слова tall:

```
wchar_t bob = L'P';       //символьная константа в расширенном алфавите
wcout << L"tall" << endl; //вывод строки в расширенном алфавите
```

В системе с 2-байтным типом wchar_t этот код хранит каждый символ в 2-байтном элементе памяти. В этой книге не используется тип расширенного алфавита, однако вы должны помнить о его существовании, особенно если вам придется работать в интернациональной команде программистов или использовать Unicode или ISO 10646.

Тип bool

В стандарт ANSI/ISO для языка C++ включен дополнительный новый тип (то есть, новый для C++), называемый bool. Он назван так в честь английского математика Джорджа Буля (George Boole), разработавшего математическое представление законов логики. В вычислительной технике *булевская переменная* может принимать всего два значения: true (истина) или false (ложь). Ранее в языке C++, как и в C, не было булевского типа. Вместо него, как вы увидите в главах 5 и 6, C++ интерпретировал ненулевые значения как true, и нулевые значения как false. Теперь для представления true и false можно использовать тип bool, а предварительно определенные литералы true и false позволяют представлять эти значения. То есть, вы можете записывать операторы, подобные следующему:

```
bool isready = true;
```

Литералы `true` и `false` могут быть преобразованы в тип `int`, причем `true` преобразовывается в `1`, а `false` в `0`:

```
int ans = true;         // переменной ans присваивается 1
int promise = false;    // переменной promise присваивается 0
```

Кроме того, любое числовое значение или значение-указатель может быть преобразовано неявно (то есть, без явного приведения типов) в значение `bool`. Любое ненулевое значение преобразовывается в `true`, а нулевое значение — в `false`:

```
bool start = -100;      // переменной start присваивается true
bool stop = 0;          // переменной stop присваивается false
```

После того как мы рассмотрим операторы `if` (в главе 6), тип `bool` будет чаще встречаться в примерах.

Квалификатор const

Теперь давайте вернемся к вопросу о символических именах констант. По символическому имени можно судить о том, что представляет константа. Также, если в нескольких местах программы используется константа и вам необходимо изменить ее значение, то для этого достаточно изменить одно описание символа. В примечании к операторам `#define` (см. врезку "Символические константы как средство препроцессора") сказано, что в языке C++ имеется более удобный способ обработки символьных констант — посредством использования ключевого слова `const` для изменения объявления и инициализации переменной. Предположим, например, что вам нужна символическая константа для выражения количества месяцев в году. В своей программе вы вводите следующую строку:

```
const int MONTHS = 12;   // Символическая константа MONTHS,
                         // значение которой равно 12
```

Теперь эту константу можно применять в программе вместо значения `12`. (Вместо `12` в программе может использоваться количество дюймов в футе, количество пончиков в дюжине, однако имя `MONTHS` говорит о том, что представляет значение `12`.) После инициализации константы, такой как `MONTHS`, устанавливается ее значение. Компилятор не допускает последующего изменения значения константы `MONTHS`. Если вы попытаетесь это сделать, Borland C++ выдаст сообщение об ошибке, требуя использовать *l-значение*. Это же сообщение можно получить, если попытаться присвоить значение `4` значению `3`. (l-значение — это значение, которое может находиться в левой части оператора присваивания.) Ключевое слово `const` называется *квалификатором*, потому что оно квалифицирует само объявление.

Как правило, на практике имена записывают в верхнем регистре, поэтому `MONTHS` можно идентифицировать как константу. Конечно, это соглашение не универсальное, однако при чтении кода программ оно помогает различать константы и переменные. В соответствии с другим соглашением буква в верхнем регистре ставится только вначале имени константы. По еще одному соглашению перед именем константы ставится буква `k`, например `kmonths`. Существует еще целый ряд соглашений. Во многих организациях принят свой формат написания кода, который надлежит использовать всем программистам.

Общая форма для создания константы выглядит следующим образом:

```
const тип имя = значение;
```

Обратите внимание, что вы инициализируете const в объявлении. Следующая последовательность строк не является хорошим примером:

```
const int toes; // значение переменной toes до этой строки является
                // неопределенным
toes = 10;      // а теперь слишком поздно!
```

Если вы не присвоите значение константе во время ее объявления, она получит неопределенное значение, которое вы не сможете изменить.

Если у вас имеется опыт в написании программ на С, вам может показаться, что оператор #define вполне может справиться с этой задачей. И все же const лучше. Во-первых, он позволяет явным образом определить тип. Во-вторых, вы можете использовать правила обзора данных, чтобы ограничить описание для определенных функций или файлов. (Правила обзора данных описывают, в какой части кода программы будет известно данное имя; этот вопрос мы рассмотрим в главе 9.) В-третьих, квалификатор const можно использовать и для более сложных типов, например массивов и структур, о чем мы поговорим в главе 4.

Совет

Если вы начали изучать С++, имея опыт в написании программ на С, и для определения символьных констант намерены применять #define, то лучше всего использовать const.

В ANSI С также используется квалификатор const, позаимствованный в С++. Если вы знакомы с версией ANSI С, то должны знать, что версия С++ немного отличается. Одним из отличий являются правила обзора данных, о которых речь пойдет в главе 9. Другое важное отличие состоит в том, что в С++ (но не в С) значение const можно использовать для объявления размера массива. Примеры применения квалификатора const вы найдете в главе 4.

Числа с плавающей точкой

Теперь, после того как вы познакомились со всеми целочисленными типами в С++, мы можем перейти к рассмотрению чисел с плавающей точкой, которые составляют вторую основную группу фундаментальных типов в С++. Эта группа позволяет представлять числа с дробными частями, например, расход топлива танка M1 (0.56 миль на галлон). Также эта группа предлагает более широкий диапазон значений. Если число слишком большое, чтобы его можно было представить как тип long, например количество звезд в нашей галактике (примерно 400 000 000 000), можно использовать один из типов чисел с плавающей точкой.

Посредством типов с плавающей точкой можно представлять такие числа, как 2.5 и 3.14159 и 122442.32 — то есть числа с дробными частями. Такие числа компьютер хранит в двух частях. Одна часть представляет значение, а другая часть увеличивает или уменьшает его. Приведем такую аналогию. Сравним два числа: 34.1245 и 34124.5. Они идентичны друг другу за исключением масштаба. Первое значение можно представить как 0.341245 (базовое значение) и 100 (масштабный множитель). Второе значение можно представить как 0.341245 (такое же базовое значение) и 100 000 (больший масштабный множитель). Масштабный множитель необходим для того, чтобы

перемещать десятичную точка, которая поэтому и называется *плавающей точкой.* В языке C++ используется еще один похожий способ внутреннего представления чисел с плавающей точкой: он отличается тем, что основан на двоичных числах, поэтому масштабирование производится с помощью множителя 2, а не 10. К счастью, вам не нужно досконально разбираться в механизме внутреннего представления чисел. Вы должны усвоить одно: с помощью чисел с плавающей точкой можно представлять дробные очень большие и очень малые значения, и что их внутреннее представление сильно отличается от представления целых чисел.

Запись чисел с плавающей точкой

В языке C++ записать числа с плавающей точкой можно двумя способами. Первый из них заключается в использовании стандартного обозначения десятичной точки, которое вы используете на практике:

```
12.34       // число с плавающей точкой
939001.32   // число с плавающей точкой
0.00023     // число с плавающей точкой
8.0         // это тоже число с плавающей точкой
```

Даже если дробная часть равна 0, как в числе 8.0, десятичная точка гарантирует, что число представлено в формате с плавающей точкой и не является целым числом. (Стандарт C++ позволяет в реализациях использовать различные специфические представления, например, европейский способ идентификации десятичной точки с помощью запятой. Однако эти представления влияют только на визуальное отображение чисел во входных и выходных данных.)

Другой способ представления значений с плавающей точкой называется экспоненциальным представлением и имеет вид, подобный следующему: 3.45E6. Эта запись означает, что значение 3.45 умножается на 1 000 000; E6 означает 10 в шестой степени, то есть не что иное, как единица с шестью нулями. Поэтому, 3.45E6 соответствует 3 450 000. В данном случае 6 называется *экспонентой*, а 3.45 — *мантиссой.* Ниже показаны примеры таких записей:

```
2.52e+8     // можно использовать E или e; знак + необязателен
8.33E-4     // экспонента может быть отрицательной
7E5         // то же, что и 7.0E+05
-18.32e13   // перед записью может стоять знак + или -
7.123e12    // государственный долг США по состоянию на начало 2004 года
5.98E24     // масса Земли в килограммах
9.11e-31    // масса электрона в килограммах
```

Как вы могли заметить, обозначение посредством E очень удобно использовать для представления очень больших и очень малых чисел.

Обозначение E гарантирует, что число будет храниться в формате с плавающей точкой, даже если десятичная точка не используется. Обратите внимание, что в записи вы можете использовать как `E`, так и `e`, и что экспонента может быть как положительной, так и отрицательной (рис. 3.3). Использование пробелов не допускается, поэтому, запись `7.2 E6` является недопустимой.

Отрицательное значение экспоненты означает деление на степень с основанием 10, а не умножение. Таким образом, 8.33E-4 означает $8.33/10^4$, или 0.000833. Точно так же рассчитывается и масса электрона:

9.11e-31 кг = 0.000000000000000000000000000000911 кг

Вы можете выбрать тот вариант, который вам больше нравится. (Между прочим, заметьте, что 911 это номер телефона службы спасения в США, а телефонные сообщения передаются посредством электронов. Совпадение или научная тайна? Решать вам.) Обратите внимание на то, что –8.33E4 означает –83300. Знак перед записью относится к числу, а знак в экспоненте относится к масштабу.

На память!

Форма `d.dddE+n` означает перемещение десятичной точки на `n` позиций вправо, а форма `d.dddE-n` означает перемещение десятичной точки на `n` позиций влево.

Рис. 3.3. Экспоненциальный формат

Типы чисел с плавающей точкой

Как и в ANCI C, язык C++ имеет три типа чисел с плавающей точкой: `float`, `double` и `long double`. Эти типы характеризуются количеством значащих цифр, которые они могут представлять, и минимальным допустимым диапазоном экспонент. *Значащими цифрами* являются значащие разряды числа. Например, запись высотной отметки горы Шаста (Shasta) в Калифорнии 14162 фута содержит пять значащих цифр, которые определяют высоту с точностью до ближайшего фута. А в записи высотной отметки этой же горы 14000 футов используется две значащие цифры, поэтому результат округляется до ближайшего тысячного фута; в данном случае остальные три разряда являются просто заполнителями. Количество значащих цифр не зависит от позиции десятичной точки. Например, высоту можно записать как 14.162 тысяч футов. В этом случае также используются пять значащих разрядов, поскольку это значение имеет точность до пятого разряда.

Требования в языках C и C++ относительно количества значащих разрядов следующие: тип `float` должен иметь как минимум 32 бита, `double` должен иметь как минимум 48 битов и, естественно, быть не меньше чем `float`, и `long double` должен быть как минимум таким же, как и тип `double`. Все три типа могут иметь одинаковый размер. Однако обычно `float` имеет 32 бита, `double` имеет 64 бита, а `long double` имеет 80, 96 или 128 битов. Кроме того, диапазон экспонент для каждого из этих трех типов ограничен в пределах от -37 до +37. Об ограничениях системы можно узнать в файле `cfloat` или `float.h`. (Файл `cfloat` является аналогом файла `float.h` в языке C.) Далее в качестве примера показаны некоторые строки из файла `float.h` для Borland C++Builder:

```
// минимальное количество значащих цифр
#define DBL_DIG 15    // double
#define FLT_DIG 6     // float
#define LDBL_DIG 18   // long double
// количество битов, используемых для представления мантиссы
#define DBL_MANT_DIG 53
#define FLT_MANT_DIG 24
#define LDBL_MANT_DIG 64
// максимальные и минимальные значения экспоненты
#define DBL_MAX_10_EXP +308
#define FLT_MAX_10_EXP +38
#define LDBL_MAX_10_EXP +4932
#define DBL_MIN_10_EXP -307
#define FLT_MIN_10_EXP -37
#define LDBL_MIN_10_EXP -4931
```

Замечание по совместимости

Не все реализации языка C++ имеют заголовочный файл `cfloat`, и также не все реализации, основанные на компиляторах, разработанных до выхода стандарта ANSI C, имеют заголовочный файл `float.h`.

В листинге 3.8 показан пример использования типов `float` и `double` и продемонстрированы также их различия в точности, с которой они представляют числа (то есть количество значащих цифр). В программе показан пример использования метода `ostream`, который называется `setf()`; о нем мы будем говорить в главе 17. В данном примере этот метод устанавливает формат вывода с фиксированной точкой, который позволяет визуально определять точность выходных данных. Благодаря этому методу программа не будет переключаться на экспоненциальное обозначение больших чисел и будет отображать шесть цифр справа от десятичной точки. Аргументы `ios_base::fixed` и `ios_base::floatfield` являются константами из файла `iostream`.

Листинг 3.8. `floatnum.cpp`

```
// floatnum.cpp -- типы с плавающей точкой
#include <iostream>

int main()
{
   using namespace std;
   cout.setf(ios_base::fixed, ios_base::floatfield); // фиксированная точка
   float tub = 10.0 / 3.0;       // подходит для 6 разрядов
   double mint = 10.0 / 3.0;     // подходит для 15 разрядов
   const float million = 1.0e6;

   cout << "tub = " << tub;
   cout << ", a million tubs = " << million * tub;
   cout << ",\nand ten million tubs = ";
   cout << 10 * million * tub << endl;
   cout << "mint = " << mint << " and a million mints = ";
   cout << million * mint << endl;
   return 0;
}
```

Ниже показан пример выполнения этой программы:

```
tub = 3.333333, a million tubs = 3333333.250000,
and ten million tubs = 33333332.000000
mint = 3.333333 and a million mints = 3333333.333333
```

Замечание по совместимости

В стандарте С++ заменены `ios::fixed` на `ios_base::fixed` и `ios::floatfield` на `ios_base::floatfield`. Если ваш компилятор не принимает формы `ios_base`, попробуйте использовать вместо них `ios`; другими словами, замените `ios::fixed` на `ios_base::fixed` и так далее. По умолчанию в старых версиях С++ при отображении значений с плавающей точкой отображались в общей сложности шесть цифр справа от десятичной точки, например 2345.831541. В стандарте С++ по умолчанию отображается шесть цифр (2345.83), при этом, если значения достигают миллиона или более, происходит переключение на экспоненциальную форму (2.34583E+06). В режимах отображения, не используемых по умолчанию, как в `fixed` из предыдущего примера, отображаются шесть цифр справа от десятичной точки в старых и в новых версиях.

В настройке по умолчанию удаляются конечные нули, отображая 23.4500 как 23.45. Реализации языка отличаются реакцией на использование оператора `setf()` для отмены настроек по умолчанию. В старых версиях, например Borland С++ 3.1 для DOS, также удаляются конечные нули. Версии, соответствующие стандарту, такие как Microsoft Visual С++ 7.0, Metrowerks CodeWarrior 9, Gnu GCC 3.3 и Borland С++ 5.5, отображают нули, как показано в листинге 3.8.

Замечания по программе

Обычно объект `cout` отбрасывает конечные нули. Например, он может отобразить 3333333.250000 как 3333333.25. Вызов функции `cout.setf()` отменяет это поведение, по крайней мере, в новых реализациях. Главное, на что следует обратить внимание в листинге 3.8, это на различия в точности между типами `float` и `double`. Значения переменных `tub` и `mint` оценивались по одному выражению `10.0/3.0`, результат которого равен приблизительно 3.33333333333333333... Поскольку `cout` выводит шесть цифр справа от десятичной точки, вы можете видеть, что `tub` и `mint` отображаются с довольно высокой точностью. Однако после того как программа умножит каждое число на миллион, вы увидите, что значение `tub` отличается от правильного результата после 7-й тройки. `tub` дает хороший результат до 7-й значащей цифры. (Эта система гарантирует 6 значащих цифр для `float`, но это самый худший вариант.) Переменная, имеющая тип `double`, показывает 13 троек, поэтому она дает хороший результат до 13-й значащей цифры. Поскольку система гарантирует 15 значащих цифр, то такой и должна быть точность этого типа. Обратите также внимание, что умножение `million tubs` на 10 дает не совсем точный результат; это еще раз указывает на ограниченную точность значений, имеющих тип `float`.

Класс `ostream`, к которому принадлежит объект `cout`, имеет функции-члены класса, которые позволяют управлять способом форматирования вывода — вы можете задавать ширину полей, определять позиции справа от десятичной точки, выбирать десятичную или экспоненциальную форму представления и так далее. Эти вопросы будут рассматриваться в главе 17. Примеры из этой книги довольно простые и обычно используют только операцию <<. Иногда после ее выполнения отображается больше разрядов, чем это необходимо, однако этот “дефект” всего лишь визуальный. О вариантах форматирования выходных данных вы узнаете в главе 17. Однако не каждый из этих вариантов полностью подойдет вам.

Пример из практики: чтение включаемых файлов

К директиве include, которую можно найти в начале файла исходного кода C++, часто относятся как к какому-то магическому элементу; изучая материал данной книги и выполняя предложенные примеры, новички в C++ смогут узнать, для каких целей предназначены определенные заголовочные файлы, и самостоятельно их использовать для обеспечения работоспособности программы. Не воспринимайте эти файлы как нечто сверхъестественное — смелее открывайте их и внимательно изучайте. Они представляют собой обыкновенные текстовые файлы, поэтому их можно без труда читать. Каждый файл, который вы включаете в свою программу, хранится в вашем компьютере или на доступном ему носителе. Постарайтесь отыскать используемые файлы и просмотрите их содержимое. Вы поймете, что используемые файлы исходного кода и заголовочные файлы являются превосходным источником информации, а в некоторых случаях в них можно найти больше информации, чем в самой лучшей документации. Впоследствии, когда вы научитесь включать в свои программы более сложные файлы и работать с другими, нестандартными библиотеками, это занятие будет приносить вам немалую пользу.

Константы с плавающей точкой

Когда в своей программе вы записываете константу с плавающей точкой, с каким именно типом программа будет хранить ее значение? По умолчанию константы с плавающей точкой, например, 8.24 и 2.4E8, имеют тип double. Чтобы константа имела тип float, необходимо указать суффикс f или F. Для типа long double используется суффикс l или L. (Поскольку начертание буквы l в нижнем регистре очень похоже на начертание цифры 1, рекомендуется использовать верхний регистр.) Далее показаны примеры использования суффиксов:

```
1.234f         // константа float
2.45E20F       // константа float
2.345324E28    // константа double
2.2L           // константа long double
```

Преимущества и недостатки чисел с плавающей точкой

Числа с плавающей точкой обладают двумя преимуществами по сравнению с целыми числами. Во-первых, они могут представлять значения между целыми числами. Во-вторых, поскольку у них есть масштабный множитель, они могут представлять более широкий диапазон значений. С другой стороны, операции над числами в формате с плавающей точкой выполняются медленнее, чем целочисленные операции, по крайней мере, на компьютерах, не имеющих математического сопроцессора. Также вы можете потерять точность вычислений, чему посвящен следующий пример.

Листинг 3.9. fltadd.cpp

```
//fltadd.cpp--потеря точности при работе с числами в формате с плавающей точкой
#include <iostream>
int main()
{
   using namespace std;
   float a = 2.34E+22f;
   float b = a + 1.0f;
```

```
    cout << "a = " << a << endl;
    cout << "b - a = " << b - a << endl;
    return 0;
}
```

Замечание по совместимости

До выхода стандарта ANSI C некоторые реализации не поддерживали суффикс `f` для обозначения констант, имеющих тип `float`. Если вы столкнетесь с этой проблемой, попробуйте заменить `2.34E+22f` на `2.34E+22` и `1.0f` на `(float) 1.0`.

В программе из листинга 3.9 берется число, к нему прибавляется единица, и затем вычитается исходное число. По идее, результат должен быть равен единице. Так ли это? Ниже показан результат выполнения программы из листинга 3.9:

```
a = 2.34e+022
b - a = 0
```

Дело в том, что 2.34E+22 представляет число с 23 цифрами слева от десятичной точки. При прибавлении единицы происходит прибавление единицы к 23-ей цифре этого числа. А тип `float` может представить только первые 6 или 7 цифр данного числа, поэтому попытка изменить 23-ю цифру не повлияет на изменение значения.

Классификация типов данных

Базовые типы данных в языке C++ объединены в семейства типов. Типы `signed char`, `short`, `int` и `long` называются *целыми типами со знаком*. Версии типов без знака называются *целыми типами без знака*, или *беззнаковыми типами*. Типы `bool`, `char`, `wchar_t`, целый тип со знаком, целые типы без знака все вместе называются *целочисленными*, или *целыми* типами. Типы `float`, `double` и `long double` называются типами с плавающей точкой. Целые типы и типы с плавающей точкой все вместе называются *арифметическими* типами.

Арифметические операции в языке C++

Наверное, у вас сохранились кое-какие воспоминания об арифметике еще со средней школы. У вас есть возможность переложить эту заботу на плечи компьютера. Для выполнения арифметических действий в языке C++ используется пять базовых арифметических операций: сложение, вычитание, умножение, деление и нахождение остатка от целочисленного деления. Каждая из этих операций выполняется над парой значений, называемых *операндами*, для нахождения конечного результата. Операция и ее операнды вместе образуют *выражение*. Например, рассмотрим следующий оператор:

```
int wheels = 4 + 2;
```

Значения `4` и `2` называются операндами, знак `+` обозначает операцию сложения, а `4 + 2` — это выражение, результатом которого является 6.

Ниже перечислены пять базовых арифметических операций в языке C++:

- Операция `+` выполняет сложение операндов. Например, `4 + 20` равно `24`.
- Операция `-` вычитает второй операнд из первого. Например, `12 - 3` равно `9`.
- Операция `*` умножает операнды. Например, `28 * 4` равно `112`.

- Операция / выполняет деление первого операнда на второй. Например, 1000 / 5 равно 200. Если оба операнда являются целыми числами, то результат будет равен целой доли частного. Например, 17 / 3 равно 5, с отброшенной дробной частью.
- Операция % находит остаток целочисленного деления первого операнда на второй. Например, 19 % 6 равно 1, поскольку 19 разбивается на 6 три раза, с остатком 1. Оба операнда при этом должны быть целочисленными; использование операции % над числами в формате с плавающей точкой приведет к ошибке времени компиляции. Если один из операндов будет отрицательным, то знак результата будет зависеть от реализации.

Естественно, на месте операндов можно использовать переменные и константы. В листинге 3.10 как раз показан пример их использования. Поскольку операция % работает только над целыми числами, мы рассмотрим ее чуть позже.

Листинг 3.10. arith.cpp

```
// arith.cpp -- примеры арифметических операций в C++
#include <iostream>
int main()
{
   using namespace std;
   float hats, heads;
   cout.setf(ios_base::fixed, ios_base::floatfield);// формат с
                                                    // фиксированной точкой
   cout << "Enter a number: ";
   cin >> hats;
   cout << "Enter another number: ";
   cin >> heads;
   cout << "hats = " << hats << "; heads = " << heads << endl;
   cout << "hats + heads = " << hats + heads << endl;
   cout << "hats - heads = " << hats - heads << endl;
   cout << "hats * heads = " << hats * heads << endl;
   cout << "hats / heads = " << hats / heads << endl;
   return 0;
}
```

Замечание по совместимости

Если ваш компилятор не принимает формы ios_base в setf(), попробуйте использовать более старые формы ios; другими словами, замените ios::fixed на ios_base::fixed и так далее.

Как видно из результатов выполнения программы из листинга 3.10, языку C++ можно смело поручать выполнение элементарных арифметических операций:

```
Enter a number: 50.25
Enter another number: 11.17
hats = 50.250000; heads = 11.170000
hats + heads = 61.419998
hats - heads = 39.080002
hats * heads = 561.292480
hats / heads = 4.498657
```

И все же полностью доверять ему нельзя. Так, при добавлении 11.17 к 50.25 результат должен быть равен 61.42, а в результатах выполнения он равен 61.419998. Такое расхождение не связано с ошибками выполнения арифметических операций, а объясняется ограниченными возможностями типа float представлять значащие цифры. Если помните, для типа float гарантируется только шесть значащих цифр. Если 61.419998 округлить до шести цифр, получим 61.4200, что является вполне корректным значением для гарантированной точности. Из этого следует, что если вам нужна более высокая точность, то в таком случае необходимо использовать тип double или long double.

Порядок выполнения операций: приоритеты операций и ассоциативность

Можете ли вы доверить C++ выполнение сложных вычислений? Да, но для этого необходимо знать о том, какими правилами руководствуется C++ при выполнении вычислений. Например, во многих выражениях выполняется несколько операций. Поэтому возникает логичный вопрос: какая из них должна быть выполнена первой? Например, рассмотрим следующий оператор:

```
int flyingpigs = 3 + 4 * 5; // 35 или 23?
```

Получается, что операнд 4 может участвовать и в сложении, и в умножении. Когда над одним и тем же операндом может быть выполнено несколько операций, C++ руководствуется правилами *старшинства*, или правилами *приоритета*, чтобы определить, какая операция должна быть выполнена первой. Арифметические операции выполняются в соответствии с алгебраическими правилами, согласно которым умножение, деление и нахождение остатка от целочисленного деления выполняются до операций сложения и вычитания. Поэтому выражение 3 + 4 * 5 следует читать как 3 + (4 * 5), но не (3 + 4) * 5. Итак, результатом этого выражения будет 23, а не 35. Конечно, чтобы обозначить свои приоритеты вы можете заключать операнды и операции в скобки. В приложении Г представлены приоритеты всех операций в C++. Обратите внимание, что в приложении Г операции *, / и % занимают одну строку. Это означает, что они имеют одинаковый уровень приоритета выполнения. Операции сложения и вычитания имеют самый низкий уровень приоритета выполнения.

Однако знания только лишь приоритетов операций не всегда бывает достаточно. Взгляните на следующий оператор:

```
float logs = 120 / 4 * 5; // 150 или 6?
```

И в этом случае над операндом 4 могут быть выполнены две операции. Операции / и * имеют одинаковый уровень приоритета, поэтому программа нуждается в уточняющих правилах, чтобы определить, нужно ли сначала делить 120 на 4, или же 4 умножать на 5. Поскольку результатом первого варианта является 150, а второго — 6, выбор здесь очень важен. В том случае, когда две операции имеют одинаковый уровень приоритета, C++ анализирует их *ассоциативность*: ассоциативность слева направо или ассоциативность справа налево. Ассоциативность слева направо означает, что если две операции, выполняемые над одним и тем же операндом, имеют одинаковый приоритет, то сначала выполняется операция слева от операнда. В случае ассоциативности справа налево будет первой выполнена операция справа от операнда. Сведения об ассоциативности также можно найти в приложении Г. В этом приложении пока-

зано, что для операций умножения и деления характерна ассоциативность слева направо. Это означает, что над операндом 4 первой будет выполнена операция слева. То есть вы делите 120 на 4, получаете в результате 30, а затем умножаете результат на 5, получая в итоге 150.

Следует отметить, что приоритет выполнения и правила ассоциативности действуют только тогда, когда две операции относятся к одному и тому же операнду. Рассмотрим следующее выражение:

```
int dues = 20 * 5 + 24 * 6;
```

В соответствии с приоритетом выполнения операций, программа должна умножить 20 на 5, затем умножить 24 на 6, и после этого выполнить сложение. Однако ни уровень приоритета выполнения, ни ассоциативность не помогут определить, какая из операций умножения должна быть выполнена в первую очередь. Можно было бы предположить, что в соответствии со свойством ассоциативности должна быть выполнена операция слева, однако в данном случае две операции умножения не относятся к одному и тому же операнду, поэтому эти правила здесь не могут быть применены. В действительности, выбор порядка выполнения операций, который будет приемлемым для системы, оставлен за конкретной реализацией C++. В данном случае при любом порядке выполнения будет получен один и тот же результат, однако иногда результат выражения зависит от порядка выполнения операций. Мы вернемся к этому вопросу в главе 5, когда будем рассматривать операцию инкремента.

Различные результаты, получаемые после деления

Мы продолжаем рассмотрение вопроса, связанного с операцией деления (/). Поведение этой операции зависит от типа операндов. Если оба операнда представлены целыми числами, то C++ выполнит целочисленное деление. Это означает, что любая дробная часть результата будет отброшена, приводя результат к целому числу. Если один или оба операнда представлены значениями в формате с плавающей точкой, то дробная часть остается, поэтому результирующим будет число в формате с плавающей точкой. В листинге 3.11 показано, как операция деления в C++ осуществляется над значениями различных типов. Как и в листинге 3.10, в листинге 3.11 вызывается функция-член setf() для изменения формата отображаемых результатов.

Листинг 3.11. divide.cpp

```
// divide.cpp -- деление целых чисел и чисел в формате с плавающей точкой
#include <iostream>
int main()
{
   using namespace std;
   cout.setf(ios_base::fixed, ios_base::floatfield);
   cout << "Integer division: 9/5 = " << 9 / 5 << endl;
   cout << "Floating-point division: 9.0/5.0 = ";
   cout << 9.0 / 5.0 << endl;
   cout << "Mixed division: 9.0/5 = " << 9.0 / 5 << endl;
   cout << "double constants: 1e7/9.0 = ";
   cout << 1.e7 / 9.0 << endl;
   cout << "float constants: 1e7f/9.0f = ";
   cout << 1.e7f / 9.0f << endl;
   return 0;
}
```

Замечание по совместимости

Если ваш компилятор не принимает формы `ios_base` в `setf()`, попробуйте использовать старые формы `ios`. До выхода стандарта ANSI C некоторые реализации не поддерживали суффикс `f` для обозначения констант, имеющих тип `float`. Если вы столкнетесь с такой проблемой, попробуйте заменить `1.e7f / 9.0f` на `(float) 1.e7 / (float) 9.0`.

В некоторых реализациях удаляются конечные нули.

Ниже показан результат выполнения программы из листинга 3.11:

```
Integer division: 9/5 = 1
Floating-point division: 9.0/5.0 = 1.800000
Mixed division: 9.0/5 = 1.800000
double constants: 1e7/9.0 = 1111111.111111
float constants: 1e7f/9.0f = 1111111.125000
```

Первая строка вывода показывает, что в результате деления целого числа `9` на целое число `5` было получено целое число `1`. Дробная часть от деления `4 / 5` (или `0.8`) отбрасывается. (Далее в этой главе вы увидите практический пример использования такого деления при рассмотрении операции нахождения остатка целочисленного деления.) Следующие две строки показывают, что если хотя бы один из операндов имеет формат с плавающей точкой, искомый результат (`1.8`) также будет представлен в этом формате. В действительности, при комбинировании смешанных типов C++ преобразовывает их к одному типу. Об этих автоматических преобразованиях мы поговорим далее в этой главе. Относительная точность в двух последних строках вывода показывает, что результат имеет тип `double`, если оба операнда имеют тип `double`, и тип `float`, если оба операнда имеют тип `float`. Не забывайте, что константы в формате с плавающей точкой имеют тип `double` по умолчанию.

Несколько слов о перегрузке операций

В листинге 3.11 операция деления представляет три различных деления: целочисленное деление `int`, деление `float` и деление `double`. Чтобы определить, какая именно подразумевается операция, язык C++ использует контекст — в данном случае тип операндов. Использование одного и того же символа для нескольких операций называется *перегрузкой операций*. В языке C++ можно найти несколько примеров перегрузки. C++ позволяет расширять перегрузку операций на определяемые пользователем классы, поэтому то, что можно увидеть в этом примере, является предвестником важного свойства ООП (рис. 3.4).

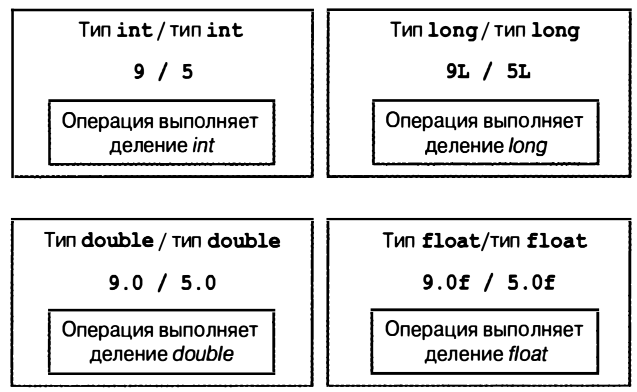


Рис. 3.4. Различные операции деления

Операция нахождения остатка целочисленного деления

Большинство людей больше знакомы с операциями сложения, вычитания, умножения и деления, чем с операцией нахождения остатка целочисленного деления, поэтому давайте остановимся на ее рассмотрении. В результате выполнения этой операции возвращается остаток, полученный от деления целых чисел. В комбинации с целочисленным делением, операция нахождения остатка особенно полезна в тех задачах, где интересуемую величину необходимо разделить на целые единицы. Например, при преобразовании дюймов в футы и дюймы или преобразовании долларов в двадцатипятицентовые, десятицентовые, пятицентовые и одноцентовые эквиваленты. В листинге 2.6 во второй главе был показан пример преобразования мер веса: британского стоуна в фунты. В листинге 3.12 демонстрируется обратный процесс: преобразование фунтов в стоуны. Напомним, что 1 стоун равен 14 фунтам, и в Великобритании большинство весов в ванных комнатах калиброваны в этих единицах. Программа использует целочисленное деление для нахождения наибольшего целого стоуна и операцию нахождения остатка для определения оставшихся фунтов.

Листинг 3.12. modulus.cpp

```
// modulus.cpp -- использует операцию % для преобразования фунтов в стоуны
#include <iostream>
int main()
{
   using namespace std;
   const int Lbs_per_stn = 14;
   int lbs;
   cout << "Enter your weight in pounds: ";
   cin >> lbs;
   int stone = lbs / Lbs_per_stn;     // целый стоун
   int pounds = lbs % Lbs_per_stn;    // остаток в фунтах
   cout << lbs << " pounds are " << stone
        << " stone, " << pounds << " pound(s).\n";
   return 0;
}
```

Ниже показан результат выполнения программы из листинга 3.12:

```
Enter your weight in pounds: 177
184 pounds are 12 stone, 9 pound(s).
```

В выражении `lbs / Lbs_per_stn` оба операнда имеют тип `int`, поэтому компьютер выполняет целочисленное деление. С учетом введенного значения (`177`), которое было присвоено переменной `lbs`, результат выражения оказался равным `12`. Умножение `12` на `14` дает `168`, поэтому остаток от деления `177` на `14` равен `9`, и это значение является значением `lbs % Lbs_per_stn`. Теперь вы теоретически, безо всяких эмоций, будете готовы ответить на вопрос о собственном весе, когда будете путешествовать по Великобритании.

Преобразования типов

Благодаря большому разнообразию типов в языке C++ вы можете выбрать для своей задачи любой необходимый тип данных. Однако помимо пользы, это разнообразие еще и усложняет компьютеру жизнь. Например, при сложении двух значений, имеющих тип short, могут использоваться аппаратные инструкции, отличные от используемых при сложении двух значений long. При наличии 11 целочисленных типов и 3 типов чисел с плавающей точкой компьютер имеет множество вариантов их обработки, особенно если вы комбинируете числа, имеющие различные типы. Чтобы справиться с потенциальной неразберихой в типах, язык C++ осуществляет автоматическое преобразование многих типов:

- C++ преобразует значения при присваивании значения одного арифметического типа переменной другого арифметического типа.
- C++ преобразует значения при комбинировании разных типов в выражениях.
- C++ преобразует значения при передаче аргументов функциям.

Если вы не знаете, что происходит во время автоматического преобразования типов, то некоторые результаты выполнения ваших программ могут оказаться несколько неожиданными, поэтому давайте рассмотрим правила преобразования типов.

Преобразование при присваивании

В языке C++ разрешается присваивание числового значения одного типа переменной другого типа. Всякий раз, когда вы это делаете, значение преобразовывается в тип переменной, которая его получает. Предположим, например, что переменная so_long имеет тип long, а переменная thirty имеет тип short, и в вашей программе используется такой оператор:

```
so_long = thirty;  // присваивание значения переменной типа short
                   // переменной типа long
```

Программа принимает значение переменной thirty (обычно 16-битное значение) и расширяет его до значения long (обычно 32-битное значение) во время присваивания. Заметьте, что в процессе расширения создается новое значение, которое будет присвоено переменной so_long; содержимое переменной thirty остается неизмененным.

Присваивание значения переменной, тип которой имеет более широкий диапазон, обычно происходит без особых проблем. Например, при присваивании значения переменной, имеющей тип short, переменной, имеющей тип long, само значение не изменяется; в этом случае значение просто получает несколько дополнительных байтов, которые остаются незанятыми. А если большое значение, имеющее тип long, например, 2111222333, присвоить переменной, имеющей тип float, то точность будет потеряна. Поскольку переменная типа float может иметь только шесть значащих цифр, то значение может быть округлено до 2.11122E9. Поэтому одни преобразования могут быть осуществлены без каких-либо ошибок, а другие могут привести к возникновению проблем. В табл. 3.3 указаны некоторые возможные проблемы, возникающие при преобразовании.

Таблица 3.3. Потенциальные проблемы при преобразовании чисел

Тип преобразования	Возможные проблемы
Больший тип с плавающей точкой в меньший тип с плавающей точкой, например, `double` в `float`.	Потеря точности (значащих цифр); исходное значение может превысить диапазон искомого типа, поэтому результат будет неопределенным.
Тип с плавающей точкой в целочисленный тип.	Потеря дробной части; исходное значение может превысить диапазон искомого типа, поэтому результат будет неопределенным.
Больший целочисленный тип в меньший целочисленный тип, например, `long` в `short`.	Исходное значение может превысить диапазон искомого типа; обычно копируются только младшие байты.

Нулевое значение, присвоенное переменной `bool`, преобразуется в `false`, а ненулевое значение — в `true`.

При присваивании значений, имеющих тип с плавающей точкой, целочисленным переменным могут возникнуть две проблемы. Во-первых, преобразование типа с плавающей точкой в целочисленный тип приводит к усечению числа (отбрасыванию дробной части). Во-вторых, значение, имеющее тип `float`, может оказаться слишком большим, чтобы его можно было присвоить переменной `int`. В этом случае С++ не определяет, каким должен быть результат, то есть разные реализации С++ будут по-разному реагировать на подобные ситуации. В листинге 3.13 можно увидеть несколько примеров преобразования при присваивании.

Листинг 3.13. `assign.cpp`

```
// assign.cpp -- изменение типа при присваивании
#include <iostream>
int main()
{
   using namespace std;
   cout.setf(ios_base::fixed, ios_base::floatfield);
   float tree = 3;      // тип int преобразован в тип float
   int guess = 3.9832; // тип float преобразован в тип int
   int debt = 7.2E12;  // результат не определен в C++
   cout << "tree = " << tree << endl;
   cout << "guess = " << guess << endl;
   cout << "debt = " << debt << endl;
   return 0;
}
```

Ниже показан результат выполнения программы из листинга 3.13:

```
tree = 3.000000
guess = 3
debt = 1634811904
```

В этом случае переменной `tree` присваивается значение в формате с плавающей точкой `3.0`. После того как переменной `guess`, имеющей тип `int`, было присвоено значение `3.9832`, это значение было усечено до `3`; при преобразовании типов с плавающей точкой в целочисленные типы значения усекаются (отбрасывается дробная

часть) без округления (нахождение ближайшего целого числа). Обратите также внимание, что переменная debt, имеющая тип int, не может хранить число 7.2E12. Это порождает ситуацию, при которой C++ не определяет, каким должен быть результат. В этой системе debt заканчивается значением 1634811904, или примерно 1.6E09. Нельзя ли таким способом подходить к проблеме сокращения внушительной задолженности?!

Некоторые компиляторы предупреждают о возможной потере данных для тех операторов, которые инициализируют целочисленные переменные по значениям в формате с плавающей точкой. Кроме того, значение переменной debt будет разным для разных компиляторов. Например, выполнение той же программы из листинга 3.13 во второй системе даст значение 2147483647.

Преобразования в выражениях

Давайте разберемся с тем, что происходит при комбинировании двух различных арифметических типов в одном выражении. В подобных случаях C++ выполняет два вида преобразований: автоматическое преобразование типов в том месте кода программы, где они встречаются, и преобразование типов при их комбинировании с другими типами в выражении.

Давайте начнем с автоматических преобразований. Когда C++ оценивает выражение, он преобразовывает типы bool, char, unsigned char, signed char и short в тип int. В частности, значение true преобразуется в 1, а false — в 0. Такие преобразования называются *целочисленными расширениями*. В качестве примера рассмотрим следующие операторы:

```
short chickens = 20;              // строка 1
short ducks = 35;                 // строка 2
short fowl = chickens + ducks;    // строка 3
```

Чтобы выполнить оператор в строке 3, программа на C++ принимает значения переменных chickens и ducks и преобразует их в тип int. Затем программа преобразует полученный результат обратно в short, поскольку конечный результат присваивается переменной, имеющей тип short. Настоящий круговорот! Однако эта схема не лишена смысла. Тип int обычно является наиболее распространенным типом, поэтому компьютер произведет все вычисления быстрее всего именно для этого типа.

Возможен еще один вариант целочисленного расширения: когда тип unsigned short преобразуется в int, если тип short короче, чем тип int. Если оба типа имеют одинаковый размер, то unsigned short преобразуется в unsigned int. Это правило гарантирует, что никакой потери данных при расширении типа unsigned short не будет. Подобным образом и тип wchar_t расширяется до первого из следующих типов, диапазон которого позволяет уместить его диапазон: int, unsigned int, long или unsigned long.

Возможны также преобразования при арифметическом комбинировании различных типов, например, при сложении значений, имеющих типы int и float. Если в арифметической операции участвуют два разных типа, то меньший тип преобразуется в больший. Например, в программе из листинга 3.11 значение 9.0 делится на 5. Так как 9.0 имеет тип double, то программа, прежде чем произвести деление, преобразует значение 5 в тип double. Вообще говоря, чтобы определить, какие преобразования необходимо осуществить в арифметическом выражении, компилятор просматривает контрольный список, представленный ниже:

1. Если один из операндов имеет тип `long double`, то другой операнд преобразуется в тип `long double`.
2. Иначе, если один из операндов имеет тип `double`, то другой операнд преобразуется в тип `double`.
3. Иначе, если один из операндов имеет тип `float`, то другой операнд преобразуется в тип `float`.
4. Иначе, операнды имеют целочисленный тип, поэтому выполняется целочисленное расширение.
5. В этом случае, если один из операндов имеет тип `unsigned long`, то другой операнд преобразуется в тип `unsigned long`.
6. Иначе, если один из операндов имеет тип `long int`, а другой операнд имеет тип `unsigned int`, то преобразование зависит от относительного размера обоих типов. Если тип `long` может представлять возможные значения, имеющие тип `unsigned int`, то тип `unsigned int` преобразуется в тип `long`.
7. Иначе, оба операнда преобразуются в тип `unsigned long`.
8. Иначе, если один из операндов имеет тип `long`, то другой операнд преобразуется в тип `long`.
9. Иначе, если один из операндов имеет тип `unsigned int`, то другой операнд преобразуется в тип `unsigned int`.
10. Если компилятор во время проверки дошел до этого пункта, значит, оба операнда должны иметь тип `int`.

В стандарте ANSI для языка С действуют те же правила, что и в языке С++, однако правила в классической версии K&R языка С немного отличаются. Например, в классической версии языка С тип `float` всегда расширяется до типа `double`, даже если оба операнда имеют тип `float`.

Преобразования при передаче аргументов

В языке С++ преобразованиями типов при передаче аргументов управляют прототипы функций, о чем мы будем говорить в главе 7. Однако от такого способа преобразования можно отказаться, хотя так поступать не рекомендуется. В этом случае язык С++ будет использовать целочисленное расширение для типов `char` и `short` (`signed` и `unsigned`). Также, чтобы обеспечить совместимость с большим объемом кода в классическом С, язык С++ расширяет аргументы `float` до `double` при передаче их функции, которая не использует прототип.

Приведение типов

Вы можете принудительно преобразовывать типы явным образом через механизм приведения типов. (Относительно использования типов С++ требует соблюдать правила, и в то же время позволяет их нарушать.) Приведение типа может быть осуществлено двумя способами. Например, чтобы преобразовать целочисленное значение переменной `thorn` в тип `long`, можно использовать какое-нибудь из следующих выражений:

```
(long) thorn // возвращает переменную thorn, преобразованную в тип long
long (thorn) // возвращает переменную thorn, преобразованную в тип long
```

В результате приведения типа значение переменной thorn не изменяется; наоборот, создается новое значение указанного типа, которое впоследствии можно будет использовать в выражении, как показано далее:

```
cout << int('Q');     // отображает целочисленный код для 'Q'
```

Вообще говоря, вы можете сделать следующее:

```
(имяТипа) значение    // преобразовывает значение в тип имяТипа
имяТипа (значение)    // преобразовывает значение в тип имяТипа
```

Первая форма представляет стиль языка C, а вторая — стиль C++. Идея новой формы заключается в том, чтобы оформить приведение типа точно так же, как и вызов функции. В результате приведение типов для встроенных типов будет выглядеть так же, как и преобразование типов, которые вы разрабатываете для определяемых пользователями классов.

C++ также предлагает четыре операции приведения типов, возможности которых ограничены. О них мы поговорим в главе 15. Что касается четвертой операции, static_cast<>, то она может использоваться для преобразования значений из одного числового типа в другой. Вы можете ее использовать, например, чтобы преобразовать значение переменной thorn в значение, имеющее тип long:

```
static_cast<long> (thorn)  // возвращает результат преобразования
                           // значения переменной thorn в тип long
```

Вообще говоря, вы можете сделать следующее:

```
static_cast<имяТипа> (значение) //преобразовывает значение в тип имяТипа
```

Как будет сказано в главе 15, Страуструп понимал, что возможности традиционного стиля приведения типов в языке C были очень ограничены.

В листинге 3.14 вкратце показаны примеры обеих форм. Представьте, что первая часть этого листинга является частью мощной программы моделирования экологической среды, вычисления которой производятся в формате с плавающей точкой, а результат преобразовывается в целочисленные значения, представляющие количество птиц и животных. Искомый результат будет зависеть от того, в какой момент вы осуществляете преобразование. Вначале выполняются операции с переменной auks: суммируются значения в формате с плавающей точкой и полученный результат преобразуется в целочисленную форму во время присваивания. Операции с переменными bats и coots выполняются иначе: сначала используется приведение типов для преобразования значений из формата с плавающей точкой в int, а затем эти значения суммируются. В завершающей части программы показан пример использования приведения типов для отображения кода ASCII значения, имеющего тип char.

Листинг 3.14. typecast.cpp

```
// typecast.cpp -- принудительное изменение типов
#include <iostream>
int main()
{
   using namespace std;
   int auks, bats, coots;
   // следующий оператор суммирует значения, имеющие тип double,
   // полученный результат преобразуется в тип int
   auks = 19.99 + 11.99;
```

```
    // эти операторы суммируют целочисленные значения
    bats = (int) 19.99 + (int) 11.99;       // старый синтаксис C
    coots = int (19.99) + int (11.99);      // новый синтаксис C++
    cout << "auks = " << auks << ", bats = " << bats;
    cout << ", coots = " << coots << endl;
    char ch = 'Z';
    cout << "The code for " << ch << " is "; // вывод в формате char
    cout << int(ch) << endl;                 // вывод в формате int
    return 0;
}
```

Ниже показан результат выполнения этой программы:

```
auks = 31, bats = 30, coots = 30
The code for Z is 90
```

Сначала суммируются два значения: 19.99 и 11.99, результатом чего является 31.98. Когда полученное значение присваивается переменной auks, которая имеет тип int, оно усекается до 31. Однако если использовать приведение типов, то значения будут усечены до 19 и 11 до суммирования, поэтому в результате переменные bats и coots получат значение 30. Последний оператор cout использует приведение типа для преобразования значения, имеющего тип char, в тип int и выводит результат на экран. Поэтому cout напечатает значение в виде целого числа, а не символа.

В этой программе показаны две причины использования приведения типов. Во-первых, у вас могут быть значения, которые хранятся в формате double, но используются для вычисления значения, имеющего тип int. Вы можете, например, подбирать координаты для сетки или моделировать целочисленные значения, например, популяции, со значениями в формате с плавающей точкой. Или вам нужно будет сделать так, чтобы в вычислениях все значения обрабатывались как целочисленные. Все это можно сделать путем приведения типов. Имейте в виду, что если сначала значения преобразовать в int и затем суммировать их, то конечный результат будет отличаться от результата, полученного путем суммирования значений в исходном формате с последующим преобразованием результата в тип int, по крайней мере, для этих значений.

Вторая часть программы демонстрирует наиболее распространенную причину использования механизма приведения типов: вы можете добиться того, чтобы данные, представленные в одном формате, вели себя так, как если бы они были представлены в другом формате. Например, в листинге 3.14 переменная ch, имеющая тип char, хранит код буквы Z. Если для вывода значения переменной ch использовать cout, то будет отображена буква Z, поскольку cout руководствуется тем, что переменная ch имеет тип char. Однако после приведения типа переменной ch к типу int объект cout переключается на режим int и выводит ASCII-код, хранящийся в переменной ch.

Резюме

Основные типы данных в языке C++ делятся на две группы. Одна группа представляет значения, которые хранятся в виде целых чисел. Другая группа представляет значения, которые хранятся в формате с плавающей точкой. Целочисленные типы отличаются друг от друга по количеству памяти, которое отводится для хранения значений, а также по тому, имеют они знак или нет. Целочисленными типами являются

следующие (в порядке возрастания диапазона представляемых значений): bool, char, signed char, unsigned char, short, unsigned short, int, unsigned int, long и unsigned long. Существует также тип wchar_t, чье положение среди них зависит от реализации. C++ гарантирует, что тип char имеет достаточно большой размер, чтобы хранить любой член расширенного набора символов в системе, тип short имеет как минимум 16 битов, int как минимум такой же, как short, а long имеет минимум 32 бита и как минимум такой же, как и int. Точный размер зависит от реализации.

Представление символов осуществляется посредством их числовых кодов. Система ввода-вывода определяет, соответствует ли код цифре или символу.

Типы с плавающей точкой могут представлять дробные значения и значения намного больше тех, которые могут быть представлены целыми типами. Существует три таких типа: float, double и long double. C++ гарантирует, что тип float не меньше, чем тип double, и что тип double не больше типа long double. Обычно тип float использует 32 бита памяти, double использует 64 бита, а long double использует от 80 до 128 битов.

Для решения конкретной задачи вы можете выбрать наиболее подходящий тип данных, поскольку язык C++ предлагает различные типы данных, характеризующиеся различными диапазонами представления чисел, и могут быть как знаковыми, так и беззнаковыми.

Над числовыми типами в языке C++ можно выполнять следующие арифметические операции: сложение, вычитание, умножение, деление и нахождение остатка целочисленного деления. Если две операции соперничают за одно значение, то первоочередность выполнения операции определяется в соответствии с правилами приоритета и ассоциативности.

C++ преобразовывает значение одного типа в другой, когда вы присваиваете значение переменной, комбинируете разные типы в арифметических операциях и используете приведение типов для принудительного преобразования типов. Многие преобразования типов являются "безопасными", то есть они могут быть выполнены без потери или изменения данных. Например, вы можете преобразовать значение int в значение long без каких-либо проблем. Другие преобразования, например, преобразование типов с плавающей точкой в целые типы, требуют осторожности.

На первый взгляд вам может показаться, что в языке C++ имеется чересчур много базовых типов значений, особенно если принять во внимание различные правила преобразования. И все же, из всего разнообразия типов вы сможете выбирать именно такой, который будет наиболее всего соответствовать вашим требованиям.

Вопросы для самоконтроля

1. Почему в языке C++ имеется несколько целочисленных типов?
2. Объявите переменные, соответствующие следующим описаниям:
   а. Целочисленная переменная short, имеющая значение 80.
   б. Целочисленная переменная unsigned int, имеющая значение 42 110.
   в. Целочисленная переменная, имеющая значение 3 000 000 000.
3. Какие меры предпринимает язык C++, чтобы не допустить превышения пределов целочисленного типа?

4. В чем состоит различие между 33L и 33?
5. Взгляните на следующие два оператора C++:

```
char grade = 65;
char grade = 'A';
```

Эквивалентны ли они друг другу?
6. Как с помощью C++ можно определить, какой символ представляет код с номером 88? Сделайте это хотя бы двумя способами.
7. Если переменной, имеющей тип `float`, присвоить значение, имеющее тип `long`, это может привести к ошибке при округлении. А что произойдет, если переменной `double` присвоить значение `long`?
8. Чему равны результаты следующих выражений:

   а. `8 * 9 + 2`

   б. `6 * 3 / 4`

   в. `3 / 4 * 6`

   г. `6.0 * 3 / 4`

   д. `15 % 4`
9. Предположим, что `x1` и `x2` являются переменными, имеющими тип `double`, которые вы хотите просуммировать как целые числа и полученный результат присвоить целочисленной переменной. Напишите для этого соответствующий оператор C++.

Упражнения по программированию

1. Напишите короткую программу, которая выдавала бы запрос на ввод роста в целых дюймах и преобразовывала бы их в футы и дюймы. Программа должна использовать символ подчеркивания для обозначения позиции, с которой начинается ввод значений. Используйте также символьную константу `const` для представления коэффициента преобразования.
2. Напишите короткую программу, которая выдавала бы запрос на ввод значения роста в футах и дюймах и веса в фунтах. (Для хранения этой информации используйте три переменных.) Программа должна выдать индекс массы тела (BMI, body mass index). Чтобы рассчитать индекс, сначала преобразуйте высоту в футах и дюймах в высоту в дюймах (1 фут = 12 дюймов). Затем преобразуйте вес в фунтах в массу в килограммах, разделив на 2.2. После этого рассчитайте свой индекс, разделив массу в килограммах на квадрат вашего роста в метрах. Используйте символьные константы для представления различных коэффициентов преобразования.
3. Напишите программу, которая выдавала бы запрос на ввод широты в градусах, минутах и секундах, после чего отображала бы широту в десятичном формате. В одной минуте 60 секунд, а в одном градусе 60 минут; представьте эти значения посредством символьных констант. Для каждого вводимого значения следует использовать отдельную переменную. Пример результата выполнения программы выглядит следующим образом:

```
Enter a latitude in degrees, minutes, and seconds:
First, enter the degrees: 37
Next, enter the minutes of arc: 51
Finally, enter the seconds of arc: 19
37 degrees, 51 minutes, 19 seconds = 37.8553 degrees
```

4. Напишите программу, которая выдавала бы запрос на ввод количества секунд в виде целого значения (используйте тип `long`), и затем отображала бы эквивалентное значение в сутках, часах, минутах и секундах. Для представления количества часов в сутках, количества минут в часе и количества секунд в минуте используйте символьные константы. Пример результата выполнения программы выглядит следующим образом:

```
Enter the number of seconds: 31600000
31600000 seconds = 365 days, 46 minutes, 40 seconds
```

5. Напишите программу, которая выдавала бы запрос на ввод миль, которые вы преодолели на автомобиле, и количества галлонов израсходованного бензина, а затем выдавала бы отчет о том, сколько миль вы преодолели, израсходовав один галлон бензина. Или, если хотите, программа может запросить расстояние в километрах и количество литров бензина, и затем представить отчет в европейском стиле — в количестве литров, израсходованных на один километр.
6. Напишите программу, которая выдавала бы запрос на ввод расхода бензина в европейском стиле (количество литров на 100 км) и преобразовывала бы его в стиль, принятый в США, — в милях на один галлон. Имейте в виду, что кроме использования других единиц измерения в США, в отличие от европейских стран, принято и другое соотношение: расстояние/топливо, а не топливо/расстояние. Учтите, что 100 километров соответствуют 62.14 милям, а 1 галлон равен 3.875 литрам. Таким образом, 19 миль/галлон примерно равно 12.4 литров на 100 км, а 27 миль/галлон примерно составляет 8.7 литров на 100 км.

ГЛАВА 4

Составные типы

В этой главе:

- Как создавать и использовать массивы
- Как создавать и использовать строки в стиле C
- Как создавать и использовать строки класса `string`
- Как использовать методы `get()` и `getline()` для чтения строк
- Как смешивать строковый и числовой ввод
- Как создавать и использовать структуры
- Как создавать и использовать объединения
- Как создавать и использовать перечисления
- Как создавать и использовать указатели
- Как управлять динамической памятью с помощью `new` и `delete`
- Как создавать динамические массивы
- Как создавать динамические структуры
- Автоматическое, статическое и динамическое хранение

Предположим, что вы разработали компьютерную игру под названием "Пользователь-Враг", в которой игроки состязаются с зашифрованным и недружественным компьютерным интерфейсом. Теперь вам необходимо написать программу, которая отслеживает ваши месячные объемы продаж в течение пятилетнего периода. Или вы хотите провести инвентаризацию торговых карт героев-хакеров. Очень скоро вы придете к выводу, что для того, чтобы накапливать и обрабатывать информацию, вам нужно нечто большее, чем простые базовые типы C++. И C++ предлагает нечто большее, а именно — составные типы. Это типы, состоящие из базовых целочисленных типов и типов с плавающей точкой. Наиболее развитый составной тип — это класс, бастион объектно-ориентированного программирования (ООП), к которому направлено наше движение. Но C++ также поддерживает несколько более скромных составных типов, которые взяты из языка C. Массив, например, может хранить множество значений одного и того же типа. Конкретные виды массивов могут хранить строки, которые являются последовательностями символов. Структуры могут хранить по нескольку значений разных типов. Затем есть еще указатели, которые представляют собой переменные, сообщающие компьютеру местонахождение данных в памяти. Все эти составные формы данных (кроме классов) мы рассмотрим в настоящей главе. Мы впервые познакомимся с операциями `new` и `delete`, а также составим первое представление о классе C++ `string`, который предлагает альтернативный способ работы со строками.

Введение в массивы

Массив — это структура данных, которая содержит множество значений, относящихся к одному и тому же типу. Например, массив может содержать 60 значений типа `int`, которые представляют информацию об объемах продаж за пять лет, 12 значений типа `short`, представляющих число дней в каждом месяце, или 365 значений типа `float`, которые указывают ваши ежедневные расходы на питание в течение года. Каждое значение сохраняется в отдельном элементе массива, и компьютер сохраняет все элементы массива в памяти последовательно — друг за другом.

Чтобы создать массив, вы используете оператор объявления. Объявление массива должно описывать три вещи:

- Тип значений каждого элемента.
- Имя массива.
- Количество элементов в массиве.

В C++ это достигается модификацией объявления простой переменной и добавлением квадратных скобок, в которых указано число элементов. Например, следующее объявление

```
short months[12];   // создает массив из 12 элементов типа short
```

создает массив `months`, который имеет 12 элементов, каждый из которых может хранить одно значение типа `short`. По сути, каждый элемент — это переменная, которую можно трактовать как простую переменную.

Так выглядит общая форма объявления массива:

```
typeName arrayName[arraySize];
```

Выражение `arraySize`, представляющее количество элементов, должно быть целочисленной константой, такой как 10, либо значением `const` или константным выражением вроде `8 * sizeof(int)`, в котором все величины известны на момент компиляции. В частности, `arraySize` не может быть переменной, чье значение устанавливается во время работы программы. Однако позднее в этой главе вы узнаете, как, используя операцию `new`, можно обойти это ограничение.

Массив как составной тип

Массив называют *составным типом*, потому что он строится из некоторого другого типа (В языке С применяется термин *унаследованный тип*, но поскольку термин *наследование* в C++ применяется для описания отношений классов, пришлось ввести новый термин.) Вы не можете просто объявить, что нечто является массивом; это всегда должен быть массив элементов определенного конкретного типа. Нет обобщенного типа массива. Вместо этого существует множество специфических типов массивов, таких как массив `char` или массив `long`. Например, рассмотрим следующее объявление:

```
float  loans[20];
```

Типом переменной `loans` будет не "массив", а "массив `float`". Это подчеркивает, что массив `loans` построен из типа `float`.

Большая часть пользы от массивов определяется тем фактом, что к его элементам можно обращаться индивидуально. Способ, который позволяет это делать, заключается в использовании *индекса* для нумерации элементов. Нумерация массивов в C++ начинается с нуля. (Это обязательно; вы должны начинать с нуля. Программисты Pascal и BASIC должны это отметить особо.) Для указания элемента массива C++ использует обозначение с квадратными скобками и индексом между ними. Например, `months[0]` — это первый элемент массива `months`, а `months[11]` — его последний элемент. Обратите внимание, что индекс последнего элемента на единицу меньше, чем размер массива (рис. 4.1). Таким образом, объявление массива позволяет вам создавать множество переменных в одном объявлении, и вы затем можете использовать индекс для идентификации и доступа к индивидуальным элементам.

Рис. 4.1. Создание массива

Важность указания правильных значений индекса

Компилятор не проверяет правильность применяемого индекса. Например, компилятор не станет жаловаться, если вы присвоите значение несуществующему элементу `months[101]`. Однако такое присваивание может вызвать проблемы во время выполнения программы — возможно, повреждение данных или кода, возможно, прерывание выполнения программы. То есть на вашей совести, как разработчика лежит обеспечение правильности значений индекса.

Небольшая программа анализа, представленная в листинге 4.1, демонстрирует несколько свойств массивов, включая их объявление, присваивание значение его элементам, а также инициализацию.

Листинг 4.1. arrayone.cpp

```
// arrayone.cpp -- небольшие массивы целых чисел
#include <iostream>
int main()
{
  using namespace std;
  int yams[3]; // создание массива из трех элементов
  yams[0] = 7; // присваивание значения первому элементу
  yams[1] = 8;
  yams[2] = 6;
  int yamcosts[3] = {20, 30, 5}; // создание и инициализация массива
```

```
    // Примечание. Если ваш компилятор C++ не может инициализировать
    // этот массив, используйте static int yamcosts[3] вместо int yamcosts[3]
    cout << "Total yams = ";
    cout << yams[0] + yams[1] + yams[2] << endl;
    cout << "The package with " << yams[1] << " yams costs ";
    cout << yamcosts[1] << " cents per yam.\n";
    int total = yams[0] * yamcosts[0] + yams[1] * yamcosts[1];
    total = total + yams[2] * yamcosts[2];
    cout << "The total yam expense is " << total << " cents.\n";
    cout << "\nSize of yams array = " << sizeof yams;
    cout << " bytes.\n";
    cout << "Size of one element = " << sizeof yams[0];
    cout << " bytes.\n";
    return 0;
}
```

Замечание по совместимости

Текущая версия C++, так же, как ANSI C, позволяет инициализировать обычные массивы, объявленные в функциях. Однако в некоторых более старых реализациях, которые используют транслятор C++ вместо полноценного компилятора, такой транслятор создает код на C для последующей обработки компилятором C, который не вполне совместим с ANSI C. В таких случаях вы можете получить сообщение об ошибке вроде приведенного ниже, полученного в системе Sun C++ 2.0:

```
"arrayone.cc", line 10: sorry, not implemented: initialization of
yamcosts (automatic aggregate) Compilation failed
```

Чтобы исправить это, используйте ключевое слово `static` в объявлении массива:

```
// инициализация, соответствующая состоянию до принятия стандарта ANSI
static int yamcosts[3] = {20, 30, 5};
```

Ключевое слово `static` заставляет компилятор использовать другую схему выделения памяти для хранения вашего массива — такую, которая позволяет инициализацию даже на языке C, несовместимом с ANSI. В главе 9 обсуждается упомянутое применение `static`.

Вот как выглядит вывод программы из листинга 4.1:

```
Total yams = 21
The package with 8 yams costs 30 cents per yam.
The total yam expense is 410 cents.
Size of yams array = 12 bytes.
Size of one element = 4 bytes.
```

Замечания по программе

Во-первых, программа в листинге 4.1 создает массив из трех элементов по имени `yams`. Поскольку `yams` имеет три элемента, они нумеруются от 0 до 2, и `arrayone.cpp` использует значения индекса от 0 до 2 для присваивания значений трем отдельным элементам. Каждый индивидуальный элемент `yams` — переменная типа `int`, со всеми правилами и привилегиями типа `int`, поэтому `arrayone.cpp` может (и делает это) присваивать значения элементам, складывать элементы, перемножать их и отображать.

В этой программе применяется длинный способ присваивания значений элементам yams. C++ также позволяет инициализировать элементы массива непосредственно в операторе объявления. В листинге 4.1 демонстрируется этот сокращенный способ путем присваивания значений элементам массива yamcosts:

```
int yamcosts[3] = {20, 30, 5};
```

Он просто представляет разделенный запятыми список значений (*список инициализации*), ограниченный фигурными скобками. Пробелы в списке не обязательны. Если вы не инициализируете массив, определенный внутри функции, его элементы остаются неопределенными. Это значит, что элементы получают случайные значения на основе предыдущего содержимого области памяти, выделенной для такого массива.

Далее программа использует значения массива в нескольких вычислениях. Эта часть программы выглядит несколько беспорядочно со всеми этими индексами и скобками. Цикл for, который будет описан в главе 5, представляет мощный способ работы с массивами и исключает необходимость явного указания индексов. Но пока мы ограничимся небольшими массивами.

Как вы, возможно, помните, операция sizeof возвращает размер в байтах типа или объекта данных. Обратите внимание, что если вы применяете sizeof с именем массива, то получаете количество байт, которые занимает весь массив. Но если вы используете sizeof в отношении элемента массива, то получите размер в байтах одного элемента. Это говорит о том, что yams — массив, но yams[1] — просто int.

Правила инициализации массивов

В C++ существует несколько правил, касающихся инициализации массивов. Они ограничивают, когда вы можете ее осуществлять, и определяют, что случится, если количество элементов массива не соответствует количеству элементов инициализатора. Рассмотрим эти правила.

Вы можете использовать инициализацию *только* при объявлении массива. Ее нельзя выполнить позже, и нельзя присвоить один массив другому:

```
int cards[4] = {3, 6, 8, 10};    // все в порядке
int hand[4];                     // все в порядке
hand[4] = {5, 6, 7, 9};          // не допускается
hand = cards;                    // не допускается
```

Однако можно использовать индексы и присваивать значения элементам массива индивидуально.

При инициализации массива можно указать меньше значений, чем в массиве объявлено элементов. Например, следующий оператор инициализирует только первые два элемента массива hotelTips:

```
float hotelTips[5] = {5.0, 2.5};
```

Если вы инициализируете массив частично, то компилятор присваивает остальным элементом нулевые значения. То есть, очень легко инициализировать весь массив нулями — для этого просто нужно явно инициализировать нулем его первый элемент, а инициализацию остальных предоставить компилятору:

```
long totals[500] = {0};
```

Следует отметить, что если вы инициализируете его с помощью {1} вместо {0}, то только первый элемент устанавливается в 1; остальные по-прежнему устанавливаются в 0.

Если вы оставите квадратные скобки пустыми при инициализации массива, то компилятор C++ пересчитает элементы за вас. Предположим, к примеру, что у вас есть такое объявление:

```
short things[] = {1, 5, 3, 8};
```

Компилятор сделает things массивом из пяти элементов.

Позволять ли компилятору делать это

Обычно позволять компилятору считать количество элементов — плохая привычка, потому что количество, которое он посчитает, может отличаться от того, что вы думаете. Однако, как вы вскоре убедитесь, такой подход может оказаться безопасным при инициализации символьных массивов как строк. И если ваш основной принцип состоит в том, что программа, а не вы, должна знать, какой размер у массивов, то можно сделать нечто вроде такого:

```
short things[] = {1, 5, 3, 8};
int num_elements = sizeof things / sizeof (short);
```

Удобно это или нет — зависит от обстоятельств.

Стандартная библиотека шаблонов C++ (STL) предлагает альтернативу массивам — *шаблонный класс* vector. Это более сложная и гибкая конструкция, нежели встроенный составной тип массива. В главе 16 обсуждается STL и шаблонный класс vector.

Строки

Строка — это серия символов, сохраненная в расположенных последовательно байтах памяти. C++ предлагает два способа работы со строками. Первый, унаследованный от C и часто называемый *строки в стиле C* — это тот, что рассматривается в настоящей главе. Далее мы расскажем об альтернативном методе, основанном на библиотечном классе string.

Идея серии символов, сохраняемых в последовательных байтах, предполагает хранение строки в массиве char, где каждый элемент содержится в отдельном элементе массива. Строки предлагают удобный способ хранения текстовой информации — такой как сообщения для пользователя ("Пожалуйста, сообщите номер вашего секретного счета в Швейцарском банке") или его ответы ("Вы, наверное, шутите?"). Строки в стиле C имеют одно специальное свойство: последним в каждой такой строке идет *нулевой символ*. Этот символ, который записывается, как \0 и представляет собой символ с ASCII-кодом 0, служит меткой конца строки. Например, рассмотрим два следующих объявления:

```
char dog [5] = { 'b', 'e', 'a', 'u', 'x'};       // не строка!
char cat[5] = {'f', 'a', 't', 's', '\0'};        // строка!
```

Обе эти переменные — массивы char, но только вторая из них является строкой. Нулевой символ играет фундаментальную роль в строках стиля C. Например, C++ имеет множество функций для обработки строк, включая те, что используются cout. Все они работают, проходя по строке символ за символом до тех пор, пока не встре-

тится нулевой символ. Если вы просите объект cout отобразить такую симпатичную строку, как cat из предыдущего примера, он выводит первые четыре символа, обнаруживает нулевой символ и на этом останавливается. Однако если вы вдруг решите вывести в cout массив dog из предыдущего примера, который не является строкой, то cout напечатает пять символов из этого массива и будет продолжать двигаться по памяти, выводя байт за байтом, интерпретируя каждый из них как символ, подлежащий печати, пока опять-таки не встретит нулевой символ. Поскольку нулевые символы, которые, по сути, представляют собой байты, содержащие нули, встречаются в памяти довольно часто, ошибка обычно обнаруживается быстро, но в любом случае, вы не должны трактовать нестроковые символьные массивы как строки.

Пример инициализации массива cat выглядит довольно громоздким и утомительным — все эти одиночные кавычки, запятые, необходимость помнить о нулевом символе... Не волнуйтесь. Есть более простой способ инициализировать массив как строку. Для этого просто используйте строку в двойных кавычках, называемую *строковой константой* или *строковым литералом*, как показано ниже:

```
char bird[10] = "Mr. Cheeps";   // символ \0 подразумевается
char fish[] = "Bubbles";  // позволяет компилятору подсчитать нужную длину
```

Строки в кавычках всегда неявно включают ограничивающий нулевой символ, поэтому вам не нужно указывать его явно. (Посмотрите на рис. 4.2.) К тому же разнообразные средства ввода C++, предназначенные для чтения строк с клавиатуры в массив char, автоматически добавляют ограничитель — нулевой символ. (Если, компилируя программу из листинга 4.1, вы обнаружите необходимость применения ключевого слова static для инициализации массива, это также нужно будет сделать с этими массивами char.)

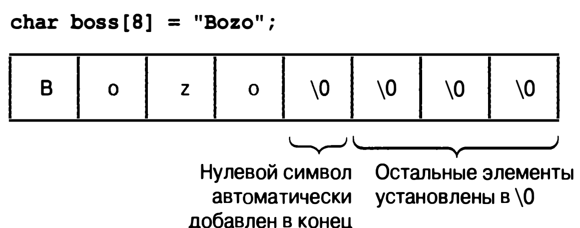


Рис. 4.2. Инициализация массива строкой

Конечно, вы должны обеспечить достаточную величину массива, чтобы в него поместились все символы строки, включая нулевой. Инициализация символьного массива строковой константой — это один из тех случаев, когда безопаснее поручить компилятору подсчет числа элементов массива. Это потому, что функции, которые работают со строками, руководствуются положением нулевого символа, а не размером массива. C++ не накладывает никаких ограничений на длину строки.

На память!

Когда определяется минимальный размер массива для хранения строки, не забудьте включить ограничивающий нулевой символ в общее количество.

Обратите внимание, что строковая константа (с двойными кавычками) не взаимозаменяема с символьными константами (с одинарными кавычками). Символьная

константа вроде 'S' — это сокращенное обозначение для кода символа. В системе ASCII 'S' — просто другой способ написать 83. Поэтому, оператор

```
char shirt_size = 'S'; // так нормально
```

присваивает значение 83 переменной shirt_size. Но "S" представляет строку, состоящую из двух символов — S и \0. Хуже того, "S" в действительности представляет адрес памяти, по которому размещается строка. Поэтому оператор

```
char shirt_size = "S"; // несоответствие типов
```

означает попытку присвоить адрес памяти переменной shirt_size! Поскольку адрес памяти — это отдельный тип в C++, компилятор не пропустит подобную бессмыслицу. (Мы вернемся к этому позднее, когда будем говорить об указателях.)

Конкатенация строковых констант

Иногда строки могут оказаться слишком большими, чтобы удобно разместиться в одной строке кода. C++ позволяет выполнять конкатенацию строковых констант — то есть комбинировать две строки в кавычках в одну. В самом деле, любые две строковые константы, разделенные только пробельным символом (пробелами, символами табуляции и перевода строки), автоматически объединяются в одну. Таким образом, следующие три оператора вывода эквивалентны:

```
cout << "I'd give my right arm to be" " a great violinist.\n";
cout << "I'd give my right arm to be a great violinist.\n";
cout << "I'd give my right ar"
     "m to be a great violinist.\n";
```

Следует отметить, что такие объединения не добавляют никаких пробелов к объединяемым строкам. Первый символ второй строки немедленно следует за последним символом первой, не считая \0 в первой строке. Символ \0 из первой строки заменяется первым символом второй строки.

Использование строк в массивах

Два наиболее распространенных метода помещения строки в массив заключаются в инициализации массива строковой константой и чтением с клавиатуры или из файла в массив. В листинге 4.2 демонстрируются эти подходы за счет инициализации одного массива строкой в кавычках и использования cin для помещения вводимой строки в другой массив. Программа также использует стандартную библиотечную функцию stlen() для получения длины строки. Стандартный заголовочный файл cstring (или string.h в более старых реализациях) представляет объявления для этой и многих других функций, работающих со строками.

Листинг 4.2. strings.cpp

```
// strings.cpp -- сохранение строк в массиве
#include <iostream>
#include <cstring> // для функции strlen()
int main()
{
   using namespace std;
   const int Size = 15;
```

```
    char name1[Size]; // пустой массив
    char name2[Size] = "C++owboy"; // инициализация массива
    // ПРИМЕЧАНИЕ: некоторые реализации могут потребовать ключевого
    // слова static для инициализации массива name2
    cout << "Howdy! I'm " << name2;
    cout << "! What's your name?\n";
    cin >> name1;
    cout << "Well, " << name1 << ", your name has ";
    cout << strlen(name1) << " letters and is stored\n";
    cout << "in an array of " << sizeof(name1) << " bytes.\n";
    cout << "Your initial is " << name1[0] << ".\n";
    name2[3] = '\0'; // нулевой символ
    cout << "Here are the first 3 characters of my name: ";
    cout << name2 << endl;
    return 0;
}
```

Замечание по совместимости

Если ваша система не включает заголовочного файла `cstring`, попробуйте включить более старую версию — `string.h`.

Ниже показан пример выполнения программы из листинга 4.2.

```
Howdy! I'm C++owboy! What's your name?
Basicman
Well, Basicman, your name has 8 letters and is stored
in an array of 15 bytes.
Your initial is B.
Here are the first 3 characters of my name: C++
```

Замечания по программе

Чему учит код в листинге 4.2? Во-первых, тому, что операция `sizeof` возвращает размер всего массива — 15 байт, но функция `strlen()` возвращает размер строки, помещенной в массив, а не размер самого массива. К тому же `strlen()` считает только видимые символы, без нулевого символа-ограничителя. То есть эта функция возвращает значение 8, а не 9, в качестве длины `"Basicman"`. Если `cosmic` — строка, то минимальный размер массива для размещения этой строки вычисляется как `strlen(cosmic) + 1`.

Поскольку `name1` и `name2` — массивы, вы можете использовать индексы для доступа к отдельным символам этих массивов. Например, программа использует `name1[0]` для поиска первого символа этого массива. К тому же программа присваивает `name2[3]` нулевой символ. Это завершает строку после трех символов, хотя в массиве и остаются еще символы (рис. 4.3).

Обратите внимание, что программа в листинге 4.2 применяет символическую константу для указания размера массива. Часто размер массива нужно указать в нескольких операторах программы. Применение символических констант для представления размера массива упрощает внесение изменений, связанных с длиной массива; в таких случаях вам нужно изменить размер только в одном месте — там, где определена символическая константа.

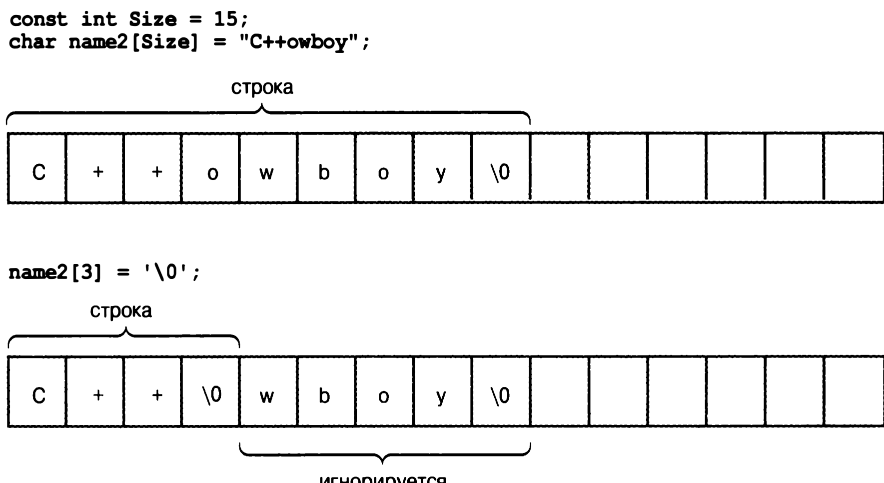


Рис. 4.3. Сокращение размера строки с помощью **\0**

Риск, связанный с вводом строк

Программа `string.cpp` имеет недостаток, скрытый за часто используемой в литературе техникой тщательного выбора примеров ввода. В листинге 4.3 демонстрируется тот факт, что строковый ввод может оказаться непростым.

Листинг 4.3. insrtl.cpp

```
// insrtl.cpp -- чтение более одной строки
#include <iostream>
int main()
{
   using namespace std;
   const int ArSize = 20;
   char name[ArSize];
   char dessert[ArSize];
   cout << "Введите свое имя: \n";
   cin >> name;
   cout << "Введите свой любимый десерт: \n";
   cin >> dessert;
   cout << "У меня есть вкусный " << dessert;
   cout << " для вас, " << name << ".\n";
   return 0;
}
```

Назначение программы из листинга 4.3 простое: прочитать имя пользователя и название его любимого десерта с клавиатуры с последующим отображением информации. Вот пример запуска:

```
Введите свое имя:
Васисуалий Цуккертордт
Введите свой любимый десерт:
У меня есть вкусный Цуккертордт для вас, Васисуалий.
```

Мы даже не получили возможности ответить на вопрос о десерте! Программа показала его и затем немедленно перешла к отображению заключительной строки.

Корень проблемы состоит в том, как cin определяет, где завершается ввод строки. Вы не можете ввести нулевой символ с клавиатуры, поэтому cin требуется что-то другое для нахождения конца строки. Техника cin заключается в использовании пробельных символов для разделения строк — пробелов, знаков табуляции и символов новой строки. Это значит, что cin читает только одно слово, когда получает вход для символьного массива. После того, как он прочитает слово, cin автоматически добавляет ограничивающий нулевой символ, когда помещает строку в массив.

Практический результат этого примера заключается в том, что cin читает слово Васисуалий как полную первую строку и помещает его в массив name. При этом второе слово, Цуккертордт, остается во входной очереди. Когда cin ищет ввод, отвечающий на вопрос о десерте, он находит там Цуккертордт. Затем cin "съедает" Цуккертордт и помещает его в массив dessert. (См. рис. 4.4).

Рис. 4.4. Видение входной строки объектом `cin`

Другая проблема, которая не проявляется при этом простом запуске, состоит в том, что входная строка может оказаться длиннее, чем целевой массив. Применение cin в данном примере не гарантирует никакой защиты от помещения 30-символьной строки в 20-символьный массив.

Многие программы зависят от строкового ввода, поэтому нам стоит рассмотреть эту тему глубже. Мы опишем более подробно некоторые из наиболее усовершенствованных средств cin в главе 17.

Построчное чтение ввода

Чтение строкового ввода по одному слову за раз — часто не самое желательное поведение. Например, предположим, что программа запрашивает пользователя ввести город, и пользователь отвечает New York или San Paulo. Вы хотите, чтобы программа прочитала и сохранила полные названия, а не только New и San. Чтобы иметь возможность вводить целые фразы вместо отдельных слов, необходим другой подход к строковому вводу. Точнее говоря, нужен строчно-ориентированный метод вместо метода, ориентированного на слова. К счастью, у класса istream, экземпляром которого является cin, есть функции-члены, предназначенные для строчно-ориентированного ввода: getline() и get(). Оба читают полную строку ввода — то есть вплоть до символа перевода строки. Однако getline() затем отбрасывает символ перевода строки, в то время как get() оставляет его во входной очереди. Рассмотрим детали, начиная с getline().

Строчно-ориентированный ввод и getline()

Функция getline() читает целую строку, используя символ перевода строки, переданный клавишей <Enter>, для обозначения конца ввода. Этот метод инициируется вызовом функции cin.getline(). Функция принимает два аргумента. Первый аргумент — имя массива назначения, который сохраняет введенную строку, а второй — максимальное количество символов, подлежащих чтению. Если, скажем, установлен предел 20, то функция читает не более 19 символов, оставляя место для автоматически добавляемого в конце нулевого символа. Функция-член getline() прекращает чтение, когда достигает указанного предела количества символов или когда читает символ новой строки — смотря, что случится раньше.

Например, предположим, что вы хотите использовать getline() для того, чтобы прочесть имя в 20-элементный массив name. Для этого следует использовать такой вызов:

```
cin.getline(name,20);
```

Он читает полную строку в массив name, предполагая, что строка состоит не более чем из 19 символов (Функция-член getline() имеет еще необязательный третий аргумент, который обсуждается в главе 17.)

В листинге 4.4 представлен модифицированный пример из листинга 4.3 с применением cin.getline() вместо простого cin. В остальном программа неизменна.

Листинг 4.4. insrt2.cpp

```
// insrt2.cpp -- чтение более одного слова с помощью getline
#include <iostream>
int main()
{
   using namespace std;
   const int ArSize = 20;
   char name[ArSize];
   char dessert[ArSize];
   cout << "Введите ваше имя: \n";
   cin.getline(name, ArSize); // читать до символа новой строки
   cout << "Введите ваш любимый десерт: \n";
   cin.getline(dessert, ArSize);
   cout << "У меня есть вкусный " << dessert;
   cout << " для вас, " << name << ".\n";
   return 0;
}
```

Замечание по совместимости

Некоторые старые версии C++ не полностью реализуют все стороны современного пакета ввода-вывода. В частности, функция-член getline() не везде доступна. Если вы столкнетесь с этим, просто прочтите этот пример и переходите к следующему, который применяет функцию-член, предшествующую getline(). Ранние версии Turbo C++ реализуют getline() немного иначе — так, что она помещает символ перевода строки в массив. Microsoft Visual C 5.0 и 6.0 содержат ошибку — getline() реализована в заголовочном файле iostream, но ее нет в ostream.h; пакет обновлений Service Pack 5 для Microsoft Visual C 6.0, доступный на Web-сайте msdn.microsoft.com/vstdio, исправляет упомянутую ошибку.

Вот пример выполнения программы из листинга 4.4:

```
Введите ваше имя:
Дик Длинныйнос
Введите ваш любимый десерт:
Редисочный торт
У меня есть вкусный Редисочный торт для вас, Дик Длинныйнос.
```

Теперь программа читает полное имя и название блюда. Функция `getline()` удобным образом принимает по одной строке за раз. Она читает ввод до новой символа строки, помечая конец строки, но не сохраняя при этом сам символ новой строки. Вместо этого она заменяет его нулевым символом при сохранении строки (рис. 4.5).

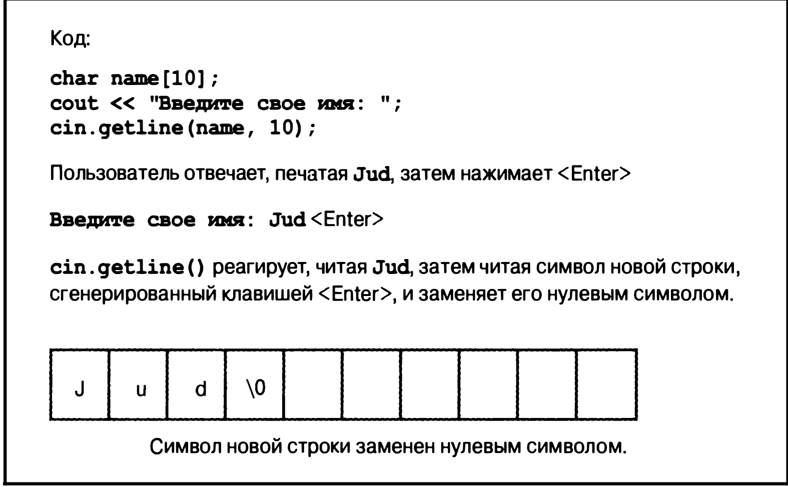


Рис. 4.5. `getline()` читает и заменяет символ новой строки

Строчно-ориентированный ввод и `get()`

Теперь попробуем другой подход. Класс `istream` включает другую функцию-член, `get()`, которая имеет различные варианты. Один из них работает почти так же, как `getline()`. Он принимает те же аргументы, интерпретирует их аналогичным образом, и читает до конца строки. Но вместо того, чтобы прочитать и отбросить символ перевода строки, `get()` оставляет его во входной очереди. Предположим, вы используете два вызова `get()` подряд:

```
cin.get(name, ArSize);
cin.get(dessert, Arsize); // проблема
```

Поскольку первый вызов оставляет символ перевода строки во входной очереди, то получается, что символ новой строки оказывается первым символом, который видит следующий вызов. Таким образом, второй вызов `get()` заключает, что он достиг конца строки, не найдя ничего интересного, что можно было бы прочитать. Без посторонней помощи `get()` просто вообще не может преодолеть этот символ новой строки.

К счастью, на помощь приходят различные варианты get(). Вызов cin.get() без аргументов читает единственный следующий символ, даже если это будет символ новой строки, поэтому вы можете использовать его для того, чтобы отбросить символ новой строки и подготовиться к вводу следующей строки. То есть, следующая последовательность будет работать правильно:

```
cin.get(name, ArSize);      // читать первую строку
cin.get();                  // читать символ перевода строки
cin.get(dessert, Arsize);   // читать вторую строку
```

Другой способ применения get() состоит в конкатенации, или соединении, двух вызовов функций-членов класса, как показано в следующем примере:

```
cin.get(name, ArSize).get();   // конкатенация функций-членов
```

Такую возможность обеспечивает то, что cin.get(name, ArSize) возвращает сам объект cin, который затем используется как объект, вызывающий функцию get(). Аналогично следующий оператор:

```
cin.getline(name1, ArSize).getline(name2, ArSize);
```

читает две последовательных строки в массивы name1 и name2, что эквивалентно двум отдельным вызовам cin.getline().

В листинге 4.5 используется конкатенацию. В главе 11 вы узнаете о том, как включить это средство в ваши собственные определения классов.

Листинг 4.5. insrt3.cpp

```
// insrt3.cpp -- чтение более одного слова с помощью get() и get()
#include <iostream>
int main()
{
   using namespace std;
   const int ArSize = 20;
   char name[ArSize];
   char dessert[ArSize];
   cout << "Введите свое имя: \n";
   cin.get(name, ArSize).get(); // читать строку и символ перевода строки
   cout << "Введите свой любимый десерт: \n";
   cin.get(dessert, ArSize).get();
   cout << "У меня есть вкусный " << dessert;
   cout << " для вас, " << name << ".\n";
   return 0;
}
```

Замечание по совместимости

В некоторых старых версиях C++ вариант get() без аргументов не реализован. Однако в них реализован другой вариант get(), принимающий единственный аргумент типа char. Чтобы использовать его вместо get() без аргументов, потребуется сначала объявить переменную char:

```
char ch;
cin.get(name, ArSize).get(ch);
```

Вы можете применять такой код вместо того, что представлен в листинге 4.5. Различные варианты get() обсуждаются в главах 5, 6 и 17.

Вот пример запуска программы из листинга 4.5:

```
Введите свое имя:
Михаил Самуэльевич
Введите свой любимый десерт:
Свежий кефир
У меня есть вкусный Свежий кефир для вас, Михаил Самуэльевич.
```

Один момент, который следует отметить — то, как C++ допускает существование множества версий функции с разными списками аргументов. Если, скажем, вы используете `cin.get(name, ArSize)`, то компилятор определяет, что вызывается форма, которая помещает строку в массив и подставляет соответствующую функцию-член. Если же вместо этого вы применяете `cin.get()`, то компилятор видит, что вам нужна форма, которая читает один символ. В главе 8 раскрывается это средство, называемое *перегрузкой функций*.

Зачем вообще может понадобиться вызывать `get()` вместо `getline()`? Во-первых, старые реализации могут не иметь `getline()`. Во-вторых, `get()` позволяет вам быть более осторожными. Предположим, например, что вы воспользовались `get()`, чтобы прочитать строку в массив. Как вы можете узнать, была ли прочитана полная строка, или же чтение прервалось в связи с наполнением массива? Для этого нужно посмотреть на следующий в очереди символ. Если это символ новой строки, значит, была прочитана вся строка. В противном случае строка была прочитана не полностью и еще есть что читать. Этот прием исследуется в главе 17. Короче говоря, `getline()` немного проще в применении, но `get()` упрощает проверку ошибок. Вы можете применять любую из этих функций для чтения ввода; просто учитывайте различия в их поведении.

Пустые строки и другие проблемы

Что случается после того, как `getline()` и `get()` читают пустую строку? Изначально предполагалось, что следующий оператор ввода должен получить указание, где завершил работу предыдущий вызов `getline()` или `get()`. Однако современная практика заключается в том, что после того, как функция `get()` (но не `getline()`) читает пустую строку, она устанавливает флажок, который называется `failbit`. Влияние этого флажка состоит в том, что последующий ввод блокируется, но вы можете восстановить его следующей командой:

```
cin.clear();
```

Другая потенциальная проблема связана с тем, что входная строка может быть длиннее, чем выделенное для нее пространство. Если входная строка длиннее, чем указанное количество символов, то и `getline()`, и `get()` оставляют избыточные символы во входной очереди. Однако `getline()` дополнительно устанавливает `failbit` и отключает последующий ввод.

В главах 5, 6 и 17 эти свойства и способы программирования с их учетом исследуются более подробно.

Смешивание строкового и числового ввода

Смешивание числового ввода со строковым может вызывать проблемы. Рассмотрим пример простой программы в листинге 4.6.

Листинг 4.6. numstr.cpp

```
// numstr.cpp -- строковый ввод после числового
#include <iostream>
int main()
{
   using namespace std;
   cout << "В каком году построен ваш дом? \n";
   int year;
   cin >> year;
   cout << "По какому адресу он расположен? \n";
   char address[80];
   cin.getline(address, 80);
   cout << "Год постройки: " << year << endl;
   cout << "Адрес: " << address << endl;
   cout << "Готово! \n";
   return 0;
}
```

В результате выполнения программы из листинга 4.6 получаем:

```
В каком году построен ваш дом?
1966
По какому адресу он расположен?
Год постройки: 1966
Адрес:
Готово!
```

Вы так и не получите возможность ввести адрес. Проблема в том, что когда `cin` читает год, то оставляет символ новой строки, сгенерированный нажатием <Enter>, во входной очереди. Затем `cin.getline()` читает символ новой строки просто как пустую строку, и присваивает нулевую строку массиву `address`. Чтобы исправить это, нужно прочитать и отбросить символ новой строки прежде, чем читать адрес. Это может быть сделано несколькими способами, включая вызов `get()` без аргументов либо с аргументом `char`, как описано в предыдущем примере. Этот вызов можно выполнить отдельно:

```
cin >> year;
cin.get(); // или cin.get(ch);
```

Или же можно сцепить вызов, воспользовавшись тем фактом, что выражение `cin >> year` возвращает объект `cin`:

```
(cin >> year).get(); // или (cin >> year).get(ch);
```

Если внести одно из таких исправлений в листинг 4.6, то он будет работать правильно:

```
В каком году построен ваш дом?
1966
По какому адресу он расположен?
43821 Безымянная Короткая улица
Год постройки: 1966
Адрес: 43821 Безымянная Короткая улица
Готово!
```

Для обработки строк программы C++ часто используют указатели вместо массивов. Мы обратимся к этому аспекту строк после того, как немного поговорим об указателях. А пока рассмотрим более современный способ обработки строк: класс C++ `string`.

Введение в класс `string`

Стандарт ISO/ANSI C++ расширил библиотеку C++, добавив класс `string`. Поэтому отныне вместо применения символьных массивов для хранения строк можно применять переменные типа `string` (или, пользуясь терминологией C++, объекты). Как вы увидите, класс `string` проще в использовании, чем массив, и к тому же предлагает более естественное представление строки как типа.

Чтобы использовать класс `string`, в программе должен быть включен заголовочный файл `string`. Класс `string` является частью пространства имен `std`, поэтому вы должны указать директиву `using` или же обращаться к классу по имени `std::string`. Определение класса скрывает природу строки как массива символов и позволяет трактовать ее как обычную переменную. В листинге 4.7 проиллюстрированы некоторые сходства и различия между объектами `string` и символьными массивами.

Листинг 4.7. `strtype1.cpp`

```
// strtype1.cpp -- использование класса C++ string
#include <iostream>
#include <string> // обеспечить доступ к классу string
int main()
{
    using namespace std;
    char charr1[20];          // создать пустой массив
    char charr2[20] = "ягуар"; // создать инициализированный массив
    string str1;              // создать пустой объект строки
    string str2 = "пантера";  // создать инициализированный объект строки
    cout << "Введите животное из семейства кошачьих: ";
    cin >> charr1;
    cout << "Введите другое животное из семейства кошачьих: ";
    cin >> str1;                // использовать cin для ввода
    cout << "Вот некоторые животные из семейства кошачьих: \n";
    cout << charr1 << " " << charr2 << " "
         << str1 << " " << str2   // для вывода использовать cout
         << endl;
    cout << "Третья буква в слове " << charr2 << " - "
         << charr2[2] << endl;
    cout << "Третья буква в слове " << str2 << " - "
         << str2[2] << endl;     // использовать нотацию массива
    return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.7:

```
Введите животное из семейства кошачьих: оцелот
Введите другое животное из семейства кошачьих: тигр
Вот некоторые животные из семейства кошачьих:
оцелот ягуар тигр пантера
Третья строка в слове ягуар - у
Третья строка в слове пантера - н
```

Из этого примера вы можете сделать вывод, что во многих отношениях объект string можно использовать так же, как символьный массив:

- Объект string можно инициализировать строкой в стиле С.
- Чтобы сохранить клавишный ввод в объекте string, можно использовать cin.
- Для отображения объекта string можно использовать cout.
- Можно использовать нотацию типа массива для доступа к индивидуальным символам, сохраненным в объекте string.

Главное отличие между объектами string и символьными массивами, продемонстрированное в листинге 4.7, заключается в том, что объект string объявляется как обычная переменная, а не массив:

```
string str1;                // создать пустой строковый объект
string str2 = "пантера";  // создать инициализированную строку
```

Дизайн класса позволяет программе обрабатывать автоматически изменение размера строк. Например, объявление str1 создает объект string нулевой длины, но при чтении ввода в str1 программа автоматически увеличивает его:

```
cin >> str1;                // str1 увеличен для того, чтобы вместить ввод
```

Это делает применение объекта string более удобным и безопасным, чем применение массива.

Концептуально главным является то, что массив строк — это коллекция хранилищ отдельных символов, представляющих строку, а класс string — единая сущность, представляющая строку.

Присваивание, конкатенация и добавление

Некоторые операции со строками класс string выполняет проще, чем это возможно в случае символьных массивов. Например, вы не можете просто присвоить один массив другому. Однако один объект string вполне можно присвоить другому:

```
char charr1[20];            // создать пустой массив
char charr2[20] = "jaguar"; // создать инициализированный массив
string str1;                // создать пустой строковый объект
string str2 = "panther";    // создать инициализированную строку
charr1 = charr2;            // НЕ ПРАВИЛЬНО, присваивание массива невозможно
str1 = str2;                // ПРАВИЛЬНО, присваивание объектов допускается
```

Класс string упрощает комбинирование строк. Вы можете применить операцию + для сложения двух объектов string вместе, и операцию += для того, чтобы сцепить строку с существующим объектом string. В отношении предшествующего кода у нас есть следующие возможности:

```
string str3;
str3 = str1 + str2;         // присвоить str3 объединение строк
str1 += str2;               // добавить str2 в конец str1
```

В листинге 4.8 приводится соответствующая иллюстрация. Обратите внимание, что вы можете складывать и добавлять к объектам string как другие объекты string, так и строки в стиле С.

Листинг 4.8. strtype2.cpp

```
// strtype2.cpp -- присваивание, сложение, добавление
#include <iostream>
#include <string> // обеспечить доступ к классу string
int main()
{
    using namespace std;
    string s1 = "penguin";
    string s2, s3;
    cout << "Вы можете присвоить один объект другому: s2 = s1\n";
    s2 = s1;
    cout << "s1: " << s1 << ", s2: " << s2 << endl;
    cout << "Вы можете присвоить объекту string строку в стиле C. \n";
    cout << "s2 = \"buzzard\"\n";
    s2 = "buzzard";
    cout << "s2: " << s2 << endl;
    cout << "Вы можете сцеплять строки: s3 = s1 + s2\n";
    s3 = s1 + s2;
    cout << "s3: " << s3 << endl;
    cout << "Вы можете добавлять строки. \n";
    s1 += s2;
    cout <<"s1 += s2 yields s1 = " << s1 << endl;
    s2 += " for a day";
    cout <<"s2 += \" for a day\" yields s2 = " << s2 << endl;
    return 0;
}
```

Вспомните, что управляющая последовательность `\"` представляет двойную кавычку, используемую как литеральный символ, а не как ограничитель строки. Ниже показан вывод программы из листинга 4.8:

```
Вы можете присвоить один объект другому: s2 = s1
s1: penguin, s2: penguin
Вы можете присвоить объекту string строку в стиле C.
s2 = "buzzard"
s2: buzzard
Вы можете сцеплять строки: s3 = s1 + s2
s3: penguinbuzzard
Вы можете добавлять строки.
s1 += s2 yields s1 = penguinbuzzard
s2 += " for a day" yields s2 = buzzard for a day
```

Дополнительные сведения об операциях над классом string

Еще до появления в C++ класса `string` программисты нуждались в таких операциях, как присваивание строк. Для строк в стиле C использовались функции из стандартной библиотеки C. Заголовочный файл `cstring` (бывший `string.h`) поддерживает эти функции. Например, вы можете применять функцию `strcpy()` для копирования строки в символьный массив, а функцию `strcat()` — для добавления строки к символьному массиву:

```
strcpy(charr1, charr2); // копировать charr2 в charr1
strcat(charr1, charr2); // добавить содержимое charr2 к char1
```

В листинге 4.9 сравнивается техника, предусматривающая использование объектов `string`, с техникой работы с символьными массивами.

Листинг 4.9. strtype3.cpp

```
// strtype3.cpp -- дополнительно о средствах класса string
#include <iostream>
#include <string> // обеспечить доступ к классу string
#include <cstring> // библиотека обработки строк в стиле C
int main()
{
    using namespace std;
    char charr1[20];
    char charr2[20] = "jaguar";
    string str1;
    string str2 = "panther";
    // присваивание объектов string и символьных массивов
    str1 = str2;              // копировать str2 в str2
    strcpy(charr1, charr2);   // копировать charr2 в charr1
    // добавление объектов string и символьных массивов
    str1 += " paste";         // добавить "paste" в конец str1
    strcat(charr1, " juice"); // добавить "juice" в конец charr1
    // определение длины объекта string и строки в стиле C
    int len1 = str1.size();    // получить длину str1
    int len2 = strlen(charr1);   // получить длину charr1
    cout << "Строка " << str1 << " содержит "
        << len1 << " символов. \n";
    cout << "Строка " << charr1 << "содержит "
        << len2 << "символов. \n";
    return 0;
}
```

Ниже представлен вывод программы из листинга 4.9:

```
Строка panther paste содержит 13 символов.
Строка jaguar juice содержит 12 символов.
```

Работа со строковыми объектами проще, чем использование строковых функций C. Это особенно проявляется в более сложных операциях. Например, библиотечный эквивалент оператора

```
str3 = str1 + str2;
```

будет таким:

```
strcpy(charr3, charr1);
strcat(charr3, charr2);
```

Более того, когда имеешь дело с массивами, всегда существует опасность, что целевой массив окажется слишком малым для того, что вместить всю информацию. Например:

```
char site[10] = "house";
strcat(site, " of pancakes"); // проблема памяти
```

Функция strcat() пытается скопировать 12 символов в массив site, таким образом, переполняя выделенную память. Это может вызвать прерывание программы, или программа продолжит работать, но с поврежденными данными. Класс string, с его автоматическим расширением при необходимости, избегает проблем подобного рода. Библиотека C предлагает функции, подобные strcat() и strcpy(), называемые strncat() и strncpy(), которые работают более безопасно, принимая третий параметр, задающий максимально допустимый размер целевого массива, но их применение усложняет написание программ.

Обратите внимание, что для получения количества символов в строке используется разный синтаксис:

```
int len1 = str1.size();      // получить длину str1
int len2 = strlen(charr1);   // получить длину charr1
```

Функция strlen() — стандартная функция, которая принимает в качестве аргумента строку в стиле C и возвращает количество символов в ней. Функция size() обычно делает то же самое, но синтаксис ее вызова отличается. Вместо передачи аргумента ее имени предшествует имя объекта str1, отделенное точкой. Как вы уже видели на примере метода put() в главе 3, этот синтаксис означает, что str1 — это объект, а size() — метод его класса. Метод — это функция, которая может быть вызвана только объектом — экземпляром класса, в котором определен данный метод. В данном конкретном случае str1 — объект типа string, а size() — метод string. Короче говоря, функции C используют аргументы для идентификации требуемой строки, а объект класса C++ string использует имя объекта и операцию точки для указания того, какую именно строку нужно взять.

Дополнительные сведения о строковом вводе-выводе

Как вы уже видели, можно использовать cin с операцией >> для чтения объекта string и cout с операцией << — для отображения объекта string, причем с тем же синтаксисом, что и в случае строк в стиле C. Но чтение за раз целой строки с пробелами вместо отдельного слова требует другого синтаксиса. В листинге 4.10 демонстрируется это отличие.

Листинг 4.10. strtype4.cpp

```
// strtype4.cpp -- ввод строки с пробелами
#include <iostream>
#include <string>   // обеспечить доступ к классу string
#include <cstring>  // библиотека обработки строк в стиле C
int main()
{
    using namespace std;
    char charr[20];
    string str;
    cout << "Длина строки charr перед вводом: "
         << strlen(charr) << endl;
    cout << "Длина строки str перед вводом: "
         << str.size() << endl;
    cout << "Введите строку текста: \n";
```

```
    cin.getline(charr, 20); // указать максимальную длину
    cout << "Вы ввели: " << charr << endl;
    cout << "Введите другую строку текста: \n";
    getline(cin, str); // теперь cin - аргумент; спецификатор длины отсутствует
    cout << "Вы ввели: " << str << endl;
    cout << "Длина строки charr после ввода: "
         << strlen(charr) << endl;
    cout << "Длина строки str после ввода: "
         << str.size() << endl;
    return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.10:

```
Длина строки charr перед вводом: 27
Длина строки str перед вводом: 0
Введите строку текста:
peanut butter
Вы ввели: peanut butter
Введите другую строку текста:
blueberry jam
Вы ввели: blueberry jam
Длина строки charr после ввода: 13
Длина строки str после ввода: 13
```

Обратите внимание, что программа сообщает длину строки для массива `charr` перед вводом, как равную 27, что больше, чем размер массива! Здесь происходят две вещи. Первая — содержимое неинициализированного массива не определено. Вторая — функция `strlen()` работает, просматривая массив, начиная с первого элемента, и подсчитывает количество байт до тех пор, пока не встретит нулевой символ. В этом случае первый нулевой символ встретился через несколько байт за пределами массива. Где именно встретится нулевой символ в неинициализированном массиве, определяется случаем, поэтому очень может быть, что получите другое значение, запустив эту программу у себя на компьютере.

Также отметьте, что длина строки `str` перед вводом равна 0. Это объясняется тем, что размер неинициализированного объекта `string` автоматически устанавливается равным 0.

Следующий код читает строку в массив:

```
cin.getline(charr, 20);
```

Точечная нотация указывает на то, что функция `getline()` — это метод класса `istream`. (Напомним, что `cin` — объект класса `istream`.) Как упоминалось ранее, первый аргумент задает целевой массив, а второй — его размер, используемый `getline()` для того, чтобы избежать переполнения массива.

Следующий код читает строку в объект `string`:

```
getline(cin,str);
```

Здесь точечная нотация не используется, а это говорит о том, что данный `getline()` не является методом класса. Поэтому он принимает объект `cin` как аргумент, сообщающий ему о том, где искать ввод. К тому же нет аргумента, задающего

размер строки, потому что объект string автоматически изменяет свой размер, чтобы вместить строку.

Так почему же один getline() — метод класса istream, а второй — нет? Класс istream появился в С++ до того, как был добавлен класс string. Поэтому дизайн istream распознает базовые типы С++, такие как double или int, но ничего не знает о типе string. Потому-то у класса istream есть методы, обрабатывающие double, int и другие базовые типы, но нет методов, обрабатывающих объекты string.

Поскольку у класса istream нет методов, обрабатывающих объекты string, вас может удивить, почему работает следующий код:

```
cin >> str;   // читать слово в str - объект типа string
```

Ведь такой код

```
cin >> x;     // читать значение в переменную базового типа C++
```

использует (в скрытом виде) функцию-член класса istream. Но эквивалент этого выражения с классом string использует дружественную функцию (также в скрытой нотации) класса string. Вам придется подождать до главы 11 объяснений — что такое дружественная функция и как эта техника работает. А между тем, пока вы смело можете использовать cin и cout с объектами string и не заботиться о том, как оно там все внутри работает.

Теперь перейдем к другим составным типам — к структурам.

Введение в структуры

Предположим, вы хотите хранить информацию о баскетболисте. Вы хотите хранить его имя, зарплату, рост, вес, среднюю результативность, процент попаданий, результативных передач и тому подобное. Вам понадобится некоторая форма данных, которая могла бы хранить всю эту информацию как единое целое. Массив здесь не подходит. Хотя массив может хранить несколько элементов, но все они должны быть одного типа. То есть один массив может хранить 20 целых чисел, а другой — 10 чисел с плавающей точкой, но он не может хранить целые значения в одних элементах и значения с плавающей точкой — в других.

Удовлетворить вашу потребность в совместном хранении всей информации о баскетболисте может структура С++. *Структура* — более многосторонняя форма данных, чем массив, потому что одна структура может хранить элементы более чем одного типа. Это позволяет унифицировать представление данных за счет сохранения всей информации, связанной с баскетболистом, в одной структурной переменной. Если вы хотите отслеживать информацию о целой команде, то можете воспользоваться массивом структур. Тип структуры — это еще и ступенька к покорению бастиона объектно-ориентированного программирования С++ - класса. Изучение структур приблизит вас к сердцу ООП на языке С++.

Структура — определяемый пользователем тип с объявлением, описывающим свойства типа. После того, как вы определите тип, вы можете создавать переменные этого типа. То есть, создание структуры — двухфазный процесс. Во-первых, вы определяете описание структуры, перечисляющей и именующей типы данных, которые могут быть сохранены в структуре. Затем вы создаете структурные переменные, или, иначе говоря, структурные объекты данных, которые следуют плану, заданному объявлением.

Например, предположим, что фирма Bloataire, Inc. желает создать тип данных, описывающий линейку ее продуктов — различного рода надувных предметов. В частности, тип должен включать наименование продукта, его объем в кубических футах, а также цену продажи. Вот описание структуры, отвечающее этим потребностям:

```
struct inflatable // объявление структуры
{
   char name[20];
   float volume;
   double price;
};
```

Ключевое слово `struct` указывает на то, что этот код определяет план структуры. Идентификатор `inflatable` — имя, или *дескриптор* этой формы, то есть имя нового типа. Таким образом, теперь вы можете создавать переменные типа `inflatable` точно так же, как вы создаете переменные типа `char` или `int`. Далее между фигурными скобками находится список типов данных, которые будут содержаться в структуре. Каждый элемент списка — это оператор объявления. Вы можете использовать здесь любые типы C++, включая массивы и другие структуры. В этом примере применяется массив `char`, который подходит для хранения строки, затем идет один элемент типа `float` и один — типа `double`. Каждый индивидуальный элемент в списке называется *членом* структуры. Таким образом, структура `inflatable` имеет три члена. (См. рис. 4.6.)

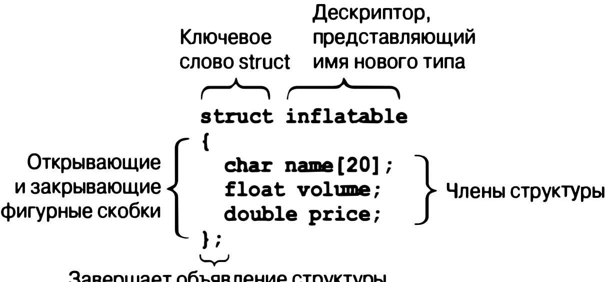


Рис. 4.6. Части описания структуры

После того, как у вас объявлен шаблон, вы можете создавать переменные этого типа:

```
inflatable hat;             // hat — структурная переменная типа inflatable
inflatable woopie_cushion;  // переменная типа inflatable
inflatable mainframe;       // переменная типа inflatable
```

Если вы знакомы со структурами C, то отметите (возможно, с удовольствием), что C++ позволяет отбросить ключевое слово `struct` при объявлении структурных переменных:

```
struct inflatable goose;    // ключевое слово struct требуется в C
inflatable vincent;         // ключевое слово struct не требуется в C++
```

В C++ дескриптор структуры используется как фундаментальное имя типа. Это изменение подчеркивает, что объявление структуры определяет новый тип. Кроме того, оно исключает пропуск слова `struct` из списка причин выдачи сообщений об ошибках компилятора.

Если переменная hat имеет тип inflatable, для доступа к ее отдельным членам вы используете операцию принадлежности, или членства (.). Например, hat.volume ссылается на член структуры по имени volume, а hat.price — на член по имени price. Аналогично, vincent.price — член price переменной vincent. Короче говоря, имена членов позволяют вам ссылаться на члены структур, почти так же, как индексы – на элементы массивов. Поскольку член price объявлен как double, hat.price и vincent.price обе являются эквивалентами переменной типа double и могут быть использованы точно так же, как любая другая переменная типа double. Кстати, метод, применяемый для доступа к функции-члену класса, такой как cin.getline(), происходит от метода доступа к переменным-членам структуры вроде vincent.price.

Использование структур в программах

Теперь, когда мы раскрыли некоторые из основных свойств структур, пришло время собрать все идеи вместе в виде программы, использующей переменные-структуры. В листинге 4.11 представлен пример такой программы. Здесь также показано, как их инициализировать.

Листинг 4.11. structur.cpp

```
// structur.cpp -- простая структура
#include <iostream>
struct inflatable // объявление структуры
{
    char name[20];
    float volume;
    double price;
};
int main()
{
  using namespace std;
  inflatable guest =
  {
    "Glorious Gloria",   // значение name
    1.88,                // значение volume
    29.99                // значение value
  }; // guest — структурная переменная типа inflatable
  // инициализируется указанными значениями
  inflatable pal =
  {
    "Audacious Arthur",
    3.12,
    32.99
  }; // pal — вторая переменная типа inflatable
  // ПРИМЕЧАНИЕ: некоторые реализации требуют
  // static inflatable guest =
  cout << "Expand your guest list with " << guest.name;
  cout << " and " << pal.name << "!\n";
  // pal.name - член name переменной pal
  cout << "You can have both for $";
  cout << guest.price + pal.price << "!\n";
  return 0;
}
```

Замечание по совместимости

Как некоторые старые версии C++ не реализуют возможности инициализации обычных массивов, объявленных внутри функции, точно так же они не реализуют возможности инициализации обычных структур, объявленных в функциях. Опять-таки решение состоит в применении к объявлению ключевого слова `static`.

Ниже показан вывод программы из листинга 4.11:

```
Expand your guest list with Glorious Gloria and Audacious Arthur!
You can have both for $62.98!
```

Замечания по программе

Относительно программы в листинге 4.11 следует отметить одно важное обстоятельство — местоположение объявления структуры. Для `structur.cpp` существует два варианта. Можно поместить объявление внутри функции `main()`. Второй вариант, использованный здесь, состоит в том, чтобы поместить объявление вне функции `main()`. Когда объявление встречается вне функции, оно называется *внешним объявлением*. Для данной конкретной программы нет практической разницы между этими двумя вариантами. Но для программ, состоящих из двух и более функций, разница может оказаться существенной. Внешнее объявление может быть использовано всеми функциями, которые следуют за ней, в то время как внутреннее объявление может быть использовано только той функцией, в которой это объявление находится. Чаще всего вам придется применять внешнее объявление, чтобы все функции могли использовать структуры этого типа (рис. 4.7).

```
#include <iostream>
using namespace std;
struct parts
{
 unsigned long part_number;
 float part_cost;
};
void mail();
int mail();
{
 struct  perks
 {
  int key_number;
  char car[12];
 };
 parts chicken;
 perks mr_blug;
 ...
 ...
}
void mail();
{
 parts studebaker;
 ...
 ...
}
```

Внешнее объявление — может быть использовано всеми функциями в файле

Локальное объявление — может бытьиспользовано только данной функцией

Переменная типа **parts**

Переменная типа **perks**

Переменная типа **parts**

Здесь нельзя объявить переменную типа **perks**

Рис. 4.7. Локальные и внешние объявления структур

Переменные также могут быть определены как внутренние или внешние, причем внешние переменные доступны всем функциям. (В главе 9 эту тема рассматривается более подробно.) Практика C++ не одобряет использования внешних переменных, но приветствует применение внешних объявлений структур. К тому же часто имеет смысл объявлять внешними и символические константы.

Теперь обратите внимание на процедуру инициализации:

```
inflatable guest =
{
   "Glorious Gloria",  // значение name
   1.88,               // значение volume
   29.99               // значение value
};
```

Как и с массивами, вы используете список значений, разделенный запятыми внутри пары фигурных скобок. Программа размещает по одному значению в строке, но их все можно было бы поместить и в одну строку. Главное — не забудьте разделить их запятыми:

```
inflatable duck = {"Daphne", 0.12, 9.98};
```

Вы можете инициализировать каждый член структуры данными соответствующего вида. Например, член структуры `name` — это символьный массив, поэтому вы можете инициализировать его строкой.

Каждый член структуры трактуется как переменная данного типа. То есть `pal.price` — переменная типа `double`, а `pal.name` — массив `char`. И когда программа использует `cout`, чтобы отобразить `pal.name`, она отображает этот член как строку. Кстати, поскольку `pal.name` — символьный массив, для доступа к его отдельным символам можно использовать индексы. Например, `pal.name[0]` содержит символ `A`. Однако `pal[0]` не имеет смысла, потому что `pal` — структура, а не массив.

Может ли структура содержать член типа `string`?

Можем ли мы использовать объект класса `string` вместо символьного массива в качестве типа члена `name`? То есть, можно ли объявить структуру следующим образом:

```
#include <string>
struct inflatable      // шаблон структуры
{
   std::string name;
   float volume;
   double price;
};
```

В принципе, ответ положительный. Однако на практике это зависит от используемого компилятора, потому что некоторые компиляторы (включая Borland C++ 5.5 и Microsoft Visual C++ до версии 7.1) не поддерживают инициализацию структур членами типа класса `string`.

Если же ваш компьютер поддерживает такое использование `string`, обеспечьте, чтобы определение структуры имело доступ к пространству имен `std`. Вы можете

сделать это, поместив директиву using таким образом, чтобы она была над определением структуры. В качестве альтернативы, как было показано ранее, вы можете объявить тип name как std::string.

Прочие свойства структур

C++ делает пользовательские типы, насколько возможно, похожими на типы встроенные. Например, вы можете передавать структуру в качестве аргумента функции, и функция может использовать структуру в качестве возвращаемого значения. Также вы можете использовать операцию присваивания (=), чтобы присвоить одну структуру другой, того же самого типа. Эта операция устанавливает значение каждого члена одной структуры равным каждому соответствующему члену другой структуры, даже если член является массивом. Такой тип присваивания называется *почленным присваиванием*. Мы отложим разговор о передаче и возврате структур до обсуждения темы функций в главе 7, но приведем небольшой пример присваивания структур в следующем листинге 4.12.

Листинг 4.12. assign_st.cpp

```
// assign_st.cpp -- присваивание структур
#include <iostream>
struct inflatable
{
   char name[20];
   float volume;
   double price;
};
int main()
{
   using namespace std;
   inflatable bouquet =
   {
     "sunflowers",
     0.20,
     12.49
   };
   inflatable choice;
   cout << "bouquet: " << bouquet.name << " for $";
   cout << bouquet.price << endl;
   choice = bouquet; // присвоить одну структуру другой
   cout << "choice: " << choice.name << " for $";
   cout << choice.price << endl;
   return 0;
}
```

Ниже — пример вывода программы из листинга 4.12:

```
bouquet: sunflowers for $12.49
choice: sunflowers for $12.49
```

Как видите, почленное присваивание работает, и все члены структуры choice получили соответствующие значения членов структуры bouquiet.

Можно комбинировать определение формы структуры с созданием структурных переменных. Чтобы сделать это, сразу после закрывающей фигурной скобки нужно указать имя переменной или нескольких переменных:

```
struct perks
{
   int key_number;
   char car[12];
} mr_smith, ms_jones; // две переменных типа perks
```

Можно даже инициализировать созданную переменную следующим образом:

```
struct perks
{
   int key_number;
   char car[12];
} mr_glitz =
{
   7,          // значение члена mr_glitz.key_number
   "Packard"   // значение члена mr_glitz.car
};
```

Однако отделение определения структуры от объявлений переменных обычно повышает читабельность программы.

Еще один трюк, который можно сделать со структурой — создать структуру без имени типа. При определении пропускается имя дескриптора и сразу следует имя переменной:

```
struct // дескриптора нет
{
   int x;   // 2 члена
   int y;
} position; // структурная переменная
```

Это создает одну структурную переменную по имени `position`. К ее членам можно обращаться через операцию точки, как в `position.x`, но никакого общего имени типа не объявляется. Вы не сможете последовательно создать другие переменные того же типа. В этой книге мы не будем использовать эту ограниченную форму структур.

Помимо того факта, что программа C++ может использовать тег структуры в качестве имени типа, все остальные свойства структур присущи как структурам C, так и C++. Но структуры C++ двигаются дальше. В отличие от структур C, например, структуры C++ могут включать в себя функции-члены в дополнение к переменным-членам. Однако эти более развитые средства чаще используются с классами, чем со структурами, поэтому мы поговорим о них, когда раскроем тему классов — в главе 10.

Массивы структур

Структура `inflatable` включает в себя массив (член `name`). Также можно создавать массивы, чьи элементы являются структурами. Техника в точности совпадает с принятой для массивов базовых типов. Например, чтобы создать массив из 100 структур `inflatable`, можно поступить так:

```
inflatable gifts[100]; // массив из 100 структур inflatable
```

Это делает gifts массивом структур inflatable. Потому каждый элемент массива, такой как gifts[0] или gifts[99], является объектом типа inflatable, и может быть использован с операцией членства:

```
cin >> gifts[0].volume;         //используется член volume первой структуры
cout << gifts[99].price << endl; //отображается член price последней структуры
```

Имейте в виду, что сам по себе gifts является массивом, а не структурой, поэтому конструкция вроде gifts.price — некорректна.

Для инициализации массива структур комбинируется правило инициализации массивов (заключенный в фигурные скобки, разделенный запятыми список значений каждого элемента) с правилом структур (заключенный в фигурные скобки, разделенный запятыми список значений каждого члена). Поскольку каждый элемента массива является структурой, его значение представляется инициализацией структуры. Таким образом, получаем следующую конструкцию:

```
inflatable guests[2] =          // инициализация массива структур
{
   {"Bambi", 0.5, 21.99},       // первая структура в массиве
   {"Godzilla", 2000, 565.99}   // следующая структура в массиве
};
```

Как обычно, вы можете форматировать все это по своему усмотрению. Например, обе инициализации могут быть расположены в одной строке или же инициализация каждого отдельного члена структуры может занимать отдельную строку.

В листинге 4.13 показан пример использования массива структур. Обратите внимание, что поскольку guests является массивом inflatable, тип элемента guests[0] — inflatable, поэтому вы можете использовать его с операцией точки для доступа к членам структуры inflatable.

Листинг 4.13. arrstruc.cpp

```
// arrstruc.cpp -- массив структур
#include <iostream>
struct inflatable
{
   char name[20];
   float volume;
   double price;
};
int main()
{
     using namespace std;
     inflatable guests[2] =         // инициализация массива структур
     {
        {"Бэмби", 0.5, 21.99},      // первая структура в массиве
        {"Годзилла", 2000, 565.99} // следующая структура в массиве
     };
     cout << "Гости " << guests[0].name << " и " << guests[1].name
          << "\nимеют общий объем в "
          << guests[0].volume + guests[1].volume << " кубических футов. \n";
     return 0;
}
```

Ниже показан вывод этой программы:

```
Гости Bambi and Godzilla
имеют общий объем в 2000.5 кубических футов.
```

Битовые поля в структурах

Язык C++, как и C, позволяет специфицировать члены структур, занимающие определенное число бит памяти. Это может пригодиться для создания структур данных, которые подходят, скажем, для регистрации на некотором аппаратном устройстве. Тип поля должен быть целочисленным или перечислимым (перечисления обсуждаются позднее в настоящей главе) и после него вслед за двоеточием ставится число, указывающее действительное количество бит. Для выравнивания можно применять безымянные поля. Каждый член называется *битовым полем*. Вот пример:

```
struct torgle_register
{
   unsigned int SN : 4;    // 4 бита для значения SN
   unsigned int : 4;       // 4 бита не используются
   bool goodIn : 1;        // допустимый ввод (1 бит)
   bool goodTorgle : 1;    // признак успешности
};
```

Вы можете инициализировать поля в обычной манере и использовать стандартную нотацию структур для доступа к битовым полям:

```
torgle_register tr = { 14, true, false };
...
if (tr.goodIn) // оператор if описан в главе 6
...
```

Битовые поля обычно применяются в низкоуровневом программировании. Случаи применения интегральных типов и битовых операций перечислены в приложении Д.

Объединения

Объединение — это формат данных, который может хранить разные типы данных в пределах одной области памяти, но в каждую единицу времени только один из них. То есть, в то время как структура может содержать, скажем, `int` *и* `long`, *и* `double`, объединение может содержать `int` *или* `long`, *или* `double`. Синтаксис похож на синтаксис структур, но смысл отличается. Например, рассмотрим следующее объявление:

```
union one4all
{
   int int_val;
   long long_val;
   double double_val;
};
```

Вы можете использовать переменную `one4all` для хранения `int`, `long` или `double`, если делать это не одновременно:

```
one4all pail;
pail.int_val = 15;        // сохранить int
```

```
cout << pail.int_val;
pail.double_val = 1.38;   // сохранить double, int потеряно
cout << pail.double_val;
```

Таким образом, pail может служить как переменная int в одном случае и как переменная double — в другом. Имя члена идентифицирует роль, в которой в данный момент выступает переменная. Поскольку объединение хранит только одно значение в единицу времени, оно должно иметь достаточный размер, чтобы вместить самый большой член. Поэтому размер объединения определяется размером его самого большого члена.

Причиной применений объединения может быть необходимость сэкономить память, когда элемент данных может использовать два или более форматов, но никогда — одновременно. Например, предположим, что вы ведете реестр каких-то предметов, некоторые из которых имеют целочисленный идентификатор, а некоторые — строковый. В этом случае можно применить следующий подход:

```
struct widget
{
   char brand[20];
   int type;
   union id              // формат зависит от типа предмета
   {
      long id_num;       // предметы первого типа
      char id_char[20];  // прочие предметы
   } id_val;
};
...
widget prize;
...
if (prize.type == 1)            // оператор if-else (Глава 6)
   cin >> prize.id_val.id_num; //использовать поле name для указания режима
else
   cin >> prize.id_val.id_char;
```

Анонимные объединения не имеют имен; по сути, их члены становятся переменными, расположенными по одному адресу в памяти. Естественно, только одна из них может быть текущей в единицу времени:

```
struct widget
{
   char brand[20];
   int type;
   union // anonymous union
   {
      long id_num; // предметы первого типа
      char id_char[20]; // прочие предметы
   };
};
...
widget prize;
...
if (prize.type == 1)
   cin >> prize.id_num;
else
   cin >> prize.id_char;
```

Поскольку объединение анонимно, id_num и id_char трактуются как два члена prize, разделяющие один и тот же адрес памяти. Необходимость в промежуточном идентификаторе id_val исключается. Какое поле активно в каждый момент времени остается на усмотрение программиста.

Перечисления

Средство C++ enum представляет собой альтернативный const способ создания символьных констант. Он также позволяет определять новые типы, но в очень ограниченной манере. Синтаксис enum подобен синтаксису структур. Например, рассмотрим следующий оператор:

```
enum spectrum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};
```

Этот оператор делает две вещи:

- Объявляет имя нового типа — spectrum; spectrum объявлен как *перечисление*, почти так же, как переменная struct называется структурой.
- Устанавливает red, orange, yellow и так далее как символические константы для целочисленных значений 0–7. Эти константы называются *перечислителями*.

По умолчанию перечислителям присваиваются целые значения, начиная с 0 для первого из них, 1 — для второго и так далее. Вы можете переопределить это правило по умолчанию, явно присвоив целочисленные значения. Чуть позже вы увидите, как это делается.

Вы можете использовать имя перечисления для объявления переменной типа этого перечисления:

```
spectrum band;        // band — переменная типа spectrum
```

Переменные типа перечислений имеют некоторые специальные свойства, которые мы сейчас рассмотрим.

Единственные допустимые значения, которые можно присвоить переменной типа перечисления без необходимости приведения типов — это те значения, которые использованы в определении этого перечисления. То есть, мы имеем следующее:

```
band = blue;          // правильно, blue - перечислитель
band = 2000;          // неправильно, 2000 — не перечислитель
```

Таким образом, переменная spectrum ограничена только восемью допустимыми значениями. Некоторые компиляторы выдают ошибку, если вы пытаетесь присвоить некорректное значение, в то время как другие лишь выдают предупредительные сообщения. Для максимальной переносимости вы должны трактовать присваивание значений, не являющихся enum, переменным типа enum как ошибку.

Для перечислений определена только операция присваивания. В частности, арифметические операции не предусмотрены:

```
band = orange;        // правильно
++band;               // неправильно, ++ обсуждается в главе 5
band = orange + red;// неправильно, но довольно хитро
...
```

Однако некоторые реализации не накладывают таких ограничений. Это позволяет нарушить ограничения типа. Например, если band равно ultraviolet, или 7, а затем выполняется ++band, и если такое разрешено компилятором, то band получает значение, недопустимое для типа spectrum. Опять-таки, для достижения максимальной переносимости, вы должны придерживаться ограничений.

Перечисления — целочисленные типы, и они могут быть представлены в виде int, однако тип int не преобразуется автоматически в тип перечисления:

```
int color = blue;    // правильно, тип spectrum приводится к int
band = 3;            // неправильно, int не преобразуется в spectrum
color = 3 + red;     // правильно, red преобразуется в int
...
```

Обратите внимание, что в этом примере, даже несмотря на то, что значение 3 соответствует перечислителю green, все же присваивание 3 переменной band вызывает ошибку несоответствия типа. Но присваивание green переменной band законно, потому что оба они имеют тип spectrum. И снова, некоторые реализации не накладывают такого ограничения. В выражении 3 + red сложение не определено для перечислений. Однако red конвертируется в тип int, а в результате получается значение типа int. Благодаря преобразованию перечисления в int в данной ситуации, вы можете использовать перечислители в арифметических выражениях, комбинируя их с обычными целыми, даже несмотря на то, что такая арифметика не определена для самих перечислителей. Предыдущий пример

```
band = orange + red; // неправильно, но довольно хитро
```

не работает по другой причине. Верно, что операция + не определена для перечислителей. Но также верно, что перечислители преобразуются в целые, когда применяются в арифметических выражениях, поэтому выражение orange + red превращается в 1 + 0, что вполне корректно. Но это выражение имеет тип int, поэтому оно не может быть присвоено переменной band типа spectrum.

Вы можете присвоить значение int переменной enum, если полученное значение правильно и применяется явное приведение типа:

```
band = spectrum(3); // приведение 3 к типу spectrum
```

Но что будет, если вы попытаетесь выполнить приведение типа для неправильного значения? Результат не определен, в том смысле, что попытка не будет воспринята как ошибочная, но вы не можете полагаться на полученное в результате значение:

```
band = spectrum(40003); // неопределенность
```

(Дискуссию относительно приемлемых и неприемлемых значений см. в разделе "Диапазоны значений перечислителей" далее в настоящей главе.)

Как видите, правила, которым подчиняются перечисления, достаточно строги. На практике перечисления чаще используются как способ определения взаимосвязанных символических констант, нежели как средство определения новых типов. Например, вы можете использовать перечисления для определения символических констант для операторов switch. (См. примеры в главе 6.) Если вы собираетесь только использовать константы и не создавать переменные перечислимого типа, то в этом случае можете пропустить имя перечислимого типа, как показано ниже:

```
enum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};
```

Установка значений перечислителей

Конкретные значения элементов перечислений можно устанавливать явно посредством операция присваивания:

```
enum bits{one = 1, two = 2, four = 4, eight = 8};
```

Присваиваемые значения должны быть целыми. Явно устанавливать можно лишь некоторые из перечислителей:

```
enum bigstep{first, second = 100, third};
```

В этом случае `first` получает значение `0` по умолчанию. Каждый последующий неинициализированный перечислитель увеличивается на единицу по сравнению с предыдущим. Поэтому `third` имеет значение `101`.

И, наконец, вы можете устанавливать более одного перечислителя с одним и тем же значением:

```
enum {zero, null = 0, one, numero_uno = 1};
```

Здесь и `zero`, и `null` имеют значение `0`, а `one` и `numero_uno` — значение `1`. В ранних версиях C++ элементам перечислений можно было присваивать только значения типа `int` (или неявно преобразуемые к `int`), но теперь это ограничение снято, и также можно использовать значения типа `long`.

Диапазоны значений перечислителей

Изначально правильными значениями перечислений являются лишь те, что названы в объявлении. Однако C++ расширяет список допустимых значений, которые могут быть присвоены перечислимым переменным, за счет приведения типа. Каждое перечисление имеет *диапазон*, и, используя приведение типа, вы можете присвоить любое целое значение в пределах этого диапазона, даже если данное значение не равно ни одному из перечислителей. Например, предположим, что `bits` и `myflag` определены следующим образом:

```
enum bits{one = 1, two = 2, four = 4, eight = 8};
bits myflag;
```

В таком случае показанный ниже оператор является правильным:

```
myflag = bits(6); // правильно, потому что 6 лежит в пределах диапазона
```

Здесь `6` не является значением ни одного из перечислителей, однако лежит в пределах данного определенного перечисления.

Диапазон задается следующим образом. Во-первых, чтобы определить верхний предел, выбирается перечислитель с максимальным значением. Затем вычисляется минимальное число, представляющее степень двойки, которое больше этого максимального значения, и из него вычитается единица. (Например, максимальное значение `bigstep`, как определено выше, равно `101`. Минимальное число, представляющее степень двойки, которое больше `101`, равно `128`, поэтому верхний предел диапазона равен `127`.) Потом для определения минимального предела выбирается минимальное значение перечислителя. Если оно равно `0` или больше, то нижним пределом диапазона будет `0`. Если же минимальное значение перечислителя отрицательное,

используется тот же подход, как и для вычисления верхнего предела, но с минусом. (Например, если минимальный перечислитель равен -6, то следующая степень двойки будет -8, и нижний предел получается -7.)

Идея состоит в том, чтобы компилятор мог определить, сколько места необходимо для хранения перечисления. Он может использовать от 1 байт или менее для перечислений с небольшим диапазоном, и до 4 байт — для перечислений со значениями типа long.

Указатели и свободное хранилище

В начале третьей главы упоминалось о трех фундаментальных свойствах, которые должна отслеживать компьютерная программа, когда она сохраняет данные. Чтобы предохранить читателя от излишнего перелистывания страниц, приведем их еще раз:

- Где сохраняется информация.
- Какие именно значения хранятся там.
- Какого рода информация хранится.

Пока вы использовали только одну стратегию: объявляли простые переменные. Оператор объявления указывает тип и символическое имя значения. Оно также заставляет программу выделить память для этого значения и отслеживать скрытым образом ее местоположение.

Теперь мы рассмотрим другую стратегию, важность которой проявляется при разработке классов C++. Эта стратегия основана на указателях, которые представляют собой переменные, хранящие адреса значений вместо самих значений. Но прежде чем обратиться к указателям, давайте поговорим о том, как явно получить адрес обычной переменной. Для этого применяется операция взятия адреса, представленная символом &, к переменной, адрес которой вас интересует. Например, если home — переменная, то &home — ее адрес. В листинге 4.14 демонстрируется использование этой операции.

Листинг 4.14. address.cpp

```
// address.cpp -- использование операции & для нахождения адреса
#include <iostream>
int main()
{
   using namespace std;
   int donuts = 6;
   double cups = 4.5;
   cout << "значение donuts = " << donuts;
   cout << " и адрес donuts = " << &donuts << endl;
// ПРИМЕЧАНИЕ: вы можете использовать беззнаковый (&donuts)
// и беззнаковый (&cups)
   cout << "значение cups = " << cups;
   cout << " и адрес cups = " << &cups << endl;
   return 0;
}
```

Замечание по совместимости

Сам по себе cout — интеллектуальный объект, но некоторые его версии более интеллектуальны, чем другие. То есть некоторые реализации могут не отвечать требованиям стандарта C++ и не распознавать типы указателей. В этом случае вам придется выполнять приведение типа адреса к распознаваемому типу, такому как unsigned int. Соответствующие приведения типа зависят от модели памяти. По умолчанию модель памяти DOS использует 2-байтный адрес, поэтому unsigned int — правильное приведение. Но некоторые модели памяти DOS, однако, применяют 4-байтную адресацию, которая потребует приведения к unsigned long.

Ниже показан вывод программы из листинга 4.14 в одной из систем:

```
значение donuts = 6 и адрес donuts = 0x0065fd40
значение cups = 4.5 и адрес cups = 0x0065fd44
```

Конкретная реализация cout, показанная здесь, использует шестнадцатеричную нотацию при отображении значений адресов, потому что это обычная нотация, применяемая для спецификации адресов памяти. (Некоторые реализации применяют десятичную нотацию.) Наша реализация сохраняет donuts в памяти ниже, чем cups. Разница между этими двумя адресами составляет 0x0065fd44 - 0x0065fd40, или 4 байта. Конечно, в разных системах вы получите разные значения этих адресов. К тому же некоторые системы могут сохранять cups перед donuts, и разница между адресами составит 8, потому что cups имеет тип double. Другие системы могут даже разместить эти переменные в памяти далеко друг от друга, не выравнивая их рядом.

Использование обычных переменных, таким образом, трактует значение как именованную величину, а ее местоположение — как производную величину. Теперь рассмотрим стратегию указателей, которая представляет важнейшую часть философии программирования C++ в части управления памятью. (См. врезку "Указатели и философия C++" ниже.)

Указатели и философия C++

Объектно-ориентированное программирование отличается от традиционного процедурного программирования в том, что ООП подчеркивает принятие решений во время выполнения вместо стадии компиляции. *Время выполнения* означает период работы программы, а *время компиляции* — период сборки программы компилятором. Решения, принимаемые во время выполнения — это вроде того, как, будучи в отпуске, вы принимаете решение о том, какие достопримечательности стоит осмотреть, в зависимости от погоды и вашего настроения, в то время как решения, принимаемые во время компиляции, больше похожи на следование заранее разработанному плану, вне зависимости от любых условий. Решения времени выполнения обеспечивают гибкость, позволяющую программе приспосабливаться к текущим условиям. Например, рассмотрим выделением памяти для массива. Традиционный способ предполагает объявление массива. Чтобы объявить массив в C++, вы должны заранее решить, какого он должен быть размера. Таким образом, размер массива устанавливается во время компиляции программы, то есть это решение времени компиляции. Возможно, вы думаете, что массив из 20 элементов будет достаточным 80% времени, но однажды программе понадобится разместить 200 элементов. Чтобы обезопасить себя, вы используете массив размером в 200 элементов. Это приводит к тому, что ваша программа большую часть времени расходует память впустую. ООП пытается сделать программы более гибкими, откладывая принятие таких решений на стадию выполнения. Таким образом, после того, как программа запущена, она самостоятельно сможет решить, когда ей нужно размещать 20 элементов, а когда 205.

Короче говоря, с помощью ООП вы можете сделать решение относительно размера массива решением времени выполнения. Чтобы обеспечить такой подход, язык должен предоставлять возможность создавать массивы — или что-то им подобное — непосредственно во время работы программы. Как вы вскоре увидите, метод, используемый C++, включает применение ключевого слова new для запроса необходимого объема памяти и использование указателей для нахождения выделенной по запросу памяти.

Новая стратегия хранения данных изменяет трактовку местоположения как именованной величины, а значения — как производной величины. Для этого предусмотрен специальный тип переменной — *указатель*, который может хранить адрес как значение. Таким образом, имя указателя представляет местоположение. Применяя операцию *, называемую *операцией разыменования*, можно получить значение, хранящееся в указанном адресом месте. (Да, это тот же символ *, который применяется для обозначения арифметической операции умножения; C++ использует контекст для определения того, что вы подразумеваете в каждом конкретном случае — умножение или разыменование.) Предположим, например, что manly — это указатель. В таком случае manly представляет адрес, а *manly — значение, находящееся по этому адресу. Комбинация *manly становится эквивалентом простой переменной типа int. В листинге 4.15 демонстрируются эти идеи. Код в этом листинге также показывает, как объявляется указатель.

Листинг 4.15. pointer.cpp

```
// pointer.cpp -- наша первая переменная-указатель
#include <iostream>
int main()
{
   using namespace std;
   int updates = 6; // объявление переменной
   int * p_updates; // объявление указателя на int
   p_updates = &updates; // присвоить адрес int указателю
// выразить значения двумя способами
   cout << "Значение: updates = " << updates;
   cout << ", *p_updates = " << *p_updates << endl;
// выразить адреса двумя способами
   cout << "Адреса: &updates = " << &updates;
   cout << ", p_updates = " << p_updates << endl;
// изменить значение через указатель
   *p_updates = *p_updates + 1;
   cout << "Теперь updates = " << updates << endl;
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.15:

```
Значения: updates = 6, *p_updates = 6
Адреса: &updates = 0x0065fd48, p_updates = 0x0065fd48
Теперь updates = 7
```

Как вы можете видеть, переменная updates типа int и переменная-указатель p_updates — это две стороны одной монеты. Переменная updates в первую очередь представляет значение, а для получения его адреса применяется операция &, в то время как p_updates представляет адрес, а для получения значения применяется операция *. (См. рис. 4.8.) Поскольку p_updates указывает на updates, *p_updates и updates полностью эквивалентны. Вы можете использовать *p_updates точно так же, как вы используете переменную типа int. Как показывает пример в листинге 4.15, вы можете даже присваивать значения *p_updates. Это изменяет значение указанной переменной — updates.

```
int jumbo = 23;
int * pe = &jumbo;
```

Это — одно и то же

jumbo
*pe

Значение
23

&jumbo
pe

Адрес
0x2ac8

Это — одно и то же

Рис. 4.8. Две стороны одной монеты

Объявление и инициализация указателей

Давайте рассмотрим процесс объявления указателей. Компьютеру нужно отслеживать тип значения, на которое ссылается указатель. Например, адрес char обычно выглядит точно так же, как и адрес double, но char и double использует разное количество байт и разный внутренний формат представления значений. Поэтому объявление указателя должно специфицировать тип данных указываемого значения.

Например, предыдущий пример содержит следующее объявление:

```
int * p_updates;
```

Этот оператор устанавливает, что комбинация *p_updates имеет тип int. Поскольку вы используете операцию *, применяя ее к указателю, сама переменная p_updates должна *быть* указателем. Мы говорим, что p_update указывает на тип int. Мы также говорим, что тип p_updates — это указатель на int, или точнее, int *. Итак, повторим еще раз: p_updates — это указатель (адрес), а *p_updates — это int, а не указатель (рис. 4.9).

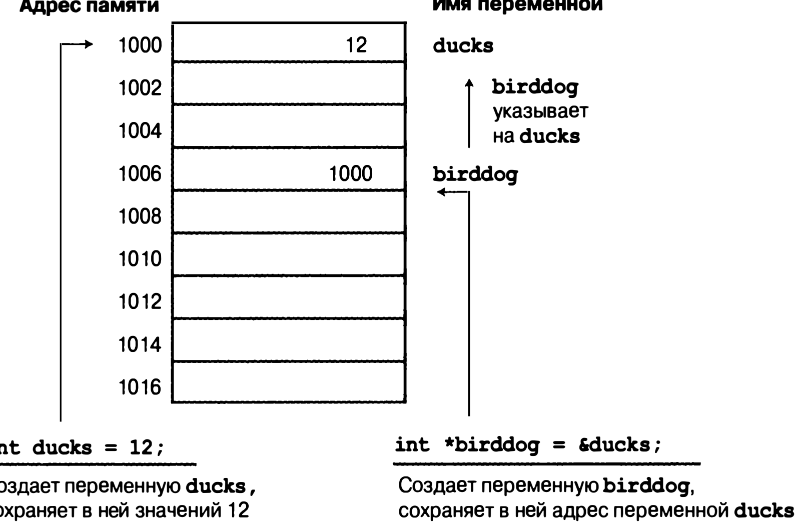


Рис. 4.9. Указатели хранят адреса

Кстати, пробелы вокруг операции * не обязательны. Традиционно программисты С используют следующую форму:

```
int *ptr;
```

Это подчеркивает идею, что комбинация *ptr имеет тип int. С другой стороны, многие программисты С++ отдают предпочтение такой форме:

```
int* ptr;
```

Это подчеркивает идею о том, что int* — это тип "указатель на int". Для компилятора не важно, с какой стороны вы поместите пробел. Однако будьте осторожны со следующим объявлением:

```
int* p1, p2;
```

Здесь создается один указатель (p1) и одна обычная переменная типа int (p2). Знак * нужно помещать возле каждой переменной-указателя.

На память!

В языке С++ комбинация int * — это составной тип, указатель на int.

Такой же синтаксис применяется для объявления указателей на другие типы:

```
double * tax_ptr;   // tax_ptr указывает на тип double
char * str;         // str указывает на тип char
```

Поскольку вы объявляете tax_ptr как указатель на double, компилятор знает, что *tax_ptr — это значение типа double. То есть он знает, что *tax_ptr представляет число, сохраненное в формате с плавающей точкой и занимающее (в большинстве систем) 8 байт. Переменная-указатель никогда не бывает просто указателем. Она всегда указывает на определенный тип. tax_ptr имеет тип "указатель на double" (или double *). Хотя оба они — указатели, но указывают на значения двух разных типов. Подобно массивам, указатели базируются на других типах.

Обратите внимание, что в то время как tax_ptr и str указывают на данные типов, имеющих разный размер, сами переменные tax_ptr и str обычно имеют одинаковый размер. То есть, адрес char имеет тот же размер, что и адрес double — точно так же, как 1016 может быть номером дома, в котором располагается огромный склад, в то время как 1024 — номером небольшого коттеджа. Размер или значение адреса на самом деле ничего не говорят о том, какого вида и размера переменная или дом находится по этому адресу. Обычно адрес требует от 2 до 4 байт, в зависимости от компьютерной системы. (Некоторые системы могут использовать и более крупные адреса, а иногда система использует разный размер адресов для разных типов.)

Вы можете использовать оператор объявления для того, чтобы инициализировать указатель. В этом случае инициализируется указатель, а не значение, на которое он указывает. То есть, операторы

```
int higgens = 5;
int * pt = &higgens;
```

устанавливают pt, а не *pt равным значению &higgens.

В листинге 4.16 демонстрируется инициализация указателя определенным адресом.

Листинг 4.16. init_ptr.cpp

```
// init_ptr.cpp -- инициализация указателя
#include <iostream>
int main()
{
   using namespace std;
   int higgens = 5;
   int * pt = &higgens;
   cout << "Значение higgens = " << higgens
       << "; Адрес higgens = " << &higgens << endl;
   cout << "Значение *pt = " << *pt
       << "; Значение pt = " << pt << endl;
   return 0;
}
```

Ниже показан вывод программы из листинга 4.16:

```
Значение higgens = 5; Адрес higgens = 0012FED4
Значение *pt = 5; Значение pt = 0012FED4
```

Вы можете видеть, что программа инициализирует `pt`, а не `*pt`, адресом переменной `higgens`.

Опасность указателей

Опасность подстерегает тех, что использует указатели неосмотрительно. Очень важно понять, что когда вы создаете указатель на C++, то компьютер выделяет память для хранения адреса, но не выделяет памяти для хранения данных, на которые указывает этот адрес. Выделение места для данных выполняется отдельным шагом. Если пропустить этот шаг, как в следующем фрагменте, то это обеспечит прямой путь к несчастью:

```
long * fellow;        // создать указатель на long
*fellow = 223323;     // поместить значение в неизвестное место
```

Конечно, `fellow` — это указатель. Но на что он указывает? Код не присваивает значения какого-либо адреса переменной `fellow`. Поэтому, куда будет помещено значение `223323`? Мы не можем на это ответить. Поскольку переменная `fellow` не была инициализирована, она может иметь какое угодно значение. Что бы в ней ни содержалось, программа будет интерпретировать это как адрес, куда и поместит `223323`. Если так случится, что `fellow` будет иметь значение `1200`, компьютер попытается поместить данные по адресу `1200`, даже если этот адрес окажется в середине вашего программного кода. На что бы ни указывал `fellow`, скорее всего, это будет не то место, куда вы хотели бы поместить число `223323`. Ошибки подобного рода порождают самое непредсказуемое поведение программы и такие ошибки очень трудно отследить.

Внимание!

Золотое правило указателей: *всегда* инициализируйте указатель, чтобы определить точный и правильный адрес, прежде чем обращаться к нему с использованием операции разыменования (`*`).

Указатели и числа

Указатели — это не целочисленные типы, даже несмотря на то, что компьютеры обычно выражают адреса целыми числами. Концептуально указатели — это типы, отличные от целых. Целые вы можете складывать, вычитать, умножать, делить и так далее. Но указатели описывают местоположение, и не имеет смысла, например, перемножать между собой два местоположения. В терминах допустимых над ними операций указатели и целые отличаются друг от друга. Следовательно, вы не можете просто присвоить целое указателю:

```
int * pt;
pt = 0xB8000000;           // несоответствие типов
```

Здесь левая часть — указатель на `int`, поэтому вы можете присваивать ему адрес, но правая часть — просто целое число. Вы можете точно знать, что `0xB8000000` — комбинация сегмент-смещение адреса видеопамяти в системе, но ничего в этом операторе не говорит программе, что данное число является адресом. Однако C++ предполагает более строгие соглашения о типах, и компилятор выдаст сообщение об ошибке, говорящее о несоответствии типов. Если вы хотите использовать числовое значение в качестве адреса, то должны выполнять приведение типа, чтобы преобразовать числовое значение к соответствующему типу адреса:

```
int * pt;
pt = (int *) 0xB8000000;  // теперь типы соответствуют
```

Теперь обе стороны оператора присваивания представляют адреса, поэтому такое присваивание корректно. Обратите внимание, что если есть значение адреса типа `int`, это не значит, что сам `pt` имеет тип `int`. Например, в большой модели памяти IBM PC под управлением DOS тип `int` имеет размер 2 байта, в то время как значение адреса является 4-байтным.

Указатели имеют и некоторые другие интересные свойства, которые мы обсудим, когда доберемся до соответствующей темы. А пока давайте рассмотрим, как указатели могут использоваться для управления выделением памяти во время выполнения.

Выделение памяти операцией new

Теперь, когда вы получили представление о работе указателей, давайте посмотрим, как они помогают реализовать важнейшую технику ООП, связанную с выделением памяти во время выполнения программы. До сих пор мы инициализировали указатели адресами переменных; переменные — это именованная память, выделенная во время компиляции, и каждый указатель, до сих пор использованный в примерах, просто представлял собой псевдоним для памяти, доступ к которой и так был возможен по именам переменных. Истинная ценность указателей проявляется тогда, когда во время выполнения выделяются неименованные области памяти для хранения переменных. В этом случае указатели становятся единственным способом доступа к такой памяти. В языке С вы можете выделять память библиотечной функцией `malloc()`. Вы можете пользоваться ею и в C++, но C++ также предлагает лучший способ — операцию `new`.

Испытаем эту новую технику, создав неименованное хранилище времени выполнения для значения типа `int`, и присвоим ему значение через указатель. Ключ ко

всему — операция C++ new. Вы сообщаете new, для данных какого типа запрашивается память; new находит блок памяти нужного размера и возвращает его адрес. Вы присваиваете этот адрес указателю, и порядок! Вот пример этой техники:

```
int * pn = new int;
```

Часть new int сообщает программе, что вам требуется некоторое новое хранилище, подходящее для хранения int. Операция new использует тип для того, чтобы определить, сколько байт необходимо выделить. Затем она находит память и возвращает адрес. Далее вы присваиваете адрес переменной pn, которая объявлена как указатель на int. Теперь pn — адрес, а *pn — значение, хранящееся по этому адресу. Сравните это с присваиванием адреса переменной указателю:

```
int higgens;
int * pt = &higgens;
```

В обоих случаях (pn и pt) вы присваиваете адрес значения int указателю. Во втором случае вы также можете обратиться к int по имени higgens. В первом случае доступ возможен только через указатель. Возникает вопрос: поскольку память, на которую указывает pn, не имеет имени, как назвать ее? Мы говорим, что pn указывает на *объект данных*. Это не есть "объект" в терминологии объектно-ориентированного программирования. Это просто объект, в смысле "вещь". Термин "объект данных" более общий, чем "переменная", потому что он означает любой блок памяти, выделенный для элемента данных. Таким образом, переменная — это тоже объект, но память, на которую указывает pn, не является переменной. Метод обращения к объектам данных через указатель может показаться поначалу несколько запутанным, однако он обеспечивает программе высокую степень управляемости памятью.

Общая форма получения и присваивания памяти для отдельного объекта данных, который может быть как структурой, так и базовым типом, выглядит так:

```
typeName pointer_name = new typeName;
```

Вы используете тип данных дважды: один раз — указывая для какого типа данных запрашивается память, и второй — для объявления подходящего указателя. Конечно, если вы уже ранее объявили указатель на корректный тип, то можете его использовать вместо объявления еще одного нового. В листинге 4.17 иллюстрируется применение new для двух разных типов.

Листинг 4.17. use_new.cpp

```
// use_new.cpp -- использование операции new
#include <iostream>
int main()
{
   using namespace std;
   int * pt = new int;        // выделить пространство для int
   *pt = 1001;                // сохранить в нем значение
   cout << "int ";
   cout << "значение = " << *pt << ": местоположение = " << pt << endl;
   double * pd = new double; // выделить пространство для double
   *pd = 10000001.0;          // сохранить в нем значение double
   cout << "double ";
   cout << "значение = " << *pd << ": местоположение = " << pd << endl;
```

```
    cout << "размер pt = " << sizeof(pt);
    cout << ": размер *pt = " << sizeof(*pt) << endl;
    cout << "размер pd = " << sizeof pd;
    cout << ":размер *pd = " << sizeof(*pd) << endl;
    return 0;
}
```

Вот что мы получим, запустив эту программу:

```
int значение = 1001: местоположение = 0x004301a8
double значение = 1e+07: местоположение = 0x004301d8
размер pt = 4: размер *pt = 4
размер pd = 4: размер *pd = 8
```

Конечно, точные значения адресов памяти будут варьироваться от системы к системе и от запуска к запуску.

Замечания по программе

Программа в листинге 4.17 использует операцию new для выделения памяти под объекты данных типа int и типа double. Это происходит во время выполнения программы. Указатели pt и pd указывают на эти объекты данных. Без них вы не смогли бы получить к ним доступ. С ними же вы можете использовать *pt и *pd подобно тому, как вы используете обычные переменные. Вы присваиваете значения новым объектам данных, присваивая их *pt и *pd. Аналогично вы печатаете *pt и *pd, чтобы отобразить эти значения.

Программа в листинге 4.17 также демонстрирует одну из причин того, что необходимо объявить тип данных, на которые указывает указатель. Сам по себе адрес относится только к началу сохраненного объекта, он не включает информации о типе или размере. К тому же заметьте, что указатель на целое имеет тот же размер, что и указатель на double. Оба они — адреса. Но поскольку use_new.cpp объявляет типы указателей, программа знает, что *pd имеет тип double размером в 8 байт, в то время как *pt — значение типа int размером в 4 байта.

Когда use_new.cpp печатает значение *pd, cout известно, сколько байт нужно прочитать и как их интерпретировать.

Нехватка памяти?

Может случиться, что у компьютера не окажется достаточно доступной памяти, чтобы удовлетворить запрос new. Когда такое случается, new возвращает значение 0. В C++ указатель со значением 0 называется *null-указателем* (нулевым указателем). C++ гарантирует, что нулевой указатель никогда не указывает на корректные данные, поэтому он часто используется в качестве признака неудачного завершения операций или функций, которые в противном случае должны возвращать корректные указатели. После того, как вы узнаете об операторе if в главе 6, вы сможете проверять возвращаемое значение new на равенство нулевому указателю, и таким образом защитить вашу программу от попыток выйти за границы памяти. В дополнение к возврату нулевого указателя при сбое выделения памяти, new может также возбуждать исключение bad_alloc. Механизм исключений описан в главе 15.

Освобождение памяти операцией delete

Используя new для запроса памяти, когда вы нуждаетесь в ней — это более красивая половина пакета управления памятью C++. Вторая половина — операция delete, которая позволяет вам вернуть память в пул свободной памяти, когда вы завершили работу с ней. Это — важный шаг к наиболее эффективному использованию памяти. Память, которую вы возвращаете, или *освобождаете*, затем может быть повторно использована другими частями программы. Вы используете delete, задавая после него указатель на блок памяти, который был выделен операцией new:

```
int * ps = new int;  // выделить память операцией new
. . .                // использовать память
delete ps;           // освободить память операцией delete,
                     // когда она больше не нужна
```

Это освобождает память, на которую указывает ps, но не удаляет сам ps. Вы можете повторно использовать ps — например, чтобы указать на другой выделенный new блок памяти. Вы всегда должны обеспечивать сбалансированное применение new и delete; в противном случае вы рискуете столкнуться с таким явлением, как *утечка памяти*, то есть ситуацией, когда память выделена, но более не может быть использована. Если утечки памяти слишком велики, то попытка программы выделить очередной блок может привести к ее аварийному завершению.

Вы не должны пытаться освобождать блок памяти, который уже был однажды освобожден. Стандарт C++ гласит, что результат таких попыток не определен, а это значит, что последствия могут оказаться любыми. Кроме того, вы не можете операцией delete освобождать память, которая была создана объявлением обычных переменных:

```
int * ps = new int;  // нормально
delete ps;           // нормально
delete ps;           // теперь не нормально!
int jugs = 5;        // нормально
int * pi = &jugs;    // нормально
delete pi;           // не допускается, память не была выделена new
```

Внимание!

Вы должны использовать delete только для освобождения памяти, выделенной new. Однако вполне безопасно применить delete к нулевому указателю.

Обратите внимание, что обязательным условием применения операции delete является использование его с памятью, выделенной операцией new. Это не значит, что вы обязаны использовать тот же указатель, который был использован с new, просто нужно использовать тот же адрес:

```
int * ps = new int;  // выделить память
int * pq = ps;       // установить второй указатель на тот же блок
delete pq;           // вызов delete для второго указателя
```

Обычно не стоит создавать два указателя на один и тот же блок памяти, потому что это может привести к ошибочной попытке освобождению одного и того же блока дважды. Но, как вы вскоре убедитесь, применение второго указателя оправдано, когда вы работаете с функциями, возвращающими указатель.

Использование new для создания динамических массивов

Если все, что нужно программе — это единственное значение, то вы можете объявить обычную переменную; это намного проще, хотя и не так впечатляет, как применение new для управления единственным небольшим объектом данных. Более типично использование new с большими фрагментами данных — такими как массивы, строки и структуры. Именно в таких случаях операция new весьма полезна. Предположим, например, что вы пишете программу, которой, может быть, понадобится массив, а, может, и нет — в зависимости от информации, поступающий во время выполнения. Если вы создаете массив простым объявлением, пространство для него распределяется раз и навсегда — во время компиляции. Понадобится массив программе, или нет — он все равно существует и занимает место в памяти. Распределение массива во время компиляции называется *статическим связыванием* и означает, что массив встраивается в программу во время компиляции. Но с помощью new вы можете создать массив при необходимости, во время выполнения программы, либо не создавать его, если потребность в нем отсутствует. Или же вы можете выбрать размер массива уже после того, как программа запущена. Это называется *динамическим связыванием* и означает, что массив будет создан во время выполнения программы. Такой массив называется *динамическим массивом.* При статическом связывании вы должны жестко закодировать размер массива во время написания программы. При динамическом связывании программа может принять решение о размере массива во время ее работы.

Пока мы рассмотрим два важных обстоятельства относительно динамических массивов: как использовать операцию new для создания массива и как использовать указатель для доступа к его элементам.

Создание динамического массива с помощью new

Создать динамический массив на C++ легко; вы сообщаете операции new тип элементов массива и требуемое количество элементов. Синтаксис, необходимый для этого, предусматривает указание имени типа с количеством элементов в квадратных скобках. Например, если нужен массив из 10 элементов int, следует написать так:

```
int * psome = new int [10]; //получить блок памяти из 10 элементов типа int
```

Операция new возвращает адрес первого элемента в блоке. В данном примере это значение присваивается указателю psome. Вы должны сбалансировать каждый вызов new соответствующим вызовом delete, когда программа завершает пользоваться этим блоком памяти.

Когда new применяется для создания массива, вы должны сопровождать его альтернативной формой delete, которая указывает на то, что освобождается массив:

```
delete [] psome;              // освободить динамический массив
```

Присутствие квадратных скобок сообщает программе, что она должна освободить весь массив, а не только один элемент, на который установлен указатель. Обратите внимание, что скобки расположены между delete и указателем. Если вы используете new без скобок, то и соответствующий delete тоже должен быть без скобок. Если же у вас new со скобками, то и соответствующий delete должен быть со скобками. Ранние версии C++ могут не распознавать нотацию с квадратными скобка-

ми. Согласно стандарту ANSI/ISO, однако, эффект от несоответствия форма `new` и `delete` не определен, то есть вы не должны рассчитывать в этом случае на какое-то определенное поведение. Вот пример:

```
int * pt = new int;
short * ps = new short [500];
delete [] pt;     // эффект не определен, не делайте так
delete ps;        // эффект не определен, не делайте так
```

Короче говоря, используя `new` и `delete`, вы должны придерживаться следующих правил:

- Не использовать `delete` для освобождения той памяти, которая не было выделена `new`.
- Не использовать `delete` для освобождения одного и того же блока памяти дважды.
- Использовать `delete[]`, если для размещения массива применялся `new[]`.
- Использовать `delete` (без скобок), если применялся `new` для размещения отдельного элемента.
- Безопасно применять `delete` к нулевому указателю.

Теперь вернемся к динамическому массиву. Заметьте, что `psome` — это указатель на отдельный `int`, первый элемент блока. Отслеживать количество элементов в блоке ложится на вашу ответственность как разработчика. То есть, поскольку компилятор не знает о том, что `psome` указывает на первые 10 целых, вы должны писать свою программу так, чтобы она самостоятельно отслеживала количество элементов.

На самом деле программе, конечно же, известен объем выделенной памяти, так что она может корректно освободить ее позднее, когда вы воспользуетесь операцией `delete[]`. Однако эта информация не является общедоступной; вы, например, не можете использовать операцию `sizeof`, чтобы узнать количество байт в выделенном блоке.

Общая форма выделения и присваивания памяти для массива выглядит следующим образом:

```
type_name pointer_name = new type_name [num_elements];
```

Вызов операции `new` выделяет достаточно большой блок памяти, чтобы в нем поместилось *num_elements* элементов типа *type_name*, и устанавливает в *pointer_name* указатель на первый элемент. Как вы вскоре увидите, *pointer_name* можно использовать точно так же, как обычное имя массива.

Использование динамического массива

После того, как вы создали динамический массив, как его можно использовать? Во-первых, подумаем о проблеме концептуально. Оператор

```
int * psome = new int [10]; // получить блок для 10 элементов типа int
```

создает указатель `psome`, который указывает на первый элемент блока из 10 значений `int`. Представьте его как палец, указывающий на первый элемент. Предположим, `int` занимает 4 байта. Затем, перемещая палец на 4 байта в правильном направлении,

вы можете указать на второй элемент. Всего имеется 10 элементов, что представляет нам допустимый диапазон, в пределах которого можно двигать палец. То есть, операция new снабжает вас всей необходимой информацией для идентификации каждого элемента в блоке.

Теперь взглянем на проблему практически. Как вы можете получить доступ к этим элементам? С первым элементом проблем нет. Поскольку psome указывает на первый элемент массива, то *psome и есть значение первого элемента. Но остается еще 9 элементов. Простейший способ доступа к этим элементам может стать сюрпризом для вас, если вы не работали с языком С; просто используйте указатель, как если бы он был именем массива. То есть, можно писать psome[0] вместо *psome для первого элемента, psome[1] — для второго и так далее. Получается, что применять указатель для доступа к динамическому массиву очень просто, хотя это и не очень понятно, почему этот метод работает. Причина в том, что С и С++ внутренне все равно работают с массивами через указатели. Эта эквивалентность указателей и массивов — одно из замечательных свойств С и С++. Ниже об этом речь пойдет более подробно. Во-первых, код в листинге 4.18 продемонстрирует, как использовать new для создания динамического массива, и затем применит нотацию обычного массива для доступа к его элементам. Он также обозначит фундаментальное отличие между указателем и настоящим именем массива.

Листинг 4.18. arraynew.cpp

```
// arraynew.cpp -- использование операции new для массивов
#include <iostream>
int main()
{
   using namespace std;
   double * p3 = new double [3];   // пространство для 3 double
   p3[0] = 0.2;                    // трактовать p3 как имя массива
   p3[1] = 0.5;
   p3[2] = 0.8;
   cout << "p3[1] равно " << p3[1] << ".\n";
   p3 = p3 + 1;      // увеличить указатель
   cout << "Теперь p3[0] равно " << p3[0] << " и ";
   cout << "p3[1] равно " << p3[1] << ".\n";
   p3 = p3 - 1;      // вернуть указатель к началу
   delete [] p3;     // освободить память
   return 0;
}
```

Вывод этой программы выглядит следующим образом:

```
p3[1] равно 0.5.
Теперь p3[0] равно 0.5 и p3[1] равно 0.8.
```

Как видите, arraynew.cpp использует указатель p3, как если бы он был именем массива, с первым элементом p3[0] и так далее. Фундаментальное отличие между именем массива и указателем проявляется в следующей строке:

```
p3 = p3 + 1;      // допускается для указателей, но не для имен массивов
```

Вы не можете изменить значение для имени массива. Но указатель — переменная, а потому ее значение можно изменить. Отметим эффект от прибавления 1 к p3. Теперь выражение p3[0] ссылается на бывший второй элемент массива. То есть, прибавление 1 к p3 заставляет p3 указывать на второй элемент вместо первого. Вычитание 1 из значения указателя возвращает его назад, в исходное значение, поэтому программа может применить delete[] с корректным адресом.

Действительные адреса соседних элементов int отличаются на 2 или 4 байта, поэтому тот факт, что добавление 1 к p3 дает адрес следующего элемента, говорит о том, что арифметика указателей устроена специальным образом. Так оно и есть.

Указатели, массивы и арифметика указателей

Родство указателей и имен массивов происходит из *арифметики указателей*, а также того, как C++ внутренне работает с массивами. Сначала рассмотрим арифметику. Прибавление единицы к целочисленной переменной увеличивает ее значение на единицу, но прибавление единицы к переменной указателя увеличивает ее значение на количество байт, составляющих размер того типа, на который она указывает. Прибавление единицы к указателю на double добавляет 8 байт к числовой величине указателя на системах с 8-байтным double, в то время как прибавление единицы к указателю на short добавляет к его значению 2 байта. Листинг 4.19 доказывает истинность этого невероятного утверждения. Он также демонстрирует еще один важный момент: C++ интерпретирует имена массивов как адреса.

Листинг 4.19. addpntrs.cpp

```
// addpntrs.cpp -- сложение указателей
#include <iostream>
int main()
{
   using namespace std;
   double wages[3] = {10000.0, 20000.0, 30000.0};
   short stacks[3] = {3, 2, 1};
// Здесь два способа получить адрес массива
   double * pw = wages;      // имя массива равно адресу
   short * ps = &stacks[0];  // или использовать операцию взятия адреса
                              // с элементами массива
   cout << "pw = " << pw << ", *pw = " << *pw << endl;
   pw = pw + 1;
   cout << "добавить 1 к указателю pw: \n";
   cout << "pw = " << pw << ", *pw = " << *pw << "\n\n";
   cout << "ps = " << ps << ", *ps = " << *ps << endl;
   ps = ps + 1;
   cout << "добавить 1 к указателю ps: \n";
   cout << "ps = " << ps << ", *ps = " << *ps << "\n\n";
   cout << "обращение к двум элементам в нотации массива \n";
   cout << "stacks[0] = " << stacks[0]
       << ", stacks[1] = " << stacks[1] << endl;
   cout << "обращение к двум элементам в нотации указателя \n";
   cout << "*stacks = " << *stacks
       << ", *(stacks + 1) = " << *(stacks + 1) << endl;
```

```
    cout << sizeof(wages) << " = размер массива wages \n";
    cout << sizeof(pw) << " = размер указателя pw \n";
    return 0;
}
```

Ниже приведен вывод программы из листинга 4.19:

```
pw = 0012FEBC, *pw = 10000
добавить 1 к указателю pw:
pw = 0012FEC4, *pw = 20000
ps = 0012FEAC, *ps = 3
добавить 1 к указателю ps:
ps = 0012FEAE, *ps = 2
обращение к двум элементам в нотации массива
stacks[0] = 3, stacks[1] = 2
обращение к двум элементам в нотации указателя
*stacks = 3, *(stacks + 1) = 2
24 = размер массива wages
4 = размер указателя pw
```

Замечания по программе

В большинстве контекстов C++ интерпретирует имя массива как адрес его первого элемента. Таким образом, оператор

```
double * pw = wages;
```

создает pw как указатель на тип double, затем инициализирует его значением wages, который, в свою очередь, является указателем первого элемента массива wages. Для wages, как и любого другого массива, справедливо следующее утверждение:

```
wages = &wages[0] = адрес первого элемента массива
```

Чтобы доказать, что это так, программа явно использует операцию адреса в выражении &stacks[0] для инициализации указателя ps адресом первого элемента массива stacks.

Далее программа инспектирует значения pw и *pw. Первое представляет адрес, а второе — значение, расположенное по этому адресу. Поскольку pw указывает на первый элемент, значение, отображаемое *pw, и будет значением первого элемента — 10000. Затем программа прибавляет единицу к pw. Как и ожидалось, это добавляет восемь (fd24 + 8 = fd2c в шестнадцатеричном виде) к числовому значению адреса, потому что double в этой системе занимает 8 байт. Это присваивает pw адрес второго элемента. Таким образом, теперь *pw равно 20000. (См. рис. 4.10.) (Значения адресов на рисунке выровнены для ясности.)

После этого программа выполняет аналогичные шаги для ps. На этот раз, поскольку ps указывает на тип short, а размер значений типа short равен 2 байтам, добавление единицы к этому указателю увеличивает его значение на 2. Опять-таки, в результате указатель устанавливается на следующий элемент массива.

На память!

Прибавление единицы к переменной указателя увеличивает его значение на количество байт, представляющее размер типа, на который он указывает.

```
double wages[3] = {10000.0, 20000.0, 30000.0};
short stacks[3] = {3, 2, 1};
double * pw = wages;
short * ps = &stacks[0];
```


Рис. 4.10. Сложение указателей

Теперь рассмотрим выражение `stacks[1]`. Компилятор С++ трактует это выражение точно так же, как если бы вы написали `*(stacks + 1)`. Второе выражение означает вычисление адреса второго элемента массива, и затем извлечение значения, сохраненного в нем. Конечный результат — значение `stacks[1]`. (Приоритет операций требует применения скобок. Без них единица была бы добавлена к `*stacks` вместо `stacks`.)

Вывод программы демонстрирует, что `*(stacks + 1)` и `stacks[1]` — это одно и то же. Аналогично `*(stacks + 1)` эквивалентно `stacks[2]`. В общем случае, всякий раз, когда вы используете нотацию массивов, С++ выполняет следующее преобразование:

```
arrayname[i] превращается в *(arrayname + i)
```

И если вы используете указатель вместо имени массива, С++ выполняет то же преобразование:

```
pointername[i] превращается в *(pointername + i)
```

Таким образом, во многих отношениях вы можете использовать имена указателей и имена массивов одинаковым образом. Нотацию квадратных скобок можно применять с обоими. К обоим можно применять операцию разыменования (`*`). В большинстве выражений каждое имя представляет адрес. Единственное отличие состоит в том, что значение указателя изменить можно, а имя массива — константа:

```
pointername = pointername + 1;  // правильно
arrayname = arrayname + 1;      // не допускается
```

Второе отличие заключается в том, что применение операции `sizeof` к имени массива возвращает размер массива в байтах, но применение `sizeof` к указателю возвращает размер указателя, даже если он указывает на массив. Например, в листинге 4.19 как `pw`, так и `wages` ссылаются на один и тот же массив. Однако применение операции `sizeof` к ним порождает разные результаты:

```
24 = размер массива wages ← отображение sizeof wages
4 = размер указателя pw ← отображение sizeof pw
```

Это один из случаев, когда C++ не интерпретирует имя массива как адрес.

Короче говоря, использовать new для создания массива и применять указатели для доступа к его различным элементам очень просто. Вы просто трактуете указатель как имя массива. Однако понять, почему это работает — интересная задача. Если вы действительно хотите понимать массивы и указатели, вы должны тщательно исследовать их поведение, связанное с изменчивостью.

Подведем итоги относительно указателей

Немного позже вы сможете углубить свои знания об указателях, а пока подведем итоги относительно того, что мы узнали об указателях и массивах к настоящему моменту.

Объявление указателей

Чтобы объявить указатель на определенный тип, нужно использовать следующую форму:

```
typeName * pointerName;
```

Вот некоторые примеры:

```
double * pn;   // pn может указывать на значение double
char * pc;     // pc может указывать на значение char
```

Здесь pn и pc — указатели, а double * и char * — нотация C++, символизирующая указатель на double и указатель на char.

Присваивание значений указателям

Указателям необходимо присваивать адреса памяти. Для этого можно применять операцию & к имени переменной, чтобы получить адрес именованной области памяти, либо операцию new, которая возвращает адрес неименованной памяти.

Вот некоторые примеры:

```
double * pn;   // pn может указывать на значение double
double * pa;   // так же и pa
char * pc;     // pc может указывать на значение char
double bubble = 3.2;
pn = &bubble;  // присвоить адрес bubble переменной pn
pc = new char; // присвоить адрес выделенной памяти char переменной pc
pa = new double[30]; // присвоить адрес массива из 30 double переменной pa
```

Разыменование указателей

Разыменование указателя означает получение значения, на которое он указывает. Для этого к указателю применяется операция разыменования (*). То есть, если pn — указатель на bubble, как в предыдущем примере, то *pn — значение, на которое он указывает, то есть в данном случае — 3.2.

Вот некоторые примеры:

```
cout << *pn;   // печатать значение bubble
*pc = 'S';     // поместить 'S' в область памяти, на которую указывает pc
```

Нотация массивов — второй способ разыменования указателя; например, pn[0] — это то же самое, что *pn. Никогда не следует разыменовывать указатель, который не был инициализирован правильным адресом.

Различие между указателем и указываемым значением

Помните, если pt — указатель на int, то *pt — не указатель на int, а полный эквивалент переменной типа int.

Вот некоторые примеры:

```
int * pt = new int; // присваивание адреса переменной pt
*pt = 5;            // сохранение 5 по этому адресу
```

Имена массивов

В большинстве контекстов C++ трактует имя массива как эквивалент адреса его первого элемента.

Вот пример:

```
int tacos[10];      // теперь tacos — то же самое, что &tacos[0]
```

Одно исключения из этого правила связано с применением операции sizeof к имени массива. В этом случае sizeof возвращает размер всего массива в байтах.

Арифметика указателей

C++ позволяет вам добавлять целые числа к указателю. Результат прибавления к указателю единицы равен исходному адресу плюс значение, эквивалентное числу байт в указываемом объекте. Вы можете также вычесть один указатель из другого, чтобы получить разницу между двумя указателями. Последняя операция, которая возвращает целое значение, осмысленна только в случае, когда два указателя указывают на элементы одного и того же массива (указание одной из позиций за границей массива также допускается); при этом результат означает расстояние между элементами массива.

Ниже показаны некоторые примеры:

```
int tacos[10] = {5,2,8,4,1,2,2,4,6,8};
int * pt = tacos;    // предположим, что pf и tacos указывают на адрес 3000
pt = pt + 1;         // теперь pt равно 3004, если int имеет размер 4 байта
int *pe = &tacos[9]; // pe равно 3036, если int имеет размер 4 байта
pe = pe - 1;         // теперь pe равно 3032 - адресу элемента tacos[8]
int diff = pe - pt;  // разница равна 7, то есть расстоянию между
                     // tacos[8] и tacos[1]
```

Динамическое и статическое связывание массивов

Вы можете использовать объявления массива для создания массива со статическим связыванием — то есть, массива, чей размер фиксирован на этапе компиляции:

```
int tacos[10];    // статическое связывание,
                  // размер фиксирован во время компиляции
```

Для создания массива с динамическим связыванием (динамического массива) используется операция new[]. Память для этого массива выделяется в соответствии с размером, указанным во время выполнения программы. Когда работа с таким массивом завершена, выделенная ему память освобождается операцией delete[]:

```
int size;
cin >> size;
int * pz = new int [size];// динамическое связывание, размер
                          // устанавливается во время выполнения
...
delete [] pz; // освобождение памяти по окончании работы с массивом
```

Нотация массивов и нотация указателей

Использование нотации массивов с квадратными скобками эквивалентно разыменованию указателя:

```
tacos[0] означает *tacos и означает значение, находящееся по адресу tacos
tacos[3] означает *(tacos + 3) и означает значение, находящееся по адресу
tacos + 3
```

Это справедливо как для имен массивов, так и для переменных-указателей, поэтому вы можете использовать обе нотации.

Вот некоторые примеры:

```
int * pt = new int [10];  // pt указывает на блок из 10 int
*pt = 5;                  // присваивает элементу 0 значение 5
pt[0] = 6;                // присваивает элементу 0 значение 6
pt[9] = 44;               // устанавливает десятому элементу
                          // (элемент номер 9) значение 44
int coats[10];
*(coats + 4) = 12;        // устанавливает coats[4] значение 12
```

Указатели и строки

Специальные отношения между массивами и указателями расширяют строки в стиле C. Рассмотрим следующий код:

```
char flower[10] = "rose";
cout << flower << "s are red\n";
```

Имя массива — это адрес его первого элемента, поэтому `flower` в операторе `cout` представляет адрес элемента `char`, содержащего символ `r`. Объект `cout` предполагает, что адрес `char` — это адрес строки, поэтому печатает символ, расположенный по этому адресу, и затем продолжает печать последующих символов до тех пор, пока не встретит нулевой символ (`\0`). Короче говоря, вы сообщаете `cout` адрес символа, он печатает все, что находится в памяти, начиная с этого символа и до нулевого.

Ключевым фактором здесь является не то, что `flower` — имя массива, а то, что `flower` трактуется как адрес значения `char`. Это предполагает, что вы можете использовать переменную-указатель на `char` в качестве аргумента для `cout`, потому что она тоже содержит адрес `char`. Конечно, этот указатель должен указывать на начало строки. Чуть позже мы проверим это.

Но как насчет финальной части предыдущего оператора `cout`? Если `flower` — на самом деле адрес первого символа строки, что есть выражение `"s are red\n"`? Согласно принципам, которыми руководствуется `cout` при выводе строк, эта строка в кавычках также должна быть адресом. И так оно и есть — в C++ строка в кавычках, как и имя массива, служит адресом его первого элемента. Предыдущий код на самом деле не посылает полную строку объекту `cout`; он просто передает адрес строки. Это

значит, что строки в массиве, строковые константы в кавычках и строки, описываемые указателями — все обрабатываются одинаково. Каждая из них передается в виде адреса. Конечно, это задает компьютеру меньше работы, чем потребовалось бы в случае передачи каждого символа строки.

На память!

С объектом `cout`, как и в большинстве других выражений C++, имя массива `char`, указатель на `char`, а также строковая константа в кавычках — все они интерпретируются как адрес первого символа строки.

В листинге 4.20 иллюстрируется применение различных форм строк. Код в этом листинге использует две функции из стандартной библиотеки строк. Функция `strlen()`, которую вы уже применяли ранее, возвращает длину строки. Функция `strcpy()` копирует строку из одного места в другое. Обе имеют прототипы в файле заголовков `cstring` (или `string.h` — в устаревших реализациях). Программа также использует комментарии, предупреждающие о возможных случаях неправильного применения указателей, которых следует избегать.

Листинг 4.20. `ptrstr.cpp`

```
// ptrstr.cpp -- использование указателей на строки
#include <iostream>
#include <cstring>              // объявление strlen(), strcpy()
int main()
{
   using namespace std;
   char animal[20] = "bear";   // animal содержит bear
   const char * bird = "wren";  // bird содержит адрес строки
   char * ps;                   // не инициализировано
   cout << animal << " и ";     // отображает bear
   cout << bird << "\n";        // отображает wren
   // cout << ps << "\n";       // может отобразить мусор, но может
                                // и вызвать крах программы
   cout << "Введите вид животного: ";
   cin >> animal;               // порядок, если ввод < 20 символов
   // cin >> ps; Очень опасная ошибка, чтобы пробовать; ps не указывает
   // на выделенное пространство
   ps = animal;                 // установить в ps указатель на строку
   cout << ps << "s!\n";        // нормально, то же, что и применение animal
   cout << "Перед использованием strcpy():\n";
   cout << animal << "по адресу " << (int *) animal << endl;
   cout << ps << "по адресу " << (int *) ps << endl;
   ps = new char[strlen(animal) + 1]; // получить новое хранилище
   strcpy(ps, animal);          // скопировать строку в новое хранилище
   cout << "После использования strcpy():\n";
   cout << animal << "по адресу " << (int *) animal << endl;
   cout << ps << "по адресу " << (int *) ps << endl;
   delete [] ps;
   return 0;
}
```

Замечание по совместимости

Если в вашей системе нет заголовочного файла cstring, используйте более старую версию — string.h.

Ниже показан пример выполнения программы из листинга 4,20:

```
bear и wren
Введите вид животного: fox
foxs!
Перед использованием strcpy():
fox по адресу 0x0065fd30
fox по адресу 0x0065fd30
После использования strcpy():
fox по адресу 0x0065fd30
fox по адресу 0x004301c8
```

Замечания по программе

Программа в листинге 4.20 создает один массив char (animal) и две переменных — указателя на char (bird и ps). Программа начинается с инициализации массива animal строкой "bear" — точно так же, как вы инициализировали массивы и раньше. Затем программа делает нечто новое. Она инициализирует указатель на char строкой:

```
const char * bird = "wren"; // bird содержит адрес строки
```

Вспомните, что "wren" на самом деле представляет собой адрес строки, поэтому данный оператор присваивает адрес "wren" указателю bird. (Обычно компилятор выделяет область памяти для размещения строк в кавычках, указанных в исходном коде, ассоциируя каждую сохраненную строку с ее адресом.) Это значит, что вы можете использовать указатель bird, как вы использовали бы строку "wren", например:

```
cout << "Эта " << bird << " разговаривает\n";
```

Строковые литералы — это константы, вот почему код использует ключевое слово const в объявлении. Применение const, таким образом, означает, что вы можете использовать bird для доступа к строке, но не можете изменять ее. В главе 7 тема константных указателей рассматривается более подробно. И, наконец, указатель ps остается неинициализированным, поэтому он не может указывать ни на какую строку (Как вы знаете, это плохая идея, и данный пример — не исключение.)

Далее программа иллюстрирует, что вы можете использовать с cout имя массива animal и указатель bird совершенно одинаково. В конце концов, оба они являются адресами строк, и cout отображает две строки ("bear" и "wren"), расположенные по указанным адресам. Если вы активизируете код, который пытается отобразить ps, вы можете получить пустую строку, какой-то мусор или даже привести программу к краху. Создание неинициализированных указателей подобно выдаче незаполненного чека с подписью: вы утрачиваете контроль над его использованием.

Что касается ввода, то здесь ситуация несколько отличается. Использовать массив animal для ввода безопасно до тех пор, пока ввод достаточно краток, чтобы поместиться в отведенный массив. Однако было бы неправильно применять для ввода указатель bird:

- Некоторые компиляторы трактуют строковые литералы как константы, доступные только для чтения, что ведет к ошибкам времени выполнения при попытках писать данные через них. То, что строковые литералы являются константами — обязательное поведение C++, но еще не все компиляторы отказались от старого подхода при их обработке.
- Некоторые компиляторы используют только одну копию строкового литерала для представления всех его копий, встреченных в тексте программы.

Давайте проясним второй пункт. C++ не гарантирует, что строковый литерал сохраняется уникально. То есть, если вы используете литерал "wren" несколько раз в программе, компилятор может сохранить либо несколько копий этой строки, либо всего одну. Если он сохраняет одну, то установка в bird адреса "wren" заставляет его указать на единственную копию этой строки. Чтение значения в одну строку может задевать те строки, которые изначально имели то же значение, но логически были совершенно независимы, встречаясь в других местах программы. В любом случае, поскольку указатель bird объявлен как const, компилятор предотвратит любые попытки изменить содержимое, расположенное по адресу, на который указывает bird.

Еще хуже обстоят дела с попытками чтения информации в место, на которое указывает ps. Поскольку ps не инициализирован, вы не можете знать, куда попадет введенная информация. Она может даже перезаписать ту информацию, которая уже имеется в памяти. К счастью, такой проблемы легко избежать: вы просто используете достаточно большой массив char, чтобы принять ввод, но не используете для ввода строчных констант и неинициализированных указателей.

Внимание!

Когда вы читаете строку в программу, вы всегда должны использовать адрес ранее распределенной памяти. Этот адрес может иметь форму имени массива либо указателя, инициализированного операцией new.

Далее обратите внимание на то, что делает следующий код:

```
ps = animal; // установить в ps указатель на строку
...
cout << animal << "по адресу " << (int *) animal << endl;
cout << ps << "по адресу " << (int *) ps << endl;
```

Он генерирует следующий вывод:

```
fox по адресу 0x0065fd30
fox по адресу 0x0065fd30
```

Обычно если вы передаете cout указатель, он печатает адрес. Но если указатель имеет тип char *, то cout отображает строку, на которую установлен указатель. Если вы хотите увидеть адрес строки, для этого нужно выполнить приведение типа к указателю на другой тип, такой как int *, что и делает этот код. Поэтому ps отображается как строка "fox", но (int *)ps выводится как адрес, по которому эта строка находится. Обратите внимание, что присваивание animal переменной ps не копирует строку; оно копирует только адрес. В результате два указателя (animal и ps) указывают на одно и то же место в памяти — то есть на одну строку.

Чтобы получить копию строки, потребуется сделать кое-что еще. Во-первых, распределить память для хранения копии строки. Это можно сделать либо объявив еще

один массив, либо используя new. Второй подход позволяет точно настроить размер хранилища для строки:

```
ps = new char[strlen(animal) + 1]; // получить новое хранилище
```

Строка "fox" не полностью заполняет массив animal, поэтому такой подход впустую тратит память. Здесь же мы видим использование strlen() для нахождения длины строки, а затем к найденной длине прибавляется единица, чтобы получить длину, включающую нулевой символ. Далее программа применяет new для выделения достаточного пространства для сохранения строки.

Вам необходим способ копирования строки из массива animal во вновь выделенное пространство. Присваивание animal ps не работает, потому что это только изменяет адрес, сохраненный в ps, при этом утрачивается единственная возможность доступа к выделенной памяти. Поэтому вместо этого нужно использовать библиотечную функцию strcpy():

```
strcpy(ps, animal); // скопировать строку в новое хранилище
```

Функция strcpy() принимает два аргумента. Первый — целевой адрес, а второй — адрес строки, которую следует скопировать. Ваша обязанность — обеспечить, чтобы место назначения действительно вместило копируемую строку. Здесь это обеспечивается применением strlen() для определения корректного размера и применением new для получения свободной памяти. Обратите внимание, что после использования strlen() и new вы получили две отдельных копии "fox":

```
fox at 0x0065fd30
fox at 0x004301c8
```

Также обратите внимание, что новое хранилище находится в памяти довольно далеко от того места, где хранится содержимое animal.

Вам часто придется сталкиваться с необходимостью размещения строки в массиве. Вы используете операцию = при инициализации массива; в противном случае применяется функция strcpy() или strncpy(). Вы уже видели функцию strcpy(); она работает следующим образом:

```
char food[20] = "carrots";   // инициализация
strcpy(food, "flan");        // альтернатива
```

Необходимо отметить, что следующий подход:

```
strcpy(food, "a picnic basket filled with many goodies");
```

может послужить причиной проблем, если массив food окажется меньше, чем строка. В этом случае функция копирует остаток строки в байты памяти, непосредственно следующие за массивом, при этом перезаписывая ее содержимое, несмотря на то, что, возможно, программа ее использует для других целей. Чтобы избежать такой проблемы, вместо strcpy() вы должны применять strncpy(). Эта функция принимает третий аргумент — максимальное количество копируемых символов. Однако при этом имейте в виду, что если эта функция скопирует указанное число символов и не достигнет конца строки, то она не добавит нулевой символ. Потому применять ее нужно так:

```
strncpy(food, "a picnic basket filled with many goodies", 19);
food[19] = '\0';
```

Этот код копирует до 19 символов в массив, затем устанавливает в последний элемент массива нулевой символ. Если строка короче, чем 19 символов, то `strncpy()` добавит нулевой символ ранее, пометив им действительный конец строки.

На память!

Для копирования строки в массив применяйте `strcpy()` или `strncpy()`, а не операцию присваивания.

Теперь, когда вы ознакомились с некоторыми аспектами применения строк в стиле C и библиотеки `cstring`, вы сможете по достоинству оценить сравнительную простоту использования типа `string` C++. Обычно вам не следует беспокоиться о переполнении массива строкой, и вы можете использовать операцию присваивания вместо `strcpy()` или `strncpy()`.

Использование new для создания динамических структур

Вы уже видели, насколько выгодно может быть создание массивов во время выполнения по сравнению с их жестким построением во время компиляции. То же самое касается структур. Вам нужно выделить пространство для стольких структур, сколько понадобится программе при каждом конкретном запуске. Опять-таки операция `new` послужит инструментом для этого. С его помощью вы можете создавать динамические структуры. *Динамические* означает выделение памяти во время выполнения, а не во время компиляции. Кстати, поскольку классы очень похожи на структуры, вы сможете использовать технику, изученную здесь, как со структурами, так и с классами.

Применение `new` со структурами состоит из двух частей: создания структуры и обращения к ее членам. Чтобы создать структуру, вы используете тип структуры с операцией `new`. Например, чтобы создать безымянную структуру типа `inflatable` и присвоить ее адрес соответствующему указателю, вы можете поступить следующим образом:

```
inflatable * ps = new inflatable;
```

Это присвоит указателю `ps` адрес участка памяти достаточной величины, чтобы вместить тип `inflatable`. Обратите внимание, что синтаксис в точности такой же, как и для встроенных типов C++.

Более сложная часть — доступ к членам. Когда вы создаете динамическую структуру, вы не можете применить операцию членства с именем структуры, потому что такая структура безымянна. Все, что у вас есть — ее адрес. В C++ предусмотрена специальная операция для этой ситуации — операция членства через указатель (`->`). Эта операция, оформленная в виде тире с последующим значком "больше", значит для указателей на структуры то же, что операция точки значит для имен структур. Например, если `ps` указывает на структуру типа `inflatable`, то `ps->price` означает член `price` структуры, на которую указывает `ps`. (См. рис. 4.11.)

На память!

Иногда новички в C++ путаются в том, когда нужно применять операцию точки, а когда — операцию стрелочки при обращении к членам структуры. Правило простое: если идентификатор структуры есть имя структуры, используйте точку. Если же идентификатор является указателем на структуру, используйте операцию стрелочки.

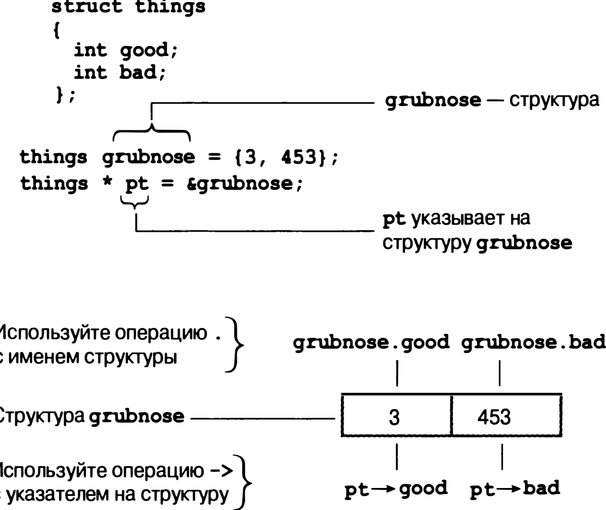


Рис. 4.11. Идентификация членов структуры

Имеется еще один, довольно безобразный подход для обращения к членам структур; он принимает во внимание, что если `ps` — указатель на структуру, то `*ps` — сама структура. Поэтому, если `ps` — структура, то `(*ps).price` — ее член `price`. Правила приоритетов операций C++ требуют применения скобок в этой конструкции.

В листинге 4.21 используется операция `new` для создания неименованной структуры и демонстрируются оба варианта нотации для доступа к ее членам.

Листинг 4.21. newstrct.cpp

```
// newstrct.cpp -- использование new со структурой
#include <iostream>
struct inflatable // шаблон структуры
{
   char name[20];
   float volume;
   double price;
};
int main()
{
   using namespace std;
   inflatable * ps = new inflatable;  // выделить память для структуры
   cout << "Укажите имя для элемента inflatable: ";
   cin.get(ps->name, 20);             // метод 1 доступа к членам
   cout << "Укажите объем в кубических футах: ";
   cin >> (*ps).volume;               // метод 2 доступа к членам
   cout << "Введите цену: $";
   cin >> ps->price;
   cout << "Имя: " << (*ps).name << endl; // метод 2
   cout << "Объем: " << ps->volume << " cubic feet\n"; // метод 1
   cout << "Цена: $" << ps->price << endl; // метод 1
   delete ps; // освободить память, использованную структурой
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.21:

```
Укажите имя для элемента inflatable: Fabulous Frodo
Укажите объем в кубических футах: 1.4
Введите цену: $27.99
Имя: Fabulous Frodo
Объем: 1.4 cubic feet
Цена: $27.99
```

Пример использования new и delete

Давайте рассмотрим пример, использующий new и delete для управления размещением строкового ввода с клавиатуры. В листинге 4.22 определяется функция getname(), которая возвращает указатель на входную строку. Эта функция читает ввод в большой временный массив, а затем использует new[] с указанием соответствующего размера, чтобы выделить фрагмент памяти точно такого размера, чтобы вместить входную строку. После этого функция возвращает указатель на этот блок. Такой подход может сэкономить огромный объем памяти в программе, которая читает большое количество строк.

Предположим, что ваша программа должна прочитать 1000 строк, и что самая длинная из них может составлять 79 символов длины, но большинство строк — значительно короче. Если вы решите использовать массивы char для хранения строк, то вам понадобится 1000 массивов по 80 символов каждый. То есть 80000 байт, причем бóльшая часть этого блока памяти останется неиспользованной. В качестве альтернативы вы можете создать массив из 1000 указателей на char и применить new для выделения ровно такого объема памяти, сколько необходимо для каждой строки. Это может сэкономить десятки тысяч байт. Вместо того чтобы заводить большой массив для каждой строки, вы выделяете память, достаточную для размещения ввода. И более того, вы можете использовать new для выделения памяти лишь для стольких указателей, сколько будет входных строк в действительности. Ну, это уже чересчур амбициозно для начала. Даже массив из 1000 указателей — чересчур амбициозное решение для настоящего момента, но листинг 4.22 иллюстрирует эту технику. К тому же, чтобы проиллюстрировать работу операции delete, программа использует ее для освобождения выделенной памяти.

Листинг 4.22. delete.cpp

```
// delete.cpp -- использование операции delete
#include <iostream>
#include <cstring>        // или string.h
using namespace std;
char * getname(void);     // прототип функции
int main()
{
   char * name;            // создать указатель, но без хранилища
   name = getname();       // присвоить строку имени
   cout << name << " по адресу" << (int *) name << "\n";
   delete [] name;         // освободить память
   name = getname();       // повторно использовать освобожденную память
   cout << name << "по адресу " << (int *) name << "\n";
   delete [] name;         // опять освободить память
   return 0;
}
```

```
char * getname()           // вернуть указатель на новую строку
{
   char temp[80];          // временное хранилище
   cout << "Введите фамилию: ";
   cin >> temp;
   char * pn = new char[strlen(temp) + 1];
   strcpy(pn, temp);       // скопировать строку в меньшее пространство
   return pn;              // temp пропадает по завершении функции
}
```

Вот пример выполнения программы из листинга 4.22:

```
Введите фамилию: Fredeldumpkin
Fredeldumpkin по адресу 0x004326b8
Введите фамилию: Pook
Pook по адресу 0x004301c8
```

Замечания по программе

Рассмотрим функцию `getname()` из программы, представленной в листинге 4.22. Она использует `cin` для размещения введенного слова в массив `temp`. Далее она обращается к `new` для выделения памяти, достаточной, чтобы вместить это слово. Включая нулевой символ, программе требуется сохранить в строке `strlen(temp) + 1` символов, поэтому именно эта величина передается `new`. После получения пространства памяти `getname()` вызывает стандартную библиотечную функцию `strcpy()`, чтобы скопировать строку `temp` в выделенный блок памяти. Функция не проверяет, поместится ли строка, но `getname()` гарантирует выделение блока памяти правильного размера. В конце функция возвращает `ps` — адрес копии строки.

Внутри `main()` возвращенное значение (адрес) присваивается указателю `name`. Этот указатель объявлен в `main()`, но указывает на блок памяти, выделенный в функции `getname()`. Затем программа печатает строку и ее адрес.

Далее, после того, как она освободит блок, на который указывает `name`, `main()` вызывает `getname()` второй раз. C++ не гарантирует, что только что освобожденная память будет выделена при следующем вызове `new`, и, как видно из вывода программы, этого и не случается.

Обратите внимание, что в этом примере `getname()` выделяет память, а `main()` освобождает ее. Обычно это не слишком хорошая идея — размещать `new` и `delete` в разных функциях, потому что в этом случае очень легко забыть вызвать `delete`. В этом примере мы разделили эти две операции просто для того, чтобы показать, что такое возможно.

Благодаря некоторым тонким аспектам данной программы, вы должны узнать немного больше о том, как C++ управляет памятью. Поэтому давайте предварительно просмотрим некоторый материал, который будет раскрыт более подробно в главе 9.

Автоматическое, статическое и динамическое хранилища

C++ предлагает три способа управления памятью для данных — в зависимости от метода ее выделения: автоматическое хранилище, статическое хранилище и динамическое хранилище, иногда называемое *свободным хранилищем* или *кучей*. Объекты дан-

ных, выделенные этими тремя способами, отличаются друг от друга тем, насколько долго они существуют. Рассмотрим кратко каждый из них.

Автоматическое хранилище

Обычные переменные, объявленные внутри функции, используют *автоматическое хранилище* и называются *автоматическими переменными.* Этот термин означает, что они создаются автоматически при вызове содержащей их функции и уничтожаются при ее завершении. Например, массив temp в листинге 4.22 существует только во время работы функции getname(). Когда управление программой возвращается main(), то память, используемая temp, освобождается автоматически. Если бы getname() возвращала указатель на temp, то указатель name в main() остался бы установленным на адрес памяти, которая скоро будет использована повторно. Вот почему внутри getname() необходимо вызывать new.

На самом деле автоматические значения являются локальными по отношению к блоку, в котором они объявлены. *Блок* — это раздел кода, ограниченный фигурными скобками. До сих пор все наши блоки были целыми функциями. Но как будет показано в следующей главе, можно иметь блоки и внутри функций. Если вы объявите переменную внутри одного из таких блоков, она будет существовать только в то время, когда программа выполняет операторы, содержащиеся внутри этого блока.

Статическое хранилище

Статическое хранилище — это хранилище, которое существует в течение всего времени выполнения программы. Есть два способа сделать переменные статическими. Один заключается в объявлении их вне функций. Другой предполагает использование при объявлении переменной ключевого слова static:

```
static double fee = 56.50;
```

Согласно правилам языка C стандарта K&R, вы можете инициализировать только статические массивы и структуры, в то время как C++ Release 2.0 (и более поздние), а также ANSI C позволяют также инициализировать автоматические массивы и структуры. Однако, как вы, возможно, обнаружите, в некоторых реализациях C++ до сих пор не реализована инициализация таких массивов и структур.

В главе 9 статическое хранилище обсуждается более подробно. Главный момент, который вы должны запомнить сейчас относительно автоматическое и статического хранения — то, что эти методы строго определяют время жизни переменных. Переменные могут либо существовать на протяжении всего выполнения программы (статические переменные) либо только в период выполнения функции или блока (автоматические переменные).

Динамическое хранилище

Операции new и delete предлагают более гибкий подход, нежели использование автоматических и статических переменных. Они управляют пулом памяти, который в C++ называется *свободным хранилищем.* Этот пул отделен от области памяти, используемой статическими и автоматическими переменными. Как показано в листинге 4.22, операции new и delete позволяют выделять память в одной функции и освобождать в другой. Таким образом, время жизни данных при этом не привязывается жестко к времени жизни программы или функции. Совместное применение new и delete

предоставляет вам возможность более тонко управлять использованием памяти, чем в случае обычных переменных.

Пример из практики: стеки, кучи и утечка памяти

Что случится, если вы не вызовите `delete` после создания переменной операцией `new` в области кучи? Переменная или конструкция, динамически выделенная в области свободного хранилища, останется там, если не вызвать `delete`, даже если память, содержащая указатель, будет освобождена в соответствии с правилами видимости и времени жизни объектов. По сути, после этого у вас не будет никакой возможности получить доступ к такой конструкции, находящейся в области свободного хранилища, поскольку уже не будет существовать указатель, который помнит ее адрес. В этом случае вы получите *утечку памяти*. Такая память остается недоступной на протяжении всего сеанса работы программы. Она была выделена, но не может быть освобождена. В крайних случаях (хотя и нечастых), утечки памяти могут привести к тому, что они поглотят всю память, выделенную программе, что вызовет ее аварийное завершение с сообщением об ошибке переполнения памяти. Вдобавок такие утечки могут негативно повлиять на некоторые операционные системы и другие приложения, использующие то же пространство памяти, приводя к их сбоям.

Даже лучшие программисты и программистские компании допускают утечки памяти. Чтобы избежать их, лучше выработать привычку сразу объединять операции `new` и `delete`, тщательно планируя создание и удаление конструкций, как только вы собираетесь обратиться к динамической памяти.

На заметку!

Указатели — одно из наиболее мощных средств C++. Однако они также и наиболее опасны, потому что открывают возможность недружественных к компьютеру действий, таких как использование неинициализированных указателей для доступа к памяти, либо попыток освобождения одного и того же блока дважды. Более того, до тех пор, пока вы не привыкнете в нотации указателей и к самой концепции указателей на практике, они будут приводить вас в замешательство. Но поскольку указатели — важнейшая часть программирования на C++, они постоянно будут присутствовать во всех материалах нашей книги. Эта книга будет обращаться к теме указателей еще много раз. Мы надеемся, что каждое объяснение поможет вам чувствовать себя все более уверенно при работе с указателями.

Резюме

Массивы, структуры и указатели — три составных типа в C++. Массив может содержать множество значений одного и того же типа в одном объекте данных. Используя индекс, вы получаете доступ к индивидуальным элементам массива.

Структура может содержать несколько значений разных типов в одном объекте данных, и для доступа к ним вы можете использовать операцию членства (`.`). Первым шагом в применении структуры является создание шаблона структуры, который определяет члены, входящие в нее. Имя, или дескриптор такого шаблона затем становится идентификатором нового типа данных. Далее вы можете объявлять структурные переменные этого типа.

Объединение может содержать единственное значение, которое может трактоваться разными типами, причем имя члена при этом указывает, какой режим (тип) используется в данный момент.

Указатели — это переменные, которые предназначены для хранения адресов памяти. Говорят, что указатель указывает на адрес, который он хранит. Объявление указателя всегда включает тип объекта, на который он указывает. Применяя операцию

разыменования (*), можно получить значение, находящееся в памяти по адресу, хранящемуся в указателе.

Строка — это серия символов, ограниченная нулевым символом. Строка может быть представлена в виде строковой константы, заключенной в кавычки, при этом нулевой символ подразумевается неявно. Вы можете сохранить строку в массиве char и представить строку как указатель на char, представляющий ее начальный символ. Функция strlen() возвращает длину строки, исключая нулевой символ-ограничитель. Функция strcpy() копирует строку из одного места в другое. Для применения этих функций необходимо включить в программу заголовочный файл cstring или string.h.

Класс C++ string, поддерживаемый заголовочным файлом string, предлагает альтернативный, более дружественный к пользователю способ обращения со строками. В частности, объекты string автоматически изменяют свои размеры, приспосабливаясь к хранимым строкам, а для копирования их можно применять операцию присваивания.

Операция new позволяет запрашивать память для объектов данных во время выполнения программы. Эта операция возвращает адрес выделенного участка памяти, и вы можете присвоить его указателю. Единственный способ доступа к такой памяти — через указатель. Если объект данных — это простая переменная, вы можете применить операцию разыменования (*) для получения значения. Если объект данных — массив, вы можете использовать указатель на него как обычное имя массива и обращаться к его элементам по индексу. Если же объект данных — структура, вы можете использовать операцию -> для доступа к ее членам.

Указатели и массивы тесно связаны. Если ar — имя массива, то выражение ar[i] интерпретируется как *(ar + i), то есть имя массива интерпретируется как адрес его первого элемента. Таким образом, имя массива играет ту же роль, что и указатель. В свою очередь, вы можете использовать имя указателя в нотации массива, чтобы обращаться к элементам массива, выделенным операцией new.

Операции new и delete позволяют явно контролировать, когда объекты данных размещаются и когда покидают пул свободной памяти. Автоматические переменные, которые объявляются внутри функций, и статические переменные, определяемые вне функций или же с помощью ключевого слова static, являются менее гибкими. Автоматические переменные создаются, когда управление приходит в содержащий их блок (обычно в теле функции), и исчезают, когда оно его покидает. Статические переменные сохраняются в течение всего времени исполнения программы.

Вопросы для самоконтроля

1. Как вы объявите следующие объекты данных?
   а. actor — массив из 30 char
   б. betsie — массив из 100 short.
   в. chuck — массив из 13 float.
   г. dipsea — массив из 64 long double.
2. Объявите массив из пяти int, и инициализируйте его первыми пятью четными числами.
3. Напишите оператор, который присваивает сумму первого и последнего элементов массива из вопроса 2 переменной even.

4. Напишите оператор, отображающий значение второго элемента массива float под именем ideas.
5. Объявите массив char и инициализируйте его строкой "cheeseburger".
6. Предложите объявление структуры, описывающей рыбу. Структура должна включать вид, вес в полных унциях и длину в дробных дюймах.
7. Объявите переменную типа, определенного в вопросе 6, и инициализируйте ее.
8. Используйте enum для определения типа по имени Response с возможными значениями Yes, No и Maybe. Yes должно быть равно 1, No — 0, а Maybe — 2.
9. Предположим, что ted — переменная типа double. Объявите указатель, указывающий на ted, и воспользуйтесь им, чтобы отобразить значение ted.
10. Предположим, что treacle — массив из 10 значений float. Объявите указатель, указывающий на первый элемент treacle, и используйте его для отображения первого и последнего элементов массива.
11. Напишите фрагмент кода, который просит пользователя ввести положительное целое число, и затем создает динамический массив указанного размера элементов типа int.
12. Правильный ли код приведен ниже? Если да, что он напечатает?

    ```
    cout << (int *) "Home of the jolly bytes";
    ```

13. Напишите фрагмент кода, который динамически выделит память для структуры, описанной в вопросе 6, и затем прочитает в нее значение члена kind.
14. В листинге 4.6 иллюстрируется проблема, вызванная тем, что числовой ввод следует за строчно-ориентированным вводом. Как замена оператора

    ```
    cin.getline(address,80);
    ```

    на оператор

    ```
    cin >> address;
    ```

    повлияет на работу этой программы?

Упражнения по программированию

1. Напишите программу C++, которая запрашивает и отображает информацию, как показано в следующем примере вывода:

    ```
    What is your first name? Betty Sue
    What is your last name? Yew
    What letter grade do you deserve? B
    What is your age? 22
    Name: Yew, Betty Sue
    Grade: C
    Age: 22
    ```

    Обратите внимание, что программа должна принимать имена, состоящие из более чем одного слова. Также отметьте, что программа должна уменьшать значение grade на один шаг — то есть, на одну букву выше. Предполагается, что пользователь может ввести A, B или C, поэтому вам не нужно беспокоиться о пропуске между D и F.
2. Перепишите листинг 4.4, применив класс C++ string вместо массивов char.

3. Напишите программу, которая просит пользователя ввести ее или его имя, затем фамилию, а затем построит, сохранит и отобразит третью строку, состоящую из фамилии пользователя, за которой следует запятая, пробел и его имя. Используйте массивы `char` и функции из заголовочного файла `cstring`. Пример запуска должен выглядеть так:

```
Enter your first name: Flip
Enter your last name: Fleming
Here's the information in a single string: Fleming, Flip
```

4. Напишите программу, которая приглашает пользователя ввести его имя и фамилию, а затем построит, сохранит и отобразит третью строку, состоящую из фамилии, за которой следует запятая, пробел и имя. Используйте объекты `string` и методы из заголовочного файла `string`. Пример запуска должен выглядеть так:

```
Enter your first name: Flip
Enter your last name: Fleming
Here's the information in a single string: Fleming, Flip
```

5. Структура `CandyBar` включает три члена. Первый из них содержит наименование коробки конфет. Второй – ее вес (который может иметь дробную часть), а третий – число калорий (целое значение). Напишите программу, объявляющую эту структуру и создающую переменную типа `CandyBar` по имени `snack`, инициализируя ее члены значениями `"Mocha Munch"`, `2.3` и `350`, соответственно. Инициализация должна быть частью объявления `snack`. И, наконец, программа должна отобразить содержимое этой переменной.

6. Структура `CandyBar` включает три члена, как описано в предыдущем упражнении. Напишите программу, которая создает массив из трех структур `CandyBar`, инициализирует их значениями на ваше усмотрение и затем отображает содержимое каждой структуры.

7. Вильям Вингейт (William Wingate) заведует службой анализа пиццы. О каждой пицце он записывает следующую информацию:

   - Наименование компании – производителя пиццы, которое может состоять из более чем одного слова.
   - Диаметр пиццы.
   - Вес пиццы.

   Разработайте структуру, которая может содержать всю эту информацию, и напишите программу, использующую структурную переменную этого типа. Программа должна запрашивать у пользователя ввод каждого из перечисленных показателей и затем отображать введенную информацию. Применяйте `cin` (или его методы) и `cout`.

8. Выполните упражнение 7, но с применением `new` для размещения структуры в свободной памяти вместо объявления структурной переменной. Кроме того, сделайте так, чтобы программа сначала запрашивала диаметр пиццы, а потом – наименование компании.

9. Выполните упражнение 6, но вместо объявления массива из трех структур `CandyBar` используйте операцию `new` для динамического размещения массива.

ГЛАВА 5

Циклы и выражения отношений

В этой главе:

- Цикл `for`
- Выражения и операторы
- Операции инкремента и декремента: `++` и `--`
- Комбинированные операции присваивания
- Составные операторы (блоки)
- Операция запятой
- Операции сравнения: `>`, `>=`, `==`, `<=`, `<` и `!=`
- Цикл `while`
- Средство `typedef`
- Цикл `do while`
- Метод ввода символов `get()`
- Условие конца файла
- Вложенные циклы и двумерные массивы

Компьютеры умеют много больше, чем просто хранить данные. Они анализируют, консолидируют, упорядочивают, извлекают, модифицируют, экстраполируют, синтезируют и выполняют другие манипуляции над данными. Иногда они даже искажают и уничтожают данные, но мы стараемся управлять таким их поведением. Чтобы творить все эти чудеса, программам необходимы инструменты для выполнения повторяющихся действий и принятия решений. Конечно, C++ представляет такие инструменты. В самом деле, он использует те же циклы `for`, `while`, `do while`, операторы `if`, `switch`, которые есть в языке C, поэтому если вы знаете C, то можете быстро пробежаться по этой и шестой главам (но не слишком быстро — вы же не хотите пропустить объяснение того, как объект `cin` обрабатывает символьный ввод!). Все эти разнообразные управляющие операторы часто используют сравнивающие и логические выражения, и глава 6 продолжит рассмотрение операторов ветвления и логических выражений.

Введение в цикл `for`

Обстоятельства часто требуют от программ выполнения повторяющихся задач, таких как сложение элементов массивов один за другим или 20-кратная распечатка похвалы за продуктивность. Цикл `for` облегчает выполнение задач подобного рода.

Давайте взглянем на цикл в листинге 5.1, посмотрим, что он делает, а затем разберемся, как он работает.

Листинг 5.1. forloop.cpp

```
// forloop.cpp -- представление цикла for
#include <iostream>
int main()
{
   using namespace std;
   int i; // создадим счетчик
   // инициализация; проверка; обновление
   for (i = 0; i < 5; i++)
       cout << "C++ знает циклы.\n";
   cout << "C++ знает, где остановиться. \n";
   return 0;
}
```

Ниже показан вывод программы из листинга 5.1:

```
C++ знает циклы.
C++ знает циклы.
C++ знает циклы.
C++ знает циклы.
C++ знает циклы.
C++ знает, где остановиться.
```

Этот цикл начинается с присваивания целочисленной переменной `i` значения `0`:

```
i = 0;
```

Это — *инициализирующая часть* цикла. Затем, в *проверочной части*, программа проверяет, меньше ли `i` числа `5`:

```
i < 5;
```

Если это так, программа выполняет следующий оператор, который называется *телом цикла*:

```
cout << "C++ знает циклы.\n";
```

После этого программа активизирует *обновляющую часть* цикла, увеличивая `i` на единицу:

```
i++
```

Обновляющая часть цикла использует операцию ++, называемую *операцией инкремента*. Она увеличивает значение своего операнда на 1. (Применение операции инкремента не ограничено циклами `for`. Например, вы можете использовать `i++;` вместо `i = i + 1;` в качестве операторы программы.) Увеличение `i` завершает первый шаг цикла.

Далее цикл начинает новый шаг, сравнивая новое значение `i` с `5`. Поскольку новое значение (`1`) также меньше `5`, цикл печатает еще одну строку и завершается новым увеличением значения `i`. Это подготавливает новый шаг цикла — проверку, выполнение оператора и обновления значения `i`. Процесс продолжается до тех пор, пока не `i` не получит значение `5`. После этого следующая проверка дает ложный результат, и программа переходит к оператору, следующему за циклом.

Части цикла for

Цикл `for` представляет собой средство пошагового выполнения повторяющихся действий. Рассмотрим более подробно, как он устроен. Обычно части цикла `for` выполняют следующие шаги:

1. Установка начального значения.
2. Выполнение проверки условия продолжения цикла.
3. Выполнение действий цикла.
4. Обновление значения (значений), используемых в проверочном условии.

Дизайн цикла C++ располагает эти элементы таким образом, чтобы их можно было сразу охватить одним взглядом. Инициализация, проверка и обновление составляют три части управляющего раздела, заключенные в скобки. Каждая часть — выражение, а точки с запятой отделяют их друг от друга. Оператор, следующий за управляющим разделом, называется *телом* цикла, и он выполняется до тех пор, пока проверочное условие остается истинным:

```
for (инициализация; проверочное выражение; обновляющее выражение)
  тело
```

Синтаксис C++ считает полный оператор `for` единым целым оператором программы, даже несмотря на то, что он может заключать в своем теле один или более других операторов. (При наличии нескольких операторов в теле цикла они должны быть заключены с блок, как будет описано ниже.)

Цикл выполняет инициализацию только однажды. Как правило, программы используют это выражение для установки переменной в некоторое начальное значение, а потом используют эту переменную в качестве счетчика цикла.

Проверочное выражение определяет, должно ли выполняться тело цикла. Обычно это выражение представляет собой выражение сравнения — то есть выражение, сравнивающее два значения. Наш пример сравнивает значение `i` с `5`. Если проверка проходит успешно (проверочное выражение истинно), программа выполняет тело цикла. На самом деле C++ не сводит проверочные выражения к сравнениям, возвращающим `true` или `false`. Вы можете использовать здесь любое выражение, и C++ приведет его к типу `bool`. Таким образом, выражение, возвращающее `0`, конвертируется в булевское значение `false`, и цикл завершается. Если выражение оценивается как ненулевое, оно приводится к булевскому значению `true`, и цикл продолжается. В листинге 5.2 это демонстрируется на примере использования выражения `i` в качестве проверочного. (В разделе обновления `i--` подобно `i++`, за исключением того, что оно уменьшает значение `i` на 1 при каждом выполнении.)

Листинг 5.2. num_test.cpp

```
// num_test.cpp -- использование числовой проверки в цикле
#include <iostream>
int main()
{
  using namespace std;
  cout << "Введите начальное значение для обратного отсчета: ";
  int limit;
  cin >> limit;
```

```
    int i;
    for (i = limit; i; i--) // quits when i is 0
        cout << "i = " << i << "\n";
    cout << "Готово, теперь i = " << i << "\n";
    return 0;
}
```

Вот как выглядит пример выполнения программы из листинга 5.2:

```
Введите начальное значение для обратного отсчета: 4
i = 4
i = 3
i = 2
i = 1
Готово, теперь i = 0
```

Обратите внимание, что цикл завершается, когда `i` достигает значения 0.

Как же сравнивающие выражения, вроде `i < 5`, вписываются в концепцию прерывания цикла при получении значения 0? До появления типа `bool` сравнивающие выражения вычислялись как `1` в случае истинности и `0` — в противоположном случае. Таким образом, значение выражения `3 < 5` было равно `1`, а значение `5 < 5` — `0`. Теперь, когда в C++ добавлен тип `bool`, сравнивающие выражения возвращают булевские литералы `true` и `false` вместо `1` и `0`. Однако это изменение не приводит к несовместимости, потому что программы C++ преобразуют `true` и `false` в `1` и `0` там, где ожидаются целые значения, а `0` преобразует в `false`, и не ноль в `true` — там, где ожидаются значения типа `bool`.

Цикл `for` является циклом с входным условием. Это значит, что проверочное условие выполняется *перед* каждым шагом цикла. Цикл никогда не выполняет тело, если проверочное условие возвращает `false`. Например, представьте, что вы запустили программу из листинга 5.2, но в качестве начального значения ввели `0`. Поскольку проверочное условие не удовлетворено при первой же проверке, тело цикла не выполняется ни разу:

```
Введите начальное значение для обратного отсчета: 0
Готово, теперь i = 0
```

Позиция "проверка перед циклом" может помочь предохранить программу от проблем.

Обновляющее выражение выполняется в конце цикла, после тела цикла. Обычно оно используется для увеличения или уменьшения значения переменной, управляющей количеством шагов цикла. Однако оно может быть любым корректным выражением C++, как и все остальные управляющие выражения. Это обеспечивает циклу `for` гораздо более широкие возможности, чем простой отсчет от 0 до 5, как это делается в первом примере. Позже вы увидите некоторые примеры этих возможностей.

Тело цикла `for` состоит из одного оператора, но вскоре вы увидите, как расширить это правило. На рис. 5.1 подводится итог описанию дизайна цикла `for`.

Оператор цикла `for` выглядит несколько похожим на вызов функции, потому что использует имя, за которым следует пара скобок. Однако статус `for` как ключевого слова C++ предотвращает восприятие его компилятором как функции. Это также предохраняет вас от попыток назвать функцию именем `for`.

```
оператор1
for (инициализационное_выражение;
     проверочное_выражение;
     обновляющее_выражение)
   оператор2
оператор3
```

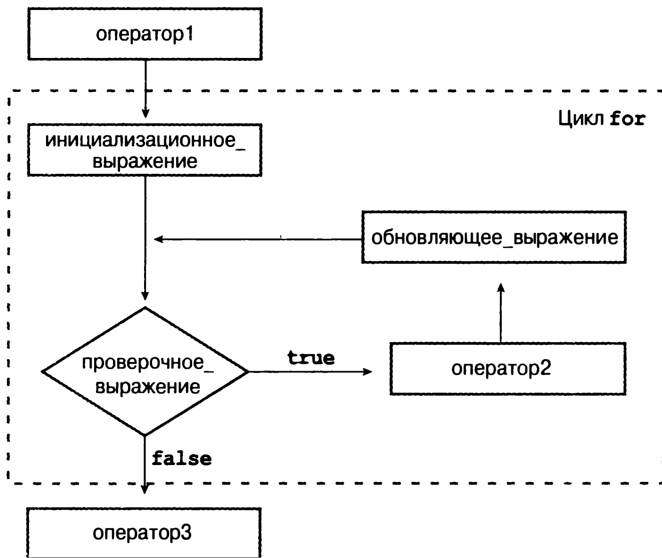


Рис. 5.1. Структура циклов `for`

На заметку!

В С++ принят стиль помещения пробелов между `for` и последующими скобками, а также пропуск пробела между именем функции и следующими за ним скобками:

```
for (i = 6; i < 10; i++)
      smart_function(i);
```

Другие управляющие операторы, такие как `if` и `while`, трактуются аналогично `for`. Это призвано визуально подчеркнуть разницу между управляющим оператором и вызовом функции. К тому же общепринятая практика заключается в снабжении тела функции отступом, чтобы выделить его визуально.

Выражения и операторы

Управляющий раздел `for` включает три выражения. В пределах ограничений, накладываемых его синтаксисом, С++ — очень выразительный язык. Любое значение или любая корректная комбинация выражений и операций составляет выражение. Например, `10` — это выражение со значением 10 (что не удивительно), а `28 * 20` — выражение со значением 560. В С++ любое выражение имеет свое значение. Часто оно вполне очевидно. Например, выражение

```
22 + 27
```

формируется из двух значений и операции сложения, и оно равно 49. Иногда значение не столь очевидно. Например:

```
x = 20
```

это тоже выражение, потому что формируется из двух значений и операции присваивания. C++ определяет значение выражения присваивания как значение его левой части, поэтому это выражение имеет значение 20. Тот факт, что выражения присваивания имеют значения, означает, что допускаются выражения вроде такого:

```
maids = (cooks = 4) + 3;
```

Выражение `cooks = 4` имеет значение 4, поэтому `maids` присваивается значение 7. Однако то, что C++ допускает подобное поведение, не значит, что вы должны злоупотреблять им. Но то же правило, которое разрешает такие специфические операторы, также значит, что разрешено следующее:

```
x = y = z = 0;
```

Это простой и быстрый способ установить значение нескольким переменным одновременно. Таблица приоритетов (см. приложение Г) показывает, что присваивание ассоциируется справа налево, поэтому первый `0` присваивается `z`, а `z = 0` присваивается `y` и так далее.

И, наконец, как упоминалось ранее, сравнивающие выражения, такие как `x < y`, вычисляются как `bool`-значения `true` и `false`. Короткая программа в листинге 5.3 иллюстрирует некоторые моменты, связанные со значениями выражений. Операция << имеет более высокий приоритет, чем операции, использованные в выражениях, поэтому этот код использует скобки, чтобы задать правильный порядок разбора.

Листинг 5.3. express.cpp

```
// express.cpp -- значения выражений
#include <iostream>
int main()
{
   using namespace std;
   int x;
   cout << "Выражение x = 100 имеет значение ";
   cout << (x = 100) << endl;
   cout << "Теперь x = " << x << endl;
   cout << "Выражение x < 3 имеет значение ";
   cout << (x < 3) << endl;
   cout << "Выражение x > 3 имеет значение ";
   cout << (x > 3) << endl;
   cout.setf(ios_base::boolalpha);     // новейшее средство C++
   cout << "Выражение x < 3 имеет значение ";
   cout << (x < 3) << endl;
   cout << "Выражение x > 3 имеет значение ";
   cout << (x > 3) << endl;
   return 0;
}
```

Замечание по совместимости

Старые реализации C++ могут потребовать применения `ios::boolalpha` вместо `ios_base::boolalpha` в качестве аргумента `cout.setf()`. Еще более старые реализации могут вообще не распознавать ни одну из форм.

Ниже показан вывод программы из листинга 5.3:

```
Выражение x = 100 имеет значение 100
Теперь x = 100
Выражение x < 3 имеет значение 0
Выражение x > 3 имеет значение 1
Выражение x < 3 имеет значение false
Выражение x > 3 имеет значение true
```

Обычно `cout` преобразует значения `bool` в `int` перед тем, как отобразить их, но вызов функции `cout.setf(ios::boolalpha)` устанавливает флаг, который инструктирует `cout` отображать `true` и `false` вместо `1` и `0`.

На память!

Выражение C++ — это значение или комбинация значений и операций, и каждое выражение C++ имеет свое значение.

Чтобы вычислить выражение `x = 100`, C++ должен присвоить значение `100` переменной x. Когда в результате вычисления выражения изменяется значение данных в памяти, мы говорим, что вычисление имеет *побочный эффект*. Таким образом, вычисление выражения присваивания в качестве побочного эффекта изменяет значение переменной, которой осуществляется присваивание. Вы можете думать о присваивании как о главном эффекте, но с точки зрения внутреннего устройства C++ первичный эффект — это вычисление выражения. Не все выражения имеют побочные эффекты. Например, вычисление `x + 15` дает новое значение, но не меняет значения x. Однако вычисление `++x + 15` имеет побочный эффект, потому что включает в себя инкремент x.

От выражения до оператора программы один шаг; для этого достаточно добавить точку с запятой. То есть

```
age = 100
```

есть выражение, в то время как

```
age = 100;
```

уже оператор. Любое выражение может стать оператором, если к нему добавить точку с запятой, но результат может не иметь смысла с точки зрения программы. Например, если `rodents` — переменная, то

```
rodents + 6; // правильное, но бессмысленное выражение
```

представляет собой корректный оператор C++. Компилятор допускает его, но этот оператор не делает ничего полезного. Программа просто вычисляет его, но результат никак не использует и переходит к следующему оператору. (Интеллектуальный компилятор может даже пропустить такой оператор.)

Не-выражения и операторы

Некоторые концепции, такие как структуры или цикл `for`, являются ключевыми для понимания С++. Но есть также относительно менее значимые аспекты синтаксиса, которые иногда могут сбить вас с толку, когда вы уже думаете, что вполне понимаете язык. Парочку из них мы сейчас рассмотрим.

Хотя справедливо то, что добавление точки с запятой к любому выражению превращает его в оператор, обратное не верно. То есть, исключение точки с запятой из оператора не обязательно преобразует его в выражение. Из всех разновидностей операторов, которые мы рассмотрели до сих пор, операторы возврата, операторы объявления и операторы `for` не укладываются в правило *оператор* = *выражение* + *точка с запятой*. Например, это — оператор:

```
int toad;
```

Но фрагмент `int toad` не является выражением и не имеет значения. Это делает следующий код неправильным:

```
eggs = int toad * 1000;    // не верно, это — не выражение
cin >> int toad;           // нельзя комбинировать объявление с cin
```

Аналогично, вы не можете присваивать цикл `for` переменной. В следующем примере цикл `for` — это не выражение, поэтому он не имеет значения, и вы не можете присваивать его:

```
int fx = for (i = 0; i< 4; i++)
   cout >> i; // невозможно
```

Отклонения от правил

Язык С++ добавляет к циклам С возможность, которая требует некоторой поправки к синтаксису цикла `for`. Таков исходный синтаксис:

```
for (выражение; выражение; выражение)
   оператор
```

В частности, управляющий раздел конструкции `for` состоит из трех выражений, разделенных точками с запятой, как это указывалось ранее в настоящей главе. Однако циклы С++ позволяют вам делать вещи вроде:

```
for (int i = 0; i < 5; i++)
```

То есть, вы можете объявить переменную в области инициализации цикла `for`. Это может быть удобно, но не вписывается в исходный синтаксис, поскольку объявление не является выражением. Это единственное незаконное поведение, которое подгоняется к правилу определением нового вида выражений — *выражения оператора объявления*, которое представляет собой объявление без точки с запятой, и которое может появляться только в операторе `for`. Однако эта поправка была отброшена. Вместо нее был модифицирован синтаксис оператора `for`:

```
for (оператор-инициализации-for условие; выражение)
   оператор
```

На первый взгляд это выглядит не вполне ясно, потому что содержит только одну точку с запятой вместо двух. Но здесь все в порядке, потому что *оператор-инициализации-for* идентифицируется как оператор, а оператор имеет свою собственную точку с запятой. Что касается оператора *оператор-инициализации-for*, он идентифицируется и как выражение-оператор, и как объявление. Это синтаксическое правило заменяет выражение с последующей точкой с запятой оператором, имеющим свою собственную точку с запятой. Следствием этого является возможность для программистов C++ объявлять и инициализировать переменные внутри оператора цикла `for`, и они могут теперь выразить все, что нужно, в синтаксисе C++ и английского языка.

Есть практический аспект объявления переменной внутри *оператор-инициализации-for*, о чем вы должны знать. Такая переменная существует только внутри оператора `for`. То есть после того, как программа покидает цикл, переменная исчезает:

```
for (int i = 0; i < 5; i++)
   cout << "C++ знает циклы. \n";
cout << i << endl; // Оп! i больше не определена
```

Еще одна вещь, о которой следует знать — это то, что некоторые реализации C++ придерживаются старых правил и трактуют приведенный выше цикл, как если бы `i` была объявлена *перед* циклом, что делает ее доступной и после окончания цикла. То есть, применение этой новой опции при объявлении переменной в инициализации цикла `for` приводит, по крайней мере, в настоящее время, к разному поведению программы на разных системах.

Внимание!

На момент написания этой книги не все компиляторы реализовывали текущее правило о том, что переменные, объявленные в разделе инициализации цикла `for`, исчезают по его завершении. Например, Microsoft Visual C++ до версии 7.1 по умолчанию следовал старому правилу — в основном потому, что большая часть кода в существующих библиотеках Microsoft была написана до появления этого правила. (Версия 7.1 предпринимает героическую, но нестандартную попытку следовать обоим правилам, принимая код, написанный в любом стиле.)

Вернемся к циклу `for`

Попробуем сделать что-то более сложное с помощью цикла. Код в листинге 5.4 использует цикл для вычисления и сохранения первых 16 факториалов. Факториалы, которые являются удобным примером автоматизации обработки, вычисляются следующим образом. Ноль факториал, записываемый как 0!, определен как равный 1. Затем, 1! равен 1*0!, то есть 1. Далее, 2! равно 2*1!, или 2. Затем 3! равно 3*2!, или 6, и так далее. То есть факториал каждого целого числа равен произведению этого числа на факториал предыдущего числа. (Одним из общеизвестных средств фонетической пунктуации, предложенной пианистом Виктором Борге (Victor Borge), является выделение восклицательного знака звуком "ффт" или "пц" с небольшим ударением. Однако в данном случае "!" произносится как "факториал".) Программа использует один цикл для вычисления значений последовательных факториалов, помещая их в массив. Затем она использует второй цикл для отображения результатов. Также программа представляет применение внешнего объявления для значений.

Листинг 5.4. formore.cpp

```
// formore.cpp -- еще о циклах for
#include <iostream>
using namespace std;
const int ArSize = 16; // пример внешнего объявления
int main()
{
   double factorials[ArSize];
   factorials[1] = factorials[0] = 1.0;
   // int i;
   for (int i = 2; i < ArSize; i++)
     factorials[i] = i * factorials[i-1];
   for (i = 0; i < ArSize; i++)
     cout << i << "! = " << factorials[i] << endl;
   return 0;
}
```

Вывод программы из листинга 5.4 выглядит следующим образом:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3.6288e+006
11! = 3.99168e+007
12! = 4.79002e+008
13! = 6.22702e+009
14! = 8.71783e+010
15! = 1.30767e+012
```

Факториалы растут очень быстро!

Замечания по программе

Программа в листинге 5.4 создает массив для значений факториалов. Элемент 0 равен 0!, элемент 1 — 1! и так далее. Поскольку первые два факториала равны 1, программа присваивает первым двум элементам массива `factorials` значение `1.0`. (Напоминаем, что первый элемент массива имеет индекс 0.) После этого программа использует цикл для вычисления каждого факториала как произведения индекса на значение предыдущего факториала. Цикл иллюстрирует, что вы можете использовать счетчик цикла в его теле как переменную.

Программа из листинга 5.4 демонстрирует, как цикл `for` работает рука об руку с массивами, предоставляя удобное средство доступа к каждому члену массива по очереди. К тому же `formore.cpp` использует `const` для создания символического представления (`ArSize`) размера массива. Затем `ArSize` применяется везде, где вступает в игру размер массива — в определении массива, а также в выражении, ограничива-

ющем количество шагов циклов обрабатывающих массив. Теперь, если вы решите расширить программу для вычисления, скажем, 20 факториалов, то для этого вам понадобится только установить ArSize равным 20 и перекомпилировать программу. Благодаря использованию символической константы вы избегаете необходимости изменять индивидуально каждое появление 16 на 20.

На заметку!

Обычно это хорошая идея — определить const-значение для представления размера массива. Вы можете использовать значение const в объявлении массива и во всех случаях ссылок на его размер, как, например, в циклах for.

Ограничивающее выражение i < ArSize отражает тот факт, что индексы элементов массива лежат в пределах от 0 до ArSize - 1, то есть значение индекса должно останавливаться за один шаг до достижения ArSize. Вы могли бы использовать проверочное условие i <= ArSize - 1, но это излишне и не меняет сути проверочного условия.

Обратите внимание, что программа объявляет переменную ArSize типа const int вне тела функции main(). Как упоминалось в конце главы 4, это делает ArSize частью внешних данных. Два последствия объявления ArSize в такой манере заключаются в том, что ArSize существует для всей программы и любая функция программы может использовать ArSize. В данном конкретном случае в программе присутствует только одна функция, поэтому внешнее объявление ArSize не имеет особого практического смысла. Однако программы с множеством функций часто выигрывают от разделения доступа к внешним константам, поэтому дальше мы еще попрактикуемся в их применении.

Изменение шага цикла

До сих пор примеры циклов в этой главе увеличивали и уменьшали счетчик цикла на единицу за каждый шаг. Вы можете изменить это, изменив обновляющее выражение. Программа в листинге 5.5, например, увеличивает счетчик цикла на величину указанного пользователем шага. Вместо применения i++ в качестве обновляющего выражения она использует выражение i = i + by, где by — выбранный пользователем шаг цикла.

Листинг 5.5. bigstep.cpp

```
// bigstep.cpp -- цикл указанным пользователем шагом
#include <iostream>
int main()
{
   using namespace std;
   cout << "Введите целое число: ";
   int by;
   cin >> by;
   cout << "Шагаем по " << by << ":\n";
   for (int i = 0; i < 100; i = i + by)
     cout << i << endl;
   return 0;
}
```

Пример запуска программы из листинга 5.5:

```
Введите целое число: 17
Шагаем по 17:
0
17
34
51
68
85
```

Когда `i` достигает значения `102`, цикл завершается. Главное, на что здесь нужно обратить внимание — в качестве обновляющего можно использовать любое корректное выражение. Например, если вы захотите на каждом шаге цикла возводить `i` в квадрат и прибавлять `10`, то можете воспользоваться выражением `i = i * i + 10`.

Внутрь строки с помощью цикла for

Цикл `for` предлагает прямой способ доступа к каждому символу в строке. Например, листинг 5.6 позволяет ввести строку и отобразить ее символ за символом в обратном порядке. В этом примере вы можете использовать либо объект класса `string`, либо массив `char`, потому что оба позволяют применять нотацию массива для доступа к индивидуальным символам строки. В листинге 5.6 используется объект класса `string`. Метод `size()` класса `string` возвращает количество символов в строке; цикл использует это значение в выражении инициализации для присваивания значения индекса последнего символа строки переменной `i`, исключая нулевой символ. Чтобы выполнять обратный отсчет, программа применяет операцию декремента (`--`) для уменьшения текущего индекса массива на каждом шаге цикла. Также листинг 5.6 использует операцию сравнения "больше или равно" (`>=`), чтобы проверить, достиг ли цикл первого элемента. Чуть позже мы подведем итог всем операций сравнения.

Листинг 5.6. forstr1.cpp

```
// forstr1.cpp -- применение for к строке
#include <iostream>
#include <string>
int main()
{
    using namespace std;
    cout << "Введите слово: ";
    string word;
    cin >> word;
    // отобразить символы в обратном порядке
    for (int i = word.size() - 1; i >= 0; i--)
      cout << word[i];
    cout << "\nГотово.\n";
    return 0;
}
```

Вот пример запуска программы из листинга 5.6:

```
Введите слово: animal
lamina
Готово.
```

Как видите, программа действительно успешно напечатала слово `animal` наоборот; выбор этого слова в качестве теста яснее демонстрирует эффект от работы программы, нежели выбор, скажем, палиндрома вроде `redder` или `stats`.

Операции инкремента и декремента

Язык C++ снабжен несколькими операциями, которые часто используются в циклах; давайте потратим немного времени на их изучение. Вы уже видели две из них: операция инкремента (`++`), которая инспирировала название самого C++, а также операция декремента (`--`). Эти операции выполняют две чрезвычайно часто встречающиеся операции циклов: увеличивают и уменьшают на единицу значение счетчика цикла. Однако к тому, что вы уже знаете о них, есть что добавить. Каждая из них имеет два варианта. *Префиксная* версия задается перед операндом, как в `++x`. *Постфиксная* версия указывается после операнда, как в `x++`. Эти две версии имеют один и тот же эффект для операнда, но отличаются в контексте применения. Это как получение оплаты за стрижку газона авансом или после завершения работы: оба метода имеют один и тот же конечный результат для вашего бумажника, но отличаются тем, когда деньги в него добавляются. В листинге 5.7 демонстрируется разница на примере операции инкремента.

Листинг 5.7. `plus_one.cpp`

```
// plus_one.cpp -- операция инкремента
#include <iostream>
int main()
{
    using namespace std;
    int a = 20;
    int b = 20;
    cout << "a = " << a << ": b = " << b << "\n";
    cout << "a++ = " << a++ << ": ++b = " << ++b << "\n";
    cout << "a = " << a << ": b = " << b << "\n";
    return 0;
}
```

Результат выполнения этой программы показан ниже:

```
a   = 20:   b = 20
a++ = 20: ++b = 21
a   = 21:   b = 21
```

Грубо говоря, нотация `a++` означает "использовать текущее значение `a` при вычислении выражения, затем увеличить `a` на единицу". Аналогично, нотация `++a` означает "сначала увеличить значение `a` на единицу, а затем использовать новое значение при вычислении выражения". Например, мы имеем следующие отношения:

```
int x = 5;
int y = ++x;    // изменить x, затем присвоить его y
                // y равно 6, x равно 6
int z = 5;
int y = z++;    // присвоить y, затем изменить z
                // y равно 5, z равно 6
```

Операции инкремента и декремента представляют собой простой удобный способ выполнения часто возникающей задачи увеличения или уменьшения значений на единицу.

Операции инкремента и декремента — симпатичные и компактные, но не стоит поддаваться соблазну и применять их к одному и тому же значению более одного раза в одном и том же операторе. Проблема в том, что при этом правила "использовать и изменить" и "изменить и использовать" становятся неоднозначными. То есть, оператор вроде следующего:

```
x = 2 * x++ * (3 - ++x); // не делайте так
```

может дать разные результаты в разных системах. C++ не определяет корректного поведения операторов подобного рода.

Побочные эффекты и последовательные точки

Давайте посмотрим внимательнее на то, что C++ может, и чего не может сказать о том, какой эффект имеют операции инкремента. Во-первых, вспомним, что *побочный эффект* — это эффект, который проявляется, когда вычисление выражения приводит к модификации чего-либо, например, значения переменной. *Последовательная точка* — точка при выполнении программы, где все побочные эффекты гарантированно будут завершены, прежде чем программа перейдет к следующему шагу. В C++ точка с запятой в операторе отмечает последовательную точку. Это значит, что все изменения, выполненные операциями присваивания, инкремента и декремента в операторе должны произойти, прежде чем программа перейдет к следующему оператору. Некоторые операторы, о которых мы поговорим в последующих разделах, включают последовательные точки. К тому же конец любого полного выражения представляет последовательную точку.

Что такое полное выражение? Это выражение, которое не является частью более крупного выражения. Примеры полных выражений включают часть операторы-выражения (без точки с запятой), а также выражения, служащие проверочными условиями в цикле `while`.

Последовательные точки помогают прояснить, когда выполняется постфиксный инкремент. Рассмотрим, например, следующий код:

```
while (guests++ < 10)
   printf("%d \n", guests);
```

Иногда новички в C++ предполагают, что "использовать значение, затем увеличить его" означает в данном контексте увеличение `guests` после того, как эта переменная использована в операторе `printf()`. Однако `guests++ < 10` является полным выражением, поскольку это проверочное условие цикла `while`, поэтому конец этого выражения — суть последовательная точка. Таким образом, C++ гарантирует, что побочный эффект (увеличение `guests`) произойдет перед тем, как программа перейдет к `printf()`. Применение постфиксной формы, однако, гарантирует, что `guests` увеличится после сравнения с `10`.

Теперь рассмотрим следующий оператор:

```
y = (4 + x++) + (6 + x++);
```

Выражение `4 + x++` не является полным выражением, поэтому C++ не гарантирует, что x будет увеличено немедленно после вычисления вложенного выражения `4 + x++`. Здесь полным выражением является весь оператор присваивания, и точка с запятой отмечает последовательную точку, поэтому все, что гарантирует C++ — это то, что x будет увеличено дважды перед тем, как программа перейдет к следующему оператору. C++ не специфицирует то, будет ли x инкрементировано после вычисления каждого подвыражения, либо после вычисления всего выражения в целом. Поэтому вы должны избегать операторов такого рода.

Сравнение префиксной и постфиксной форм

Понятно, что разница между префиксной и постфиксной формами операций проявляется, если значение их операнда используется для каких-то целей — в качестве аргумента функции или для присваивания переменной. Но что если инкрементируемое или декрементируемое значение не используется? Например, отличается ли

```
x++;
```

от

```
++x;
```

или нет? Или же отличаются ли друг от друга

```
for (n = lim; n > 0; --n)
    ...;
```

и

```
for (n = lim; n > 0; n--)
    ...;
```

Рассуждая логически, можно предположить, что префиксная форма не отличается от постфиксной в этих двух ситуациях. Значения выражений не используются, поэтому единственный эффект от них — побочный, то есть увеличение или уменьшение операнда. Здесь выражения, использующие эти операции, являются полными выражениями, поэтому побочный эффект от инкремента x и декремента n гарантированно будет получен на момент, когда программа переходит к следующему шагу; префиксная и постфиксная формы дают один и тот же результат.

Однако, несмотря на то, что выбор между двумя формами никак не отражается на поведении программы, все же он может несколько повлиять на скорость выполнения. Для встроенных типов и современных компиляторов это может показаться неважным. Но C++ разрешает вам определять самостоятельно эти операции для классов. В этом случае пользователь определяет префиксную функцию, которая работает, увеличивая значение и затем возвращая его. Но постфиксная версия работает, сначала запоминая копию значения, увеличивает его и возвращает сохраненную копию. Таким образом, для классов префиксная версия немного более эффективна, чем постфиксная.

Короче говоря, для встроенных типов, скорее всего, разницы нет между двумя формами этих операций. Но для типов, определенных пользователем, оснащенных операциями инкремента и декремента, префиксная форма более эффективна.

Операции инкремента и декремента и указатели

Вы можете использовать операции инкремента с указателями так же, как и с базовыми переменными. Вспомним, что прибавление единицы к указателю увеличивает его на число байт, представляющих размер указываемого типа. То же правило остается в силе при инкременте и декременте указателей:

```
double arr[5] = {21.1, 32.8, 23.4, 45.2, 37.4};
double *pt = arr;   // pt указывает на arr[0], то есть на 21.1
++pt;               // pt указывает на arr[1], то есть на 32.8
```

Вы также можете использовать эти операции для изменения значений, на которые указывают указатели, используя их в сочетании с операцией *. Применение * и ++ к указателю вызывает вопросы о том, что собственно должно разыменовываться, а что — инкрементироваться. Префиксный инкремент, префиксный декремент и операция разыменования имеют одинаковый приоритет и ассоциируются слева направо. Постфиксный инкремент и декремент имеют одинаковый приоритет, более высокий, чем приоритет префиксных форм. Эти две операции также ассоциируются слева направо.

Правило ассоциации слева направо для префиксных операций подразумевает, что в `*++pt` сначала применяется ++ к pt (потому что ++ находится справа от *), а затем применяется * к новому значению pt:

```
*++pt; // увеличить указатель, получить значение; то есть arr[2], или 23.4
```

С другой стороны, `++*pt` означает — получить значение, на которое указывает pt, а затем увеличить указатель:

```
++*pt; // увеличить указываемое значение; то есть изменить 23.4 на 24.4
```

Здесь pt по-прежнему будет указывать на `arr[2]`.

Теперь рассмотрим следующую комбинацию:

```
(*pt)++; // увеличить указатель на значение
```

Скобки указывают, что сначала выполняется разыменование указателя, порождая значение `24.4`. Затем операция ++ увеличивает значение до `25.4`; pt по-прежнему указывает на `arr[2]`.

И, наконец, рассмотрим такую комбинацию:

```
*pt++; // разыменовать исходный указатель, затем увеличить его
```

Более высокий приоритет постфиксной операции ++ означает, что ++ увеличит pt, а не `*pt`, поэтому инкремент касается указателя. Но тот факт, что использована постфиксная операция, означает, что разыменованный адрес является адресом `&arr[2]`, а не новым адресом. То есть значение `*pt++` равно `arr[2]`, или `25.4`, но после завершения оператора pt будет указывать на `arr[3]`.

На память!

Инкремент и декремент указателей следуют правилам арифметики указателей. То есть, если pt указывает на первый элемент массива, `++pt` изменяет его так, что он после этого указывает на второй его элемент.

Комбинация операций присваивания

В листинге 5.5 используется следующее выражение для обновления счетчика цикла:

```
i = i + by
```

В C++ предусмотрена комбинированная операция сложения с присваиванием, которая позволяет получить тот же результат более кратко:

```
i += by
```

Операция += складывает значение своих двух операндов и присваивает результат левому операнду. Это предполагает, что левый операнд должен быть чем-то таким, чему можно присваивать значения, вроде переменной, элемента массива, члена структуры либо элемента данных, полученного через разыменование указателя:

```
int k = 5;
k += 3;                // нормально, k присвоено 8
int *pa = new int[10]; // pa указывает на pa[0]
pa[4] = 12;
pa[4] += 6;            // нормально, pa[4] присвоено 18
*(pa + 4) += 7;        // нормально, pa[4] присвоено 25
pa += 2;               // нормально, pa указывает на бывший pa[2]
34 += 10;              // неверно!
```

Каждая арифметическая операция имеет соответствующую операцию присваивания, как показано в табл. 5.1. Каждая такая операция работает аналогично +=. То есть, например, оператор

```
k *= 10;
```

заменяет текущее значение k в 10 раз большим значением.

Таблица 5.1. Комбинированные операции присваивания

Операция	Эффект (L=левый операнд, R=правый операнд)
+=	Присваивает L + R операнду L
-=	Присваивает L - R операнду L
*=	Присваивает L * R операнду L
/=	Присваивает L / R операнду L
%=	Присваивает L % R операнду L

Составные операторы, или блоки

Формат, или синтаксис, написания оператора for может показаться чересчур ограниченным, поскольку тело цикла должно состоять всего лишь из одного оператора. Это весьма неудобно, если вы хотите, чтобы тело цикла состояло из нескольких операторов. К счастью, C++ предлагает синтаксис, позволяющий включить в тело цикла любое количество операторов программы. Трюк заключается в использовании пары фигурных скобок, с помощью которых конструируется *составной оператор*, или *блок*. Блок состоит из пары фигурных скобок с заключенными между ними операторами и синтаксически воспринимается как один оператор. Например, программа в

листинге 5.8 использует скобки для того, чтобы скомбинировать три отдельных оператора в единый блок. Это позволяет в теле цикла организовать вывод приглашения пользователю, прочитать его ввод и выполнить вычисления. Программа вычисляет накопительную сумму вводимых значений, что представляет удобный случай для применения операции +=.

Листинг 5.8. block.cpp

```
// block.cpp -- использование блока операторов
#include <iostream>
int main()
{
   using namespace std;
   cout << "Изумительный Счетчик сложит и вычислит для вас среднее ";
   cout << "из пяти значений. \n";
   cout << "Пожалуйста, введите пять значений: \n";
   double number;
   double sum = 0.0;
   for (int i = 1; i <= 5; i++)
   { // здесь — начало блока
     cout << "Значение " << i << ": ";
     cin >> number;
     sum += number;
   } // конец блока здесь
   cout << "В самом деле, пять замечательных значений! ";
   cout << "Их сумма равна " << sum << endl;
   cout << "а среднее - " << sum / 5 << ".\n";
   cout << "Изумительный Счетчик прощается с вами! \n";
   return 0;
}
```

Ниже показан пример запуска программы из листинга 5.8:

```
Изумительный Счетчик сложит и вычислит для вас среднее из пяти значений.
Пожалуйста, введите пять значений:
Значение 1: 1942
Значение 2: 1948
Значение 3: 1957
Значение 4: 1974
Значение 5: 1980
В самом деле, пять замечательных значений! Их сумма равна 9801
а среднее - 1960.2.
Изумительный Счетчик прощается с вами!
```

Предположим, вы сохраните отступ, но удалите скобки:

```
for (int i = 1; i <= 5; i++)
   cout << "Значение " << i << ": ";
   cin >> number;
   sum += number;
cout << "В самом деле, пять замечательных значений! ";
```

Компилятор проигнорирует отступы, поэтому в тело цикла попадет только первый оператор. То есть цикл напечатает пять приглашений и ничего более. После

завершения цикла программа перейдет к выполнению последующих строк, читая и складывая только одно значение.

Составные операторы обладают еще одним интересным свойством. Если вы определяете новую переменную внутри блока, она будет существовать только до тех пор, пока выполняются операторы внутри этого блока. Когда поток выполнения покидает блок, такая переменная уничтожается. То есть это значит, что переменная определена только внутри блока.

```
#include <iostream>
using namespace std;
int main()
{
   int x = 20;
   { // начало блока
      int y = 100;
      cout << x << endl;  // ok
      cout << y << endl;  // ok
   } // конец блока
   cout << x << endl;     // ok
   cout << y << endl;     // неверно, не скомпилируется!
   return 0;
}
```

Обратите внимание, что переменная, объявленная вне блока, определена и внутри него.

А что случится, если вы объявите внутри блока переменную с тем же именем, что и у одной из объявленных вне блока? Новая переменная скроет старую, начиная с точки ее появления и до конца блока. Потом старая переменная опять станет видимой, как в следующем примере:

```
int main()
{
   int x = 20;            // исходная переменная x
   { // начало блока
      cout << x << endl;  // используется исходная переменная x
      int x = 100;        // новая переменная x
      cout << x << endl;  // используется новая переменная x
   } // конец блока
   cout << x << endl;     // используется исходная переменная x
   return 0;
}
```

Операция запятой (или еще о синтаксических трюках)

Как вы уже видели, блок позволяет поместить два и более операторов туда, где синтаксис C++ разрешает лишь один. Операция запятой (`,`) делает то же самое с выражениями, позволяя вставлять два выражения туда, где синтаксис C++ допускает только одно. Например, предположим, что у вас есть цикл, в котором одна переменная увеличивается на единицу при каждом шаге, а вторая — на единицу уменьшается. Было бы удобно сделать и то, и другое в обновляющей части цикла `for`, но синтаксис

цикла разрешает там только одно выражение. Решение состоит в применении операции запятой для комбинации двух выражений в одно:

```
++j, --i    // два выражения воспринимаются как одно для удовлетворения
            // требований синтаксиса
```

Запятая — не всегда операция. Например, запятая в следующем объявлении служит для разделения имен в списке объявляемых переменных:

```
int i, j;   // здесь запятая — разделитель, а не операция
```

В листинге 5.9 операция запятой используется дважды в программе, которая переставляет содержимое объекта класса `string`. (Вы можете также написать программу, используя массив `char`, но длина слова должна ограничиваться определенным размером массива.) Обратите внимание, что листинг 5.6 отображает содержимое массива в обратном порядке, а листинг 5.9 на самом деле перемещает символы по кругу в пределах массива. Программа из листинга 5.9 также использует блок для объединения нескольких операторов в один.

Листинг 5.9. `forstr2.cpp`

```
// forstr2.cpp -- обращение порядка элементов массива
#include <iostream>
#include <string>
int main()
{
   using namespace std;
   cout << "Введите слово: ";
   string word;
   cin >> word;
   // физическая модификация объекта string
   char temp;
   int i, j;
   for (j = 0, i = word.size() - 1; j < i; --i, ++j)
   { // начало блока
       temp = word[i];
       word[i] = word[j];
       word[j] = temp;
   } // конец блока
   cout << word << "\nГотово.\n";
   return 0;
}
```

Вот пример выполнения программы из листинга 5.9:

```
Введите слово: parts
strap
Готово.
```

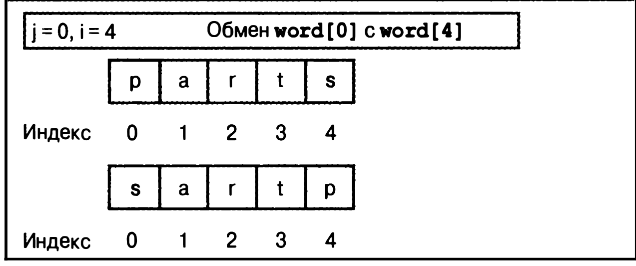
Кстати, класс `string` предлагает более удобный способ обращения строки, но мы отложим его описание до главы 16.

Замечания по программе

Взгляните еще раз на управляющий раздел цикла `for` в программе 5.9.

Во-первых, он использует операцию запятой для указания последовательно двух инициализаций в одном выражении — первой части управляющего раздела.

Далее, посмотрите на тело цикла. Программа использует фигурные скобки для комбинации нескольких операторов в одно целое. В этом теле программа обращает порядок символов в слове, меняя местами первый элемент массива с его последним элементом. Затем она увеличивает значение `j` и уменьшает значение `i`, так что теперь они ссылаются на следующий за первым элемент и предшествующий последнему. После того, как это сделано, программа меняет местами эти элементы. Обратите внимание, что проверочное условие `j<i` останавливает цикл, когда он достигает центра массива. Если бы он продолжался дальше, то начался бы обратный обмен символов на их старые места. (См. рис. 5.2.)



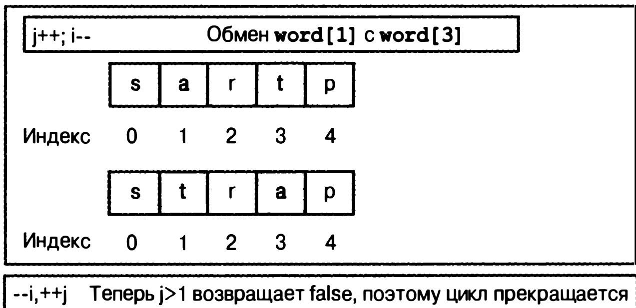


Рис. 5.2. Обращение строки

Здесь следует отметить еще одну вещь — местоположения объявлений переменных `temp`, `i` и `j`. Код объявляет `i` и `j` перед началом цикла, поскольку вы не можете комбинировать два объявления подряд, разделяя их операцией запятой. Это потому, что объявления уже используют запятую для другой цели — в качестве разделителя элементов списка. Вы можете использовать один оператор-объявление для создания и инициализации двух переменных, но выглядит это несколько запутано:

```
int j = 0, i = word.size() - 1;
```

В этом случае запятая служит просто разделителем, а не операцией, поэтому оператор объявляет и инициализирует обе переменных. Но смотрится это так, будто объявляется только `j`.

Между прочим, вы можете объявить temp внутри цикла for:

```
int temp = word[i];
```

Это может привести к тому, что temp будет размещаться и освобождаться на каждом шаге цикла. Это может замедлить работу программы по сравнению с вариантом, когда переменная temp объявлена один раз, перед началом цикла. С другой стороны, при объявлении внутри цикла после его завершения temp уничтожается.

Особенности операции запятой

До сих пор наиболее часто мы использовали операцию запятой для размещения двух или более выражений в одном выражении цикла for. Но C++ снабжает эту операцию двумя дополнительными свойствами. Во-первых, она гарантирует, что первое выражение вычисляется перед вторым. (Другими словами, операция запятой — это последовательная точка.) Безопасны выражения вроде следующего:

```
i = 20, j = 2 * i // i присваивается 20, j присваивается 40
```

Во-вторых, C++ устанавливает, что значением выражения с запятой является значение его второй части. Значение предыдущего выражения, например, равно 40, потому что такого значение выражения j = 2 * i.

Операция запятой обладает наименьшим приоритетом среди всех операций. Например, следующий оператор:

```
cats = 17,240;
```

читается как

```
(cats = 17), 240;
```

То есть, cats присваивается 17, а 240 не делает ничего. Но поскольку скобки имеют более высокий приоритет, то следующий оператор:

```
cats = (17,240);
```

дает результат 240 — выражение, находящееся справа от запятой.

Выражения отношений

Компьютеры — это нечто большее, чем неустанные обработчики чисел. Они также умеют сравнивать значения, и это их свойство лежит в основе автоматического принятия решений. В C++ эту возможность реализуют операции отношений. В C++ имеются шесть операций отношений для сравнения чисел. Поскольку символы представлены их ASCII-кодами, вы можете применять эти операции также и для сравнения символов. Они не работают со строками в стиле C, но работают с объектами класса string. Каждое сравнивающее выражение возвращает булевское (типа bool) значение true, если сравнение истинно, и false — в противном случае, поэтому эти операции хорошо подходят для применения в проверочных условиях циклов. (Старые реализации оценивали истинные выражения как 1 и ложные — как 0.) В табл. 5.2 предлагается список операций отношений.

Таблица 5.2. Операции отношений

Операция	Значение
<	Меньше чем
<=	Меньше или равно
==	Равно
>	Больше чем
>=	Больше или равно
!=	Не равно

Эти шесть операций отношений исчерпывают возможности, предусмотренные в C++ для сравнения чисел. Если вы хотите сравнить два значения на предмет того, какое из них более красиво или более удачно, вам придется поискать в другом месте.

Вот некоторые примеры сравнений:

```
for (x = 20; x > 5; x--)           // продолжать, пока x больше чем 5
for (x = 1; y != x; ++x)           // продолжать, пока y не равно x
for (cin >> x; x == 0; cin >> x))  // продолжать, пока x равно 0
```

Операции отношений обладают более низким приоритетом, нежели арифметические операции. Это значит, что следующее выражение:

```
x + 3 > y - 2     // Выражение 1
```

соответствует такому:

```
(x + 3) > (y - 2) // Выражение 2
```

и не соответствует этому:

```
x + (3 > y) - 2   // Выражение 3
```

Поскольку выражение `(3 > y)` после приведения значения `bool` к типу `int` даст либо `0`, либо `1`, оба выражения 2 и 3 корректны. Но большинство из нас подразумевают под выражением 1 то, что записано в выражении 2, так поступает и C++.

Вероятные ошибки

Не следует путать операцию проверки равенства (`==`) с операцией присваивания (`=`). Следующее выражение:

```
musicians == 4    // сравнение
```

задает вопрос: равно ли значение `musicians` четырем? Выражение может принимать значение `true` или `false`. А следующее выражение:

```
musicians = 4     // присваивание
```

присваивает `musicians` значение `4`. Полное выражение в данном случае имеет значение `4`, потому что таково значение левой части.

Гибкий дизайн цикла `for` создает любопытную возможность ошибки. Если вы нечаянно удалите один символ `=` из операции сравнения `==` и примените операцию присваивания вместо операции сравнения в проверочной части цикла `for`, это будет расценено компилятором как вполне корректный код. Это потому, что вы можете ис-

пользовать любое корректное выражение C++ в качестве проверочного условия цикла for. Вспомните, что ненулевые значения оцениваются как true, а нулевые — как false. Выражение, которое присваивает 4 переменной musicians, имеет значение 4 и трактуется как true. Если вы перешли в C++ от таких языков, как Pascal или BASIC, которые используют = для проверки равенства, то вы будете склонны к подобного рода ошибкам.

В листинге 5.10 показана ситуация, когда есть риск допустить такую ошибку. Программа пытается проверить массив quizscores и останавливается, когда достигает первого значения, которое не равно 20. Она сначала демонстрирует цикл, который корректно использует сравнение, а затем — цикл, в котором в проверочном условии ошибочно вместо операции равенства использована операция присваивания. Кроме этой, программа также содержит в себе еще одну вопиющую ошибку, исправление которой вы увидите позднее. (На ошибках учатся, и листинг 5.10 помогает в этом.)

Листинг 5.10. equal.cpp

```
// equal.cpp -- равенство или присваивание
#include <iostream>
int main()
{
   using namespace std;
   int quizscores[10] =
     { 20, 20, 20, 20, 20, 19, 20, 18, 20, 20};
   cout << "Так правильно: \n";
   int i;
   for (i = 0; quizscores[i] == 20; i++)
       cout << "quiz " << i << " равно 20\n";
   cout << "Так — неправильно и опасно: \n";
   for (i = 0; quizscores[i] = 20; i++)
       cout << "quiz " << i << " равно 20\n";
   return 0;
}
```

Поскольку программа в листинге 5.10 имеет серьезную проблему, вы можете предпочесть почитать о ней, а не запускать. Вот так будет выглядеть вывод:

```
Так правильно:
quiz 0 равно 20
quiz 1 равно 20
quiz 2 равно 20
quiz 3 равно 20
quiz 4 равно 20
Так — неправильно и опасно:
quiz 0 равно 20
quiz 1 равно 20
quiz 2 равно 20
quiz 3 равно 20
quiz 4 равно 20
quiz 5 равно 20
quiz 6 равно 20
quiz 7 равно 20
```

```
quiz 8 равно 20
quiz 9 равно 20
quiz 10 равно 20
quiz 11 равно 20
quiz 12 равно 20
quiz 13 равно 20
...
```

Первый цикл корректно завершается после отображения первых пяти значений из массива. Но второй цикл начинает отображать весь массив. Хуже того, он сообщает, что все значения в нем равны 20. И еще хуже того: он не останавливается по достижении конца массива!

Что здесь неправильно — это, конечно, то, что содержится в проверочном условии:

```
quizscores[i] = 20
```

Во-первых, поскольку здесь присваивается элементу массива ненулевое значение, выражение всегда вычисляется как истинное. Во-вторых, поскольку это выражение присваивает значения элементам массива, оно на самом деле изменяет данные. В-третьих, поскольку проверочное условие остается истинным, программа продолжает изменять данные и за концом массива. Оно просто продолжает и продолжает вставлять в память значения 20! Это нехорошо.

Проблема с ошибками подобного рода состоит в том, что код синтаксически корректен, поэтому компилятор не может распознать ошибку. (Однако годы и годы программирования на C и C++ заставили многих поставщиков компиляторов, по крайней мере, выдавать предупреждения, которые запрашивают — действительно ли вы имеете в виду то, что написали.)

Внимание!

Не используйте = для проверки равенства, используйте ==.

Подобно языку C, C++ предлагает вам большую свободу, чем большинство других языков программирования. Это даром не обходится — на разработчика возлагается большая ответственность. Ничто другое, кроме тщательного планирования, не предохранит вашу программу от выхода за пределы стандартного массива C++. Однако, используя классы C++, вы можете проектировать безопасные типы массивов, которые предохранят вас от подобной бессмыслицы. Пример этого приведен в главе 13. А пока вы должны встраивать защиту в свои программы, когда нуждаетесь в ней. Например, цикл из листинга 5.10 должен включать проверку, которая не позволит ему выйти за пределы массива. Это верно даже для "хороших" циклов. Если все элементы массива из этого примера содержали бы значение 20, то этот самый "хороший" цикл также вышел бы за пределы массива. Короче говоря, цикл должен проверять как значения массива, так и индекс. В главе 6 мы покажем, как использовать логические операции для комбинирования таких проверок в единое условие.

Сравнение строк в стиле C

Предположим, что вы хотите узнать, хранится ли в символьном массиве слово `mate`. Если `word` — имя массива, то следующая проверка не сделает того, что вы ожидаете:

```
word == "mate"
```

Напомним, что имя массива — синоним его адреса. Аналогично, константная строка в кавычках — синоним ее адреса. Таким образом, приведенное выражение сравнения не проверяет идентичности строк; оно проверяет, находятся ли они по одному и тому же адресу. Ответом будет — нет, даже если эти две строки состоят из одинаковых символов.

Поскольку C++ обрабатывает строки в стиле C как адреса, вы получите мало удовольствия, если попытаетесь воспользоваться операциями отношений для сравнения строк. Вместо этого вы можете обратиться к библиотеке строк в стиле C и применить для их сравнения функцию `strcmp()`. Эта функция принимает в виде аргументов два адреса строк. Это значит, что аргументы могут быть указателями, строковыми константами либо именами символьных массивов. Если две строки идентичны, функция возвращает значение 0. Если первая строка предшествует второй в алфавитном порядке, `strcmp()` возвращает отрицательное значение, если же первая строка следует за второй в алфавитном порядке, то `strcmp()` возвращает положительное значение. Говорить "в порядке системной схемы сортировки" точнее, чем "в алфавитном порядке". Это значит, что символы сравниваются в соответствии с их системными кодами. Например, в коде ASCII заглавные буквы имеют меньшие коды, чем строчные, поэтому заглавные предшествуют строчным в порядке сортировки. То есть, строка `"Zoo"` предшествует строке `"aviary"`. Тот факт, что сравнение основано на значениях кодов, также означает, что заглавные и строчные буквы отличаются, поэтому строка `"FOO"` отличается от `"foo"`.

В некоторых языках, таких как BASIC и стандартный Pascal, строки, сохраненные в массивах разных размеров, по определению не равны друг другу. Но строки в стиле C ограничиваются нулевым символом, а не размером содержащего их массива. Это значит, что две строки могут быть идентичными, даже если содержатся в массивах разного размера:

```
char big[80] = "Daffy";      // 5 букв плюс \0
char little[6] = "Daffy";    // 5 букв плюс \0
```

Кстати, хотя вы и не можете применять операции отношений для сравнения строк, вы можете использовать их для сравнения символов, потому что символы относятся к целочисленным типам. Поэтому

```
for (ch = 'a'; ch <= 'z'; ch++)
   cout << ch;
```

является корректным кодом, по крайней мере, в наборе символов ASCII, для отображения символов по алфавиту.

Программа в листинге 5.11 использует `strcmp()` в проверочном условии цикла `for`. Программа отображает слово, изменяет его первую букву, отображает его снова и продолжает это делать до тех пор, пока `strcmp()` не определит, что `word` содержит строку `"mate"`. Обратите внимание, что листинг включает файл `cstring`, поскольку в нем содержится прототип `strcmp()`.

Листинг 5.11. `compstr1.cpp`

```
// compstr1.cpp -- сравнение строк с использованием массивов
#include <iostream>
#include <cstring> // прототип для strcmp()
```

```
int main()
{
   using namespace std;
   char word[5] = "?ate";
   for (char ch = 'a'; strcmp(word, "mate"); ch++)
   {
     cout << word << endl;
     word[0] = ch;
   }
   cout << "По окончании цикла word содержит " << word << endl;
   return 0;
}
```

Замечание по совместимости

Вы можете использовать `string.h` вместо `cstring`. К тому же, код в листинге 5.11 предполагает, что в системе применяется набор символов ASCII. В этом наборе коды символов от `a` до `z` идут последовательно, а код символа `?` непосредственно предшествует коду символа `a`.

Ниже показан результат выполнения программы из листинга 5.11:

```
?ate
aate
bate
cate
date
eate
fate
gate
hate
iate
jate
kate
late
По окончании цикла word содержит mate
```

Замечания по программе

Программа в листинге 5.11 содержит несколько интересных моментов. Один из них, конечно же — проверка цикла. Вы хотите, чтобы цикл продолжался до тех пор, пока `word` не совпадает с `"mate"`. То есть, до тех пор, пока `strcmp()` сообщает, что строки не одинаковы. Наиболее очевидный способ сделать это выглядит следующим образом:

```
strcmp(word, "mate") != 0 // строки не одинаковы
```

Этот оператор имеет значение `1` (`true`), если строки не одинаковы, и значение `0` (`false`), если они совпадают. Но как насчет самого вызова `strcmp(word, "mate")`? Он возвращает ненулевое значение (`true`), если строки отличаются, и `0` (`false`) — если строки эквивалентны. По сути, функция возвращает `true`, если строки разные, и `false`, если одинаковые. Вы можете использовать только саму функцию вместо всего сравнивающего выражения. Это даст тот же результат при меньшем объеме кода. К тому же, это — традиционный способ применения `strcmp()` в языках C и C++.

На память!

Вы можете применять `strcmp()` для проверки строк в стиле С на эквивалентность или порядок. Выражение

```
strcmp(str1,str2) == 0
```

истинно, если `str1` и `str2` идентичны; выражения

```
strcmp(str1, str2) != 0
```

и

```
strcmp(str1, str2)
```

истинны, когда `str1` и `str2` не идентичны; выражение

```
strcmp(str1,str2) < 0
```

истинно, если `str1` по порядку предшествует `str2`; выражение

```
strcmp(str1, str2) > 0
```

истинно, когда `str1` следует за `str2`. Таким образом, функция `strcmp()` может играть роль операций ==, !=, < и >, в зависимости от того, как вы составите проверочное условие.

Далее `compstr1.cpp` применяет операцию инкремента для прохода переменной `ch` по алфавиту:

```
ch++
```

Вы можете применять операции инкремента и декремента в отношении символьных переменных, потому что тип `char` — на самом деле целочисленный, поэтому эта операция в действительности изменяет целый код, сохраненный в переменной. К тому же заметьте, что применение индекса массива упрощает изменение отдельного символа в строке:

```
word[0] = ch;
```

Сравнение строк — объектов класса string

Жизнь станет немного легче, если вы используете строки класса `string` вместо строк в стиле С, потому что дизайн класса позволяет применять операции отношений для выполнения сравнений. Это возможно, поскольку определены функции класса, которые "перегружают", или переопределяют, операции. В главе 12 будет описано, как включить это средство в дизайн класса, но с практической точки зрения все, что вам нужно знать сейчас — это то, что с объектами класса `string` вы можете использовать операции сравнения. Листинг 5.12 представляет собой преобразованный код из листинга 5.11, в котором используется объект `string` вместо массива `char`.

Листинг 5.12. compstr2.cpp

```
// compstr2.cpp -- сравнение строк с использованием класса string
#include <iostream>
#include <string> // класс string
int main()
{
   using namespace std;
   string word = "?ate";
   for (char ch = 'a'; word != "mate"; ch++)
   {
```

```
    cout << word << endl;
    word[0] = ch;
  }
  cout << "По окончании цикла word содержит " << word << endl;
  return 0;
}
```

Вывод этой программы в точности такой же, как у программы из листинга 5.11.

Замечания по программе

В листинге 5.12 проверочное условие

```
word != "mate"
```

использует операцию сравнения с объектов `string` в левой части и строкой в стиле С — в правой части выражения. Способ перегрузки классом `string` операции `!=` позволяет использовать ее, если хотя бы один из операндов является объектом `string`; второй операнд при этом может быть либо объектом `string`, либо строкой в стиле С.

Дизайн класса `string` позволяет использовать объект `string` либо как единую сущность, как в сравнительных выражениях, либо как агрегатный объект, допускающий нотацию массива для извлечения индивидуальных символов.

И, наконец, в отличие от большинства циклов `for`, которые вы видели до сих пор, два последних цикла не имеют счетчиков. То есть они не выполняют блок операторов определенное количество раз. Вместо этого каждый из этих циклов проверяет определенное условие (равенство слову `"mate"`), которое сигнализирует о необходимости завершения. Более типично для программ С++ применять в таких случаях цикл `while`, поэтому давайте рассмотрим его в следующем разделе.

Цикл `while`

Цикл `while` — это цикл `for`, у которого удалены инициализирующая и обновляющая части; в нем имеется только проверочное условие и тело:

```
while (проверочное условие)
   тело
```

В начале программа выполняет выражение *проверочное условие* в скобках. Если выражение оценивается как истинное, программа выполняет оператор (или операторы), содержащийся в теле цикла. Как и в случае с циклом `for`, тело состоит из единственного оператора либо блока, ограниченного фигурными скобками. После того, как завершено выполнение тела, программа возвращается к проверочному условию и заново оценивает его. Если условие возвращает `true` или не ноль, программа снова выполняет тело. Этот цикл проверки и выполнения продолжается до тех пор, пока проверочное условие не вернет `false`. (См. рис. 5.3.) Понятно, что если вы хотите в конечном итоге прервать цикл, то в теле цикла должно происходить нечто такое, что повлияет на выражение проверочного условия. Например, цикл может увеличивать значение переменной, используемой в проверочном условии, либо читать новое значение, вводимое с клавиатуры. Подобно циклу `for`, цикл `while` является циклом

с входным условием. То есть, если проверочное условие оценивается как ложное в самом начале, то программа ни разу не исполнит тело цикла.

В листинге 5.13 представлен пример работы цикла while. Цикл проходит по каждому из символов строки и отображает его ASCII-код. По достижении нулевого символа цикл завершается. Эта техника прохода по символам строки до нулевого ограничителя является стандартным методом обработки строк в C++. Поскольку строка содержит маркер конца, программа часто не нуждается в явной информации о длине строки.

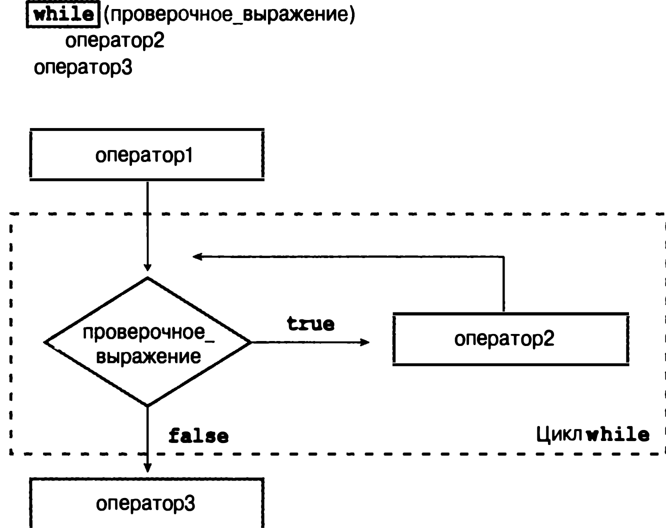


Рис. 5.3. Структура циклов **while**

Листинг 5.13. while.cpp

```
// while.cpp -- представление цикла while
#include <iostream>
const int ArSize = 20;
int main()
{
   using namespace std;
   char name[ArSize];
   cout << "Ваше имя, пожалуйста: ";
   cin >> name;
   cout << "Вот ваше имя, посимвольно и в кодах ASCII:\n";
   int i = 0;                 // начать с начала строки
   while (name[i] != '\0')    // обрабатывать до конца строки
   {
     cout << name[i] << ": " << int(name[i]) << endl;
     i++;                     // не забудьте этот шаг
   }
   return 0;
}
```

Ниже показан пример выполнения этой программы:

```
Ваше имя, пожалуйста: Muffy
Вот ваше имя, посимвольно и в кодах ASCII:
M: 77
u: 117
f: 102
f: 102
y: 121
```

Замечания по программе

Условие `while` в листинге 5.13 выглядит следующим образом:

```
while (name[i] != '\0')
```

Оно проверяет, является ли определенный символ массива нулевым. Чтобы эта проверка в конечном итоге была успешной, тело цикла должно изменять значение индекса `i`. Это достигается инкрементом `i` в конце тела цикла. Если пропустить этот шаг, то цикл застрянет на одном элементе массива, печатая один и тот же символ и его код до тех пор, пока вы принудительно не завершите программу. Возможность создания бесконечной последовательности — одна из наиболее часто возникающих проблем при работе с циклами. Часто это получается именно из-за того, что вы забываете изменить что-то в теле цикла, что связано с проверочным условием.

Строку с `while` можно переписать так:

```
while (name[i])
```

С этим изменением программа будет работать точно так же, как и раньше. Это объясняется тем, что `name[i]` — обычный символ, а его значение является кодом символа, который отличен от нуля, что соответствует `true`. Но когда в `name[i]` содержится нулевой символ, его код равен `0`, то есть `false`. Такая нотация более кратка и чаще используется, но не так ясна, как та, что приведена в листинге 5.13. Некоторые компиляторы попроще могут порождать более эффективный код во втором случае, но более интеллектуальные компиляторы генерируют в обоих случаях одинаковый код.

Чтобы напечатать ASCII-код символа, программа использует приведение типа для преобразования `name[i]` в целочисленный тип. После этого `cout` печатает значение символа как целое число.

В отличие от строк в стиле C, объекты класса `string` не используют нулевые символы для идентификации конца строки, поэтому вы не можете изменить листинг 5.13 для использования `string` простой заменой массива `char` на объект `string`. В главе 16 обсуждается техника, которую можно применить с объектами `string` для идентификации последнего символа.

Сравнение циклов `for` и `while`

В C++ циклы `for` и `while`, по сути, эквивалентны. Например, следующий цикл `for`:

```
for (инициализирующее-выражение; проверочное-выражение; обновляющее-выражение)
{
    оператор(ы)
}
```

может быть переписан так:

```
инициализирующее-выражение;
while (проверочное-выражение)
{
   оператор(ы)
   обновляющее-выражение;
}
```

Аналогично представленный ниже цикл while:

```
while (проверочное-выражение)
   тело
```

можно переписать так:

```
for ( ; проверочное-выражение;)
   тело
```

Цикл for требует трех выражений (или, технически, одного оператора и следующих за ним двух выражений), но они могут быть пустыми выражениями (или операторами). Обязательны только два знака точки с запятой. Кстати, пропуск проверочного выражения в цикле for трактуется как true, поэтому следующий цикл будет продолжаться бесконечно:

```
for( ; ; )
   тело
```

Поскольку циклы for и while почти эквивалентны, какой именно использовать — вопрос стиля. (Существует небольшое отличие, если тело включает оператор continue, который описан в главе 6.) Обычно программисты применяют циклы for для циклов со счетчиками, потому что формат for позволяет разместить всю необходимую информацию — начальное значение, конечное значение и метод обновления счетчика — в одном месте. Цикл while используется программистами, когда они не знают в точности заранее, сколько раз должен выполниться цикл.

На заметку!

Проектируя цикл, необходимо руководствоваться следующими указаниями:

- Идентифицировать условие его завершения.
- Инициализировать это условие перед первой проверкой.
- Обновлять условие на каждом шаге цикла, прежде чем оно будет проверено вновь.

Одним из преимуществ цикла for является то, что его структура предоставляет место для реализации всех этих требований, что помогает вам помнить о них.

Плохая пунктуация

Оба цикла — и for, и while — имеют тело, состоящее из одного оператора, следующего за выражениями в скобках. Как вы уже видели, этот единственный оператор может быть блоком, содержащим несколько операторов. Помните, что фигурные скобки, а не отступ, формируют блок. Например, посмотрите на следующий фрагмент кода:

```
i = 0;
while (name[i] != '\0')
    cout << name[i] << endl;
    i++;
cout << "Done\n";
```

Отступ говорит о том, что автор программы, вероятно, намеревался включить оператор `i++;` в тело цикла. Но отсутствие фигурных скобок говорит компилятору, что тело цикла состоит из единственного первого оператора `cout`. Таким образом, этот цикл будет бесконечно печатать первый символ массива. Программа никогда не достигнет оператора `i++;`, потому что он находится вне цикла.

Следующий пример демонстрирует другую потенциальную ловушку:

```
i = 0;
while (name[i] != '\0'); // проблема состоит в точке с запятой
{
    cout << name[i] << endl;
    i++;
}
cout << "Done\n";
```

На этот раз в коде верно установлены фигурные скобки, но перед ними идет лишняя точка с запятой. Напомним, что точка с запятой завершает оператор, поэтому здесь она завершает цикл `while`. Другими словами, телом цикла является *пустой оператор* — то есть *ничего*, за которым следует точка с запятой. Все, что находится в фигурных скобках, происходит *после* завершения цикла, и никогда не будет выполнено, потому что цикл, не делающий ничего, бесконечен. Опасайтесь беспорядочных точек с запятой.

Подождем минуточку — построение цикла задержки

Иногда возникает необходимость приостановить выполнение программы на некоторое время. Например, ваша программа может выдавать мгновенное сообщение на экран и тут же начать делать что-то еще, до того, как вы успеете его прочитать. В этом случае есть опасность пропустить жизненно важную информацию незамеченной. Было бы гораздо удобнее, если бы в этом случае программы приостановилась на 5 секунд, прежде чем продолжать работу. Цикл `while` удобен для создания такого эффекта. С давних времен существования персональных компьютеров для приостановки выполнения программы применялся способ, заключающийся в запуске простого счетчика:

```
long wait = 0;
while (wait < 10000)
   wait++; // молча посчитать
```

Проблема этого подхода состоит в том, что при переходе на компьютер с другой скоростью процессора приходилось менять предел значения счетчика. Некоторые игры, написанные для оригинального IBM PC, например, становились неуправляемо быстрыми при переносе на более быстрые компьютеры. Более правильный подход заключается в использовании системных часов для организации задержки.

Библиотеки ANSI C и C++ включают функцию, которая поможет вам в этом. Она называется `clock()` и возвращает системное время, прошедшее с момента запуска программы. Однако с ней связано несколько сложностей. Во-первых, `clock()` не обязательно возвращает время в секундах. Во-вторых, тип возврата этой функции может быть в одних системах `long`, в других — `unsigned long`, а в третьих — еще какимнибудь.

Однако заголовочный файл ctime (time.h в более старых реализациях) предлагает решение этой проблемы. Во-первых, он определяет символическую константу CLOCKS_PER_SEC, которая содержит количество единиц системного времени, приходящихся на секунду. То есть, разделив показание системного времени на эту константу, вы получите секунды. Или же вы можете умножить секунды на CLOCKS_PER_SEC, чтобы получить время в системных единицах. Во-вторых, ctime устанавливает псевдоним clock_t для типа возврата clock(). (См. ниже сноску "Псевдонимы типов".) Это значит, что вы можете объявить переменную типа clock_t, и компилятор преобразует ее в long или unsigned int либо в любой другой допустимый для вашей системы тип.

Замечание по совместимости

Системы, в которых еще нет заголовочного файла ctime, могут использовать вместо него time.h. У некоторых реализаций C++ могут быть проблемы с программой waiting.cpp, если библиотечные компоненты данной библиотеки не полностью совместимы с ANSI C. Дело в том, что функция clock() — это дополнение ANSI к традиционной библиотеке C. Кроме того, некоторые ранние реализации ANSI C используют CLK_TCK или TCK_CLK вместо более длинного CLOCKS_PER_SEC. Некоторые среды имеют проблемы с символом звукового сигнала \a, а также с согласованием дисплея при задержке времени.

В листинге 5.14 демонстрируется использование clock() и ctime для организации цикла задержки.

Листинг 5.14. waiting.cpp

```
// waiting.cpp -- использование clock() в цикле временной задержки
#include <iostream>
#include <ctime> // описывает функцию clock() и тип clock_t
int main()
{
    using namespace std;
    cout << "Введите время задержки в секундах: ";

    float secs;
    cin >> secs;

    clock_t delay = secs * CLOCKS_PER_SEC;   // преобразовать в системные
                                             // единицы времени (тики)
    cout << "starting\a\n";
    clock_t start = clock();

    while (clock() - start < delay )         // ждать истечения времени
       ; // обратите внимание на эту точку с запятой
    cout << "готово \a\n";

    return 0;
}
```

За счет подсчета времени задержки в системных единицах вместо секунд программа из листинга 5.14 избегает необходимости преобразования времени в секунды на каждом шаге цикла.

Псевдонимы типов

В C++ предусмотрены два способа установки нового имени в качестве псевдонима для типа. Один — через препроцессор:

```
#define BYTE char // препроцессор заменяет BYTE на char
```

Препроцессор затем заменяет все появления `BYTE` на `char` во время компиляции программы, таким образом, рассматривая `BYTE` в качестве псевдонима `char`.

Второй способ заключается в применении ключевого слова С (или C++) `typedef` для создания псевдонима. Например, чтобы объявить `byte` псевдонимом `char`, вы поступаете следующим образом:

```
typedef char byte; // делает byte псевдонимом char
```

Вот обобщенная форма:

```
typedef typeName aliasName;
```

Другими словами, если вы хотите, чтобы *aliasName* был псевдонимом для определенного типа, вы объявляете *aliasName*, как если бы он был переменной этого типа, и предваряете объявление ключевым словом `typedef`. Например, чтобы сделать `byte_pointer` псевдонимом указателя на `char`, вы должны объявить `byte_pointer` как указатель на `char` и поставить впереди `typedef`:

```
typedef char * byte_pointer; // указатель на тип char
```

Можете попробовать что-то подобное реализовать через `#define`, но это не работает, когда объявляется список переменных. Например, рассмотрим следующее:

```
#define FLOAT_POINTER float *
FLOAT_POINTER pa, pb;
```

Препроцессор выполнит подстановку и превратит это объявление в следующее:

```
float * pa, pb; // pa — указатель на float, a pb - просто float
```

Подход `typedef` лишен этой проблемы. Способность объявлять более сложные псевдонимы типов делает применение `typedef` более предпочтительным выбором перед `#define`, а иногда и единственно возможным.

Обратите внимание, что `typedef` не создает нового типа. Он просто дает существующему типу новое имя. Если вы объявляете `word` как псевдоним `int`, `cout` трактует значения типа `word`, как если бы они были типа `int`.

Цикл do while

Вы познакомились с циклами `for` и `while`. Третий цикл C++ — `do while`. Он отличается от двух других тем, что это цикл с проверкой на выходе. Это значит, что цикл "черт его знает" сначала выполнит свое тело, и только потом оценит проверочное условие, чтобы узнать, нужно ли продолжать дальше. Если условие оценивается как `false`, цикл прерывается; в противном случае выполняется новый шаг с последующей проверкой условия. Такой цикл всегда исполняется, как минимум, один раз, потому что поток управления программы проходит через его тело до того, как достигает проверочного условия. Вот синтаксис цикла `do while`:

```
do
   тело
while (проверочное-выражение);
```

Часть *тело* может быть единственным оператором либо блоком операторов, заключенным в фигурные скобки. На рис. 5.4 показан поток управления в цикле do while.

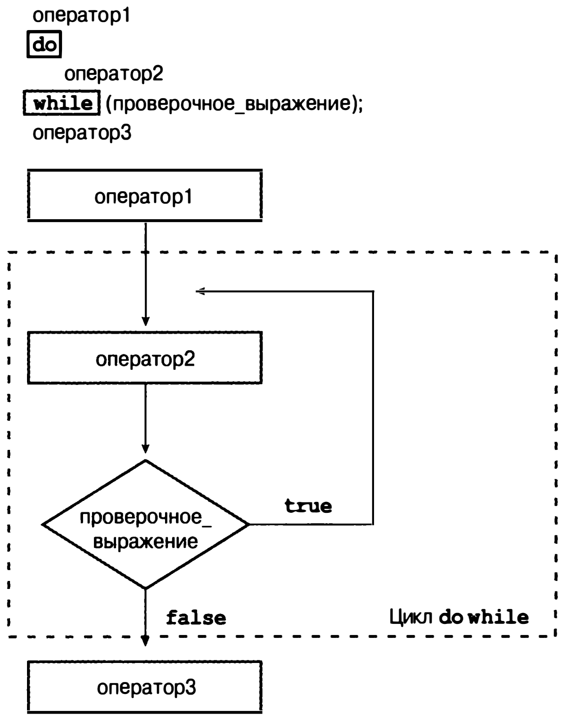


Рис. 5.4. Структура циклов **do while**

Обычно цикл с проверкой на входе — лучший выбор, чем цикл с проверкой на выходе, потому что проверка выполняется до его запуска. Например, предположим, что в листинге 5.13 использовался бы do while вместо while. В этом случае цикл должен будет печатать нулевой символ и его код, прежде чем обнаружит, что он уже достиг конца строки. Но все же иногда проверка в стиле do while имеет смысл. Например, если вы запрашиваете пользовательский ввод, то программа должна получить его, а только затем проверить. В листинге 5.15 показано, как можно использовать do while в такой ситуации.

Листинг 5.15. dowhile.cpp

```
// dowhile.cpp -- цикл с проверкой на выходе
#include <iostream>

int main()
{
   using namespace std;
   int n;

   cout << "Угадайте мое любимое число в диапазоне 1-10 \n";
```

```
  do
  {
    cin >> n;          // выполнить тело
  } while (n != 7);    // затем проверить
  cout << "Да, 7 — мое любимое. \n" ;
  return 0;
}
```

Ниже показан пример выполнения этой программы:

```
Угадайте мое любимое число в диапазоне 1-10
9
4
7
Да, 7 — мое любимое.
```

Пример из практики: странные циклы for

Не очень часто, но иногда вам может встретиться код, который представляет нечто такое:

```
for(;;) // иногда называется "бесконечным циклом"
{
    I++;
    // делать что-то...
    if (30 >= I) break; // операторы if и break (глава 6)
}
```

или другой вариант:

```
for(;;I++)
{
    if (30 >= I) break;
       // делать что-то...
}
```

Этот код полагается на тот факт, что пустое проверочное условие в цикле `for` трактуется как истинное. Ни один из этих примеров не является читабельным, и ни один не стоит использовать в качестве общего образца для написания циклов. Функциональность первого примера может быть выражена яснее циклом `do while`:

```
do {
    I++;
    // делать что-то...
while (30 < I);
```

Соответственно, второй пример может быть выражен яснее циклом `while`:

```
while (I < 30)
{
    // делать что-то...
    I++;
}
```

В общем случае, написание ясного, хорошо понятного кода — более важная цель, нежели демонстрация умения применять малоизвестные средства языка.

Циклы и текстовый ввод

Теперь, когда вы узнали, как работают циклы, давайте рассмотрим одну из наиболее часто встречающихся и важных задач, выполняемых циклами: чтение текстового ввода из файла или с клавиатуры символ за символом.

Например, вам может понадобиться написать программу, которая подсчитывает количество символов, строк и слов во входном потоке. Традиционно C++, как и C, использует цикл `while` для выполнения задач подобного рода. Давайте-ка посмотрим, как это делается. Если вы уже знаете C, не переходите слишком быстро к следующему разделу. Хотя цикл C++ `while` — точно такой же, как в C, средства ввода-вывода в C++ отличаются. Это может придать циклу C++ несколько другой вид, чем у цикла C. Фактически, объект `cin` поддерживает три разных режима односимвольного ввода, каждый со своим собственным пользовательским интерфейсом. Давайте посмотрим, как использовать эти варианты в циклах `while`.

Применение для ввода простого `cin`

Если программа собирается использовать цикл для чтения текстового ввода с клавиатуры, она должна каким-то образом узнать, когда ей следует остановиться. Как она узнает об этом? Один способ заключается в использовании некоторого специального символа, иногда называемого *сигнальным символом* (sentinel character) в качестве сигнала останова. Например, листинг 5.16 прекращает чтение ввода, когда программа встречает символ #. Программа считает количество прочтенных символов и отображает их. То есть она повторно выводит прочтенные символы. (Нажатие буквенной клавиши не приводит к автоматическому помещению соответствующего символа на экран; программы должны выполнять эту нудную работу по отображению введенного символа. Обычно этим занимается операционная система. В данном случае это будет делать и операционная система, и тестовая программа, отображая ввод.) По завершении работы программа выдаст отчет об общем количестве обработанных символов. В листинге 5.16 показан исходный код этой программы.

Листинг 5.16. textin1.cpp

```
// textin1.cpp -- чтение символов в цикле while
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   int count = 0;    // использовать базовый ввод
   cout << "Вводите символы; для выхода введите #:\n";
   cin >> ch;        // получить символ
   while (ch != '#') // проверить символ
   {
     cout << ch;     // отобразить символ
     ++count;        // посчитать символ
     cin >> ch;      // получить следующий символ
   }
   cout << endl << count << " символов прочитано \n";
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 5.16:

```
Вводите символы; для выхода введите #:
see ken run#really fast
seekenrun
9 символов прочитано
```

Похоже, Кен работает так быстро, что успевает уничтожить символы пробела — по крайней мере, пробельные символы при вводе.

Замечания по программе

Обратите внимание на структуру программы в листинге 5.16. Программа читает первый введенный символ до входа в цикл. Таким образом, первый символ может быть проверен, когда программа достигает оператора цикла. Это важно, потому что первым символом может сразу оказаться #. Поскольку `textin1.cpp` использует цикл с проверкой на входе, в этом случае программа корректно пропустит весь цикл. А поскольку переменной `count` было предварительно присвоено значение `0`, `count` будет содержать правильное значение.

Предположим, что первый прочтенный символ — не #. В этом случае программа входит в цикл, отображает символ, увеличивает значение счетчика и читает следующий символ. Последний шаг жизненно важен. Без него цикл бесконечно обрабатывал бы первый введенный символ. Но благодаря этому последнему шагу, программа может перейти к следующему символу.

Обратите внимание, что дизайн цикла следует правилам, упомянутым ранее. Условие, прерывающее выполнение цикла — прочтение последним символа #. Это условие инициализируется первым чтением символа перед входом в цикл. Условие обновляется чтением следующего символа в конце тела цикла.

Все это звучит разумно. Но почему же программа не выводит пробелы? Виноват `cin`. Когда он читает значения типа `char`, как и при чтении других базовых типов, `cin` пропускает пробелы и символы перевода строки. Эти символы не вводятся, не могут быть подсчитаны и не отображаются.

Чтобы еще более усложнить ситуацию, сообщим, что ввод в `cin` буферизуется. Это значит, что символы, которые вы вводите, не попадают в программу до тех пор, пока вы не нажмете клавишу <Enter>. Вот почему программа из листинга 5.16 позволяет печатать символы и после #. После того, как вы нажимаете <Enter>, вся последовательность символов посылается программе, но программа прекращает обработку ввода после прочтения #.

Спасение в виде `cin.get(char)`

Обычно программы, принимающие ввод символ за символом, должны обрабатывать каждый введенный символ, включая пробелы, знаки табуляции и символы новой строки. Класс `istream` (определенный в `iostream`), к которому относится объект `cin`, включает функцию-член, которая отвечает этому требованию. В частности, функция `cin.get(ch)` читает из ввода следующий символ, даже если это пробел, и присваивает его переменной `ch`. Заменив `cin >> ch` вызовом этой функции, вы можете устранить недостатки программы из листинга 5.16. В листинге 5.17 показан результат.

Листинг 5.17. textin2.cpp

```
// textin2.cpp -- использование cin.get(char)
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   int count = 0;
   cout << Вводите символы; для выхода введите #:\n";
   cin.get(ch);     // использовать функцию cin.get(ch)
   while (ch != '#')
   {
     cout << ch;
     ++count;
     cin.get(ch);   // использовать ее снова
   }
   cout << endl << count << " символов прочитано \n";
   return 0;
}
```

Ниже можно видеть пример выполнения этой версии программы:

```
Вводите символы; для выхода введите #:
Did you use a #2 pencil?
Did you use a
14 символов прочитано
```

Теперь программа отображает и подсчитывает все символы, включая пробелы. Ввод по-прежнему буферизуется, поэтому по-прежнему можно ввести больше информации, чем на самом деле попадет в программу.

Если вы знакомы с языком С, эта программа может показаться вам опасно ошибочной. Вызов `cin.get(ch)` помещает значение в переменную `ch`, а это означает, что она изменяет значение переданной переменной. В С вы должны передавать адрес переменной функции, если хотите, чтобы она изменила ее значение. Но вызов `cin.get()` в листинге 5.17 передает `ch`, а не `&ch`. В языке С подобный код не работает. В С++ он может работать, если функция объявляет свой аргумент как *ссылку*. Это — новое средство С++. Заголовочный файл `iostream` объявляет аргумент `cin.get(ch)` как ссылочный тип, поэтому данная функция может изменять значение своего аргумента. Более подробно об этом вы узнаете в главе 8. А пока знатоки С могут расслабиться; обычно аргументы, передаваемые в С++, работают так же, как и в С. Но не в случае `cin.get(ch)`.

Который из `cin.get()`?

Листинг 4.5 в главе 4 содержит следующий код:

```
char name[ArSize];
...
cout << "Введите свое имя: \n";
cin.get(name, ArSize).get();
```

Последняя строка эквивалентна следующим последовательным вызовам функции:

```
cin.get(name, ArSize);
cin.get();
```

Одна версия cin.get() принимает два аргумента: имя массива, который представляет адрес строки (технически char*), и ArSize, который является целым типа int. (Напомним, что имя массива — это адрес его первого элемента, поэтому имя символьного массива имеет тип char*.) Затем программа использует cin.get() без аргументов. Но в последнем примере мы использовали cin.get() следующим образом:

```
char ch;
cin.get(ch);
```

На этот раз cin.get() принимает один аргумент, и его типом является char.

И здесь опять приходит время тем из вас, кто знаком с языком С, прийти в замешательство. В С, если функция принимает в качестве аргументов указатель на char и int, то вы не можете с таким же успехом использовать ту же функцию с одним аргументом, да еще и другого типа. Но с C++ такое возможно, потому что этот язык поддерживает возможность ООП, называемую *перегрузкой функций*. Перегрузка функций позволяет создавать разные функции с одним и тем же именем, при условии, что списки их аргументов отличаются. К примеру, если вы используете cin.get(name, ArSize) в C++, компилятор находит ту версию cin.get(), которая принимает аргументы char* и int. Но если вы используете cin.get(ch), то компилятор найдет версию cin.get(), которая принимает единственный аргумент типа char. И, наконец, если код не передает никаких аргументов, то компилятор использует версию cin.get() без аргументов. Перегрузка функций позволяет использовать одно и то же имя для взаимосвязанных функций, выполняющих одну и ту же задачу разными способами или с разными типами данных. Это — еще одна тема, знакомство с которой ожидает вас в главе 8. А пока вы можете привыкнуть к перегрузке функций, используя примеры get(), поставляемые классом istream. Чтобы отличать друг от друга разные версии одной функции, упоминая их, мы будем указывать список аргументов. То есть cin.get() будет означать версию, не принимающую аргументов, а cin.get(char) — версию с одним аргументом.

Условие конца файла

Как показано в листинге 5.17, применение символа вроде # для обозначения конца ввода, не всегда подходит, потому что такой ввод может быть частью совершенно легитимного ввода. То же верно и любого другого символа, такого как @ или %. Если ввод поступает из файла, вы можете задействовать более мощную технику — обнаружение конца файла (end-of-file — EOF). Средства ввода C++ взаимодействуют с операционной системой для обнаружения момента достижения конца файла и предоставляют эту информацию программе.

На первый взгляд чтение информации из файла имеет мало общего с cin и клавиатурным вводом, но между ними существуют две связи. Во-первых, многие операционные системы, включая Unix и MS-DOS, поддерживают *перенаправление*, что позволяет легко подставлять файл вместо клавиатурного ввода. Например, предположим, что в MS-DOS у вас есть исполняемая программа gofish.exe и текстовый файл по имени fishtale. В этом случае вы можете вести следующую команду в ответ на приглашение командной строки:

```
gofish <fishtale
```

Это заставит программу принять ввод из файла `fishtale` вместо клавиатуры. Символ < — это операция перенаправления, как в Unix, так и в DOS.

Во-вторых, многие операционные системы позволяют эмулировать условие EOF с клавиатуры. В Unix вы можете сделать это, нажав <Ctrl+D> в начале строки. В DOS для этого нужно нажать <Ctrl+Z> и после этого <Enter> в любом месте строки. Некоторые реализации C++ поддерживают подобное поведение, даже если его не поддерживает операционная система. Концепция EOF для клавиатурного ввода в действительности унаследована от сред командной строки. Однако Symantec C++ для Mac имитирует Unix и распознает <Ctrl+D> как эмуляцию EOF. Metrowerks Codewarrior распознает <Ctrl+Z> в средах Macintosh и Windows. Microsoft Visual C++ 7.0, Borland C++ 5.5 и GNU C++ для PC распознают <Ctrl+Z>, когда он встречается в начале строки, но требует последующего нажатия <Enter>. Короче говоря, многие среды программирования для PC распознают комбинацию <Ctrl+Z> как эмуляцию EOF, но конкретные детали (в любом месте строки или же только в начале, нужно последующее нажатие <Enter> или нет) могут варьироваться.

Если среда программирования предусматривает проверку EOF, вы можете использовать программу, подобную приведенной в листинге 5.17, с перенаправленными файлами либо с клавиатурным вводом, в котором эмулируется EOF. Это выглядит удобным, поэтому давайте посмотрим, как это делается.

Когда `cin` обнаруживает EOF, он устанавливает двум битам (`eofbit` и `failbit`) значение 1. Вы можете использовать функцию-член по имени `eof()` для проверки состояния `eofbit`; вызов `cin.eof()` возвращает булевское значение `true`, если EOF был обнаружен, и `false` — в противоположном случае. Аналогично, функция-член `fail()` возвращает `true`, если `eofbit` или `failbit` установлены в 1. Обратите внимание, что методы `eof()` и `fail()` сообщают результат самой последней попытки чтения; то есть они сообщают о прошлом, а не заглядывают в будущее. Поэтому проверки `cin.eof()` и `cin.fail()` должны всегда следовать за попытками чтения. Дизайн программы в листинге 5.18 отражает этот факт. Она использует `fail()` вместо `eof()`, потому что первый из этих методов работает в более широком диапазоне реализаций.

Замечание по совместимости

Некоторые системы не поддерживают эмуляцию EOF с помощью клавиатуры. Другие ее поддерживают, но не лучшим образом. Если вы использовали `cin.get()` для приостановки вывода на экран, чтобы вы могли прочесть его, то здесь это не работает, потому что обнаружение EOF отключает дальнейшие попытки чтения. Однако вы можете использовать цикл задержки, подобный тому, что применен в листинге 5.14, чтобы на время оставить экран видимым.

Листинг 5.18. textin3.cpp

```
// textin3.cpp -- чтение символов до конца файла
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   int count = 0;
   cin.get(ch);                    // попытка прочитать символ
   while (cin.fail() == false)  // проверка на EOF
   {
```

```
    cout << ch;     // отобразить символ
    ++count;
    cin.get(ch);    // попытка читать следующий символ
  }
  cout << endl << count << " символов прочитано \n";
  return 0;
}
```

Вот пример выполнения этой программы:

```
The green bird sings in the winter.<ENTER>
The green bird sings in the winter.
Yes, but the crow flies in the dawn.<ENTER>
Yes, but the crow flies in the dawn.
<CTRL><Z>
73 символов прочитано
```

Поскольку эта программа запускалась под управлением Windows XP, для эмуляции условия EOF нажималась комбинация <Ctrl+Z> и затем <Enter>. Пользователи Unix и Linux должны вместо этого нажимать <Ctrl+D>.

Применив перенаправление, вы можете использовать программу из листинга 5.18 для отображения текстового файла и сообщения о количестве содержащихся в нем символов. На этот раз мы используем ее для чтения, отображения и подсчета символов файла, содержащего две строки на системе Unix (`$` — это приглашение командной строки Unix):

```
$ textin3 < stuff
I am a Unix file. I am proud
to be a Unix file.
49 символов прочитано
$
```

EOF завершает ввод

Напомним, что когда метод `cin` обнаруживает EOF, он устанавливает флаг в объекте `cin`, обозначающий условие EOF. Когда установлен этот флаг, `cin` не читает больше никакого ввода, и последующие вызовы `cin` не имеют никакого эффекта. Для файлового ввода это имеет смысл, поскольку вы не можете читать ничего за концом файла. Однако при клавиатурном вводе вы можете эмулировать EOF для прерывания цикла, но захотите позже продолжать вводить информацию. Метод `cin.clear()` очищает флаг EOF и позволяет продолжить обработку ввода. В главе 17 это обсуждается более подробно. Однако имейте в виду, что на некоторых системах нажатие <Ctrl+Z> полностью прекращает ввод и вывод, несмотря на возможности `cin.clear()` восстановить их.

Общие идиомы символьного ввода

Ниже представлен дизайн цикла, предназначенного для чтения текста по одному символу вплоть до получения EOF:

```
cin.get(ch);                  // попытка прочесть символ
while (cin.fail() == false)  // проверка на EOF
{
```

```
    ... // делать что-то полезное
    cin.get(ch);     // попытка прочесть следующий символ
}
```

Этот код можно несколько сократить. В главе 6 будет представлена операция !, которая обращает true в false и наоборот. Применяя ее, вы можете переписать проверочное условие while следующим образом:

```
while (!cin.fail()) // пока ввод не сбоит
```

Возвращаемое значение cin.get(char) — это сам объект cin. Однако класс istream предусматривает функцию, которая преобразует такой объект istream, как cin, в булевское значение; эта функция преобразования неявно вызывается, когда cin появляется в выражениях, где ожидается значение типа bool, например, в проверочном условии цикла while. Более того, это значение bool равно true, только если последняя попытка чтения завершилась успешно, в противном случае оно равно false. Это значит, что проверочное условие цикла while можно переписать так:

```
while (cin)          // пока ввод успешен
```

Эта проверка несколько более широка, чем применение !cin.fail() или !cin.eof(), потому что обнаруживает также некоторые другие причины сбоев, такие как отказ диска.

И, наконец, поскольку возвращаемое значение cin.get(ch) — это сам cin, вы можете сократить цикл до следующего формата:

```
while (cin.get(ch)) // пока ввод успешен
{
    ... // делать что-то полезное
}
```

Здесь cin.get(char) вызывается лишь однажды — в составе проверочного условия, а не дважды — один раз перед циклом и один — в конце тела цикла, как было раньше. Чтобы оценить проверочное условие цикла, программа сначала должна выполнить cin.get(ch), в случае успеха которого введенный символ будет помещен в ch. Затем программа получает возвращаемое значение этого вызова — сам объект cin. Затем она применяет преобразование cin к bool, принимающему значение true, если ввод отработал, и false — в противоположном случае. Три рекомендации (идентификация условия прерывания, инициализация этого условия и обновление) — все они оказались упакованы в одно проверочное условие цикла.

Еще одна версия cin.get()

Ностальгирующие пользователи С могут тосковать по функциям getchar() и putchar(), предлагаемым языком С. Они доступны в С++, если вы в них нуждаетесь. Для этого просто нужно включить заголовочный файл stdio.h, как это делалось и в программах С (либо обратиться к более современной версии cstdio). Или же вы можете применять функции классов istream и ostream, которые работают почти точно так же. Давайте теперь рассмотрим и этот подход.

Замечание по совместимости

Некоторые старые реализации C++ не поддерживают обсуждаемую здесь функцию-член `cin.get()` (без аргументов).

Функция-член `cin.get()` без аргументов возвращает следующий символ ввода. То есть, вы можете использовать ее следующим образом:

```
ch = cin.get();
```

(Вспомните, что `cin.get(ch)` возвращает объект, а не прочитанный символ.) Эта функция работает почти так же, как `getchar()` в языке C, возвращая код символа как значение типа `int`. Аналогично вы можете использовать функцию `cout.put()` (см. главу 3) для отображения символа:

```
cout.put(ch);
```

Она работает так же, как `putchar()` в языке C, с единственным отличием — аргумент должен быть типа `char`, а не `int`.

Замечание по совместимости

Изначально функция-член `put()` имеет единственный прототип — `put(char)`. Вы можете передать ей аргумент `int`, который неявно приводится к `char`. Стандарт также утверждает один прототип. Однако многие современные реализации C++ предлагают три прототипа: `put(char)`, `put(signed char)` и `put(unsigned char)`. Вызов `put()` с аргументом типа `int` в этих реализациях генерирует сообщение об ошибке, поскольку более одного варианта допускают преобразование в `int`. Явное приведение типа, такое как `cin.put(char(ch))`, позволяет передавать `int`.

Чтобы успешно применять `cin.get()`, вы должны знать, как эта версия функции обрабатывает условие EOF. Когда функция достигает EOF, не остается символов, которые должны быть возвращены. Вместо этого `cin.get()` возвращает специальное значение, представленное символической константой `EOF`. Эта константа определена в заголовочном файле `iostream`. Значение `EOF` должно отличаться от любого допустимого значения символа, чтобы программа не могла спутать `EOF` с обычным символом. Обычно `EOF` определяется как `-1`, потому что ни один из символов не имеет ASCII-кода, равного `-1`. Однако вам не обязательно знать действительное значение. В программе вы просто используете `EOF`. Например, центральная часть листинга 5.18 выглядит так:

```
char ch;
cin.get(ch);
while (cin.fail() == false) // проверка на EOF
{
   cout << ch;
   ++count;
   cin.get(ch);
}
```

Вы можете использовать `int ch`, заменить `cin.get(ch)` на `cin.get()`, заменить `cout` на `cout.put()` и заменить `cin.fail()` проверкой на `EOF`:

```
int ch; /// для совместимости со значением EOF
ch = cin.get();
```

```
while (ch != EOF)
{
   cout.put(ch); // cout.put(char(ch)) для некоторых реализаций
   ++count;
   ch = cin.get();
}
```

Если ch — символ, цикл отображает его. Если же ch — EOF, цикл прекращается.

На заметку!

Следует понимать, что EOF не представляет вводимый символ. Это просто сигнал о том, что символов больше нет.

Существует один тонкий, но важный момент, касающийся применения cin.get(), о котором еще не было сказано. Поскольку EOF представляет значение, находящееся вне допустимых кодов символов, может случиться, что оно будет не совместимо с типом char. Например, в некоторых системах тип char является беззнаковым, поэтому переменная типа char никогда не получит обычного значения EOF, равного -1. По этой причине, если вы используете cin.get() (без аргументов) и проверяете возврат на EOF, то должны присваивать возвращенное значение int вместо char. К тому же, если вы объявите ch типа int вместо char, то, возможно, вам придется выполнить приведение к типу char для его отображения.

В листинге 5.19 представлен подход cin.get() в новой версии программы из листинга 5.18. Он также сокращает код за счет комбинации символьного ввода с проверочным условием цикла while.

Листинг 5.19. textin4.cpp

```
// textin4.cpp -- чтение символов с помощью cin.get()
#include <iostream>
int main(void)
{
   using namespace std;
   int ch; // должно быть типа int, а не char
   int count = 0;
   while ((ch = cin.get()) != EOF) // проверка конца файла
   {
     cout.put(char(ch));
     ++count;
   }
   cout << endl << count << " символов прочитано\n";
   return 0;
}
```

Замечание по совместимости

Некоторые системы, которые либо не поддерживают эмуляцию EOF с клавиатуры, либо поддерживают ее не лучшим образом, могут помешать выполнению примера 5.19 так, как описано. Если вы используете cin.get() для приостановки вывода на экран с целью прочтения, то это не будет работать, потому что обнаружение EOF отключает дальнейшие попытки чтения ввода. Однако вы можете использовать цикл задержки вроде показанного в листинге 5.14, чтобы на время сохранить экран видимым.

Ниже можно видеть пример выполнения программы из листинга 5.19:

```
The sullen mackerel sulks in the shadowy shallows.<ENTER>
The sullen mackerel sulks in the shadowy shallows.
Yes, but the blue bird of happiness harbors secrets.<ENTER>
Yes, but the blue bird of happiness harbors secrets.
^Z
104 символов прочитано
```

Давайте проанализируем следующее условие цикла:

```
while ((ch = cin.get()) != EOF)
```

Скобки, в которые заключено подвыражение `ch = cin.get()`, заставляют программу вычислить его в начале. Чтобы выполнить вычисление, программа сначала вызывает функцию `cin.get()`. Затем она присваивает возвращенное значение функции переменной `ch`. Поскольку значением оператора присваивания является значение левого операнда, то полное подвыражение сводится к значению `ch`. Если это значение равно `EOF`, цикл прерывается, в противном случае — продолжается. Проверочному условию нужны все эти скобки. Предположим, что вы уберете пару скобок:

```
while (ch = cin.get() != EOF)
```

Операция `!=` имеет более высокий приоритет, чем =, поэтому сначала программа сравнит возвращаемое значение `cin.get()` с `EOF`. Сравнение даст в результате `true` или `false`; значение `bool` преобразуется в `0` или `1`, и это присваивается `ch`.

С другой стороны, применение `cin.get(ch)` (с аргументом) для ввода не создает никаких проблем, связанных с типом. Вспомним, что `cin.get(char)` не присваивает специального значения `ch` по достижении `EOF`. Фактически, в этом случае она не присваивает `ch` ничего. `ch` никогда не приходится хранить несимвольные значения. В табл. 5.3 представлены различия между `cin.get(char)` и `cin.get()`.

Таблица 5.3. Сравнение `cin.get(char)` и `cin.get()`

Свойство	`cin.get(char)`	`cin.get()`
Метод доставки вводимого символа	Присваивание аргументу	Возвращаемое значение
Возвращаемое значение функции при символьном вводе	Объект класса `istream` (после преобразования к `bool` — `true`)	Код символа, как значение типа `int`
Возвращаемое значение функции при `EOF`	Объект класса `istream` (после преобразования к `bool` — `false`)	`EOF`

Итак, что вы должны использовать — `cin.get()` или `cin.get(char)`? Форма с символьным аргументом более полно интегрирована в объектный подход, поскольку ее возвращаемым значением является объект `istream`. Это значит, например, что вы можете связывать вызовы в цепочку. Например, приведенный ниже код означает чтение следующего входящего символа в `ch1`, и еще одного следующего — в `ch2`:

```
cin.get(ch1).get(ch2);
```

Это работает, потому что вызов функции `cin.get(ch1)` возвращает объект `cin`, который затем работает как объект, к которому присоединен следующий вызов `get(ch2)`.

Возможно, основное назначение формы get() — обеспечить возможность быстрого чернового перехода от функций getchar() и putchar() из stdio.h к методам класса iostream — cin.get() и cout.put(). Вы просто заменяете один заголовочный файл другим и проводите глобальную замену getchar() и putchar() на их эквиваленты — методы. (Если старый код использует переменную типа int для ввода, то вы должны будете также внести соответствующие изменения, если ваша реализация включает несколько прототипов put().)

Вложенные циклы и двумерные массивы

Ранее в этой главе вы уже видели, что цикл for — это естественный инструмент для обработки массивов. Теперь мы сделаем ещё один шаг и посмотрим, как цикл for, внутри которого находится еще один цикл for (вложенные циклы), может применяться для обработки двумерных массивов.

Во-первых, что такое двумерный массив? Массивы, которые мы использовали до сих пор в этой главе, относятся к *одномерным массивам*, потому что каждый из них можно визуализировать как одну строку данных. Двумерный массив можно визуализировать в виде таблицы, состоящей из строк и столбцов. Вы можете использовать двумерный массив, например, чтобы представить поквартальные уровни продаж по разным регионам, причем каждая строка данных будет представлять один регион. Или же вы можете использовать двумерный массив для представления положения робота на поле компьютерной игры.

В C++ не предусмотрен специальный тип представления двумерных массивов. Вместо этого вы создаете массив, каждый элемент которого является массивом. Например, предположим, вы хотите сохранить данные о максимальной температуре в четырех городах за 4-летний период. В этом случае вы можете объявить массив следующим образом:

```
int maxtemps[4][5];
```

Это объявление означает, что maxterms — массив из четырех элементов. Каждый из этих элементов сам является массивом из пяти элементов. (См. рис. 5.5.) Вы можете интерпретировать массив maxterms как представление четырех строк, по пять значений температуры в каждой.

Выражение maxtemps[0] означает первый элемент массива maxtemps. Таким образом, maxtemps[0] — сам по себе массив из пяти int. Первый элемент массива maxtemps[0] — maxtemps[0][0], и этот элемент имеет тип int. Таким образом, вы должны использовать два индекса для доступа к элементам int. Первый индекс можно представлять как строку таблицы, а второй — как ее столбец (рис. 5.6).

Предположим, что требуется распечатать все содержимое массива. В этом случае вы можете использовать цикл for для прохода по строкам и второй вложенный цикл for — для прохода по столбцам:

```
for (int row = 0; row < 4; row++)
{
   for (int col = 0; col < 5; ++col)
      cout << maxtemps[row][col] << "\t";
   cout << endl;
}
```

Для каждого значения row вложенный цикл for проходит по значениям col. Этот пример печатает символ табуляции (\t в нотации управляющих символов C++) после каждого значения и символ новой строки после каждой полной строки.

maxterms — массив из четырех элементов

```
int maxtemps[4][5];
```

Каждый элемент — массив из 5 значений int

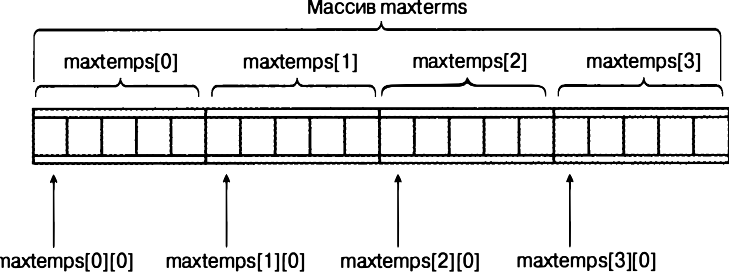


Рис. 5.5. Массив массивов

```
int maxtemps[4][5];
```

Массив maxterms, представленный в виде таблицы

		0	1	2	3	4
maxtemps[0]	0	maxtemps[0][0]	maxtemps[0][1]	maxtemps[0][2]	maxtemps[0][3]	maxtemps[0][4]
maxtemps[1]	1	maxtemps[1][0]	maxtemps[1][1]	maxtemps[1][2]	maxtemps[1][3]	maxtemps[1][4]
maxtemps[2]	2	maxtemps[2][0]	maxtemps[2][1]	maxtemps[2][2]	maxtemps[2][3]	maxtemps[2][4]
maxtemps[3]	3	maxtemps[3][0]	maxtemps[3][1]	maxtemps[3][2]	maxtemps[3][3]	maxtemps[3][4]

Рис. 5.6. Доступ к элементам массива по индексам

Инициализация двумерного массива

Когда вы создаете двумерный массив, то у вас есть возможность инициализировать каждый элемент. Техника основана на способе инициализации одномерного массива. Напомним, что это делается указанием разделенного запятыми списка элементов, заключенного в фигурные скобки:

```
// инициализация одномерного массива
int btus[5] = { 23, 26, 24, 31, 28};
```

В двумерном массиве каждый элемент сам по себе является массивом, поэтому вы можете инициализировать каждый элемент, как показано в предыдущем примере. То есть инициализация состоит из разделенной запятыми серии одномерных инициализаций, каждая из которых заключена в фигурные скобки:

```
int maxtemps[4][5] = // 2-мерный массив
{
   {94, 98, 87, 103, 101},   // значения для maxtemps[0]
   {98, 99, 91, 107, 105},   // значения для maxtemps[1]
   {93, 91, 90, 101, 104},   // значения для maxtemps[2]
   {95, 100, 88, 105, 103}   // значения для maxtemps[3]
};
```

Выражение {94, 98, 87, 103, 101} инициализирует первую строку, представленную maxtemps[0]. Стиль размещения каждой строки данных в отдельной строке кода повышает читабельность.

Листинг 5.20 представляет в одной программе инициализацию двумерного массива и проход по его элементам во вложенном цикле. На этот раз программа меняет порядок циклов, помещая проход по столбцам (индекс города) во внешний цикл, а проход по строкам (индекс года) — во внутренний цикл. К тому же здесь используется общепринятая в C++ практика инициализации массива указателей набором строковых констант. То есть, cities объявлен как массив указателей на char. Это делает каждый его элемент, такой как cities[0], указателем на char, который может быть инициализирован адресом строки. Программа инициализирует cities[0] адресом строки "Gribble City" и так далее. Таким образом, массив указателей ведет себя как массив строк.

Листинг 5.20. nested.cpp

```
// nested.cpp -- вложенные циклы и двумерный массив
#include <iostream>
const int Cities = 5;
const int Years = 4;
int main()
{
   using namespace std;
   const char * cities[Cities] = // массив указателей
   { // для 5 строк
     "Gribble City",
     "Gribbletown",
     "New Gribble",
     "San Gribble",
     "Gribble Vista"
   };
   int maxtemps[Years][Cities] = // 2-мерный массив
   {
     {95, 99, 86, 100, 104},    // значения для maxtemps[0]
     {95, 97, 90, 106, 102},    // значения для maxtemps[1]
     {96, 100, 940, 107, 105},  // значения для maxtemps[2]
     {97, 102, 89, 108, 104}    // значения для maxtemps[3]
   };
   cout << "Максимальные температуры за 2003 — 2006 годы \n\n";
   for (int city = 0; city < Cities; ++city)
   {
     cout << cities[city] << ":\t";
     for (int year = 0; year < Years; ++year)
        cout << maxtemps[year][city] << "\t";
     cout << endl;
   }
   return 0;
}
```

Ниже показан вывод программы из листинга 5.20:

```
Максимальные температуры за 2003 — 2006 годы
Gribble City:    95    95    96    97
Gribbletown:     99    97    100   102
New Gribble:     86    90    940   89
San Gribble:     100   106   107   108
Gribble Vista:   104   102   105   104
```

Применение знаков табуляции позволяет разместить данные более равномерно, чем с помощью пробелов. Однако разные установки позиций табуляции могут привести к тому, что на разных системах вывод будет выглядеть немного по-разному. В главе 17 будут представлены более точные, но и более сложные методы форматирования вывода.

Более грубо вы могли бы использовать массив массивов char вместо массива указателей для размещения строковых данных. Объявление выглядело бы так:

```
char cities[25][Cities] = // массив из 5 массивов по 25 символов
{
   "Gribble City",
   "Gribbletown",
   "New Gribble",
   "San Gribble",
   "Gribble Vista"
};
```

Такой подход ограничивает длину каждой из 5 строк максимум 24 символами. Массив указателей сохраняет адреса пяти строковых литералов, но массив массивов char копирует каждый из пяти строчных литералов в соответствующий массив из 25 символов. То есть, массив указателей более экономичен в отношении используемой памяти. Однако если вы намерены модифицировать любую из этих пяти строк, то в этом случае двумерный массив символов — более удачный выбор. Это довольно странно, но оба варианта используют одинаковый список инициализации и один и тот же код циклов for для отображения строк.

К тому же вы можете использовать массив объектов класса string вместо массива указателей для сохранения данных. Объявление будет выглядеть следующим образом:

```
const string cities[Cities] = // массив из 5 строк
{
   "Gribble City",
   "Gribbletown",
   "New Gribble",
   "San Gribble",
   "Gribble Vista"
};
```

Если вам нужны модифицируемые строки, можете пропустить квалификатор const. Эта форма будет использовать тот же список инициализации и тот же цикл for для отображения строк, как и две другие формы. Если строки будут модифицируемыми, то свойство автоматического изменения размера класса string делает такой подход более удобным, чем применение двумерного массива символов.

Резюме

В C++ представлены три варианта циклов: for, while и do while. Цикл позволяет повторно выполнять один и тот же набор инструкций до тех пор, пока проверочное условие цикла оценивается как true или не ноль, и цикл прекращает их выполнение, когда это проверочное условие возвращает false или 0. Циклы for и while — циклы с проверкой на входе, это означает, что они оценивают проверочное условие перед выполнением операторов, находящихся в теле цикла. Цикл do while проверяет условие на выходе, то есть после выполнения операторов, содержащихся в его теле.

Синтаксис каждого цикла позволяет размещать в теле только один оператор. Однако этот оператор может быть составным, или блоком в виде группы операторов, заключенных в фигурные скобки.

Сравнивающие выражения (выражения отношений), которые сравнивают два значения, часто применяются в качестве проверочных условий цикла. Эти выражения формируются с использованием одной из шести операций отношений: <, <=, ==, >=, > или !=. Сравнивающие выражения возвращают значения типа `bool`: `true` или `false`.

Многие программы читают текстовый ввод или текстовые файлы символ за символом. Класс `istream` представляет несколько способов сделать. Если `ch` — переменная типа `char`, то оператор

```
cin >> ch;
```

читает очередной символ ввода в `ch`. Однако при этом пропускаются пробелы, переносы строки и символы табуляции. Вызов функции-члена

```
cin.get(ch);
```

читает очередной входной символ, независимо от его значения, и помещает его в `ch`. Вызов функции-члена `cin.get()` возвращает следующий символ ввода, включая пробелы, переносы строк и символы табуляции, поэтому он может быть использован следующим образом:

```
ch = cin.get();
```

Функция-член `cin.get(char)` сообщает о встреченном состоянии EOF, возвращая значение, преобразуемое в тип `bool`, как `false`, в то время как функция-член `cin.get()` сообщает о EOF, возвращая значение `EOF`, определенное в файле заголовка `iostream`.

Вложенный цикл — это цикл, находящийся в теле другого цикла `for`. Вложенные циклы обеспечивают естественный способ обработки двумерных массивов.

Вопросы для самоконтроля

1. В чем разница между циклами с проверкой на входе и циклами с проверкой на выходе? Какой из циклов C++ к какой категории относится?
2. Что должен напечатать следующий фрагмент кода, если использовать его в программе?

```
int i;
for (i = 0; i < 5; i++)
    cout << i;
    cout << endl;
```

3. Что напечатает следующий фрагмент кода, если использовать его в программе?

```
int j;
for (j = 0; j < 11; j += 3)
    cout << j;
cout << endl << j << endl;
```

4. Что напечатает следующий фрагмент кода, если использовать его в программе?

```
int j = 5;
while ( ++j < 9)
    cout << j++ << endl;
```

5. Что напечатает следующий фрагмент кода, если использовать его в программе?

```
int k = 8;
do
   cout <<" k = " << k << endl;
while (k++ < 5);
```

6. Напишите цикл `for`, который напечатает значения 1 2 4 8 16 32 64, увеличивая вдвое значение переменной счетчика на каждом шаге.
7. Как включить в тело цикла более одного оператора?
8. Правилен ли следующий оператор? Если нет — почему? Если да — что он делает?

```
int x = (1,024);
```

Как насчет следующего?

```
int y;
y = 1,024;
```

9. В чем отличие представления ввода между `cin>>ch`, `cin.get(ch)` и `ch=cin.get()`?

Упражнения по программированию

1. Напишите программу, запрашивающую у пользователя ввести два целых числа. Затем программа должна вычислить и выдать сумму всех целых чисел, лежащих между этими двумя целыми. Предполагается, что меньшее значение введено первым. Например, если пользователь ввел 2 и 9, программа должна сообщить, что сумма целых от 2 до 9 равна 44.
2. Напишите программу, которая приглашает пользователя вводить числа. После каждого введенного значения программа должна выдавать накопленную сумму введенных значений. Программа должна завершаться при вводе 0.
3. Дафна инвестировала $100 под простых 10%. То есть, каждый год инвестиция должна приносить 10% инвестированной суммы, то есть $10 каждый год:

   прибыль = 0.1 × *исходный баланс*

   В то же время Клео инвестировала $100 под сложных 5%. То есть прибыль составит 5% от текущего баланса, включая предыдущую накопленную прибыль:

   прибыль = 0.05 × *текущий баланс*

   Клео зарабатывает 5% от $100 в первый год, что дает ей $105. На следующий год она зарабатывает 5% от $105, что составляет $5.25, и так далее. Напишите программу, которая вычислит, сколько лет понадобится для того, чтобы сумма баланса Клео превысила сумму баланса Дафны, с отображением значений обоих балансов за каждый год.
4. Вы продаете книгу "Язык С++ для чайников". Напишите программу, которая позволит ввести помесячные объемы продаж в течение года (в количестве книг, а не в деньгах). Программа должна использовать цикл, в котором выводится приглашение с названием месяца, применяя массив указателей на `char` (или массив объектов `string`, если вы предпочитаете его), инициализированный строками — названиями месяцев, и сохраняя введенные значения в массиве `int`. Затем программа должна найти сумму содержимого массива и выдать общий объем продаж за год.
5. Выполните упражнение 4, но используя двумерный массив для сохранения данных о месячных продажах за 3 года. Выдайте общую сумму продаж за каждый год и за все годы вместе.

6. Спроектируйте структуру по имени `car`, которая будет хранить следующую информацию об автомобиле: наименование производителя как строку в символьном массиве или в объекте `string`, а также год его выпуска как целое число. Напишите программу, которая запросит пользователя, сколько автомобилей необходимо включить в каталог. Затем программа должна применить `new` для создания динамического массива структур `car` указанного пользователем размера. Далее она должна пригласить пользователя ввести наименование производителя и год выпуска для наполнения данными каждой структуры в массиве (см. главу 4). И, наконец, она должна отобразить содержимое каждой структуры. Пример запуска программы должен выглядеть примерно так:

```
Сколько автомобилей поместить в каталог? 2
Автомобиль #1:
Введите производителя: Hudson Hornet
Укажите год выпуска: 1952
Автомобиль #2:
Введите производителя: Kaiser
Укажите год выпуска: 1951
Вот ваша коллекция:
1952 Hudson Hornet
1951 Kaiser
```

7. Напишите программу, использующую массив `char` и цикл для чтения по одному слову за раз до тех пор, пока не будет введено слово `done`. Затем программа должна сообщить количество введенных слов (исключая `done`). Пример запуска должен быть таким:

```
Вводите слова (для завершения введите слово done):
anteater birthday category dumpster
envy finagle geometry готово for sure
Вы ввели 7 слов.
```

Вы должны включить заголовочный файл `cstring` и использовать функцию `strcmp()` для выполнения проверки.

8. Напишите программу, соответствующую описанию программы из упражнения 7, но применив объект класса `string` вместо символьного массива. Включите заголовочный файл `string` и используйте операции отношений для выполнения проверки.

9. Напишите программу, использующую вложенные циклы, которая запросит у пользователя значение количества строк для отображения. Затем она должна отобразить указанное число строк со звездочками, с одной звездочкой в первой строке, двумя — во второй и так далее. В каждой строке звездочкам должны предшествовать точки — в таком количестве, чтобы общее число символов в каждой строке было равно числу строк. Пример запуска программы должен выглядеть следующим образом:

```
Введите количество строк: 5
....*
...**
..***
.****
*****
```

ГЛАВА 6

Операторы ветвления и логические операции

В этой главе:

- Оператор `if`
- Оператор `if else`
- Логические операции: `&&`, `||` и `!`
- Библиотека символьных функций `cctype`
- Условная операция `?:`
- Оператор `switch`
- Операторы `continue` и `break`
- Циклы чтения чисел
- Базовый файловый ввод-вывод

Одним из ключей к проектированию интеллектуальных программ является предоставление им возможности принятия решений. В главе 5 был продемонстрирован один из видов принятия решений — циклы — когда программа решает, следует ли продолжать циклическое выполнение части кода. Здесь же мы исследуем, как C++ позволяет с помощью операторов ветвления принимать решения относительно выполнения одного из альтернативных действий. Какую технику защиты от вампиров (чеснок или крест) следует выбрать? Какой пункт меню выбрал пользователь? Ввел ли пользователь ноль? Для принятия подобных решений в C++ предусмотрены операторы `if` и `switch`, и именно они будут предметом рассмотрения настоящей главы. Здесь мы также поговорим об условной операции, которая представляет другой способ принятия решений, а также логических операциях, позволяющих комбинировать две проверки в одну. И, наконец, в этой главе мы также впервые ознакомимся с файловым вводом-выводом.

Оператор `if`

Когда программа C++ должна принять решение о том, какое из альтернативных действий следует выполнить, такой выбор обычно реализуется оператором `if`. Этот оператор имеет две формы: просто `if` и `if else`. Давайте сначала исследуем простой `if`. Он создан по образцу обычного английского языка, как в выражении "If you have a Captain Cookie card, you get a free cookie" (игра слов на основе созвучности фамилии Кук и слова "cookie" (печенье) — *прим. перев.*). Оператор `if` разрешает программе выполнять оператор или блок операторов при условии истинности проверочного условия, и пропускает этот оператор или блок, если проверочное условие

оценивается как ложное. Таким образом, `if` позволяет программе принимать решение относительно того, нужно ли выполнять некоторую часть кода.

Синтаксис оператора `if` подобен `while`:

```
if (проверочное-условие)
   оператор
```

Истинность выражения *проверочное-условие* заставляет программу выполнить *оператор*, который может быть единственным оператором или блоком операторов. Ложность выражения *проверочное-условие* заставляет программу пропустить *оператор* (рис. 6.1). Как и с проверочными условиями циклов, тип проверочного условия `if` приводится к `bool`, поэтому ноль трактуется как `false`, а все, что отличается от нуля — как `true`. Вся конструкция `if` рассматривается как единственный оператор.

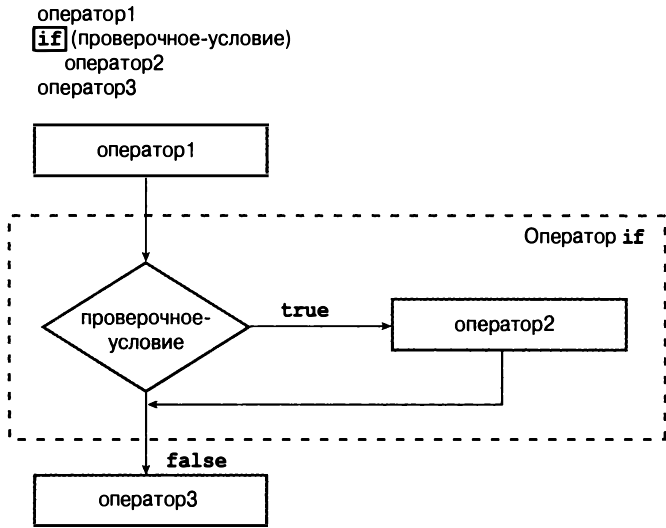


Рис. 6.1. Структура операторов `if`

Чаще всего *проверочное-условие* — выражение сравнения, вроде таких, которые управляют циклами. Например, предположим, что вы хотите запрограммировать подсчет пробелов во входной строке, а также общее число символов. Для чтения символов можно использовать оператор `cin.get(char)` внутри цикла `while`, а затем использовать оператор `if`, чтобы идентифицировать и посчитать пробельные символы. Листинг 6.1 реализует этот алгоритм, используя точку, как признак конца входного оператора.

Листинг 6.1. `if.cpp`

```
// if.cpp -- использование оператора if
#include <iostream>
int main()
{
    using namespace std;
    char ch;
    int spaces = 0;
```

```
    int total = 0;
    cin.get(ch);
    while (ch != '.')  // выйти по окончании оператора
    {
      if (ch == ' ')   // проверка ch на равенство пробелу
        ++spaces;
      ++total;         // выполняется на каждом шаге цикла
      cin.get(ch);
    }
    cout << spaces << " пробелов, " << total;
    cout << " — общее количество символов в предложении\n";
    return 0;
}
```

Ниже — пример вывода программы из листинга 6.1:

```
The balloonist was an airhead
with lofty goals.
6 пробелов, 46 - общее количество символов в предложении
```

Как следует из комментариев в листинге 6.1, оператор `++spaces;` выполняется только в том случае, если `ch` равен пробелу. Поскольку оператор `++total;` находится вне `if`, он выполняется на каждом шаге цикла. Обратите внимание, что в общее количество символов входит также символ перевода строки, который генерируется нажатием клавиши <Enter>.

Оператор if else

В то время как оператор `if` позволяет программе принять решение о том, должен ли выполняться определенный оператор или блок, `if else` позволяет решить, какой из двух операторов или блоков следует выполнить. Это незаменимое средство для программирования альтернативных действий. Оператор C++ `if else` моделирует простой английский язык, как в предложении "If you have a Captain Cookie card, you get a Cookie Plus Plus, else you just get a Cookie d'Ordinaire" (непереводимая игра слов с применением местных идиоматических выражений — *прим. перев.*). Оператор `if else` имеет следующую общую форму:

```
if (проверочное-условие)
  оператор1
else
  оператор2
```

Если *проверочное-условие* равно `true` или не ноль, то программа исполняет *оператор1* и пропускает *оператор2*. В противном случае, когда *проверочное-условие* равно `false` или ноль, программа выполняет *оператор2* и пропускает *оператор1*. Потому следующий фрагмент кода:

```
if (answer == 1492)
  cout << "Это правильно! \n";
else
  cout << "Вам следует перечитать еще раз первую главу.\n";
```

печатает первое сообщение, если `answer` равно `1492`, и второе — в противном случае. Каждый оператор может быть либо отдельным оператором, либо блоком опе-

раторов, заключенным в фигурные скобки. (См. рис. 6.2.) Вся конструкция if else трактуется синтаксически как единственный оператор.

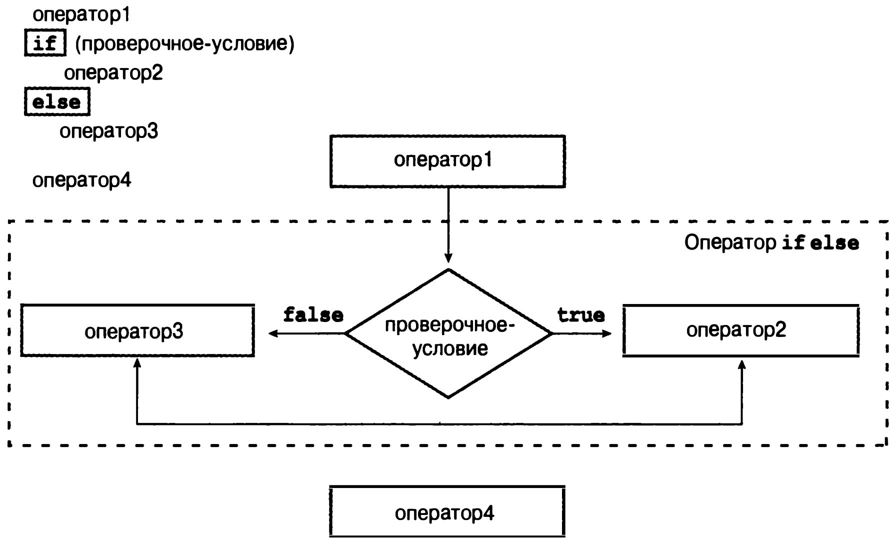


Рис. 6.2. Структура операторов **if else**

Например, предположим, что вы хотите преобразовать входящий текст, шифруя буквы и оставляя нетронутыми символы перевода строки. Это значит, что нужно заставить программу выполнять одно действие для символов перевода строки и другое — для всех прочих символов. Как показано в листинге 6.2, оператор if else позволяет легко решить эту задачу.

Листинг 6.2. ifelse.cpp

```
// ifelse.cpp -- использование оператора if else
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   cout << "Печатайте, а я буду повторять. \n";
   cin.get(ch);
   while (ch != '.')
   {
     if (ch == '\n')
        cout << ch;        // выполнять для символов перевода строки
     else
          cout << ++ch;    // выполнять в противном случае
     cin.get(ch);
   }
   // попробуйте ch + 1 вместо of ++ch , чтобы увидеть интересный эффект
   cout << "\nИзвините за легкий беспорядок.\n";
   return 0;
}
```

Ниже показан пример вывода программы из листинга 6.2:

```
Печатайте, а я буду повторять.
I am extraordinarily pleased
J!bn!fyusbpsejobsjmz!qmfbtfe
to use such a powerful computer.
up!vtf!tvdi!b!qpxfsgvm!dpnqvufs
Извините за легкий беспорядок.
```

Обратите внимание, что один из комментариев в листинге 6.2 предлагает заменить ++ch на ch+1, чтобы увидеть интересный эффект. Можете ли вы предположить, что получится? Если нет, сделайте это и посмотрите, что получится, после чего попробуйте объяснить. (Подсказка: это касается того, как cout обрабатывает разные типы аргументов.)

Форматирование операторов if else

Имейте в виду, что две альтернативы в операторе if else должны быть одиночными операторами. Если вам нужно более одного оператора в каждой логической ветви, вам следует использовать фигурные скобки, чтобы собрать их в единый блок. В отличие от других языков — таких, как BASIC и FORTRAN, C++ не воспринимает автоматически все, что находится между if и else, как один блок, поэтому вы должны использовать фигурные скобки для объединения операторов в блок. Следующий код, например, послужит причиной ошибки компиляции:

```
if (ch == 'Z')
   zorro++; // if заканчивается здесь
   cout << "Another Zorro candidate\n";
else // неверно
   dull++;
   cout << "Not a Zorro candidate\n";
```

Компилятор рассматривает это как простой оператор if, который заканчивается на zorro++;. Затем идет оператор cout. До этого места все хорошо. Но далее идет то, что воспринимается компилятором как бесхозный else, а потому он считает это синтаксической ошибкой.

Чтобы код делал то, что нужно, следует использовать фигурные скобки:

```
if (ch == 'Z')
{ // блок, исполняемый, если условие истинно
   zorro++;
   cout << "Another Zorro candidate\n";
}
else
{ // блок, исполняемый, если условие ложно
   dull++;
   cout << "Not a Zorro candidate\n";
}
```

Поскольку C++ — язык свободной формы, вы можете размещать фигурные скобки как вам угодно, до тех пор, пока они ограничивают операторы языка. В предыдущем примере демонстрируется один популярный формат. А вот и другой формат:

```
if (ch == 'Z') {
   zorro++;
   cout << "Another Zorro candidate\n";
   }
else {
   dull++;
   cout << "Not a Zorro candidate\n";
   }
```

Первая форма подчеркивает блочную структуру операторов, в то время как вторая более тесно связывает блоки с ключевыми словами `if` и `else`. Любой стиль ясен и согласован, а потому будет служить вам хорошо; однако вы можете столкнуться с руководителем или работодателем, который имеет свой строгий и специфический взгляд на эту тему.

Конструкция if else if else

В компьютерных программах, как и в жизни, иногда приходится выбирать из более чем двух вариантов. Оператор C++ `if else` можно расширить, чтобы он отвечало таким потребностям. Как уже говорилось, за `else` должен следовать единственный оператор, который может быть и блоком операторов. Поскольку конструкция `if else` сама является единым оператором, она может следовать за `else`:

```
if (ch == 'A')
   a_grade++;     // альтернатива # 1
else
   if (ch == 'B') // альтернатива # 2
      b_grade++;  // подальтернатива # 2a
   else
      soso++;     // подальтернатива # 2b
```

Если ch не равен `'A'`, программа переходит к `else`. Там второй оператор `if else` разделяет эту альтернативу еще на два варианта. Свойство свободного форматирования C++ позволяет расположить эти элементы в более читабельном виде:

```
if (ch == 'A')
   a_grade++;     // альтернатива # 1
else if (ch == 'B')
   b_grade++;     // альтернатива # 2
else
   soso++;        // альтернатива # 3
```

Это выглядит как совершенно новая управляющая структура — `if else if else`. Но на самом деле это один оператор `if else`, вложенный в другой. Пересмотренный формат выглядит намного яснее и позволяет даже при поверхностном взгляде увидеть все альтернативы. Вся эта конструкция по-прежнему трактуется как единственный оператор.

В листинге 6.3 это форматирование используется для построения простой программы загадок и отгадок.

Листинг 6.3. `ifelseif.cpp`

```
// ifelseif.cpp -- использование оператора if else if else
#include <iostream>
const int Fave = 27;
int main()
{
    using namespace std;
    int n;
    cout << "Попробуйте угадать мое любимое число из диапазона 1-100: ";
    do
    {
        cin >> n;
        if (n < Fave)
            cout << "Слишком маленькое — попробуйте еще: ";
        else if (n > Fave)
            cout << "Слишком большое — попробуйте еще: ";
        else
            cout << Fave << " - правильно!\n";
    } while (n != Fave);
    return 0;
}
```

Ниже представлен пример вывода программы из листинга 6.3:

```
Попробуйте угадать мое любимое число из диапазона 1-100: 50
Слишком большое — попробуйте еще: 25
Слишком маленькое — попробуйте еще: 37
Слишком большое — попробуйте еще: 31
Слишком большое — попробуйте еще: 28
Слишком большое — попробуйте еще: 27
27 - правильно!
```

Пример из практики: условные операции и предотвращение ошибок

Многие программисты превращают более интуитивно понятное выражение `variable == value` в `value == variable`, дабы предотвратить ошибки, связанные с опечатками, когда вместо операции проверки равенства (==) вводится операция присваивания (=). Например, такое условие, как

```
If (3 == myNumber)
```

корректно и будет работать правильно. Однако если случайно ввести

```
If (3 = myNumber)
```

то компилятор выдаст сообщение об ошибке, ибо расценит это как попытку присвоить значение переменной литералу (3 всегда равно 3, и ему нельзя присвоить ничего другого). Предположим, вы допустили ту же опечатку, используя следующий формат:

```
if (myNumber = 3)
```

В этом случае компилятор просто присвоит значение 3 переменной `myNumber`, и блок внутри `if` будет выполнен — очень распространенная ошибка, которую трудно обнаружить. В качестве общего правила примите следующее: написать код, позволяющий компилятору обнаружить ошибку, гораздо легче, чем разбираться с непонятными мистическими результатами неверного выполнения программ.

Логические выражения

Часто приходится проверять более одного условия. Например, чтобы символ относился к прописным буквам, его значение должно быть больше или равно 'a' и меньше или равно 'z'. Либо же, если вы просите пользователя ответить y или n, то наряду с прописными должны принимать и заглавные буквы (Y или N). Чтобы обеспечить такие возможности, C++ предлагает три логических операции, с помощью которых можно комбинировать или модифицировать существующие выражения: логическое ИЛИ (которое записывается как ||), логическое И (записывается как &&) и логическое НЕ (записывается как !). Рассмотрим каждую из них.

Логическая операция ИЛИ: ||

На английском языке слово "or" (или) означает, что одно из двух условий либо оба сразу удовлетворяют некоторому требованию. Например, вы можете поехать на пикник, устроенный компанией MegaMicro, если вы *или* ваш(а) супруг(а) работаете в этой компании. Эквивалент логической операции ИЛИ на C++ записывается как ||. Эта операция комбинирует два выражения в одно. Если одно или оба исходных выражения возвращают true или не ноль, то результирующее выражение имеет значение true (истина). В противном случае выражение имеет значение false. Ниже показаны некоторые примеры:

```
5 ==5 || 5 == 9     // истинно, потому что первое выражение истинно
5 > 3 || 5 > 10     // истинно, потому что первое выражение истинно
5 > 8 || 5 < 10     // истинно, потому что второе выражение истинно
5 < 8 || 5 > 2      // истинно, потому что оба выражения истинны
5 > 8 || 5 < 2      // ложно, потому что оба выражения ложны
```

Поскольку || имеет более низкий приоритет, чем операции сравнения, нет необходимости использовать в этих выражениях скобки. В табл. 6.1 показано, как работает операция ||.

Таблица 6.1. Операция ||

	Значение expr1 || expr2	
	expr1 == true	**expr1 == false**
expr2 == true	true	true
expr2 == false	true	false

В C++ предполагается, что операция || является *последовательной точкой*. То есть, любое изменение, проведенное в левой части, вычисляется прежде, чем вычисляется правая часть. Например, рассмотрим следующее выражение:

```
i++ < 6 || i == j
```

Предположим, изначально i имело значение 10. К моменту сравнения с j переменная i получает значение 11. Таким образом, C++ не заботится о правой части выражения, если выражение слева истинно, потому что одного истинного выражения достаточно, чтобы все составное выражение было оценено как истинное (напоминаем, что точка с запятой и запятая — также последовательные точки).

Листинг 6.4 использует операцию || в операторе if для того, чтобы проверить заглавную и прописную версии символа. К тому же применяется средство конкатенации строк C++ (см. главу 4) для разбиения одной строки на три строки в коде.

Листинг 6.4. or.cpp

```
// or.cpp -- использование логической операции ИЛИ
#include <iostream>
int main()
{
  using namespace std;
  cout << "Эта программа может переформатировать ваш жесткий диск\n"
         "и уничтожить все данные.\n"
         "Хотите продолжить? <y/n> ";
  char ch;
  cin >> ch;
  if (ch == 'y' || ch == 'Y') // y или Y
    cout << "Вас предупредили! \a\a\n";
  else if (ch == 'n' || ch == 'N') // n или N
    cout << "Мудрый выбор... прощайте\n";
  else
    cout << "Это не было ни y, ни an n, потому я предполагаю, "
            "что нужно очистить диск в любом случае.\a\a\a\n";
  return 0;
}
```

Вот как выглядит пример выполнения программы из листинга 6.4:

```
Эта программа может переформатировать ваш жесткий диск
и уничтожить все данные.
Хотите продолжить? <y/n> N
Мудрый выбор... прощайте
```

Эта программа читает только один символ, поэтому принимается во внимание только первый символ ответа. Это значит, что пользователь может ввести NO! вместо N. Программа прочтет только N. Но если ей пришлось бы читать еще входные символы, то первым оказался бы символ O.

Логическая операция И: &&

Логическая операция И, которая записывается как &&, также комбинирует два выражения в одно. Результирующее выражение имеет значение true только в том случае, когда оба исходных выражения также равны true. Вот некоторые примеры:

```
5 == 5 && 4 == 4  // истинно, потому что оба выражения истинны
5 == 3 && 4 == 4  // ложно, потому что первое выражение ложно
5 > 3 && 5 > 10   // ложно, потому что второе выражение ложно
5 > 8 && 5 < 10   // ложно, потому что первое выражение ложно
5 < 8 && 5 > 2    // истинно, потому что оба выражения истинны
5 > 8 && 5 < 2    // ложно, потому что оба выражения ложны
```

Поскольку && имеет меньший приоритет, чем операции сравнения, нет необходимости использовать скобки.

Подобно операции ||, && действует как последовательная точка, а потому возможны любые побочные эффекты перед тем, как будет вычислено правое выражение. Если левое выражение ложно, то и все составное выражение также ложно, поэтому в этом случае C++ можно не беспокоиться о вычислении правой части. В табл. 6.2 демонстрируется работа операции &&.

Таблица 6.2. Операция &&

	Значение expr1 && expr2	
	expr1 == true	**expr1 == false**
expr2 == true	true	false
expr2 == false	false	false

Листинг 6.5 демонстрирует использование && в стандартной ситуации, когда нужно прервать цикл while по двум разным причинам. В этом листинге цикл while читает значения в массив. Одно условие (i < ArSize) прерывает цикл, когда массив полон. Вторая (temp >= 0) предоставляет пользователю возможность прервать цикл раньше, введя отрицательное значение. Программа использует операцию &&, чтобы скомбинировать эти две проверки в одно условие. Программа также использует два оператора if, один оператор if else и цикл for, то есть демонстрирует несколько тем из этой главы и главы 5.

Листинг 6.5. and.cpp

```
// and.cpp -- использование логической операции И
#include <iostream>
const int ArSize = 6;
int main()
{
   using namespace std;
   float naaq[ArSize];
   cout << "Enter the NAAQs (New Age Awareness Quotients) "
        << "of\nyour neighbors. Program terminates "
        << "when you make\n" << ArSize << " entries "
        << "or enter a negative value.\n";
   int i = 0;
   float temp;
   cout << "First value: ";
   cin >> temp;
   while (i < ArSize && temp >= 0) // 2 критерия прерывания
   {
     naaq[i] = temp;
     ++i;
     if (i < ArSize)              // в массиве еще есть место,
     {
        cout << "Next value: ";
        cin >> temp;              // поэтому получаем новое значение
     }
   }
   if (i == 0)
     cout << "No data--bye\n";
```

```
    else
    {
      cout << "Enter your NAAQ: ";
      float you;
      cin >> you;
      int count = 0;
      for (int j = 0; j < i; j++)
         if (naaq[j] > you)
           ++count;
      cout << count;
      cout << " of your neighbors have greater awareness of\n"
           << "the New Age than you do.\n";
    }
    return 0;
}
```

Следует отметить, что программа в листинге 6.5 помещает вывод во временную переменную `temp`. И только после того, как введенное значение будет проверено, и станет ясно, что введено корректное значение, программа помещает новое значение в массив.

Ниже показаны два примера выполнения программы. В первом примере программа прерывается после шести введенных значений:

```
Enter the NAAQs (New Age Awareness Quotients) of
your neighbors. Program terminates when you make
6 entries or enter a negative value.
First value: 28
Next value: 72
Next value: 15
Next value: 6
Next value: 130
Next value: 145
Enter your NAAQ: 50
3 of your neighbors have greater awareness of
the New Age than you do.
```

Второй раз программа прерывается после ввода отрицательного значения:

```
Enter the NAAQs (New Age Awareness Quotients) of
your neighbors. Program terminates when you make
6 entries or enter a negative value.
First value: 123
Next value: 119
Next value: 4
Next value: 89
Next value: -1
Enter your NAAQ: 123.031
0 of your neighbors have greater awareness of
the New Age than you do.
```

Замечания по программе

Ниже представлена часть программы из листинга 6.5, отвечающая за ввод:

```
   cin >> temp;
   while (i < ArSize && temp >= 0) // 2 критерия прерывания
   {
      naaq[i] = temp;
      ++i;
      if (i < ArSize) // в массиве еще есть место,
      {
         cout << "Next value: ";
         cin >> temp; // поэтому получаем новое значение
      }
```

Программа начинается с чтения первого входного значения во временную переменную по имени `temp`. Затем проверочное условие `while` проверяет, есть ли еще место в массиве (`i < ArSize`), а также, не является ли введенное значение отрицательным (`temp >= 0`). Если это так, программа копирует значение `temp` в массив и увеличивает индекс массива на единицу. В этот момент, поскольку нумерация массива начинается с нуля, `i` равно общему числу введенных значений. То есть, если `i` начинается с нуля, то первый проход цикла присваивает значение `naaq[0]` и устанавливает `i` равным `1`.

Цикл прерывается, когда массив будет заполнен, либо когда пользователь введет отрицательное значение. Обратите внимание, что цикл читает новое значение в `temp`, только если `i` меньше, чем `ArSize` — то есть, только если в массиве еще есть место.

После получения данных программа использует оператор `if else` для проверки того, вводились ли вообще данные (то есть, если первым же введенным элементом было отрицательное число), и обрабатывает данные, если они есть.

Установка диапазонов с помощью &&

Операция `&&` также позволяет установить серию операторов `if else if else`, где каждый выбор соответствует определенному диапазону значений. В листинге 6.6 иллюстрируется такой подход. В нем также демонстрируется полезная техника обработки серии сообщений. Точно так же, как переменная-указатель на `char` может идентифицировать целую строку, указывая на ее начало, массив указателей на `char` может идентифицировать серию строк. Вы просто присваиваете адрес каждой строки различным элементам массива. Код в листинге 6.6 использует массив `qualify` для хранения адресов четырех строк. Например, `qualify[1]` содержит адрес строки `"mud tug-of-war\n"`. Программа затем может использовать `qualify[1]` как любой другой указатель на строку — например, с `cout` либо `strlen()` или `strcmp()`. Применение квалификатора `const` защищает эти строки от непреднамеренных изменений.

Листинг 6.6. `more_and.cpp`

```
// more_and.cpp -- использование логической операции И
#include <iostream>
const char * qualify[4] = // массив указателей
{                         // на строки
   "гонки на 10,000 метров.\n",
   "перетягивания каната.\n",
   "состязания мастеров каноэ.\n",
   "фестиваля по бросанию пирожков.\n"
};
```

```
int main()
{
  using namespace std;
  int age;
  cout << "Введите свой возраст в годах: ";
  cin >> age;
  int index;
  if (age > 17 && age < 35)
    index = 0;
  else if (age >= 35 && age < 50)
    index = 1;
  else if (age >= 50 && age < 65)
    index = 2;
  else
    index = 3;
  cout << "Вы квалифицированы для " << qualify[index];
  return 0;
}
```

Замечание по совместимости

Вы можете вспомнить, что некоторые реализации C++ требуют использования в объявлении массива ключевого слова `static`, чтобы можно было инициализировать массив. Это ограничение, как говорится в главе 9, относится к массивам, объявленным внутри тела функции. Когда массив объявляется вне тела функции, как это имеет место с `quality` в листинге 6.6, он называется ***внешним массивом*** и может быть инициализирован даже в реализациях, предшествующих появлению стандарта ANSI.

Вот пример выполнения программы из листинга 6.6:

```
Введите свой возраст в годах: 87
Вы квалифицированы для фестиваля по бросанию пирожков.
```

Введенный возраст не соответствует ни одному из проверяемых диапазонов, поэтому программа присваивает `index` значение `3` и затем печатает соответствующую строку.

Замечания по программе

В листинге 6.6 выражение `age > 17 && age < 35` проверяет возраст на предмет попадания в диапазон между двумя значениями — то есть он должен быть от 18 до 34 лет включительно. Выражение `age >= 35 && age < 50` использует операцию <= для включения в диапазон 35, то есть представляет диапазон возрастов от 35 до 49 включительно. Если бы программа использовала выражение `age > 35 && age < 50`, то значение 35 было бы потеряно для всех проверок. Применяя проверки диапазонов, вы должны проверять, чтобы диапазоны не имели "прорех" между собой, а также не перекрывались. К тому же следует убедиться в том, что диапазоны заданы корректно; см. врезку "Проверка диапазонов" ниже в настоящем разделе.

Оператор `if else` служит для выбора индекса массива, который, в свою очередь, идентифицирует определенную строку.

Проверка диапазонов

Обратите внимание, что каждая часть проверки диапазонов должна использовать операцию И для объединения двух полных сравнительных выражений:

```
if (age > 17 && age < 35) // Нормально
```

не заимствуйте из математики и не применяйте следующую нотацию:

```
if (17 < age < 35) // Не делайте так!
```

Если вы допустите ошибку подобного рода, компилятор не сможет ее обнаружить, потому что это корректный синтаксис C++. Операция ассоциируется слева направо, поэтому последнее выражение эквивалентно такому:

```
if ( (17 < age) < 35)
```

Но `17 < age` может быть либо истинно (или `1`), либо ложно (или `0`). В любом случае выражение (`17<age`) всегда будет меньше 35, а потому такое выражение всегда истинно!

Логическая операция НЕ: !

Операция `!` отрицает, или обращает, истинность выражения, следующего за ней. То есть, если `expression` равно `true`, то `!expression` равно `false`, и наоборот. Точнее говоря, если `expression` есть `true`, либо не ноль, то `!expression` есть `false`.

Обычно выражение отношения можно представить яснее без применения операции `!`:

```
if (!(x > 5))     // if (x <= 5) яснее
```

Однако операция `!` может быть полезна с функциями, которые возвращают значения `true/false`, либо значения, которые могут интерпретироваться подобным образом. Например, `strcmp(s1, s2)` возвращает не ноль (`true`), если две строки в стиле C, `s1` и `s2`, отличаются друг от друга, и ноль, если они одинаковы. Это значит, что `!strcmp(s1, s2)` равно `true`, если две строки эквивалентны.

Код в листинге 6.7 использует технику применения операции `!` к значению, возвращаемому функцией для проверки экранного ввода на предмет возможности присваивания типу `int`. Пользовательская функция `is_int()`, о которой мы поговорим позже, возвращает `true`, если ее аргумент находится в диапазоне допустимых значений для присваивания типу `int`. Затем программа применяет проверку условия `while(!is_int(num))`, чтобы отклонить значения, которые не входят в диапазон.

Листинг 6.7. `not.cpp`

```
// not.cpp -- использование логической операции НЕ
#include <iostream>
#include <climits>
bool is_int(double);
int main()
{
   using namespace std;
   double num;
   cout << "Yo, dude! Enter an integer value: ";
   cin >> num;
   while (!is_int(num)) // продолжать, пока num не выражает int
   {
```

```
        cout << "Out of range -- please try again: ";
        cin >> num;
    }
    int val = int (num); // приведение типа
    cout << "You've entered the integer " << val << "\nBye\n";
    return 0;
}
bool is_int(double x)
{
    if (x <= INT_MAX && x >= INT_MIN) // проверка предельных значений climits
        return true;
    else
        return false;
}
```

Замечание по совместимости

Если в вашей системе нет `climits`, используйте `limits.h`.

Ниже показан пример выполнения программы из листинга 6.7 в системе с 32-битовым типом `int`:

```
Yo, dude! Enter an integer value: 6234128679
Out of range -- please try again: -8000222333
Out of range -- please try again: 99999
You've entered the integer 99999
Bye
```

Замечания по программе

Если вы вводите слишком большое значение в программу, читающую тип `int`, многие реализации C++ просто усекают значение, не сообщая о потере данных. Программа в листинге 6.7 избегает этого за счет того, что читает потенциальный `int` как `double`. Тип `double` имеет более чем достаточную точность для того, чтобы сохранить обычное значение `int`, а его диапазон допустимых значений намного больше.

Булевская функция `is_int()` использует две символические константы (`INT_MAX` и `INT_MIN`), определенные в файле `climits` (обсуждается в главе 3), для того, чтобы убедиться, что значение ее аргумента находится в допустимых пределах. Если так, программа возвращает `true`; в противном случае — `false`.

Главная функция программы `main()` использует условие цикла для того, чтобы отклонить неправильный ввод пользователя. Можно сделать программу более дружественной за счет отображения допустимых границ `int`, когда введено неправильное значение. После того, как введенное значение проверено, программа присваивает его переменной типа `int`.

Факты о логических операциях

Как уже упоминалось в этой главе, логические операции C++ ИЛИ и И обладают более низким приоритетом, чем операции сравнения. Это значит, что такое выражение, как

```
x > 5 && x < 10
```

читается следующим образом:

```
(x > 5) && (x < 10)
```

С другой стороны, операция НЕ (`!`) имеет более высокий приоритет, чем любые арифметические операции и операции сравнения. Таким образом, для отрицания выражения его необходимо заключить в скобки:

```
!(x > 5)    // x не больше, чем 5
!x > 5      // не x больше, чем 5
```

Кстати, второе выражение из этих двух всегда даст `false`, потому что `!x` принимает значения `true` или `false`, что преобразуется, соответственно, в `1` и `0`.

Логическая операция И имеет более высокий приоритет, чем логическая операция ИЛИ. Поэтому следующее выражение:

```
age > 30 && age < 45 || weight > 300
```

означает вот что:

```
(age > 30 && age < 45) || weight > 300
```

То есть здесь первое условие говорит о том, что возраст должен быть от 31 до 44 включительно, а второе — что вес должен быть больше 300 (фунтов). Все выражение истинно, если истинно одно из выражений, либо оба сразу.

Конечно, вы можете использовать скобки, чтобы явно указать программе, как следует интерпретировать выражение. Например, предположим, что вы хотите использовать `&&` для того, чтобы скомбинировать условия о том, что возраст (`age`) должен быть больше 50 или вес (`weight`) — больше 300 с условием, что размер пожертвования (`donation`) должен быть больше 1000. Для этого нужно часть ИЛИ поместить в скобки:

```
(age > 50 || weight > 300) && donation > 1000
```

Иначе компилятор скомбинирует условие `weight` с условие `donation`, вместо того чтобы скомбинировать его с условием `age`.

Хотя правила приоритетов C++ часто позволяют писать составные выражения без использования скобок, все же проще всего всегда использовать скобки для группирования проверок, независимо от того, нужны они или нет. Это повышает читабельность кода, исключает неправильное понимание порядка приоритетов и снижает вероятность ошибки из-за того, что вы не помните точно правил приоритетов.

C++ гарантирует, что когда программа вычисляет логическое выражение, то делает это слева направо, и прекращает вычисление, как только становится ясен ответ. Предположим, например, что у вас есть такое выражение:

```
x != 0 && 1.0 / x > 100.0
```

Если первое условие дает `false`, то и все выражение будет ложно. Это потому, что для того, чтобы составное выражение вернуло `true`, нужно, чтобы обе его части были истинны. Зная, что первое выражение дает `false`, программе незачем вычислять второе. И это — удача для данного примера, поскольку вычисление второго выражения может привести к делению на ноль, что не входит в список допустимых действий компьютера.

Альтернативные представления

Не все клавиатуры предоставляют возможность ввода необходимых символов логических операций, поэтому стандарт C++ предусматривает альтернативные их представления, показанные в табл. 6.3. Идентификаторы `and`, `or` и `not` — зарезервированные слова C++, а это значит, что их нельзя использовать в качестве имен переменных и тому подобного. Они не рассматриваются как ключевые слова, потому что являются альтернативными представлениями существующих средств языка. Кстати, они не являются зарезервированными словами в С, но программы на С могут использовать их в качестве операций, если они включают заголовочный файл `iso646.h`. C++ не нуждается в применении заголовочного файла.

Таблица 6.3. Логические операции: альтернативные представления

Операция	Альтернативное представление
`&&`	`and`
`||`	`or`
`!`	`not`

Библиотека символьных функций cctype

Язык C++ унаследовал от С удобный пакет функций, работающих с символами, чьи прототипы находятся в заголовочном файле `cctype` (в старом стиле — `ctype.h`), и это упрощает такие задачи, как определение того, является ли символ символом верхнего регистра, десятичной цифрой или знаком препинания. Например, функция `isaplha(ch)` возвращает ненулевое значение, если `ch` — буква, и ноль — в противоположном случае. Аналогично, `ispunct(ch)` возвращает значение `true`, только если `ch` — знак препинания, такой как запятая или точка. (Эти функции возвращают значение типа `int`, а не `bool`, но `int` неявно конвертируется в `bool`.)

Использовать эти функции намного удобнее, чем операции И и ИЛИ. Например, вот как пришлось бы использовать И и ИЛИ, чтобы убедиться, что `ch` — буквенный символ:

```
if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
```

Сравните с применением `isalpha()`:

```
if (isalpha(ch))
```

Однако преимущество заключается не только в том, что `isalpha()` легче использовать. Форма И/ИЛИ предполагает, что коды символов, находящихся в пределах A-Z, составляют последовательность, в которую не входят никакие небуквенные символы. Это предположение верно для кодировки ASCII, но не всегда верно в общем случае.

В листинге 6.8 демонстрируются некоторые функции из семейства `cctype`. В частности, в нем используется `isalpha()`, проверяющая буквенные символы, `isdigits()`, проверяющая символы десятичных цифр, таких как 3, `isspace()`, проверяющая пробельные символы, такие как пробелы, символы перевода строк и табуляции, и `ispunct()`, проверяющая знаки пунктуации. Программа также предлагает повторный обзор структуры `if else if` с применением цикла `while` с `cin.get(char)`.

Листинг 6.8. cctypes.cpp

```
// cctypes.cpp -- использование библиотеки ctype.h
#include <iostream>
#include <cctype> // прототипы символьных функций
int main()
{
    using namespace std;
    cout << "Введите текст для анализа; для окончания ввода введите @.\n";
    char ch;
    int whitespace = 0;
    int digits = 0;
    int chars = 0;
    int punct = 0;
    int others = 0;
    cin.get(ch);                // получить первый символ
    while(ch != '@')            // проверить признак окончания ввода
    {
        if(isalpha(ch))         // буквенный символ?
            chars++;
        else if(isspace(ch))    // пробельный символ?
            whitespace++;
        else if(isdigit(ch))    // десятичная цифра?
            digits++;
        else if(ispunct(ch))    // знак пунктуации?
            punct++;
        else
            others++;
        cin.get(ch);            // получить следующий символ
    }
    cout << chars << " букв, "
        << whitespace << " пробелов, "
        << digits << " цифр, "
        << punct << " знаков препинания, "
        << others << " прочих.\n";
    return 0;
}
```

Ниже показан пример выполнения программы из листинга 6.8 (обратите внимание, что переводы строк относятся к пробельным символам):

```
Введите текст для анализа; для окончания ввода введите @.
Jody "Java-Java" Joystone, noted restaurant critic,
celebrated her 39th birthday with a carafe of 1982
Chateau Panda.@
89 букв, 16 пробелов, 6 цифр, 6 знаков препинания, 0 прочих.
```

В табл. 6.4 перечислены доступные функции из пакета cctype. В некоторых системах не все эти функции присутствуют, к тому же есть и дополнительные функции.

Таблица 6.4. Символьные функции cctype

Имя функции	Возвращаемое значение
isalnum()	Функция возвращает true, если аргумент — буква или десятичная цифра.
isalpha()	Функция возвращает true, если аргумент — буква.
isblank()	Функция возвращает true, если аргумент — пробел или знак горизонтальной табуляции.
iscntrl()	Функция возвращает true, если аргумент — управляющий символ.
isdigit()	Функция возвращает true, если аргумент — десятичная цифра (0–9).
isgraph()	Функция возвращает true, если аргумент — любой печатаемый символ, отличный от пробела.
islower()	Функция возвращает true, если аргумент — символ в нижнем регистре.
isprint()	Функция возвращает true, если аргумент — любой печатаемый символ, включая пробел.
ispunct()	Функция возвращает true, если аргумент — знак препинания.
isspace()	Функция возвращает true, если аргумент — пробельный символ (пробел, прогон страницы, перевод строки, возврат каретки, горизонтальная табуляция, вертикальная табуляция).
isupper()	Функция возвращает true, если аргумент — символ в верхнем регистре.
isxdigit()	Функция возвращает true, если аргумент — шестнадцатеричная цифра (то есть 0–9, a–f или A–F).
tolower()	Если аргумент — символ верхнего регистра, возвращает его вариант в нижнем регистре, иначе возвращает аргумент без изменений.
toupper()	Если аргумент — символ нижнего регистра, возвращает его вариант в верхнем регистре, иначе возвращает аргумент без изменений.

Операция ?:

Язык C++ включает операцию, которая часто может использоваться вместо оператора if else. Она называется *условной операцией* и записывается как ?:, и, к вашему сведению, является единственной операцией C++, которая требует трех операндов. Ее общая форма выглядит следующим образом:

```
выражение1 ? выражение2 : выражение3
```

Если *выражение1* истинно, то значением всего условного выражения будет значение *выражение2*. В противном случае значением всего выражения будет *выражение3*. Вот два примера, показывающие, как это работает:

```
5 > 3 ? 10 : 12   // 5 > 3 истинно, поэтому значение всего выражения - 10
3 == 9? 25 : 18   // 3 == 9 ложно, поэтому значение всего выражения — 18
```

Первый пример можно перефразировать так: если 5 больше, чем 3, то выражение оценивается как 10; иначе оно оценивается как 12. В реальных ситуациях программирования, конечно, выражения могут включать в себя переменные.

Код в листинге 6.9 использует условную операцию для нахождения большего из двух значений.

Листинг 6.9. condit.cpp

```
// condit.cpp -- использование условной операции
#include <iostream>
int main()
{
    using namespace std;
    int a, b;
    cout << "Введите два числа: ";
    cin >> a >> b;
    cout << "Большее из " << a << " и " << b;
    int c = a > b ? a : b; // c = a если a > b, иначе c = b
    cout << " - " << c << endl;
    return 0;
}
```

Вот пример выполнения программы из листинга 6.9:

```
Введите два числа: 25 28
Большее из 25 и 28 - 28
```

Ключевой частью программы является следующий оператор:

```
int c = a > b ? a : b;
```

Он выдает тот же результат, что и приведенные ниже операторы:

```
int c;
if (a > b)
   c = a;
else
   c = b;
```

По сравнению с последовательностью `if else` условная операция более короткая, но, во-первых, она не столь очевидна. Одно отличие между этими двумя подходами заключается в том, что условная операция порождает выражение, а потому — единственное значение, которое может быть присвоено или встроено в более крупное выражение, как это сделано в программе из листинга 6.9, где значение условного выражения присваивается переменной `c`. Краткая форма и необычный синтаксис условной операции ценится некоторыми программистами. Их любимый трюк, достойный порицания, состоит в использовании вложенных друг в друга условных выражений, как показано в следующем несложном примере:

```
const char x[2] [20] = {"Jason ","at your service\n"};
const char * y = "Quillstone ";
for (int i = 0; i < 3; i++)
   cout << ((i < 2)? !i ? x [i] : y : x[1]);
```

Это лишь нечеткий (в высшей степени нечеткий!) способ напечатать три строки в следующем порядке:

```
Jason Quillstone at your service
```

В целях читабельности условная операция больше подходит для простых отношений и простых значений выражений вроде:

```
x = (x > y) ? x : y;
```

Если же код становится более сложным, то, возможно, более ясно его можно выразить с помощью оператора `if else`.

Оператор switch

Предположим, что вы создаете экранное меню, которое предлагает пользователю на выбор один из четырех возможных вариантов, например, "Дешевый", "Умеренный", "Дорогой", "Экстравагантный" и "Непомерный". Вы можете расширить последовательность `if else if else` для обработки этих пяти альтернатив, но оператор C++ `switch` позволяет легче обработать выбор их большого списка. Ниже представлена общая форма оператора `switch`:

```
switch (целочисленное-выражение)
{
   case метка1 : оператор(ы)
   case метка2 : оператор(ы)
   ...
   default : оператор(ы)
}
```

Оператор C++ `switch` действует как маршрутизатор, который сообщает компьютеру, какую строку кода исполнять следующей. По достижении оператора `switch` программа перепрыгивает к строке, помеченной значением, соответствующим текущему значению *целочисленное-выражение*. Например, если *целочисленное-выражение* имеет значение 4, то программа переходит к строке с меткой `case 4:`. Как следует из наименования, выражение *целочисленное-выражение* должно быть целочисленным. Также каждая метка должна быть целым константным выражением. Чаще всего метки бывают константами типа `char` или `int`, такими как `1` или `'q'`, либо же перечислителями. Если *целочисленное-выражение* не соответствует ни одной метке, программа переходит к метке `default`. Метка `default` не обязательна. Если она пропущена и соответствия не найдено, программа переходит к оператору, следующему за `switch`. (См. рис. 6.3.)

Оператор `switch` языка C++ отличается от подобных операторов в других языках, например, Pascal, в одном очень важном отношении. Каждая метка C++ `case` работает только как метка строки, а не граница между выборами. То есть, после того, как программа перейдет на определенную строку в `switch`, она последовательно выполнит все операторы, следующие за этой строкой внутри `switch`, если только вы явно не направите ее в другое место. Выполнение *не* останавливается автоматически на следующем `case`. Чтобы прекратить выполнение в конце определенной группы операторов, вы должны использовать оператор `break`. Это передаст управление за пределы блока `switch`.

Листинг 6.10 демонстрирует, как можно использовать вместе `switch` и `break` для реализации простого меню. Для отображения возможных вариантов выбора в программе применяется функция `showmenu()`. Затем оператор `switch` выбирает действие на основе выбора пользователя.

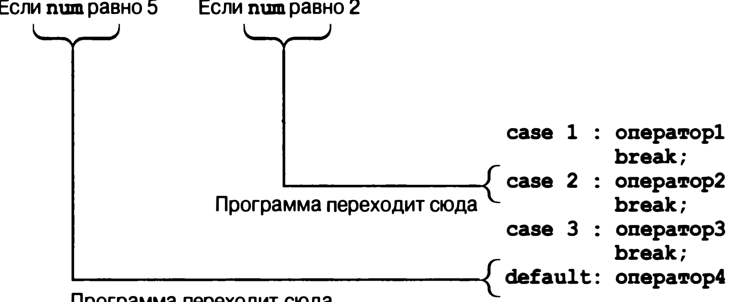


Рис. 6.3. Структура операторов **switch**

Замечание по совместимости

Некоторые реализации C++ трактуют управляющую последовательность `\a` (использованную в `case 1` листинга 6.10) как не генерирующую звуковой сигнал.

Листинг 6.10. switch.cpp

```
// switch.cpp -- использование оператора switch
#include <iostream>
using namespace std;
void showmenu(); // прототипы функций
void report();
void comfort();
int main()
{
   showmenu();
   int choice;
   cin >> choice;
   while (choice != 5)
   {
        switch(choice)
        {
          case 1 : cout << "\a\n";
                   break;
          case 2 : report();
                   break;
          case 3 : cout << "The boss was in all day.\n";
                   break;
          case 4 : comfort();
                   break;
          default : cout << "That's not a choice.\n";
        }
        showmenu();
        cin >> choice;
   }
   cout << "Bye!\n";
   return 0;
}
```

```
void showmenu()
{
   cout << "Please enter 1, 2, 3, 4, or 5:\n"
           "1) alarm 2) report\n"
           "3) alibi 4) comfort\n"
           "5) quit\n";
}
void report()
{
   cout << "It's been an excellent week for business.\n"
        "Sales are up 120%. Expenses are down 35%.\n";
}
void comfort()
{
    cout << "Your employees think you are the finest CEO\n"
            "in the industry. The board of directors think\n"
            "you are the finest CEO in the industry.\n";
}
```

Ниже показан пример выполнения программы из листинга 6.10:

```
Please enter 1, 2, 3, 4, or 5:
1) alarm 2) report
3) alibi 4) comfort
5) quit
4
Your employees think you are the finest CEO
in the industry. The board of directors think
you are the finest CEO in the industry.
Please enter 1, 2, 3, 4, or 5:
1) alarm   2) report
3) alibi   4) comfort
5) quit
2
It's been an excellent week for business.
Sales are up 120%. Expenses are down 35%.
Please enter 1, 2, 3, 4, or 5:
1) alarm 2) report
3) alibi 4) comfort
5) quit
6
That's not a choice.
Please enter 1, 2, 3, 4, or 5:
1) alarm 2) report
3) alibi 4) comfort
5) quit
5
Bye!
```

Цикл `while` прерывается, когда пользователь вводит **5**. Ввод от **1** до **4** активизирует соответствующий выбор из списка `switch`, а ввод значения **6** вызывает действие по умолчанию.

Как отмечалось ранее, этой программе необходимы операторы break, чтобы ограничить выполнение определенной частью оператора switch. Дабы убедиться, что это именно так, вы можете удалить операторы break из листинга 6.10 и посмотреть, как программа будет работать после этого. Так, например, вы обнаружите, что ввод **2** заставит программу выполнить *все* операторы, ассоциированные с метками 2, 3, 4 и default. C++ ведет себя подобным образом, потому что такое поведение иногда может быть полезным. Во-первых, за счет этого можно легко использовать множественные метки. Например, предположим, что вы перепишете листинг 6.10, используя в качестве выборов меню и меток case символы вместо целых значений. В этом случае можно использовать символы и верхнего, и в нижнего регистра:

```
char choice;
cin >> choice;
while (choice != 'Q' && choice != 'q')
{
   switch(choice)
   {
      case 'a':
      case 'A':  cout << "\a\n";
                 break;
      case 'r':
      case 'R':  report();
                 break;
      case 'l':
      case 'L':  cout << "The boss was in all day.\n";
                 break;
      case 'c':
      case 'C':  comfort();
                 break;
      default :  cout << "That's not a choice.\n";
   }
   showmenu();
   cin >> choice;
}
```

Поскольку сразу за case 'a' не следует break, управление программой передается следующей строке, которая является оператором, следующим за case 'A'.

Использование перечислителей в качестве меток

В листинге 6.11 иллюстрируется применение enum для определения набора взаимосвязанных констант в операторе switch. В общем случае входной поток cin не распознает перечислимые типы (он не может знать, как вы определите их), поэтому программа читает выбор как int. Когда оператор switch сравнивает значение int с перечислимой меткой case, он приводит перечисление к типу int. Точно также перечисления приводятся к int в проверочном условии цикла while.

Листинг 6.11. enum.cpp

```
// enum.cpp -- использование enum
#include <iostream>
// create named constants for 0 - 6
enum {red, orange, yellow, green, blue, violet, indigo};
```

```
int main()
{
    using namespace std;
    cout << "Введите код цвета (0-6): ";
    int code;
    cin >> code;
    while (code >= red && code <= indigo)
    {
      switch (code)
      {
        case red      : cout << "Ее губы были красными.\n"; break;
        case orange   : cout << "Ее волосы были рыжими.\n"; break;
        case yellow   : cout << "Ее туфли были желтыми.\n"; break;
        case green    : cout << "Ее ногти были зелеными.\n"; break;
        case blue  : cout << "Ее костюм был синим.\n"; break;
        case violet   : cout << "Ее глаза были фиолетовыми.\n"; break;
        case indigo   : cout << "Ее настроение было цвета индиго.\n"; break;
      }
      cout << "Введите код цвета (0-6): ";
      cin >> code;
    }
    cout << "Всего наилучшего\n";
    return 0;
}
```

Вот пример вывода программы из листинга 6.11:

```
Введите код цвета (0-6): 3
Ее ногти были зелеными.
Введите код цвета (0-6): 5
Ее глаза были фиолетовыми.
Введите код цвета (0-6): 2
Ее туфли были желтыми.
Введите код цвета (0-6): 8
Всего наилучшего
```

switch и if else

Оба оператора — и switch, и if else — позволяют программе выбирать из списка альтернатив. Однако if else — более гибкий оператор из этих двух. Например, он позволяет обрабатывать диапазоны, как показано в следующем примере:

```
if (age > 17 && age < 35)
   index = 0;
else if (age >= 35 && age < 50)
   index = 1;
else if (age >= 50 && age < 65)
   index = 2;
else
   index = 3;
```

В отличие от этого, оператор switch не позволяет обрабатывать диапазоны. Каждая метка case оператора switch должна быть представлена одиночным значением. К тому же значение должно быть целым (что включает char), поэтому опера-

тор switch не может проверять значения с плавающей точкой. К тому же значение метки case должно быть константой. Если вам необходимо проверять диапазоны, выполнять проверку значений с плавающей точкой или сравнивать две переменные, то вам следует использовать if else.

Если же, однако, все альтернативы могут быть идентифицированы целыми константами, то вы можете применять как switch, так и if else. А поскольку это та ситуация, для обработки которой специально был спроектирован оператор switch, то его применение в этом случае более эффективно в смысле размера кода и скорости выполнения, если только речь не идет всего о паре возможных альтернатив выбора.

На заметку!

Если в конкретном случае можно использовать и оператор switch, и последовательность if else if, то обычное правило состоит в том, чтобы использовать switch, когда имеется три или более альтернатив.

Операторы break и continue

Операторы break и continue позволяют программе пропускать часть кода. Оператор break можно использовать в операторе switch и в любых циклах. Он вызывает немедленную передачу управления за пределы текущего оператора switch или цикла. Оператор continue применяется только в циклах и вынуждает программу пропустить остаток тела цикла и сразу начать следующую итерацию. (См. рис. 6.4.)

```
while (cin.get(ch))
{
  оператор1
  if (ch == '\n')
    continue;
  оператор2
}
оператор3
```

Оператор **continue** пропускает остаток тела цикла и начинает новую итерацию

```
while (cin.get(ch))
{
  оператор1
  if (ch == '\n')
    break;
  оператор2
}
оператор3
```

Оператор **break** пропускает цикл и переходит на оператор, следующий за циклом

Рис. 6.4. Структура операторов **break** *и* **continue**

В листинге 6.12 демонстрируется работа этих двух операторов. Программа позволяет ввести строку текста. Цикл отображает каждый ее символ и использует break, чтобы прервать цикл, если очередной символ строки окажется точкой. Это показывает, как можно использовать break, чтобы прервать цикл изнутри, если некоторое условие окажется истинным. Далее программа считает пробелы, но пропускает остальные символы. Здесь в цикле используется continue, чтобы пропустить оставшуюся часть цикла, если окажется, что символ — не пробел.

Листинг 6.12. jump.cpp

```
// jump.cpp -- использование continue и break
#include <iostream>
const int ArSize = 80;
int main()
{
   using namespace std;
   char line[ArSize];
   int spaces = 0;
   cout << "Введите строку текста:\n";
   cin.get(line, ArSize);
   cout << "Полная строка:\n" << line << endl;
   cout << "Строка до первой точки: \n";
   for (int i = 0; line[i] != '\0'; i++)
   {
       cout << line[i];       // отобразить символ
       if (line[i] == '.')    // выйти, если символ - точка
         break;
       if (line[i] != ' ')    // пропустить остаток цикла
         continue;
       spaces++;
   }
   cout << "\n" << spaces << " пробелов\n";
   cout << "Готово. \n";
   return 0;
}
```

Ниже можно видеть пример выполнения программы из листинга 6.12:

```
Введите строку текста:
Let's do lunch today. You can pay!
Полная строка:
Let's do lunch today. You can pay!
Строка до первой точки:
Let's do lunch today.
3 пробела
Готово.
```

Замечания по программе

Обратите внимание, что в то время как оператор continue вынуждает программу из листинга 6.12 пропустить оставшуюся часть тела цикла, он не пропускает выражение обновления цикла. В цикле for оператор continue заставляет программу перейти непосредственно к выражению обновления, а затем — к проверочному вы-

ражению. В цикле while, однако, continue заставляет программу сразу выполнить проверочное условие. Поэтому любое обновляющее выражение в теле цикла while, которое следует за continue, будет пропущено. В некоторых случаях это может представлять собой проблему.

Эта программа могла бы обойтись без continue. Вместо этого можно было бы использовать следующий код:

```
if (line[i] == ' ')
   spaces++;
```

Однако оператор continue может сделать программу более читабельной, когда за continue следует несколько операторов. Таким образом, нет необходимости делать эти операторы частью if.

В C++, как и в C, присутствует оператор goto. Следующий оператор:

```
goto paris;
```

означает, что нужно перейти в место, помеченное меткой paris:. То есть, у вас в программе может присутствовать следующий код:

```
char ch;
cin >> ch;
if (ch == 'P')
   goto paris;
cout << ...
...
paris: cout << "Вы только что попали в Париж. \n";
```

В большинстве случаев (некоторые скажут — во всех случаях) применение goto — плохое решение, и для управления потоком выполнения программы вы должны стараться применять структурные управляющие конструкции вроде if else, switch, continue и им подобных.

Циклы для чтения чисел

Предположим, что вы пишете программу, которая должна читать серию чисел в массив. Вы хотите дать возможность пользователю прервать ввод до наполнения массива. Один из способов сделать это — воспользоваться поведением cin. Рассмотрим следующий код:

```
int n;
cin >> n;
```

Что случится, если пользователь ответит на запрос, введя слово вместо числа? При этом произойдет пять событий:

- Значение n останется неизменным.
- Некорректный ввод останется во входной очереди.
- Будет выставлен флаг ошибки в объекте cin.
- Вызов метода cin, будучи преобразованным к типу bool, вернет false.

Тот факт, что метод вернет false, означает, что вы можете использовать нецифровой ввод для прерывания цикла чтения чисел. Тот факт, что нецифровой ввод вы-

ставляет флаг ошибки cin, означает, что вы должны сбросить этот флаг перед тем, как программа снова сможет читать ввод. Метод clear(), который также сбрасывает условие конца файла (end-of-file — EOF) (см. главу 5), позволяет сбросить флажок некорректного ввода. (Как некорректный ввод, так и EOF могут заставить cin вернуть false. В главе 17 описано, как различать эти ситуации.) Рассмотрим несколько примеров, иллюстрирующих эту технику.

Пусть вы хотите написать программу, которая вычисляет средний вес вашего ежедневного улова рыбы. Допустим, что в день ловится максимум пять рыб, поэтому массив из пяти элементов вместит все данные, но бывает, что ловится меньше пяти рыб. Код в листинге 6.13 использует цикл, который прекращается, когда массив полон, или когда вы вводите что-то, отличное от числа.

Листинг 6.13. cinfish.cpp

```
// cinfish.cpp -- нечисловой ввод прерывает цикл
#include <iostream>
const int Max = 5;
int main()
{
   using namespace std;
// получить данные
   double fish[Max];
   cout << "Пожалуйста, введите вес пойманных рыб.\n";
   cout << "Можно ввести до " << Max
        << " рыб <q — для завершения>.\n";
   cout << "рыба #1: ";
   int i = 0;
   while (i < Max && cin >> fish[i]) {
     if (++i < Max)
        cout << "рыба #" << i+1 << ": ";
   }
// вычислить среднее
   double total = 0.0;
   for (int j = 0; j < i; j++)
   total += fish[j];
// вывести результаты
   if (i == 0)
     cout << "Нет рыбы\n";
   else
     cout << total / i << " = средний вес "
          << i << " рыб\n";
   cout << "Готово.\n";
   return 0;
}
```

Замечание по совместимости

Некоторые старые компиляторы Borland выдают предупреждения, встретив такое:

```
cout << "рыба #" << i+1 << ": ";
```

Это говорит о том, что неоднозначные операции требуют скобок. Не беспокойтесь. Такие компиляторы только предупреждают о возможных ошибках группирования, если << используется в ее оригинальном значении в качестве операции сдвига влево.

Выражение cin >> fish[i] в листинге 6.13 — это на самом деле вызов функции-метода cin, которая возвращает сам cin. Если cin используется как часть проверочного условия, он приводится к типу bool. Приведенное значение равно true, если ввод прошел успешно, и false — в противном случае. Значение false выражения прерывает цикл. Кстати, вот пример запуска этой программы:

```
Пожалуйста, введите вес пойманных рыб.
Можно ввести до 5 рыб <q — для завершения>.
рыба #1: 30
рыба #2: 35
рыба #3: 25
рыба #4: 40
рыба #5: q
32.5 = средний вес 4 рыб
Готово.
```

Обратите внимание на следующую строку кода:

```
while (i < Max && cin >> fish[i]) {
```

Вспомним, что C++ не вычисляет правую часть логического выражения И, если левая часть дает false. В данном случае вычисление правой части означает использование cin для помещения ввода в массив. Если i равно Max, цикл прерывается без попыток чтения значений в место, выходящее за пределы массива.

Предыдущий пример не пытается читать никакой информации после получения нецифрового ввода. Рассмотрим случай, когда это все-таки нужно делать. Предположим, что от вас требуется ввести ровно пять результатов игры в гольф в программу, вычисляющую ваш средний результат. Если пользователь вводит нечисловое значение, программа должна напомнить ему, что требуется число. Предположим, что программа обнаружила неправильный ввод пользователя. Она должна выполнить три действия:

1. Сбросить состояние cin для получения нового ввода.
2. Освободиться от некорректного ввода.
3. Пригласить пользователя попытаться снова.

Следует отметить, что программа должна сбросить cin прежде, чем отклонит неверный ввод. В листинге 6.14 показано, как все это может быть сделано.

Листинг 6.14. cingolf.cpp

```
// cingolf.cpp -- нечисловой ввод пропускается
#include <iostream>
const int Max = 5;
int main()
{
   using namespace std;
// ввести данные
   int golf[Max];
   cout << "Пожалуйста, введите ваши результаты в гольфе.\n";
   cout << "Необходимо ввести " << Max << " раундов.\n";
   int i;
   for (i = 0; i < Max; i++)
   {
```

```
    cout << "раунд #" << i+1 << ": ";
    while (!(cin >> golf[i])) {
      cin.clear();                 // очистить ввод
      while (cin.get() != '\n')
        continue;                  // отбросить неверный ввод
      cout << "Пожалуйста введите число: ";
    }
  }
// вычислить среднее
  double total = 0.0;
  for (i = 0; i < Max; i++)
    total += golf[i];
// сообщить результат
  cout << total / Max << " = средний результат "
      << Max << " раундов\n";
  return 0;
}
```

Замечание по совместимости

Некоторые старые компиляторы Borland выдают предупреждения, встретив такое:

```
cout << "раунд #" << i+1 << ": ";
```

Это говорит о том, что неоднозначные операции требуют скобок. Не беспокойтесь. Такие компиляторы только предупреждают о возможных ошибках группирования, если << используется в ее оригинальном значении в качестве операции сдвига влево.

Ниже показан пример запуска программы из листинга 6.14:

```
Пожалуйста, введите ваши результаты в гольфе.
Необходимо ввести 5 раундов.
раунд #1: 88
раунд #2: 87
раунд #3: must i?
Пожалуйста, введите число: 103
раунд #4: 94
раунд #5: 86
91.6 = средний результат 5 раундов
```

Замечания по программе

Центральной частью обрабатывающего ошибки кода в листинге 6.14 является:

```
while (!(cin >> golf[i])) {
  cin.clear(); // очистить ввод
  while (cin.get() != '\n')
    continue;         // отбросить неверный ввод
  cout << "Пожалуйста введите число: ";
}
```

Если пользователь введет **88**, то выражение `cin` равно `true`, значение помещается в массив, выражение `!(cin>>golf[i])` принимает значение `false`, и внутренний цикл прерывается. Но если пользователь вводит **must i?**, то выражение `cin` принимает значение `false`, ничего не помещается в массив, выражение `!(cin>>golf[i])`

принимает значение `true`, и программа входит во вложенный цикл `while`. Первый оператор в цикле использует метод `clear()` для очистки ввода. Если этот оператор пропустить, программа не сможет прочитать никакого нового ввода. Далее программа использует `cin.get()` в цикле `while`, чтобы прочитать остаток ввода до конца строки. Это позволяет удалить некорректный ввод вместе со всем, что может еще содержаться в строке. Другой подход заключается в чтении до следующего пробела, что позволит удалять по одному слову за раз, вместо того, чтобы удалять всю некорректную строку. После этого программа напоминает пользователю, что нужно вводить число.

Простой файловый ввод-вывод

Иногда ввод с клавиатуры — не самый лучший выбор. Например, предположим, что вы пишете программу для анализа биржевой активности и загрузили файл, содержащий 1000 цен на акции. Было бы намного удобнее иметь программу, которая прочтет этот файл напрямую, нежели вводить все значения вручную. Точно так же было бы удобно, чтобы программа записывала свой вывод в файл, чтобы у вас сохранились результаты для последующего использования.

К счастью, C++ дает возможность применить ваш опыт программирования клавиатурного ввода и экранного вывода (вместе это называется *консольным вводом-выводом*) к работе с файлами (*файловый ввод-вывод*). В главе 17 эта тема раскрывается более подробно, а пока что мы рассмотрим простейший ввод-вывод текстовых файлов.

Текстовый ввод-вывод и текстовые файлы

Давайте еще раз рассмотрим концепцию текстового ввода-вывода. Когда вы используете `cin` для ввода, программа рассматривает ввод в форме последовательности байт, интерпретируемых как коды символов. Независимо от типа данных назначения, ввод начинается как символьные данные — то есть текстовые. Объект `cin` затем берет на себя ответственность за преобразование текста в другие типы. Чтобы увидеть, как это работает, давайте рассмотрим, как разный код обрабатывает одну и ту же строку ввода.

Предположим, что у вас есть следующая простая входная строка:

```
38.5 19.2
```

Давайте посмотрим, как эта строка ввода обрабатывается `cin` при использовании с данными разных типов. Для начала попробуем тип `char`:

```
char ch;
cin >> ch;
```

Первый символ входной строки присваивается `ch`. В данном случае первый символ — десятичная цифра `3`, и двоичный код этой цифры помещается в `ch`. Введенное значение и целевая переменная имеют тип `char`, поэтому никакого преобразования не требуется. (Обратите внимание, что сохраняется не числовое значение `3`, а код символа `'3'`.) После этого десятичная цифра `8` будет следующим символом во входной очереди, и будет следующим значением, которое поступит в следующей операции ввода.

Теперь попробуем тип int:

```
int n;
cin >> n;
```

В этом случае cin читает до первого нецифрового символа. То есть он читает цифру 3 и цифру 8, оставляя точку в качестве следующего символа во входной очереди. cin вычисляет, что эти два символа соответствуют числовому значению 38, и двоичный код 38 копируется в n.

Дальше попробуем тип double:

```
double x;
cin >> x;
```

В этом случае cin выполняет чтение до первого символа, который не может быть частью числа с плавающей точкой. То есть, он читает цифру 3, цифру 8, символ точки и цифру 5, оставляя пробел в качестве следующего символа во входной очереди. cin вычисляет, что эти четыре символа соответствует числовому значению 38.5, и двоичный код (в формате плавающей точки) числа 38.5 копируется в x.

Теперь попробуем тип символьного массива:

```
char word[50];
cin >> word;
```

В этом случае cin читает до первого пробельного символа. То есть он читает цифру 3, цифру 8, символ точки, цифру 5, оставляя пробел во входной очереди. cin помещает коды этих четырех символов в массив word и добавляет ограничивающий нулевой символ. Никаких преобразований не требуется.

И, наконец, попробуем другой вариант ввода для типа символьного массива:

```
char word[50];
cin.getline(word, 50);
```

Теперь cin читает вплоть до символа новой строки. Все символы до заключительной цифры 2 помещаются в массив word, и к ним добавляется нулевой символ-ограничитель. Символ новой строки отбрасывается, и следующим символом во входной очереди будет первый символ после перевода строки. Опять-таки, никакого преобразования не осуществляется.

При выводе происходит обратный процесс. То есть целые преобразуются в последовательности десятичных цифр, а числа с плавающей точкой — в последовательности десятичных цифр и некоторых других символов (например, 284.53 или -1.587E+06). Символьные данные преобразования не требуют.

Основная идея состоит в том, что всякий ввод начинается с текста. Поэтому файловый эквивалент консольного ввода — это текстовый файл, то есть файл, в котором каждый байт хранит код некоторого символа. Не все файлы являются текстовыми. Например, базы данных и электронные таблицы хранят числовые данные в числовом виде — то есть в двоичной целочисленной форме, или двоичном представлении чисел с плавающей точкой. Также файлы, созданные текстовыми процессорами, могут хранить текстовую информацию, но они также хранят и нетекстовые данные, описывающие форматирование, шрифты, принтеры и тому подобное.

Файловый ввод-вывод в этой главе обсуждается параллельно с консольным вводом-выводом, а потому касается только текстовых файлов. Чтобы создать тексто-

вый файл, вы используете текстовый редактор вроде EDIT для DOS, Notepad — для Windows, либо vi или emacs — для Unix/Linux. Вы можете использовать текстовый процессор до тех пор, пока сохраняете файлы в текстовом формате. Редакторы кода, являющиеся частью интегрированных сред разработки (integrated development environment — IDE), также создают текстовые файлы; на самом деле файлы исходного кода программ являются примерами текстовых файлов. Аналогично вы можете использовать текстовые редакторы для просмотра файлов, созданных текстовым выводом.

Запись текстового файла

Для файлового вывода C++ использует аналог `cout`. Поэтому, чтобы подготовиться к файловому выводу, давайте рассмотрим некоторые основные факты относительно использования `cout` в консольном выводе:

- Вы должны включать заголовочный файл `iostream`.
- Заголовочный файл `iostream` определяет класс `ostream` для обработки вывода.
- Заголовочный файл `iostream` объявляет переменную `ostream`, или объект, по имени `cout`.
- Вы должны принимать во внимание пространство имен `std`; например, вы можете использовать директиву `using` или префикс `std::` для таких элементов, как `cout` или `endl`.
- Вы можете использовать `cout` с операцией >> для чтения данных различных типов.

Файловый вывод является очень похожей аналогией:

- Вы должны включать заголовочный файл `fstream`.
- Заголовочный файл `fstream` определяет класс `ofstream` для обработки вывода.
- Вы должны объявлять один или более переменных типа `ofstream`, или объектов, которые можете именовать по своему усмотрению, пока учитываете принятые для этого соглашения.
- Вы должны учитывать пространство имен `std`; например, вы можете использовать директиву `using` или префикс `std::` для таких элементов, как `ofstream`.
- Вы должны ассоциировать конкретный объект `ofstream` с определенным файлом; одним из способов сделать это является применение метода `open()`.
- По окончании работы с файлом вы должны использовать метод `close()` для закрытия файла.
- Вы можете использовать `ofstream` с операцией >> для чтения данных различных типов.

Следует отметить, что хотя заголовочный файл `iostream` представляет предопределенный объект `ostream` по имени `cout`, вы должны объявлять свой собственный объект `ofstream`, выбирая для него имя и ассоциируя с файлом. Вот как вы объявляете такие объекты:

```
ofstream outFile;   // outFile — объект типа ofstream
ofstream fout;      // fout — объект типа ofstream
```

А вот как вы можете ассоциировать объекты с конкретными файлами:

```
outFile.open("fish.txt"); //outFile используется для записи в файл fish.txt
char filename[50];
cin >> filename;          // пользователь указывает имя файла
fout.open(filename);      // fout используется для чтения указанного файла
```

Обратите внимание, что метод `open()` требует в качестве аргумента строки в стиле С. Это может быть строковый литерал или строка, сохраненная в символьном массиве.

Вот как вы можете использовать эти объекты:

```
double wt = 125.8;
outFile << wt;            // записать число в fish.txt
char line[81] = "Objects are closer than they appear.";
fin << line << endl;      // записать строку текста
```

Важный момент заключается в том, что после того, как вы объявили объект `ofstream` и ассоциировали его с файлом, то используете его точно так же, как использовали бы `cout`. Все операции и методы, доступные cout, такие как <<, `endl` и `setf()`, также доступны для всех объектов `ofstream`, подобных `outFile` и `fout` из предыдущих примеров.

Короче говоря, ниже представлены основные шаги, которые нужно пройти, чтобы использовать файловый вывод:

1. Включить заголовочный файл `fstream`.
2. Создать объект `ofstream`.
3. Ассоциировать объект `ofstream` с файлом.
4. Использовать объект `ofstream`.в той же манере, как вы используете `cout`.

Программа в листинге 6.15 демонстрирует этот подход. Она запрашивает информацию от пользователя, посылает вывод на экран, а затем посылает тот же вывод в файл. Вы можете просмотреть полученный файл в текстовом редакторе.

Листинг 6.15. `outfile.cpp`

```
// outfile.cpp -- запись файла
#include <iostream>
#include <fstream> // для файлового ввода-вывода
int main()
{
   using namespace std;
   char automobile[50];
   int year;
   double a_price;
   double d_price;
   ofstream outFile;              // создать объект вывода
   outFile.open("carinfo.txt");   // ассоциировать его с файлом
   cout << "Введите производителя и модель автомобиля: ";
   cin.getline(automobile, 50);
   cout << "Введите год выпуска: ";
   cin >> year;
   cout << "Введите запрашиваемую цену: ";
   cin >> a_price;
   d_price = 0.913 * a_price;
```

```
// отобразить информацию на экране через cout
   cout << fixed;
   cout.precision(2);
   cout.setf(ios_base::showpoint);
   cout << "Производитель и модель: " << automobile << endl;
   cout << "Год: " << year << endl;
   cout << "Начальная цена $" << a_price << endl;
   cout << "Окончательная цена $" << d_price << endl;
// теперь ту же информацию вывести через outFile вместо cout
   outFile << fixed;
   outFile.precision(2);
   outFile.setf(ios_base::showpoint);
   outFile << "Производитель и модель: " << automobile << endl;
   outFile << "Год: " << year << endl;
   outFile << "Начальная цена $" << a_price << endl;
   outFile << "Окончательная цена $" << d_price << endl;
   outFile.close(); // закрыть файл
   return 0;
}
```

Обратите внимание, что заключительный раздел программы в листинге 6.15 дублирует раздел cout, но вместо cout применяется outFile. Вот пример запуска этой программы:

```
Введите производителя и модель автомобиля: Flitz Pinata
Введите год выпуска: 2001
Введите запрашиваемую цену: 28576
Производитель и модель: Flitz Pinata
Год: 2001
Начальная цена $28576.00
Окончательная цена $26089.89
```

Экранный вывод обеспечивается cout. Если вы проверите каталог или папку, которая содержит исполняемую программу, то найдете там новый файл carinfo.txt. Он содержит вывод, сгенерированный с помощью outFile. Если открыть его в текстовом редакторе, то в нем обнаружится следующее содержимое:

```
Производитель и модель: Flitz Pinata
Год: 2001
Начальная цена $28576.00
Окончательная цена $26089.89
```

Как видите, outFile отправляет точно ту же последовательность символов в файл carinfo.txt, что cout посылает на экран.

Замечания по программе

После того, как в программе из листинга 6.15 объявлен объект ofstream, вы можете использовать метод open(), чтобы ассоциировать объект с определенным файлом:

```
ofstream outFile;                // создать объект вывода
outFile.open("carinfo.txt");     // ассоциировать его с файлом
```

Когда программа завершает работу с файлом, она должна закрыть соединение:

```
outFile.close();
```

Обратите внимание, что метод `close()` не требует имени файла. Дело в том, что `outFile` уже был ассоциирован с конкретным файлом. Если вы забудете закрыть файл, программа закроет его автоматически при нормальном завершении.

Следует отметить, что `outFile` может использовать те же методы, что и `cout`. Он может применять не только операцию <<, но также разнообразные методы форматирования, такие как `setf()` и `precision()`. Эти методы влияют только на объект, который их вызывает. Например, вы можете указать разные значения точности для разных объектов:

```
cout.precision(2);     // использовать точность 2 для вывода на экран
outFile.precision(4); // использовать точность 4 для файлового вывода
```

Главное, что вы должны помнить — после установки такого объекта `ofstream`, как `outFile`, его можно использовать точно так же, как вы используете стандартный `cout`.

Вернемся к методу `open()`:

```
outFile.open("carinfo.txt");
```

В этом случае файл `carinfo.txt` перед запуском программы не существует. То есть здесь метод `open()` создает замечательный новый файл с таким именем. Но если файл `carinfo.txt` уже существует, что случится, если вы запустите программу вновь? По умолчанию `open()` первым делом усечет его до нулевой длины, уничтожив старое содержимое. Затем содержимое будет заменено новым выводом. В главе 17 описано, как можно переопределить это поведение по умолчанию.

Внимание!

Когда вы открываете существующий файл для вывода, по умолчанию он усекается до нулевой длины и его старое содержимое утрачивается.

Бывает, что попытка открыть файл для вывода не удается. Например, файл с указанным именем может уже существовать и иметь ограничения доступа. Поэтому внимательный программист должен проверить, удалась ли попытка открытия. Мы продемонстрируем необходимую для этого технику в следующем примере.

Чтение текстового файла

Теперь рассмотрим файловый ввод. Он базируется на консольном вводе, который имеет множество элементов. Поэтому начнем с перечисления этих элементов.

- Вы должны включить заголовочный файл `iostream`.
- Заголовочный файл `iostream` определяет класс `istream` для обработки ввода.
- Заголовочный файл `iostream` объявляет переменную, или объект, типа `istream` по имени `cin`.
- Вы должны принимать во внимание пространство имен `std`; например, вы можете использовать директиву `using` или префикс `std::` для таких элементов, как `cin`.

- Вы можете использовать cin с операцией << для чтения данных разнообразных типов.
- Вы можете использовать cin с методом get() для чтения индивидуальных символов и с методом getline() для чтения целых строк символов за раз.
- Вы можете использовать cin с такими методами, как eof() и fail(), чтобы отслеживать успешность попыток ввода.
- Сам объект cin, когда присутствует в проверочных условиях, преобразуется в булевское значение true, если последняя попытка чтения была успешной, и false — в противном случае.

Файловый ввод является очень похожей аналогией:

- Вы должны включить заголовочный файл fstream.
- Заголовочный файл fstream определяет класс ifstream для обработки ввода.
- Вы должны объявлять одну или более переменных, или объектов, типа ifstream которые можете назвать по своему усмотрению, учитывая принятые для этого соглашения.
- Вы должны принимать во внимание пространство имен std; например, вы можете использовать директиву using или префикс std:: для таких элементов, как ifstream.
- Вы должны ассоциировать конкретный объект ifstream с конкретным файлом; один из способов сделать это — воспользоваться методом open().
- Завершив работу с файлом, вы должны вызвать метод close(), чтобы закрыть его.
- Вы можете использовать объект ifstream с операцией << для чтения данных различных типов.
- Вы можете использовать объект ifstream с методом get() для чтения отдельных символов и с методом getline() — для чтения целых строк.
- Вы можете использовать объект ifstream с такими методами, как eof() и fail(), чтобы отслеживать успешность попыток ввода.
- Сам объект ifstream, когда присутствует в проверочных условиях, преобразуется в булевское значение true, если последняя попытка чтения была успешной, и false — в противном случае.

Следует отметить, что хотя заголовочный файл iostream представляет предопределенный объект istream по имени cin, вы должны объявлять свой собственный объект ifstream, выбирая для него имя и ассоциируя с файлом. Вот как объявляются такие объекты:

```
ifstream inFile;          // inFile - объект типа ifstream
ifstream fin;             // fin - объект типа ifstream
```

А вот как с ними можно ассоциировать конкретные файлы:

```
inFile.open("bowling.txt");//inFile используется для чтения файла bowling.txt
char filename[50];
cin >> filename;          // имя файла указывает пользователь
fin.open(filename);       // fin используется для чтения указанного файла
```

Метод open() требует в качестве аргумента строки в стиле C. Это может быть литеральная строка или же строка, сохраненная в символьном массиве. Вот как можно использовать эти объекты:

```
double wt;
inFile >> wt;          // читает число из bowling.txt
char line[81];
fin.getline(line, 81); // читать строку текста
```

Важный момент, который следует отметить: когда вы объявили объект ifstream и ассоциировали его с определенным файлом, то после этого можете использовать его точно так же, как используете cin. Все операции и методы, доступные cin, также доступны объектам ifstream, как это демонстрируют inFile и fin в предыдущих примерах.

Что случится, если вы попытаетесь открыть для ввода несуществующий файл? Эта ошибка приведет к тому, что все последующие попытки использовать объект ifstream для ввода будут обречены на провал. Предпочтительный способ проверки того, удалось ли открыть файл, заключается в применении метода is_open(). Вы можете использовать для этого код вроде следующего:

```
inFile.open("bowling.txt");
if (!inFile.is_open())
{
   exit(EXIT_FAILURE);
}
```

Метод is_open() возвращает true, если файл открыт успешно, поэтому выражение !inFile.is_open() вернет true, если попытка не удастся. Прототип функции exit() находится в заголовочном файле cstdlib, где также определена константа EXIT_FAILURE как значение аргумента, используемого для взаимодействия программы с операционной системой. Функция exit() прерывает программу.

Метод is_open() относительно новый в C++. Если ваш компилятор не поддерживает его, вы можете применить вместо него метод good(). Как говорится в главе 17, good() не проверяет возможные проблемы настолько тщательно, как это делает is_open().

Программа в листинге 6.16 открывает файл, указанный пользователем, читает из файла числа, после чего сообщает количество прочитанных значений, их сумму и их среднюю величину. Здесь важно правильно спроектировать входной цикл, что обсуждается подробно в разделе "Замечания по программе". Отметим, что в этой программе интенсивно используются операторы if.

Листинг 6.16. sumafile.cpp

```
// sumfile.cpp -- чтение файла
#include <iostream>
#include <fstream> // поддержка файлового ввода-вывода
#include <cstdlib> // поддержка exit()
const int SIZE = 60;
int main()
{
   using namespace std;
   char filename[SIZE];
   ifstream inFile;          // объект для обработки файлового ввода
```

```
    cout << "Введите имя файла данных: ";
    cin.getline(filename, SIZE);
    inFile.open(filename);   // ассоциировать inFile с файлом
    if (!inFile.is_open())   // не удалось открыть файл
    {
      cout << "Не удалось открыть файл " << filename << endl;
      cout << "Программа прервана.\n";
      exit(EXIT_FAILURE);
    }
    double value;
    double sum = 0.0;
    int count = 0;           // количество прочитанных элементов
    inFile >> value;         // получить первое значение
    while (inFile.good())    // пока ввод успешен и не достигнут EOF
    {
      ++count;               // еще один элемент прочитан
      sum += value;          // вычислить текущую сумму
      inFile >> value;       // получить следующее значение
    }
    if (inFile.eof())
      cout << "Достигнут конец файла.\n";
    else if (inFile.fail())
      cout << "Ввод прекращен из-за несоответствия данных.\n";
    else
      cout << "Ввод прекращен по неизвестной причине.\n";
    if (count == 0)
      cout << "Никакие данные не обработаны.\n";
    else
    {
      cout << "Прочитано элементов: " << count << endl;
      cout << "Сумма: " << sum << endl;
      cout << "Среднее: " << sum / count << endl;
    }
    inFile.close();          // завершить работу с файлом
    return 0;
}
```

Замечание по совместимости

Некоторые старые компиляторы не распознают метод `is_open()`. Для них следующую строку

```
if (!inFile.is_open())
```

можно заменить на

```
if (!inFile.good())
```

Это обеспечит чуть менее строгую проверку успешности открытия файла.

Чтобы использовать программу из листинга 6.16, вам сначала нужно создать текстовый файл, содержащий числа. Для этого можно воспользоваться текстовым редактором, который вы применяете для подготовки исходных текстов программ. Предположим, что файл называется `scores.txt` и содержит следующие данные:

```
18 19 18.5 13.5 14
16 19.5 20 18 12 18.5
17.5
```

Внимание!

Правильный формат текстовых файлов DOS/Windows требует наличия символа перевода строки в конце каждой строки файла. Некоторые текстовые редакторы вроде редактора из состава IDE-среды Metrowerks CodeWarrior автоматически не добавляют символ перевода строки к последней строке файла. Поэтому, если вы пользуетесь таким редактором, то должны нажимать клавишу <Enter> после ввода последней строки текста и перед выходом из файла.

Ниже показан пример запуска программы из листинга 6.16:

```
Введите имя файла данных: scores.txt
Достигнут конец файла.
Прочитано элементов: 12
Сумма: 204.5
Среднее: 17.0417
```

Замечания по программе

Вместо жесткого кодирования имени файла программа из листинга 6.16 сохраняет введенное пользователем имя в символьном массиве `filename`. Затем этот массив используется в качестве аргумента `open()`:

```
inFile.open(filename);
```

Как говорилось ранее в настоящей главе, весьма желательно проверять, удалась ли попытка открытия файла. Вот несколько причин возможных неудач: файл может не существовать, он может находиться в другом каталоге или папке, к нему может быть запрещен доступ, либо пользователь может допустить опечатку, вводя его имя, или пропустить его расширение. Многие начинающие программисты тратят массу времени, пытаясь разобраться, почему неправильно работает цикл чтения файла, когда реальная проблема заключается в том, что программе не удалось его открыть. Проверка успешности открытия файла может сэкономить немало времени и усилий.

Вы должны уделять особое внимание правильному проектированию цикла чтения файла. Есть несколько вещей, которые нужно проверять при чтении файла. Во-первых, программа не должна пытаться читать после достижения EOF. Метод `eof()` возвращает `true`, когда последняя попытка чтения данных столкнулась с EOF. Во-вторых, программа может столкнуться с несоответствием типа. Например, программа из листинга 6.16 ожидает файла, содержащего только числа. Метод `fail()` возвращает `true`, когда последняя попытка чтения сталкивается с несоответствием типа. (Этот метод также возвращает `true` при достижении EOF.) И, наконец, что-нибудь может пойти не так — например, файл окажется поврежденным или случится сбой оборудования. Метод `bad()` вернет `true`, если самая последняя попытка чтения столкнется с такой проблемой. Вместо того чтобы проверять все эти условия индивидуально, проще применить `good()`, которая возвращает `true`, если все идет хорошо:

```
while (inFile.good()) // пока ввод работает и не достигнут EOF
{
   ...
}
```

Затем при желании можно воспользоваться другими методами, чтобы определить, почему именно был прерван цикл:

```
if (inFile.eof())
   cout << "Достигнут конец файла.\n";
else if (inFile.fail())
   cout << "Ввод прекращен из-за несоответствия данных.\n";
else
   cout << "Ввод прекращен по неизвестной причине.\n";
```

Этот код находится сразу после цикла, поэтому он исследует, почему цикл был прерван. Поскольку `eof()` проверяет только EOF, а `fail()` проверяет как EOF, так и несоответствие типа, здесь вначале проверятся именно EOF. Таким образом, если выполнение дойдет до `else if`, то достижение конца файла (EOF), как причина выхода из цикла, будет исключена, и значение `true`, полученное от `fail()`, недвусмысленно укажет на несоответствие типа.

Важно также понимать, что `good()` сообщает лишь о самой последней попытке чтения ввода. Это значит, что попытка чтения должна *непосредственно* предшествовать вызову `good()`. Стандартный способ обеспечения этого — иметь один оператор ввода непосредственно перед началом цикла, перед первой проверкой условия цикла, а второй — в конце цикла, непосредственно перед следующей проверкой условия цикла:

```
// стандартный дизайн цикла чтения
inFile >> value;        // получить первое значение
while (inFile.good())  // пока ввод хорош и нет EOF
{
   // здесь находится тело цикла
   inFile >> value;    // получить следующее значение
}
```

Код можно несколько сократить, используя тот факт, что выражение

```
inFile >> value
```

возвращает сам `inFile`, а этот `inFile`, помещенный в контекст, в котором ожидается значение `bool`, вычисляется как `inFile.good()` — то есть как `true` или `false`. Поэтому вы можете заменить два оператора ввода одним, используя его в качестве условия цикла. То есть предыдущую структуру цикла можно заменить следующей:

```
// сокращенный дизайн цикла чтения
// пропускает предшествующий циклу ввод
while (inFile >> value) // читать и проверить успешность
{
   // здесь находится тело цикла
   // пропускает ввод в конце цикла
}
```

Этот дизайн по-прежнему следует принципу попытки чтения перед проверкой, потому что для того, чтобы оценить выражение `inFile >> value`, программа сначала пытается выполнить его, читая число в переменную `value`.

Теперь вы знакомы с основами файлового ввода-вывода.

Резюме

Программы и процесс программирования становятся более интересными, когда вводятся операторы, которые позволяют программе выбрать альтернативные действия. В C++ имеются операторы `if`, `if else`, а также `switch`, представляющие собой средства управления выбором пути выполнения. `if` позволяет программе выполнить оператор или блок операторов в случае удовлетворения некоторого условия. То есть программа выполняет этот оператор или блок, только если конкретное условие истинно. `if else` позволяет программе выбрать для выполнения одно из двух операторов или блоков. Вы можете добавлять дополнительные операторы `if else` для представления серии вариантов выбора. Оператор C++ `switch` направляет поток управления программы в определенное место из списка возможных.

В C++ также доступны операции, помогающие принимать решения. В главе 5 обсуждаются выражения отношений, которые сравнивают два значения. `if` и `if else` обычно в качестве проверочных условий используют выражения сравнения. Используя логические операции C++ (`&&`, `||` и `!`), вы можете комбинировать или модифицировать выражения сравнения для конструирования более сложных тестов. Условная операция (`?:`) предлагает компактный способ выбора одного из двух значений по условию.

Библиотека символьных функций `cctype` предлагает удобный и мощный набор инструментов анализа символьного ввода.

Циклы и операторы выбора — это полезные инструменты для организации файлового ввода-вывода, который во многом повторяет консольный. После того как вы объявляете объекты `ifstream` и `ofstream` и ассоциируете их с файлами, их можно использовать в той же манере, что и стандартные `cin` и `cout`.

Используя циклы и условные операторы C++, вы можете писать интересные, интеллектуальные и мощные программы. Но мы только начали исследовать реальную мощь языка C++. Далее мы обратимся к функциям.

Вопросы для самоконтроля

1. Рассмотрите следующие два фрагмента кода для подсчета пробелов и переводов строк:

```
// Версия 1
while (cin.get(ch)) // выйти по eof
{
    if (ch == ' ')
       spaces++;
    if (ch == '\n')
       newlines++;
}
// Версия 2
while (cin.get(ch)) // выйти по eof
{
    if (ch == ' ')
       spaces++;
    else if (ch == '\n')
       newlines++;
}
```

Какие преимущества (если они есть) у второй формы перед первой?

2. Какой эффект будет иметь замена в листинге 6.2 `++ch` на `ch+1`?
3. Внимательно рассмотрите следующую программу:

```
#include <iostream>
using namespace std;
int main()
{
   char ch;
   int ct1, ct2;
   ct1 = ct2 = 0;
   while ((ch = cin.get()) != '$')
   {
      cout << ch;
      ct1++;
      if (ch = '$')
         ct2++;
      cout << ch;
   }
   cout <<"ct1 = " << ct1 << ", ct2 = " << ct2 << "\n";
   return 0;
}
```

Предположим, вы осуществите следующий ввод, где ↳ означает нажатие клавиши <Enter>:

```
Hi!↳
Send $10 or $20 now! ↳
```

Каким будет вывод? (Помните, что ввод буферизуется.)

4. Сконструируйте логические выражения для представления следующих условий:

   а. `weight` больше или равен 115, но меньше 125.

   б. `ch` равно `q` или `Q`.

   в. `x` — четное, но не равно 26.

   г. `x` — четное, но не кратно 26.

   д. `donation` находится в диапазоне 1000–2000 или `guest` равно 1.

   е. `ch` — буква в нижнем или верхнем регистре (предполагая, что буквы нижнего регистра кодируются последовательно и буквы верхнего регистра также кодируются последовательно, но между нижним и верхним регистром имеется промежуток).

5. На английском языке предложение “I will not not speak” означает то же, что и “I will speak”. На языке C++ `!!x` — это то же самое, что и `x`?
6. Сконструируйте условное выражение, которое эквивалентно абсолютному значению переменной. То есть, если переменная `x` положительна, значением выражения будет просто значение `x`, но если `x` — отрицательна, то значением выражения должно быть `-x`, то есть положительное.
7. Перепишите следующий фрагмент с применением `switch`:

```
if (ch == 'A')
   a_grade++;
```

```
else if (ch == 'B')
   b_grade++;
else if (ch == 'C')
   c_grade++;
else if (ch == 'D')
   d_grade++;
else
   f_grade++;
```

8. В листинге 6.10 каково преимущество использования символьных меток, таких как a и c, вместо цифр для выбора в меню и в операторе switch? (Подсказка: подумайте о том, что случится, если пользователь введет q в любом регистре, и что произойдет, когда он введет в любом регистре 5.)
9. Рассмотрите следующий фрагмент кода:

```
int line = 0;
char ch;
while (cin.get(ch))
{
   if (ch == 'Q')
      break;
   if (ch != '\n')
      continue;
   line++;
}
```

Перепишите этот код без операторов break и continue.

Упражнения по программированию

1. Напишите программу, которая читает клавиатурный ввод до символа @ и повторяет его, за исключением десятичных цифр, преобразуя каждую букву верхнего регистра в букву нижнего регистра и наоборот. (Не забудьте о семействе cctype.)
2. Напишите программу, читающую в массив double до 10 значений пожертвований. Программа должна прекращать ввод при получении нечисловой величины. Она должна выдавать среднее значение полученных чисел, а также количество значений в массиве, превышающих среднее.
3. Напишите предшественник программы, управляемой меню. Она должна отображать меню из четырех пунктов, каждый из них помечен буквой. Если пользователь вводит букву, отличающуюся от четырех допустимых, программа должна повторно приглашать его ввести правильное значение до тех пор, пока он этого не сделает. Затем она должна выполнить некоторое простое действие на основе пользовательского выбора. Работа программы должна выглядеть примерно так:

```
Пожалуйста, введите одно из следующих значений:
c) хищник   p) пианист
t) дерево   g) игра
f
Пожалуйста, введите a c, p, t, или g: q
Пожалуйста, введите a c, p, t, или g: t
Клен — это дерево.
```

4. Когда вы вступите в Благотворительный Орден Программистов (БОП), к вам могут обращаться на заседаниях БОП по вашему настоящему имени, по должности либо секретному имени БОП. Напишите программу, которая может выводить списки членов по настоящим именам, должностям, секретным именам либо по предпочтению самого члена. В основу положите следующую структуру:

```
// Структура имен Благотворительного Ордена Программистов (БОП)
struct bop {
    char fullname[strsize];   // настоящее имя
    char title[strsize];      // должность
    char bopname[strsize];    // секретное имя БОП
    int preference;           // 0 = полное имя, 1 = титул, 2 = имя БОП
};
```

В этой программе создайте небольшой массив таких структур и инициализируйте его соответствующими значениями. Пусть программа запустит цикл, который даст возможность пользователю выбирать разные альтернативы:

```
a. отображать по именам      b. Отображать по должностям
c. отображать по именам БОП  d. отображать по предпочтениям
q. выйти
```

Обратите внимание, что "отображать по предпочтениям" — не значит, что нужно отобразить предпочтение члена; это значит, что нужно отобразить значение того поля структуры, которое соответствует предпочтению. Например, если `preference` равно `1`, то выбор `d` должен вызвать отображение должности для данного программиста. Пример запуска этой программы может выглядеть примерно так:

```
Отчет о Благотворительно Ордене Программистов
a. отображать по именам      b. Отображать по должностям
c. отображать по именам БОП  d. отображать по предпочтениям
q. выйти
Ваш выбор: a
Wimp Macho
Raki Rhodes
Celia Laiter
Hoppy Hipman
Pat Hand
Следующий выбор: d
Wimp Macho
Junior Programmer
MIPS
Analyst Trainee
LOOPY
Следующий выбор: q
Пока!
```

5. Королевство Нейтрония, где денежной единицей служит тварп, использует следующую шкалу налогообложения:

   Первые 5 000 тварпов — налог 0%

   Следующие 10 000 тварпов — налог 10%

   Следующие 20 000 тварпов — налог 15%

   Свыше 35 000 тварпов — налог 20%

Например, если некто зарабатывает 38 000 тварпов, то он должен заплатить налогов 5000 × 0.00 + 10000 × 0.10 + 20000 × 0.15 + 3000 × 0.20, или 4 600 тварпов. Напишите программу, которая использует цикл для запроса доходов и выдачи подлежащего к выплате налога. Цикл должен прерываться, когда пользователь вводит отрицательное или нечисловое значение.

6. Скомпонуйте программу, которая отслеживает пожертвования в Общество Защиты Влиятельных Лиц. Она должна запрашивать у пользователя количество меценатов, а затем приглашать вводить их имена и суммы пожертвований от каждого. Информация должна сохраняться в динамически выделенном массиве структур. Каждая структура должна иметь два члена: символьный массив (или объект `string`) для хранения имени и переменную-член типа `double` — для хранения суммы пожертвования. После чтения всех данных программа должна отображать имена и суммы пожертвований тех, кто не пожалел $10 000 и более. Этот список должен быть озаглавлен меткой Grand Patrons. После этого программа должна выдать список остальных жертвователей. Он должен быть озаглавлен Patrons. Если в какой-то из двух категорий не окажется никого, программа должна напечатать “none”. Помимо отображения двух категорий, никакой другой сортировки делать не нужно.

7. Напишите программу, которая читает слова по одному за раз, пока не будет введена отдельная буква **q**. После этого программа должна сообщить количество слов, начинающихся с гласных, количество слов, начинающихся с согласных, а также количество слов, не попадающих ни в одну из этих категорий. Одним из возможных подходов может быть применение `isalpha()` для различия между словами, начинающимися с букв, и остальными, с последующим применением `if` или `switch` для идентификации тех слов, прошедших проверку `isalpha()`, которые начинаются с гласных. Пример запуска может выглядеть так:

```
Вводите слова (q — для выхода):
The 12 awesome oxen ambled
quietly across 15 meters of lawn. q
5 слов начинаются с гласных
4 слов начинаются с согласных
2 остальных
```

8. Напишите программу, которая открывает текстовый файл, читает его символ за символом до самого конца и сообщает количество символов в файле.

9. Выполните упражнение 6, но измените его так, чтобы данные можно было получать из файла. Первым элементом файла должно быть количество жертвователей, а остальная часть состоять из пар строк, в которых первая строка содержит имя, а вторая — сумму пожертвования. То есть файл должен выглядеть примерно так:

```
4
Sam Stone
2000
Freida Flass
100500
Tammy Tubbs
5000
Rich Raptor
55000
```

ГЛАВА 7

Функции: программные модули C++

В этой главе:

- Основы функций
- Прототипы функций
- Как передавать аргументы функциям по значению
- Как проектировать функции для обработки массивов
- Как использовать параметры типа указателей `const`
- Как проектировать функции для обработки текстовых строк
- Как проектировать функции для объектов класса `string`
- Функции, вызывающие сами себя (рекурсия)
- Указатели на функции

Во всем можно найти удовольствие. Присмотритесь внимательнее, и вы найдете его в функциях. Язык C++ сопровождается огромной библиотекой полезных функций (стандартная библиотека ANSI C плюс несколько классов C++), но истинное удовольствие от программирования вам доставит написание собственных функций. В этой и следующей главах мы рассмотрим, как определять функции, как доставлять им информацию и как ее оттуда получать. После небольшого обзора работы функций мы сосредоточим свое внимание в этой главе на том, как использовать функции с массивами, строками и структурами. В конце мы коснемся темы рекурсии и указателей на функции. Если вы имеете опыт программирования на C, то большая часть настоящей главы покажется вам знакомой. Но не поддавайтесь ложному ощущению, что здесь нет ничего нового. C++ внес некоторые существенные дополнения к тому, что делали функции C, и в главе 8 мы рассмотрим их более подробно. А пока что обратимся к основам.

Обзор функций

Давайте посмотрим, что вы уже знаете о функциях. Для того чтобы использовать функцию в C++, вы должны выполнить следующие шаги:

- Представить определение функции.
- Представить прототип функции.
- Вызвать функцию.

Если вы используете библиотечную функцию, то эта функция уже определена и скомпилирована для вас. К тому же вы можете воспользоваться стандартным библиотечным заголовочным файлом, чтобы предоставить своей программе доступ к прототипу. Все что вам остается — правильно вызвать эту функцию. В примерах, которые рассматривались до сих пор в настоящей книге, это делалось много раз. Например, перечень стандартных библиотечных функций С включает функцию `strlen()` для нахождения длины строки. Ассоциированный стандартный заголовочный файл `cstring` содержит прототип функции для `strlen()` и ряда других связанных со строками функций. Благодаря предварительной работе, выполненной поставщиками компилятора, вы используете `strlen()` без всяких забот.

Когда вы создаете собственные функции, то должны самостоятельно обработать все три аспекта — определение, прототипирование и вызов. В листинге 7.1 демонстрируются все три шага на небольшом примере.

Листинг 7.1. calling.cpp

```
// calling.cpp -- определение, прототипирование и вызов функции
#include <iostream>
void simple(); // прототип функции
int main()
{
   using namespace std;
   cout << "main() вызовет функцию simple():\n";
   simple(); // вызов функции
   return 0;
}
// определение функции
void simple()
{
   using namespace std;
   cout << "Я — функция simple.\n";
}
```

Ниже показан вывод программы из листинга 7.1:

```
main() вызовет функцию simple():
Я — функция simple.
```

Этот пример включает директиву `using` внутрь каждого определения функции, потому что каждая функция использует `cout`. Альтернативно программа могла бы поместить единственную директиву `using` над определением функции.

Давайте рассмотрим эти три шага подробнее.

Определение функции

Все функции можно разбить на две категории: те, которые не возвращают значений, и те, которые их возвращают. Функции, не возвращающие значений, называются функциями типа `void`, и имеют следующую общую форму:

```
void имяФункции(списокПараметров)
{
   оператор(ы)
   return;     // не обязательно
}
```

Здесь *списокПараметров* указывает типы и количество аргументов (параметров), передаваемых функции. Позднее в этой главе мы исследуем эту часть более подробно. Необязательный оператор `return` отмечает конец функции. При его отсутствии функция завершается на закрывающей фигурной скобке. Тип функции `void` соответствует процедуре в Pascal, подпрограмме FORTRAN, а также процедурам подпрограмм в современной версии BASIC. Обычно вы используете функции `void` для выполнения каких-то действий. Например, функция, которая должна напечатать слово "Cheers!" заданное число раз (`n`) может выглядеть следующим образом:

```
void cheers(int n) // возвращаемого значения нет
{
   using namespace std;
   for (int i = 0; i < n; i++)
      cout << "Cheers! ";
   cout << endl;
}
```

Параметр `int n` означает, что `cheers()` ожидает передачи значения типа `int` в качестве аргумента при вызове функции.

Функция с возвращаемым значением отдает генерируемой ею значение функции, которая ее вызвала. Другими словами, если функция возвращает квадратный корень из 9.0 (`sqrt(9.0)`), то вызывающая ее функция получает значение 3.0. Такая функция объявляется, как имеющая тип, — такой, как у возвращаемого ею значения. Вот общая форма:

```
имяТипа имяФункции(списокПараметров)
{
   оператор(ы)
   return значение;        // значение приводится к типу имяТипа
}
```

Функции с возвращаемыми значениями требуют использования оператора `return` таким образом, чтобы вызывающей функции было возвращено значение. Само значение может быть константой, переменной либо более общим выражением. Единственное требование — чтобы выражение по типу совпадало с *имяТипа* либо могло быть преобразовано в этот тип. (Если, скажем, объявлен возвращаемый тип `double`, и функция возвращает выражение `int`, то `int` неявно приводится к `double`.) Затем функция возвращает конечное значение в ту функцию, которая ее вызвала. Язык C++ накладывает ограничения на типы возвращаемых значений: возвращаемое значение не может быть массивом. Все остальное допускается — целые числа, числа с плавающей точкой, указатели и даже структуры и объекты! (Интересно, что несмотря на то, что функция C++ не может вернуть массив непосредственно, она все же может вернуть его в составе структуры или объекта.)

Как программист, вы не обязаны знать, каким образом функция возвращает значение, но это знание может существенно прояснить концепцию. (К тому же это даст вам тему для разговоров с друзьями и членами семьи.) Обычно функция возвращает значение, копируя его в определенный регистр центрального процессора либо в определенное место памяти. Затем вызывающая программа читает его оттуда. И вызывающая, и возвращающая функции должны подчиняться общему соглашению относительно типа данных, хранящихся в этом месте. Прототип функции сообщает вызывающей программе, что следует ожидать, а определение функции сообщает программе, что именно она возвращает (рис. 7.1). Предоставление одной и той же ин-

формации в прототипе и определении может показаться излишней работой, но это имеет глубокий смысл. Конечно, если вы хотите, чтобы курьер взял что-то с вашего рабочего стола в офисе, то вы увеличите шансы того, что задача будет выполнена правильно, если представите описание того, что требуется, как курьеру, так и кому-то, кто находится в офисе.

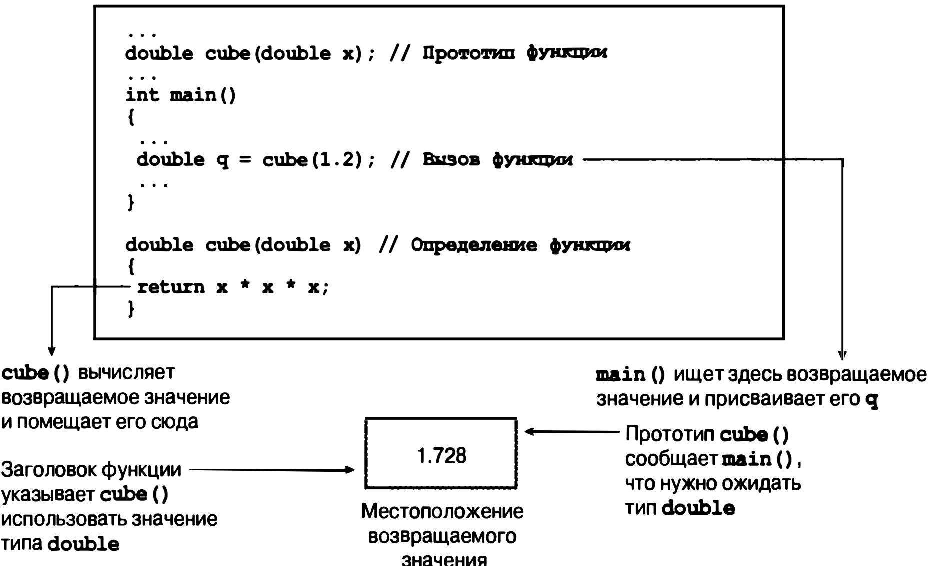


Рис. 7.1. Типичный механизм возврата значений

Функция завершается после выполнения оператора `return`. Если функция содержит более одного оператора `return`, например, в виде альтернатив разных выборов `if else`, в этом случае она прекращает свою работу по достижении первого оператора `return`. Например, в следующем примере `else` излишне, но оно помогает читателю понять намерение разработчика:

```
int bigger(int a, int b)
{
   if (a > b )
      return a; // если a > b, функция завершается здесь
   else
      return b; // в противном случае функция завершается здесь
}
```

Функции, возвращающие значения, очень похожи на функции в языках Pascal, FORTRAN и BASIC. Они возвращают значение вызывающей программе, которая затем может присвоить это значение переменной, отобразить его на экране либо использовать каким-то иным способом. Ниже показан простой пример функции, которая возвращает куб значения типа `double`:

```
double cube(double x)  // x умножить на x и еще раз умножить на x
{
   return x * x * x;   // значение типа double
}
```

Например, вызов функции cube(1.2) вернет значение 1.728. Обратите внимание, что здесь в операторе return присутствует выражение. Функция вычисляет значение выражения (в данном случае 1.728) и возвращает его.

Прототипирование и вызов функции

Вы уже знакомы с тем, как вызываются функции, но, возможно, менее уверенно себя чувствуете в том, что касается их прототипирования, поскольку зачастую прототипы функций скрываются во включаемых (#include) файлах. В листинге 7.2 демонстрируется использование функций cheers() и cube(); обратите внимание на их прототипы.

Листинг 7.2. protos.cpp

```
// protos.cpp -- использование прототипов и вызовы функций
#include <iostream>
void cheers(int);          // прототип: нет значения возврата
double cube(double x);     // прототип: возвращает double
int main(void)
{
   using namespace std;
   cheers(5);                     // вызов функции
   cout << "Введите число: ";
   double side;
   cin >> side;
   double volume = cube(side);  // вызов функции
   cout <<  side <<" в кубе равно ";
   cout << volume << " кубических футов. \n";
   cheers(cube(2));               // защита прототипа в действии
   return 0;
}
void cheers(int n)
{
   using namespace std;
   for (int i = 0; i < n; i++)
   cout << "Спасибо! ";
   cout << endl;
}
double cube(double x)
{
   return x * x * x;
}
```

Программа из листинга 7.2 помещает директиву using только в те функции, которые используют члены пространства имен std. Вот пример запуска:

```
Cheers! Cheers! Cheers! Cheers! Cheers!
Введите число: 5
5 в кубе равно 125 кубических футов.
Спасибо! Спасибо! Спасибо! Спасибо! Спасибо! Спасибо! Спасибо! Спасибо!
```

Обратите внимание, что main() вызывает функцию cheers() типа void, используя имя функции и аргументы, за которыми следует точка с запятой: cheers(5);. Это при-

мер оператора вызова функции. Но поскольку cube() возвращает значение, main() может использовать его как часть оператора присваивания:

```
double volume = cube(side);
```

Но как говорилось ранее, вы должны сосредоточиться на прототипах. Что вы должны знать о прототипах? Во-первых, вы должны понять, почему C++ требует их. Затем, поскольку C++ требует прототипы, вам должен быть известен правильный синтаксис их написания. И, наконец, вы должны оценить, что они вам дают. Рассмотрим все эти вопросы по очереди, используя листинг 7.2 в качестве основы для обсуждения.

Зачем нужны прототипы?

Прототип описывает интерфейс функции для компилятора. То есть он сообщает компилятору, каков тип возвращаемого значения, если он есть у функции, а также количество и типы аргументов данной функции. Рассмотрим для примера, как влияет прототип на вызов функции в листинге 7.2:

```
double volume = cube(side);
```

Во-первых, прототип сообщает компилятору, что cube() должен принимать один аргумент типа double. Если программа не предоставит этот аргумент, то прототипирование позволит компилятору перехватить такую ошибку. Во-вторых, когда функция cube() завершает вычисление, она помещает возвращаемое значение в некоторое определенное место — возможно, в регистр центрального процессора, возможно, в память. Затем вызывающая функция — main() в данном случае — извлекает значение из этого места. Поскольку прототип устанавливает, что cube() имеет тип double, компилятор знает, сколько байт следует извлечь и как их интерпретировать. Без этой информации он может только предполагать, а это то, чего он делать вовсе не должен.

Но вы все еще можете задаваться вопросом: почему компилятору нужен прототип? Не может ли он просто заглянуть дальше в файл и увидеть, как определена функция? Одна из проблем такого подхода в том, что он не слишком эффективен. Компилятору пришлось бы приостановить компиляцию main() на то время, пока он прочтет остаток файла. Однако имеется еще более серьезная проблема: функции может и не находиться в том же файле. C++ позволяет разбивать программу на множество файлов, которые компилируются независимо и позднее собираются вместе в одну исполняемую программу.

Синтаксис прототипа

Прототип функции — это оператор, поэтому он должен завершаться точкой с запятой. Простейший способ получить прототип — скопировать заголовок функции из ее определения и добавить точку с запятой. Что, собственно, и делает программа из листинга 7.2 с функцией cube():

```
double cube(double x); // добавлено ; к заголовку для получения прототипа
```

Однако прототип функции не требует предоставления имен переменных-параметров; достаточно списка типов. Программа из листинга 7.2 прототипирует cheers(), используя только тип аргумента:

```
void cheers(int);//в прототипе можно опустить имена переменных-параметров
```

В общем случае, в прототипе вы можете указывать или не указывать имена переменных в списке аргументов. Имена переменных в прототипе служат просто указанием места аргумента, поэтому если они заданы, то не обязательно должны совпадать с именами в определении функции.

Сравнение прототипирования в C++ и ANSI C

Прототипирование ANSI C позаимствовано из C++, но в этих двух языках имеются различия. Наиболее важно то, что ANSI C, сохраняя совместимость с классическим C, считает прототипирование не обязательным, в то время как в C++ прототипирование обязательно. Например, рассмотрим следующее объявление функции:

```
void say_hi();
```

В C++ пустые скобки означают то же, что указание ключевого слова `void` между ними. Это значит, что функция не имеет аргументов. В ANSI C пустые скобки означают просто, что список аргументов не указан — возможно, вы забыли это сделать, либо функция имеет переменное число аргументов. Эквивалент этого в C++ выглядит следующим образом:

```
void say_bye(...);  // C++ снимает с себя ответственность
```

Обычно такое использование многоточия необходимо только для взаимодействия с функциями C, имеющими переменное количество аргументов, например, `printf()`.

Что вам дают прототипы

Вы увидели, что прототипы помогают компиляторам. Но чем они полезны вам? Прототипы значительно снижают вероятность ошибок в программе. В частности, они обеспечивают следующее:

- Компилятор корректно обрабатывает возвращаемое значение.
- Компилятор проверяет, указано ли правильное количество аргументов.
- Компилятор проверяет правильность типов аргументов. Если тип не подходит, компилятор преобразует его в правильный, когда это возможно.

Мы уже говорили о том, как корректно обработать возвращаемое значение. Теперь посмотрим, что случится, если вы зададите неверное количество аргументов. Например, предположим, что в программе выполнен следующий вызов:

```
double z = cube();
```

Без прототипирования функции компилятор пропустит это. Когда функция будет вызвана, он обнаружит, что `cube()` должна принимать число, и подставит любое подвернувшееся значение. Именно так работал C до того, как ANSI C позаимствовал прототипирование из C++. Поскольку прототипирование в ANSI C не обязательно, именно так программы C ведут себя и до сих пор. Но в C++ прототипирование обязательно, поэтому вы гарантированы от ошибок подобного рода.

Далее предположим, что вы указали список аргументов, но некоторые из них неправильного типа. В C это может привести к роковым ошибкам. Например, если функция ожидает тип `int` (пусть он имеет размер 16 бит), а вы передали `double` (предположим, 64 бита), то функция видит только первые 16 бит их 64 и пытается интерпретировать их как значение типа `int`. Однако C++ автоматически преобразует переданное значение к типу, указанному в прототипе, предполагая, что оба типа арифметические.

Например, листинг 7.2 содержит два несоответствия типа в одном операторе:

```
cheers(cube(2));
```

Во-первых, программа передает целое значение 2 функции cube(), которая ожидает тип double. Компилятор, замечая, что прототип cube() специфицирует тип аргумента double, преобразует 2 в 2.0, то есть в значение типа double. Затем cube() возвращает значение 8.0 типа double, которое должно быть использовано в качестве аргумента cheers(). Опять-таки, компилятор проверяет прототип и замечает, что cheers() требует аргумента int. Он преобразует возвращенное значение в целочисленное 8. В общем случае, прототипирование позволяет осуществить автоматическое приведение к ожидаемым типам. (Перегрузки функций, которые мы обсудим в главе 8, однако, могут породить неоднозначные ситуации, которые предотвращают некоторые автоматические приведения типов.)

Автоматическое преобразование типов не позволяет исключить все возможные ошибки. Например, если вы передаете значение 8.33E27 в функцию, ожидающую аргумента int, то такое большое значение не может быть корректно преобразовано в обычный int. Некоторые компиляторы предупреждают о возможных потерях данных при автоматическом преобразовании больших типов в малые.

К тому же прототипирование позволяет выполнять преобразования типов только тогда, когда они имеют смысл. Например, невозможно конвертировать целое в структуру или указатель.

Прототипирование происходит во время компиляции и называется *статическим контролем типов*. Статический контроль типов, как вы уже видели, обнаруживает многие ошибки, которые было бы трудно перехватить во время выполнения.

Аргументы функций и передача по значению

Наступило время внимательнее взглянуть на аргументы функций. C++ обычно передает их *по значению*. Это означает, что числовое значение аргумента передается в функцию, где присваивается новой переменной. Например, в листинге 7.2 присутствует следующий вызов функции:

```
double volume = cube(side);
```

Здесь side — переменная, которая в примере запуска получает значение 5. Вспомним, что заголовок функции cube() был таким:

```
double cube(double x)
```

Когда эта функция вызывается, она создает новую переменную типа double по имени x и присваивает ей значение 5. Это позволяет изолировать данные в main() от того, что происходит в cube(), потому что cube() работает с копией side вместо исходных данных. Вскоре вы увидите пример такой защиты. Переменная, которая используется для приема переданного значения, называется *формальным аргументом* или *формальным параметром*. Значение, переданное функции, называется *действительным аргументом* или *действительным параметром*. Чтобы еще немного упростить ситуацию, стандарт C++ использует слово *аргумент* для обозначения действительного аргумента или параметра, а слово *параметр* — для обозначения формального аргумента или параметра. Применяя эту терминологию, можно сказать, что передача аргумента присваивает его значение параметру. (См. рис. 7.2.)

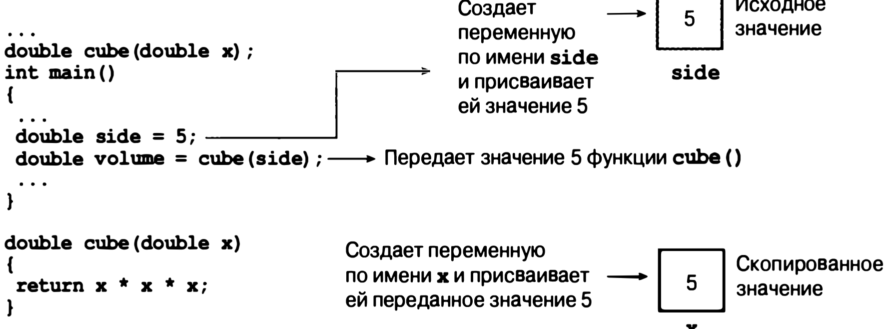


Рис. 7.2. Передача по значению

Переменные, включая параметры, объявленные в функции, являются приватными по отношению к ней. Когда функция вызывается, компьютер выделяет память, необходимую для этих переменных. Когда функция завершается, компьютер освобождает память, которая была использована этими переменными. (В некоторых источниках по C++ это выделение и освобождение памяти для переменных называется *созданием и уничтожением переменных*. Это звучит более выразительно.) Такие переменные называются *локальными переменными*, потому что они локализованы в пределах функции. Как уже упоминалось ранее, это помогает сохранить целостность данных. Это также означает, что если вы объявили переменную x в `main()` и другую переменную x в некоторой другой функции, то это будут две совершенно различных, никак не связанных переменных, подобно тому, как Олбани в Калифорнии никак не связан с Олбани в штате Нью-Йорк (рис. 7.3). Такие переменные также называются *автоматическими переменными*, поскольку они размещаются и освобождаются автоматически во время выполнения программы.

Множественные аргументы

Функция может принимать более одного аргумента. При вызове функции вы просто отделяете такие аргументы друг от друга запятыми:

```
n_chars('R', 25);
```

Это передает два аргумента функции `n_chars()`, определение которой мы приведем чуть позже.

Аналогично, когда вы определяете функцию, то используете разделенный запятыми список параметров в ее заголовке:

```
void n_chars(char c, int n)  // два аргумента
```

Этот заголовок функции устанавливает, что функция `n_chars()` принимает один аргумент типа `char` и один типа `int`. Параметрам `c` и `n` присваиваются значения, переданные функции. Если функция имеет два аргумента одного и того же типа, вы должны задавать тип каждого параметра отдельно. Нельзя комбинировать объявления аргументов, как это делается с обычными переменными:

```
void fifi(float a, float b)  // объявляет каждую переменную отдельно
void fufu(float a, b)        // недопустимо
```

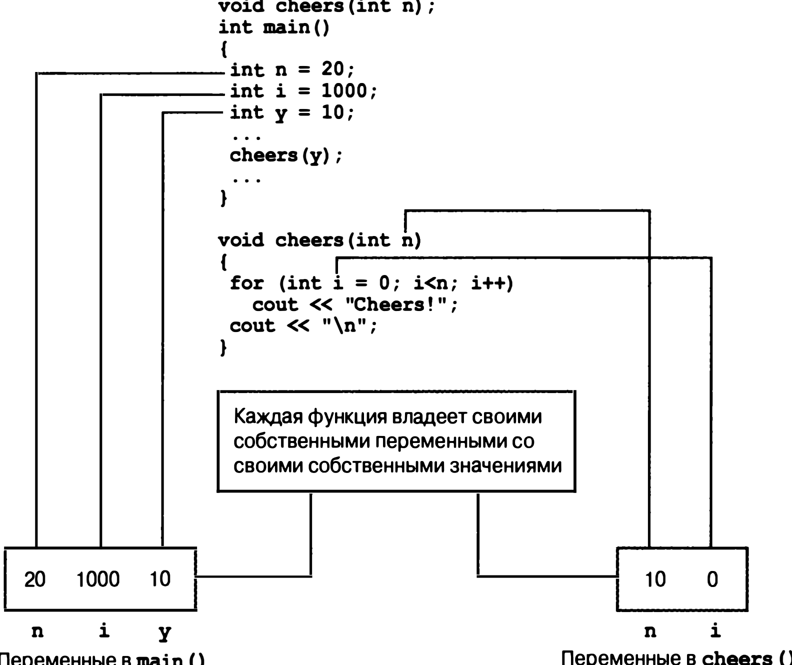


Рис. 7.3. Локальные переменные

Как и с другими функциями, вы добавляете точку с запятой, чтобы получить прототип:

```
void n_chars(char c, int n);    // прототип, стиль 1
```

Как и с единственным аргументом, вы не обязаны использовать одинаковые имена переменных в прототипе и определении, и также можно пропускать имена переменных в прототипе:

```
void n_chars(char, int);        // прототип, стиль 2
```

Однако указание имен переменных все же может сделать прототип более понятным, особенно если два параметра имеют один и тот же тип. Позднее имена параметров могут напомнить вам — какой аргумент для чего предназначен:

```
double melon_density(double weight, double volume);
```

В листинге 7.3 представлен пример функции с двумя аргументами. Он также иллюстрирует, что изменение значения формального параметра внутри функции никак не влияет на данные вызывающей программы.

Листинг 7.3. twoarg.cpp

```
// twoarg.cpp -- функция с двумя аргументами
#include <iostream>
using namespace std;
void n_chars(char, int);
```

```
int main()
{
  int times;
  char ch;
  cout << "Введите символ: ";
  cin >> ch;
  while (ch != 'q')            // q для выхода
  {
   cout << "Введите число: ";
   cin >> times;
   n_chars(ch, times);         // функция с двумя аргументами
   cout << "\nВведите другой символ или нажмите"
        " клавишу q для выхода: ";
   cin >> ch;
  }
  cout << "Значение переменной times равно " << times << ".\n";
  cout << "Готово\n";
  return 0;
}
void n_chars(char c, int n)    // отобразить c n раз
{
  while (n-- < 0)              // продолжать до достижения n значения 0
   cout << c;
}
```

Программа в листинге 7.3 иллюстрирует помещение директивы `using` над определением функции вместо помещения ее внутри функции. Ниже показан пример ее выполнения:

```
Введите символ: W
Введите число: 50
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
Введите другой символ или нажмите клавишу q для выхода: a
Введите число: 20
aaaaaaaaaaaaaaaaaaaa
Введите другой символ или нажмите клавишу q для выхода: q
Значение переменной times равно 20.
Готово
```

Замечания по программе

Функция `main()` в листинге 7.3 использует цикл `while` для организации повторяющегося ввода (и чтобы освежить ваши знания о циклах). Обратите внимание, что для чтения символа она применяет `cin >> ch` вместо `cin.get(ch)` или `ch = cin.get()`. На то имеется серьезная причина. Вспомним, что две функции `cin.get()` читают все входные символы, включая пробелы и символы перевода строки, в то время как `cin >>` пропускает пробелы и переводы строк. Когда вы отвечаете на приглашение программы, то должны нажимать <Enter> в конце каждой строки, генерируя тем самым символ перевода строки. Подход `cin >> ch` легко пропускает эти лишние символы, тогда как оба варианта `cin.get()` читают перевод строки, следующий за каждым введенным числом, как очередной символ для отображения. Это неудобство можно обойти программным путем, но проще использовать `cin`, как это делается в программе из листинга 7.3.

Функция `n_chars()` принимает два аргумента: символ `c` и целое число `n`. Затем она использует цикл для отображения символа столько раз, сколько указано в `n`:

```
while (n-- > 0) // продолжать до достижения n значения 0
    cout << c;
```

Обратите внимание, что программа выполняет подсчет, уменьшая на каждом шаге значение переменной `n`, которая является формальным параметром из списка аргументов. Этой переменной присваивается значение переменной `times` из функции `main()`. Цикл `while` уменьшает `n` до `0`, но, как доказывает пример выполнения, изменение значения `n` никак не отражается на значении `times`.

Еще одна функция с двумя аргументами

Теперь давайте создадим более сложную функцию — такую, которая будет выполнять нетривиальные вычисления. К тому же помимо применения формальных параметров функция проиллюстрирует использование локальных переменных.

Сейчас многие штаты США организуют различного рода лотереи. Эти лотереи предлагают вам выбрать определенные числа из многих, представленных на карточке. Например, вы можете выбрать 6 чисел в карточке, содержащей всего 51 число. Затем ведущий лотереи выбирает случайным образом 6 номеров. Если ваш вариант полностью совпал с тем, что выбрал ведущий, вы получаете несколько миллионов долларов или около того. Наша функция будет вычислять вероятность выигрыша. (Конечно, функция, которая могла бы успешно угадывать выигрышные номера, была бы более полезной, но C++, несмотря на всю его мощность, не может учитывать психологические факторы.)

Во-первых, нам понадобится формула. Если вы должны угадать 6 номеров из 51, математики говорят, что у вас имеется 1 шанс выигрыша из R, где R вычисляется по следующей формуле:

$$R = \frac{51 \times 50 \times 49 \times 48 \times 47 \times 46}{6 \times 5 \times 4 \times 3 \times 2 \times 1}$$

Для 6 чисел в знаменателе будет произведение первых 6 целых чисел, или 6! (шесть факториал). Числитель же вычисляется как произведение шести последовательных чисел, на этот раз, начиная с 51 и ниже. В общем, если нужно выбрать `pick` вариантов из `numbers` чисел, то числитель будет `pick` факториал, а знаменатель – произведение `pick` целых чисел, начиная со значения `numbers` и ниже. Для выполнения этого вычисления можно воспользоваться циклом `for`:

```
long double result = 1.0;
for (n = numbers, p = picks; p > 0; n--, p--)
    result = result * n / p ;
```

Вместо того чтобы сразу перемножить все составляющие числителя, цикл начинает с умножения `1.0` на первую составляющую числителя и затем делит его на первую составляющую знаменателя. Затем на следующем шаге цикл умножает и делит результат на следующие составляющие числителя и знаменателя. Это позволяет сохранять текущее произведение меньшим, чем если бы сначала выполнялось все умножение. Например, сравните

```
(10 * 9) / (2 * 1)
```

с

```
(10 / 2) / (9 / 1)
```

Первое выражение вычисляет 90/2, получая 45, в то время как второе вычисляет 5×9, получая те же 45. Результат одинаков, но в первом случае получается больший промежуточный результат (90), нежели во втором. Чем больше у вас множителей, тем существеннее разница. Для больших чисел эта стратегия замены умножения делением может предохранить процесс вычисления от переполнения максимально возможного значения с плавающей точкой.

В листинге 7.4 эта формула заключена в функции `probability()`. Поскольку количество вариантов выбора и общее количество чисел должны быть положительными значениями, программа использует тип `unsigned int` (сокращенно — `unsigned`) для этих величин. Перемножение нескольких целых может породить достаточно большие результаты, поэтому `lotto.cpp` использует тип `long double` для возвращаемого значения функции. К тому же такие выражения, как 49/6, порождают ошибки округления при работе с целочисленными типами.

Замечание по совместимости

Некоторые реализации C++ не поддерживают тип `long double`. Если ваша реализация относится к ним, используйте просто `double`.

Листинг 7.4. `lotto.cpp`

```
// lotto.cpp -- вероятность выигрыша
#include <iostream>
// Примечание: некоторые реализации требуют double вместо long double
long double probability(unsigned numbers, unsigned picks);
int main()
{
   using namespace std;
   double total, choices;
   cout << "Укажите общее количество номеров и\n"
         "количество номеров, которые нужно угадать:\n";
   while ((cin >> total >> choices) && choices <= total)
   {
     cout << "У вас один шанс из ";
     cout << probability(total, choices); // вычисление числа вариантов
     cout << " чтобы выиграть.\n";
     cout << "Следующие два числа (q для выхода): ";
   }
   cout << "Удачи!\n";
   return 0;
}
// следующая функция вычисляет вероятность правильного
// угадывания picks чисел из numbers возможных
long double probability(unsigned numbers, unsigned picks)
{
   long double result = 1.0; // здесь — локальные переменные
   long double n;
   unsigned p;
```

```
    for (n = numbers, p = picks; p > 0; n--, p--)
      result = result * n / p ;
    return result;
}
```

Ниже показан пример выполнения программы из листинга 7.4:

```
Укажите общее количество номеров и
количество номеров, которые нужно угадать:
49 6
У вас один шанс из 1.39838e+007 чтобы выиграть.
Следующие два числа (q для выхода): 51 6
У вас один шанс из 1.80095e+007 чтобы выиграть.
Следующие два числа (q для выхода): 38 6
У вас один шанс из 2.76068e+006 чтобы выиграть.
Следующие два числа (q для выхода): q
Удачи!
```

Обратите внимание, что увеличение количества вариантов в игровой карточке существенно снижает шансы выигрыша.

Замечания по программе

Функция probability() из листинга 7.4 иллюстрирует два вида локальных переменных, которые встречаются в функциях. Первый — это формальные параметры (numbers и picks), объявленные в заголовке функции внутри скобок. Затем идут другие локальные переменные (result, n и p). Они объявлены между фигурными скобками, ограничивающими определение функции. Основная разница между формальными параметрами и прочими локальными переменными состоит в том, что формальные параметры получают свои значения от функции, которая вызывает probability(), в то время как локальные переменные получают свои значения только внутри функции.

Функции и массивы

До сих пор все примеры функций, приведенные в данной книге, были простыми и использовали в своих аргументах и возвращаемых значениях только значения базовых типов. Однако функции могут служить инструментами и для обработки более сложных типов, таких как массивы и структуры. Давайте посмотрим, как относятся друг к другу массивы и функции.

Предположим, что вы используете массив, чтобы отследить, сколько печенья съел каждый участник семейного пикника. (Каждый индекс массива соответствует определенному лицу, а значение элемента — количеству съеденного печенья.) Нужен также общий итог. Его легко вычислить: для этого нужно просто применить цикл, чтобы суммировать все элементы массива. Но сложение элементов массива — настолько часто встречающаяся операция, что имеет смысл спроектировать функцию для выполнения этой задачи. Тогда вам не придется писать новый цикл каждый раз, когда понадобится сложить элементы массива.

Посмотрим, как должен выглядеть интерфейс функции. Поскольку функция вычисляет сумму, она должна возвращать ответ. Если вы едите печенье целиком, то можно использовать функцию с типом возврата int. Чтобы функция знала, какой массив

суммировать, вы захотите передать ей в качестве аргумента имя массива. И чтобы не ограничивать ее массивами определенного размера, нужно будет также передавать ей размер массива. Единственный новый ингредиент здесь — это имя массива в качестве одного из формальных аргументов. Давайте посмотрим, что получается:

```
int sum_arr(int arr[], int n) // arr = имя массива, n = размер
```

Выглядит вполне прилично. Квадратные скобки указывают на то, что `arr` — массив, а тот факт, что они пусты, говорит о том, что вы можете применять эту функцию с массивами любого размера. Но бывает, что некоторые вещи не являются тем, чем они кажутся: `arr` — на самом деле не массив, а указатель! Однако хорошей новостью для вас будет то, что вы можете писать остальную часть вашей функции так, как если бы `arr` все-таки был массивом. Во-первых, убедимся, что такой подход работает, а потом разберемся, почему он работает.

В листинге 7.5 иллюстрируется применение указателя, как если бы он был именем массива. Программа инициализирует массив некоторыми значениями и использует функцию `sum_arr()` для вычисления суммы. Обратите внимание, что `sum_arr()` использует `arr`, как если бы он был именем массива.

Листинг 7.5. `arrfun1.cpp`

```
// arrfun1.cpp -- функция с аргументом-массивом
#include <iostream>
const int ArSize = 8;
int sum_arr(int arr[], int n); // прототип
int main()
{
   using namespace std;
   int cookies[ArSize] = {1,2,4,8,16,32,64,128};
// некоторые системы требуют предварить int словом static,
// чтобы разрешить инициализацию массива

   int sum = sum_arr(cookies, ArSize);
   cout << "Всего съедено печенья: " << sum << "\n";
   return 0;
}

// вернуть сумму элементов массива целых чисел
int sum_arr(int arr[], int n)
{
   int total = 0;
   for (int i = 0; i < n; i++)
      total = total + arr[i];
   return total;
}
```

Вывод программы из листинга 7.5 выглядит следующим образом:

```
Всего съедено печенья: 255
```

Как видите, программа работает. Теперь разберемся, почему она работает.

Как указатели позволяют функциям обрабатывать массивы

Ключом к программе 7.5 является то, что C++, подобно С, в большинстве контекстов трактует имя массива как указатель. Вспомните из главы 4, что имя массива рассматривается как адрес его первого элемента:

```
cookies == &cookies[0] // имя массива – это адрес его первого элемента
```

(Существуют два исключения из этого правила. Во-первых, объявление массива использует имя массива в качестве метки хранилища. Во-вторых, применение sizeof к имени массива порождает размер всего массива в байтах.)

В листинге 7.5 присутствует следующий вызов функции:

```
int sum = sum_arr(cookies, ArSize);
```

Здесь cookies — имя массива, поэтому, согласно правилам C++, cookies представляет собой адрес первого элемента этого массива. То есть функции передается адрес. Поскольку массив имеет тип элементов int, cookies должно иметь тип указателя на int, или int *. Это предполагает, что корректный заголовок функции должен быть таким:

```
int sum_arr(int * arr, int n) // arr = имя массива, n = размер
```

Здесь int * arr заменяет int arr[]. На самом деле оба варианта заголовка корректны, потому что в C++ нотации int * arr и int arr[] имеют идентичный смысл, когда (и *только* в этом случае) применяются в заголовке или прототипе функции. Оба варианта означают, что arr — указатель на int. Однако, версия с нотацией массива (int arr[]) символически напоминает вам о том, что arr — не просто указатель на int, но указатель на первый int в массиве. В этой книге мы применяем нотацию массивов, когда указатель указывает на первый элемент в массиве, и нотацию указателя — когда имеется в виду указатель на отдельный элемент. Помните, однако, что нотации int * arr и int arr[] не являются синонимами ни в каком другом контексте. Например, вы не можете использовать нотацию int tip[] для объявления указателя в теле функции.

Принимая во внимание, что arr — на самом деле указатель, становится понятен смысл остальной части функции. Если вспомнить дискуссию о динамических массивах из главы 4, чтобы обращаться к индивидуальным элементам массива, нотация квадратными скобками полностью подходит как для работы с именами массивов, так и для работы с указателями. Будь arr указателем или именем массива, выражение arr[3] в любом случае означает четвертый элемент массива. Не помешает также напомнить о справедливости следующих утверждений:

```
arr[i] == *(ar + i)     // значение в двух нотациях
&arr[i] == ar + i       // адреса в двух нотациях
```

Вспомните, что прибавление единицы к указателю, включая и имя массива, на самом деле прибавляет к адресу значение размера в байтах того типа, на который указывает данный указатель. Увеличение указателя и увеличение индекса — это два эквивалентных способа подсчета элементов от начала массива.

Последствия использования массивов в качестве аргументов

Рассмотрим, что следует из листинга 7.5. Вызов функции sum_arr(cookies, ArSize) передает адрес первого элемента массива cookies и количество его элементов в функцию sum_arr(). Функция sum_arr() присваивает адрес cookies переменной-указателю arr , а значение ArSize — переменной n типа int. Это значит, что в листинге 7.5. на самом деле в функцию не передается содержимое массива. Вместо этого программа сообщает функции, где находится массив (то есть сообщает адрес), каков тип его элементов (тип массива) и сколько в нем содержится элементов (переменная n). (См. рис. 7.4.) Вооруженная этой информацией, функция затем использует исходный массив. Если вы передаете обычную переменную, то функция работает с ее копией. Но если вы передаете массив, то функция работает с его оригиналом. На самом деле эта разница не нарушает подхода C++ к передаче аргументов по значению. Функция sum_arr() принимает значение, которое присваиваются новой переменной. Но это значение — отдельный адрес, а не содержимое массива.

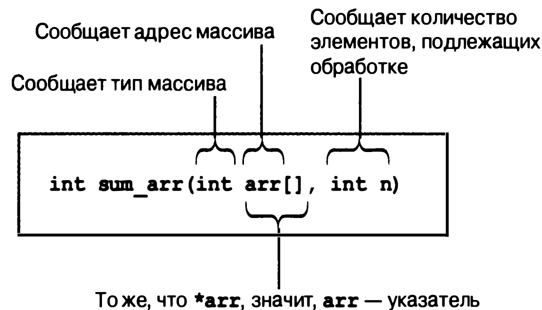


Рис. 7.4. Передача функции информации о массиве

Хорошо ли то, что между именами массивов и указателями есть соответствие? Безусловно. Проектное решение использовать адреса массивов в качестве аргументов экономит время и память, необходимые для копирования всего массива. Накладные расходы, связанные с использованием таких копий, могли быть весьма ощутимыми при работе с большими массивами. С копиями программам понадобилось бы не только больше компьютерной памяти, но и больше времени, чтобы копировать крупные блоки данных. С другой стороны, работа с исходными данными чревата возможностью непреднамеренного их повреждения. Это — реальная проблема в классическом С, но ANSI С и C++ предусматривают модификатор const, который обеспечивает необходимую защиту. Скоро вы увидите пример. Но сначала давайте изменим листинг 7.5, чтобы проиллюстрировать некоторые моменты, связанные с тем, как работают функции массивов. Код в листинге 7.6 демонстрирует, что cookie и arr содержат одни и те же значения. Он также показывает, как концепция указателей делает функцию sum_arr() более изменчивой и гибкой, нежели могло показаться вначале. Чтобы внести немного разнообразия и показать, на что это похоже, программа использует квалификатор std:: вместо директивы using для обеспечения доступа к cout и endl.

Листинг 7.6. arrfun2.cpp

```
// arrfun2.cpp -- функция с аргументом-массивом
#include <iostream>
const int ArSize = 8;
int sum_arr(int arr[], int n);
// используем std:: вместо директивы using
int main()
{
   int cookies[ArSize] = {1,2,4,8,16,32,64,128};
// некоторые системы требуют предварить int словом static,
// чтобы разрешить инициализацию массива
   std::cout << cookies << " = адрес массива, ";
// некоторые системы требуют приведения типа: unsigned (cookies)
   std::cout << sizeof cookies << " = sizeof cookies\n";
   int sum = sum_arr(cookies, ArSize);
   std::cout << "Всего съедено печенья: " << sum << std::endl;
   sum = sum_arr(cookies, 3);
   std::cout << "Первые три едока умяли " << sum << " печений. \n";
   sum = sum_arr(cookies + 4, 4);
   std::cout << "Последние пять едоков схрумкали " << sum << " печений. \n";
   return 0;
}
// возвращает сумму элементов целочисленного массива
int sum_arr(int arr[], int n)
{
   int total = 0;
   std::cout << arr << " = arr, ";
// некоторые системы требуют приведения типа: unsigned (arr)
   std::cout << sizeof arr << " = sizeof arr\n";
   for (int i = 0; i < n; i++)
     total = total + arr[i];
   return total;
}
```

Вывод программы из листинга 7.6 выглядит следующим образом:

```
0x0065fd24 = адрес массива, 32 = sizeof cookies
0x0065fd24 = arr, 4 = sizeof arr
Всего съедено печенья: 255
0x0065fd24 = arr, 4 = sizeof arr
Первые три едока умяли 7 печений.
0x0065fd34 = arr, 4 = sizeof arr
Последние пять едоков схрумкали 240 печений.
```

Обратите внимание, что значения адресов и размеров могут изменяться от системы к системе. К тому же, некоторые реализации отображают адреса в десятичной системе вместо шестнадцатеричной.

Замечания по программе

Код в листинге 7.6 иллюстрирует некоторые очень интересные моменты, касающиеся функций, работающих с массивами. Во-первых, обратите внимание, что cookies и arr, как и утверждалось, находятся по одному и тому же адресу в памяти.

Но sizeof cookies равно 16, в то время как sizeof arr составляет 4. Это потому, что sizeof cookies — размер всего массива, тогда как sizeof arr — размер переменной-указателя. (Программа выполнялась на системе, использующей 4-байтную адресацию.) Кстати, именно поэтому нужно передавать размер массива вместо использования sizeof arr в sum_arr().

Поскольку единственный способ для sum_arr() узнать количество элементов в массиве — через второй аргумент, вы можете схитрить. Например, второй вызов функции у нас выглядит следующим образом:

```
sum = sum_arr(cookies, 3);
```

Сообщая функции, что cookies имеет только три элемента, вы заставляете ее подсчитать сумму первых трех элементов.

Но зачем на этом останавливаться? Вы можете также схитрить относительно местоположения массива:

```
sum = sum_arr(cookies + 4, 4);
```

Поскольку cookies является адресом первого элемента, то cookies + 4 — адрес пятого элемента. Этот оператор суммирует пятый, шестой, седьмой и восьмой элементы массива. Следует отметить, что при третьем вызове функции передается другой адрес arr, отличный от того, что был передан в первых двух вызовах. Конечно же, вы могли бы использовать в качестве аргумента &cookies[4] вместо cookies + 4 — оба варианта означают одно и то же.

На память!

Чтобы указать вид массива и количество элементов для функции, обрабатывающей массив, вы передаете информацию в двух отдельных аргументах:

```
void fillArray(int arr[], int size); // прототип
```

Не пытайтесь передать размер массива в нотации квадратных скобок:

```
void fillArray(int arr[size]);       // НЕТ - плохой прототип
```

Дополнительные примеры функций с массивами

Когда вы решаете использовать массив для представления данных, то тем самым принимаете проектное решение. Однако проектные решения не должны ограничиваться тем, как данные хранятся, они также должны учитывать, как они используются. Иногда вы предпочтете написать специфические функции для выполнения специфических операций с данными. (Среди преимуществ такого подхода — повышение надежности программы, простота ее модификаций и облегчение процесса отладки.) Также когда, обдумывая программу, вы начинаете интеграцию средств хранения с операциями, то тем самым делаете важный шаг в сторону объектно-ориентированного образа мышления, что может принести существенные выгоды в будущем.

Рассмотрим простой случай. Предположим, что вы хотите использовать массив для отслеживания стоимости вашей недвижимости (предположим, что она у вас есть). Вы должны решить, какой тип данных для этого использовать. Конечно, double менее ограничен по диапазону допустимых значений, нежели int или long, и он представляет достаточно значимых разрядов, чтобы точно представлять значения. Далее вы должны решить, сколько понадобится элементов. (В случае динамических массивов, созданных с помощью new, это решение можно отложить, но пока

не будем усложнять.) Допустим, что у вас будет не более пяти единиц недвижимости, поэтому вы сможете использовать массив из пяти double.

Теперь рассмотрим, какие операции вы можете пожелать выполнить над этим массивом. Две самых главных — это чтение значений в массив и отображение его содержимого. Добавим в список еще одну операцию: переоценку стоимости объектов. Для простоты предположим, что ваши объекты растут или падают в цене синхронно. (Помните, что это книга по C++, а не по операциям с недвижимостью.) Далее разработаем функцию для каждой операции и затем напишем соответствующий код. Итак, сначала выполним все эти шаги по разработке частей программы, а потом соберем их в единое целое.

Наполнение массива

Поскольку функция с аргументом — именем массива обращается к исходному массиву, а не к копии, вы можете использовать вызов функции для присваивания значений его элементам. Одним аргументом такой функции будет имя наполняемого массива. В общем случае, программа может управлять инвестициями более чем одного лица, а потому в ней может быть более одного массива, следовательно, вы не захотите встраивать размер массива в саму функцию. Вместо этого вы передадите размер массива во втором аргументе, как это делалось в предыдущем примере. К тому же, возможно, что вы захотите прекратить чтение данных до того, как массив будет заполнен, поэтому такая возможность должна быть встроена в функцию. Так как вы можете ввести меньше, чем максимальное число элементов, имеет смысл сделать так, чтобы функция возвращала действительное количество введенных значений. Все эти соглашения приводят к следующему прототипу:

```
int fill_array(double ar[], int limit);
```

Функция принимает аргумент — имя массива и аргумент, указывающий максимальное количество элементов для чтения, а возвращает количество действительно введенных элементов. Например, если вы используете эту функцию с массивом из пяти элементов, то передаете во втором аргументе 5. Если затем вы вводите только три элемента, функция возвращает 3.

Вы можете воспользоваться циклом для чтения в массив следующих друг за другом значений, но как прервать этот цикл раньше? Одним из способов может быть применение специального значения для обозначения завершения ввода. Поскольку ни одно из вводимых значений не может быть отрицательным, то для указания конца ввода можно использовать отрицательное значение. К тому же функция должна как-то обрабатывать неправильный ввод, например, прекращая дальнейшие запросы значений. Учитывая все это, вы можете написать эту функцию следующим образом:

```
int fill_array(double ar[], int limit)
{
    using namespace std;
    double temp;
    int i;
    for (i = 0; i < limit; i++)
    {
        cout << "Введите значение #" << (i + 1) << ": ";
        cin >> temp;
```

```
      if (!cin) // неправильный ввод
      {
         cin.clear();
         while (cin.get() != '\n')
            continue;
         cout << "Неправильный ввод; процесс ввода прекращен.\n";
         break;
      }
      else if (temp < 0) // сигнал завершения
         break;
      ar[i] = temp;
   }
   return i;
}
```

Обратите внимание, что этот код включает приглашение пользователю. Если пользователь вводит неотрицательное значение, оно записывается в массив. В противном случае цикл прерывается. Если пользователь вводит только правильные значения, то цикл прекращается после чтения `limit` значений. Последнее, что делает цикл — увеличивает `i`, поэтому после завершения цикла `i` равно величине, на единицу большей, чем последний индекс массива, поэтому `i` равно количеству введенных элементов. Затем функция возвращает это значение.

Отображение массива и защита его посредством const

Построить функции для отображения содержимого массива очень просто. Вы передаете ей имя массива и количество заполненных элементов, а она использует цикл отображения каждого из них. Но есть еще одно обстоятельство — нужно гарантировать, что функция отображения не внесет в исходный массив никаких изменений. Если только назначение функции не предусматривает внесения изменений в переданные ей данные, вы должны каким-то образом предохранить ее от этого. Такая защита обеспечивается автоматически для обычных аргументов, потому что C++ передает их по значению, и функция имеет дело с копиями. Но функция, работающая с массивом, обращается к оригиналу. В конце концов, именно поэтому предыдущая функция, `fill_array()`, в состоянии выполнять свою работу. Чтобы предотвратить случайное изменение содержимого массива-аргумента, вы можете использовать ключевое слово `const` (описанное в главе 3), объявляя формальный аргумент:

```
void show_array(const double ar[], int n);
```

Это объявление устанавливает, что указатель `ar` указывает на константные данные. Это значит, что вы не можете применить `ar` для изменения данных. То есть вы можете обратиться к такому значению как к `ar[0]`, но не можете его изменять. Следует отметить, что это вовсе не означает, что исходный массив должен быть константным; это значит лишь, что вы не можете использовать `ar` внутри функции `show_array()` для изменения данных. Другими словами, `show_array()` трактует массив как данные, доступные только для чтения. Предположим, вы нечаянно нарушили это ограничение, попытавшись внутри функции `show_array()` сделать что-то вроде такого:

```
ar[0] += 10;
```

В этом случае компилятор пресечет ваши “преступные” действия. Например, Borland C++ выдаст сообщение об ошибке такого вида:

```
Cannot modify a const object in function
   show_array(const double *,int)
```

Другие компиляторы могут выразить свое неудовольствие иначе.

Сообщение подобного рода напомнит вам, что C++ интерпретирует `const double ar[]` как `const double *ar`. Таким образом, это объявление действительно говорит о том, что `ar` указывает на константное значение. Мы поговорим об этом подробнее, когда покончим с данным примером. А пока вот код функции `show_array()`:

```
void show_array(const double ar[], int n)
{
   using namespace std;
   for (int i = 0; i < n; i++)
   {
      cout << "Элемент #" << (i + 1) << ": $";
      cout << ar[i] << endl;
   }
}
```

Модификация массива

Третья операция с массивом в этом примере — умножение каждого элемента на один и тот же коэффициент. Для ее выполнения функции придется передавать три аргумента: коэффициент, массив и количество элементов. Возвращать какое-либо значение не требуется, поэтому функция будет выглядеть следующим образом:

```
void revalue(double r, double ar[], int n)
{
   for (int i = 0; i < n; i++)
      ar[i] *= r;
}
```

Поскольку эта функция предназначена для изменения элементов массива, вы не должны указывать слово `const` в объявлении `ar`.

Собираем все вместе

Теперь, когда вы определили данные в терминах их хранения (в виде массива), а также в терминах их использования (в трех функциях), вы можете собрать вместе программу, использующую этот дизайн. Поскольку все операции управления массивом уже построены, это значительно упрощает программирование `main()`. Большая часть оставшейся работы по программированию состоит из вызовов разработанных функций в теле `main()`. В листинге 7.7 показан результат. Директива `using` находится только в тех функциях, которые используют средства `iostream`.

Листинг 7.7. arrfun3.cpp

```
// arrfun3.cpp -- функция массивов и const
#include <iostream>
const int Max = 5;
```

```
// прототипы функций
int fill_array(double ar[], int limit);
void show_array(const double ar[], int n); // не изменяет данных
void revalue(double r, double ar[], int n);
int main()
{
  using namespace std;
  double properties[Max];
  int size = fill_array(properties, Max);
  show_array(properties, size);
  cout << "Введите коэффициент переоценки: ";
  double factor;
  cin >> factor;
  revalue(factor, properties, size);
  show_array(properties, size);
  cout << "Готово.\n";
  return 0;
}
int fill_array(double ar[], int limit)
{
    using namespace std;
    double temp;
    int i;
    for (i = 0; i < limit; i++)
    {
      cout << "Введите значение #" << (i + 1) << ": ";
      cin >> temp;
      if (!cin) // неправильный ввод
      {
        cin.clear();
        while (cin.get() != '\n')
        continue;
        cout << "Неправильный ввод; процесс ввода прекращен.\n";
        break;
      }
      else if (temp < 0) // сигнал завершения
        break;
      ar[i] = temp;
    }
    return i;
}
// следующая функция может использовать,
// но не изменять, массив по адресу ar
void show_array(const double ar[], int n)
{
  using namespace std;
  for (int i = 0; i < n; i++)
  {
    cout << "Элемент #" << (i + 1) << ": $";
    cout << ar[i] << endl;
  }
}
```

Ниже показан пример выполнения программы из листинга 7.7:

```
Введите значение #1: 100000
Введите значение #2: 80000
Введите значение #3: 222000
Введите значение #4: 240000
Введите значение #5: 118000
Элемент #1: $100000
Элемент #2: $80000
Элемент #3: $222000
Элемент #4: $240000
Элемент #5: $118000
Введите коэффициент переоценки: 1.10
Элемент #1: $110000
Элемент #2: $88000
Элемент #3: $244200
Элемент #4: $264000
Элемент #5: $129800
Готово.
Введите значение #1: 200000
Введите значение #2: 84000
Введите значение #3: 160000
Введите значение #4: -2
Элемент #1: $200000
Элемент #2: $84000
Элемент #3: $160000
Введите коэффициент переоценки: 1.20
Элемент #1: $240000
Элемент #2: $100800
Элемент #3: $192000
Готово.
```

Напомним, что `fill_array()` предполагает прекращение ввода, когда пользователь введет пять элементов либо укажет отрицательное значение — в зависимости от того, что случится раньше. Первый пример вывода иллюстрирует достижение предела по количеству элементов, а второй — прекращение приема значений по вводу отрицательной величины.

Замечания по программе

Мы уже обсудили важные детали программирования примера, поэтому обратимся к процессу в целом. Мы начали с обдумывания типа данных и разработали соответствующий набор функций для их обработки. Затем мы встроили эти функции в программу. Это то, что иногда называют *программированием снизу вверх* (*восходящим программированием*), поскольку процесс дизайна идет от частей-компонентов к целому. Этот подход хорошо стыкуется с объектно-ориентированным программированием (ООП), которое сосредоточено в первую очередь на данных и манипуляциях ими. Традиционное процедурное программирование, с другой стороны, следует парадигме программирования *сверху вниз* (*нисходящей*), когда вы сначала разрабатываете укрупненный модульный дизайн, а затем обращаете свое внимание на детали. Оба метода полезны, и оба ведут к модульным программам.

Функции, работающие с диапазонами массивов

Как вы уже видели, функции С++, обрабатывающие массивы, нуждаются в информации относительно типа данных, хранящихся в массиве, местоположения его начала и количества его элементов. Традиционный подход С/С++ к функциям, обрабатывающим массивы, состоит в передаче указателя на начало массива в одном аргументе и размера массива — в другом. (Указатель сообщает функции сразу и то, где искать массив, и тип его элементов.) Это предоставляет функции исчерпывающую информацию, необходимую для нахождения данных.

Существует и другой подход к тому, как передать функции нужную информацию — указать *диапазон* элементов. Это можно сделать, передав два указателя — один, идентифицирующий начальный элемент массива, и второй, указывающий его конец. Стандартная библиотека шаблонов С++ (STL, представлена в главе 16), например, обобщает такой подход с применением диапазона. Подход STL использует концепцию "следующий после конца" для указания границы диапазона. То есть в случае массива аргумент, идентифицирующий конец массива, должен быть указателем, установленным на адрес, следующий сразу за последним элементом. Например, предположим, что имеется такое объявление:

```
double elbuod[20];
```

Тогда два указателя, `elbuod` и `elbuod + 20`, определяют диапазон. Первый, `elbuod` — это имя массива; он указывает на первый элемент. Выражение `elbuod + 19` указывает на последний элемент (то есть `elbuod[19]`), поэтому `elbuod + 20` указывает на элемент, следующий сразу за последним. Передавая функции диапазон, вы сообщаете ей, какие элементы следует обрабатывать. В листинге 7.8 модифицируется код листинга 7.6 с использованием двух указателей для задания диапазона.

Листинг 7.8. arrfun4.cpp

```
// arrfun4.cpp -- функция с диапазоном массива
#include <iostream>
const int ArSize = 8;
int sum_arr(const int * begin, const int * end);
int main()
{
   using namespace std;
   int cookies[ArSize] = {1,2,4,8,16,32,64,128};
// некоторые системы требуют предварить int словом static,
// чтобы разрешить инициализацию массива
   int sum = sum_arr(cookies, cookies + ArSize);
   cout << "Всего съедено печенья: " << sum << endl;
   sum = sum_arr(cookies, cookies + 3);       // первые 3 элемент
   std::cout << "Первые три едока умяли " << sum << " печений. \n";
   sum = sum_arr(cookies + 4, cookies + 8);   // последние 4 элемента
   std::cout << "Последние пять едоков схрумкали " << sum << " печений. \n";
   return 0;
}
// возвращает сумму элементов целочисленного массива
int sum_arr(const int * begin, const int * end)
{
   const int * pt;
```

```
    int total = 0;
    for (pt = begin; pt != end; pt++)
        total = total + *pt;
    return total;
}
```

Ниже показан пример вывода программы из листинга 7.8:

```
Всего съедено печенья: 255
Первые три едока умяли 7 печений.
Последние пять едоков схрумкали 240 печений.
```

Замечания по программе

В листинге 7.8 обратите внимание на цикл внутри функции `sum_array()`:

```
for (pt = begin; pt != end; pt++)
    total = total + *pt;
```

Здесь указатель pt устанавливается на первый обрабатываемый элемент (тот, что задан в begin) и прибавляет *pt (значение самого элемента) к общей сумме total. Затем цикл обновляет pt, увеличивая его на единицу, после чего он указывает на следующий элемент. Процесс продолжается до тех пор, пока pt != end. Когда pt, наконец, становится равен end, он указывает на позицию, следующую за последним элементом диапазона, поэтому цикл завершается.

Второе, что следует отметить — это то, как разные вызовы функций задают различные диапазоны в пределах массива:

```
int sum = sum_arr(cookies, cookies + ArSize);
...
sum = sum_arr(cookies, cookies + 3);      // первые 3 элемента
...
sum = sum_arr(cookies + 4, cookies + 8); // последние 4 элемента
```

Значение cookies + ArSize указывает на позицию, следующую за последним элементом массива. (Размер массива равен ArSize, потому cookies[ArSize-1] — последний элемент, а его адрес — cookies + ArSize - 1.) Поэтому диапазон cookies, cookies + ArSize специфицирует весь массив. Аналогично cookies, cookies + 3 означает первые три элемента и так далее.

Кстати, обратите внимание, что правило вычитания указателей предполагает, что в sum_arr() выражение end - begin возвращает количество элементов в диапазоне.

Указатели и const

Использование ключевого слова const с указателями имеет некоторые тонкие аспекты (с указателями всегда связаны тонкие аспекты), поэтому присмотримся к нему повнимательнее. Вы можете применять ключевое слово const с указателями двумя разными способами. Первый — заставить указатель указывать на константный объект, тем самым предотвращая модификацию объекта через указатель. Второй способ — сделать сам указатель константным, запретив переустанавливать его на что-нибудь другое. Теперь обратимся к деталям.

Во-первых, объявим pt как указатель на константу:

```
int age = 39;
const int * pt = &age;
```

Это объявление устанавливает, что pt указывает на const int (в данном случае — 39). Таким образом, вы не сможете использовать pt для изменения этого значения. Другими словами, значение *pt является константным и не может быть изменено:

```
*pt += 1;       // НЕВЕРНО, потому что pt указывает на const int
cin >> *pt;     // НЕВЕРНО по той же причине
```

Теперь проанализируем тонкие моменты. Такое объявление pt не обязательно значит, что значение, на которое он указывает, действительно является константой; это значит лишь, что значение постоянно, только если к нему обращаться через pt. Например, pt указывает на age, а age — не константа. Вы можете изменить значение age непосредственно, используя переменную age, но вы не можете изменить это значение через указатель pt:

```
*pt = 20;       // НЕПРАВИЛЬНО, потому что pt указывает на const int
age = 20;       // ПРАВИЛЬНО, потому что age не объявлено const
```

До сих пор вы присваивали адрес обычной переменной обычному указателю. Теперь вы присваиваете адрес обычной переменной указателю на константу. Это оставляет две другие возможности: присваивание адреса константной переменной указателю на константу и присваивание адреса константной переменной обычному указателю. Возможно ли то и другое? Первое — да, второе — нет:

```
const float g_earth = 9.80;
const float * pe = &g_earth; // ПРАВИЛЬНО
const float g_moon = 1.63;
float * pm = &g_moon;        // НЕПРАВИЛЬНО
```

В первом случае вы не сможете использовать ни g_earth, ни pe для изменения значения 9.8. Второй случай в C++ не допускается по простой причине: если вы можете присвоить адрес g_moon указателю pm, то можете смошенничать и применить pm для изменения g_moon. Это сводит на нет константный статус g_moon, поэтому C++ запрещает присваивание адреса константной переменной не константному указателю. (При крайней необходимости вы можете использовать приведение типа для преодоления подобного ограничения; см. в главе 15 дискуссию об операторе const_cast.)

Ситуация становится немного более сложной, когда вы имеете дело с указателями на указатели. Как вы видели ранее, присваивание не константного указателя константному разрешается, если вы имеете дело только с одним уровнем хитрости:

```
int age = 39;           // age++ допустимая операция
int * pd = &age;        // *pd = 41 допустимая операция
const int * pt = pd;    // *pt = 42 недопустимая операция
```

Однако присваивания указателей, которые смешивают константы и не константы в такой манере, становятся небезопасными, когда это делается в два "слоя" и более. Если смешивание констант и не констант было разрешено, вы могли бы написать что-нибудь вроде такого:

```
const int **pp2;
int *p1;
const int n = 13;
pp2 = &p1;  // не разрешено, но предположим иначе
*pp2 = &n;  // правильно, оба const, но устанавливает в p1 указатель на n
*p1 = 10;   // правильно, но изменяет const n
```

Здесь код присваивает не константный адрес (&p1) константному указателю (pp2), что позволяет p1 применяться для изменения константных данных. Таким образом, правило, гласящее, что вы можете присваивать не константный адрес или указатель константному указателю, работает только в том случае, когда есть лишь один уровень смешения — например, если указатель указывает на базовый тип данных.

На память!

Вы можете присваивать адрес как константных, так и не константных данных указателю на константу, предполагая, что эти данные сами не являются указателем, но вы можете присвоить адрес не константных данных только не константному указателю.

Предположим, у вас есть массив константных данных:

```
const int months[12] = {31,28,31,30,31,30, 31, 31,30,31,30,31};
```

Запрет на присваивание адреса константного массива означает, что вы не можете передать имя такого массива в качестве аргумента функции, используя не константный формальный аргумент:

```
int sum(int arr[], int n);   // должен быть const int arr[]
...
int j = sum(months, 12);     // не допускается
```

Этот вызов функции пытается присвоить const-указатель (months) не константному указателю (arr), и компилятор не пропускает такой вызов.

Используйте const, где это возможно

Существуют две серьезных причины объявлять аргументы-указатели указателями на константные данные:

- Это защищает вас от программных ошибок, которые могут нечаянно изменить данные.
- Использование const позволяет функции обрабатывать как константные, так и не константные аргументы, в то время как функции, которые пропускают const в прототипе, могут принимать только не константные данные.

Вы должны объявить формальные аргументы-указатели как указатели на const, где это только возможно.

Касательно еще одного тонкого момента рассмотрим следующее объявление:

```
int age = 39;
const int * pt = &age;
```

const во втором объявлении только предотвращает изменение того значения, на которое указывает pt, в данном случае 39. Это не предотвращает изменение самого pt. То есть вы вполне можете присвоить ему другой адрес:

```
int sage = 80;
pt = &sage;    // может указывать на другое место
```

Но вы по-прежнему не сможете использовать pt для изменения того значения, на которое он указывает.

Второй способ применения const делает невозможным изменение самого указателя:

```
int sloth = 3;
const int * ps = &sloth;        // указатель на const int
int * const finger = &sloth;    // const-указатель на int
```

Обратите внимание, что последнее объявление переустанавливает ключевое слово const. Это объявление ограничивает finger тем, что он может указывать только на sloth и ни на что другое. Однако оно позволяет применить finger для изменения значения самого sloth. Второе из трех приведенных объявлений не разрешает применять ps для изменения значения sloth, но разрешает ps указывать на другое место памяти. Короче говоря, finger и *ps — оба являются константами, а *finger и ps — нет. (См. рис. 7.5.)

Если хотите, можете объявить константный указатель на константный объект:

```
double trouble = 2.0E30;
const double * const stick = &trouble;
```

Здесь stick может указывать только на trouble, и stick не может применяться для изменения значения trouble. Короче говоря, и stick, и *stick являются const.

```
int gorp = 16;
int chips = 12;
const int * p_snack = &gorp;
```

*p_snack = 20 — NO

Не позволяет изменение значения, на которое указывает p_snack

p_snack = &chips; — OK

p_snack может указывать на другую переменную

```
int gorp = 16;
int chips = 12;
int * const p_snack = &gorp;
```

*p_snack = 20; — OK

p_snack может быть использован для изменения значения

p_snack = &chips; — NO

Не позволяет изменять переменную, на которую указывает p_snack

Рис. 7.5. Указатели на константы и константные указатели

Обычно вы используете форму указателя на const, чтобы защитить данные, когда передаете указатель в качестве аргумента функции. Например, вспомните прототип show_array() из листинга 7.5:

```
void show_array(const double ar[], int n);
```

Применение const в этом объявлении означает, что функция show_array() не может изменять никакие значения в переданном ей массиве. Эта техника работает до тех пор, пока есть хоть один уровень обхода. Здесь, например, элементы массива относятся к базовому типу, но если бы они были указателями на указатели на указатели, вы не могли бы использовать const.

Функции и двумерные массивы

Чтобы написать функцию, которая принимает в качестве аргумента двумерный массив, вам необходимо помнить, что имя массива трактуется как его адрес, поэтому соответствующий формальный параметр является указателем — так же, как и в случае одномерного массива. Сложность заключается в том, чтобы правильно объявить указатель. Предположим, например, что вы начинаете с такого кода:

```
int data[3][4] = {{1,2,3,4}, {9,8,7,6}, {2,4,6,8}};
int total = sum(data, 3);
```

Как должен выглядеть прототип sum()? И почему функция передает количество строк (3), но не передает количество столбцов (4)?

Так, data — имя массива из трех элементов. Первый элемент сам по себе является массивом их четырех значений типа int. То есть, тип data — “указатель на массив из четырех int”, поэтому соответствующий прототип должен быть таким:

```
int sum(int (*ar2)[4], int size);
```

Скобки необходимы, потому что объявление

```
int *ar2[4]
```

определило бы массив их четырех указателей на int вместо одного указателя на массив из четырех int, а параметр функции не может быть массивом. Существует альтернативный формат, который означает в точности то же самое, что и первый прототип, но, возможно, является более простым для чтения:

```
int sum(int ar2[][4], int size);
```

Любой из двух прототипов устанавливает, что ar2 — указатель, а не массив. Также обратите внимание, что тип указателя явно говорит о том, что он указывает на массив из четырех int. Таким образом, тип указателя специфицирует количество столбцов — вот почему количество столбцов не передается в отдельном аргументе функции.

Поскольку тип указателя специфицирует количество столбцов, функция sum() работает только с массивами из четырех столбцов. Однако число строк задается переменной size, поэтому sum() может работать с произвольным количеством строк:

```
int a[100][4];
int b[6][4];
...
```

```
int total1 = sum(a, 100);    // сумма всех элементов a
int total2 = sum(b, 6);      // сумма всех элементов b
int total3 = sum(a, 10);     // сумма первых 10 строк a
int total4 = sum(a+10, 20);  // сумма следующих 20 строк a
```

Зная, что ar2 — указатель на массив, как вы можете использовать его в определении функции? Простейший способ — использовать ar2 как имя двумерного массива. Вот возможный вариант определения функции:

```
int sum(int ar2[][4], int size)
{
   int total = 0;
   for (int r = 0; r < size; r++)
      for (int c = 0; c < 4; c++)
         total += ar2[r][c];
   return total;
}
```

Еще раз отметим, что число строк передается в параметре size, но количество столбцов — фиксировано и равно 4, как в объявлении ar2, так и во вложенном цикле.

Вот почему вы можете использовать нотацию массива. Поскольку ar2 указывает на первый элемент (элемент 0) массива, чьи элементы являются массивами из четырех int, то выражение ar2+r указывает на элемент номер r. Таким образом, ar2[r] — это элемент номер r. Этот элемент сам по себе является массивом из четырех int, поэтому ar2[r] — имя этого массива из четырех int. Применение индекса к имени массива дает нам его элемент, поэтому ar2[r][c] — элемент массива из четырех int, то есть отдельное значение типа int. Указатель ar2 должен быть разыменован дважды, чтобы получить данные. Простейший способ сделать это — дважды использовать квадратные скобки — как в ar2[r][c]. Если это неудобно, можно использовать дважды операцию *:

```
ar2[r][c] == *(*(ar2 + r) + c) // одно и то же
```

Чтобы понять это, нужно должны разобрать выражение по частям, начиная изнутри:

```
ar2         // указатель на первую строку — массив из 4 int
ar2 + r     // указатель на строку r (массив из 4 int)
*(ar2 + r)  // строка r (массив из 4 int, то есть имя массива,
            // то есть указатель на первый int в строке, то есть - ar2[r]
*(ar2 +r) + c     // указатель на элемент int под номером c в строке r,
                  // то есть, ar2[r] + c
*(*(ar2 + r) + c  //значение int под номером c в строке r, то есть, ar2[r][c]
```

Кстати, код sum() не использует const в объявлении параметра ar2, потому что эта техника предназначена для указателей на базовые типы, а ar2 — это указатель на указатель.

Функции и строки в стиле С

Вспомните, что строки в стиле С состоят из последовательностей символов, ограниченных нулевым символом. Большая часть того, что вы изучили о проектировании функций массивов, также касается функций, обрабатывающих строки.

Например, передача строки как аргумента означает передачу ее адреса, и вы можете использовать const для защиты содержимого строки от нежелательных изменений. Однако существует несколько особенностей, связанных со строками, о которых мы поговорим сейчас.

Функции с аргументами — строками в стиле С

Предположим, что вы хотите передать строку функции в виде аргумента. У вас есть три варианта представления строки:

- Массив char.
- Константная строка в двойных кавычках (также называемая *строковым литералом*).
- Указатель на char, содержащий адрес начала строки.

Все три варианта, однако, имеют тип указателя на char (или, короче, тип char *), поэтому вы можете использовать все три в качестве аргументов функций, обрабатывающих строки:

```
char ghost[15] = "galloping";
char * str = "galumphing";
int n1 = strlen(ghost);         // ghost есть &ghost[0]
int n2 = strlen(str);           // указатель на char
int n3 = strlen("gamboling");   // адрес строки
```

Неформально вы можете сказать, что вы передаете строку как аргумент, но на самом деле вы передаете адрес ее первого символа. Это подразумевает, что прототип строковой функции должен использовать char * как тип формального параметра, представляющего строку.

Одно существенное отличие между строкой в стиле С и обычным массивом состоит в том, что строка имеет встроенный ограничивающий нулевой символ. (Вспомним, что массив char, который содержит символы, но не содержит нулевой символ — это просто массив, а не строка.) Это значит, что вы не должны передавать размер строки в качестве аргумента. Вместо этого функция может использовать цикл для поочередного чтения каждого символа строки до тех пор, пока цикл не достигнет ограничивающего нулевого символа. Код в листинге 7.9 иллюстрирует этот подход для функции, выполняющей подсчет количества появлений определенного символа в строке.

Листинг 7.9. strgfun.cpp

```
// strgfun.cpp -- функция со строковым аргументом
#include <iostream>
int c_in_str(const char * str, char ch);
int main()
{
    using namespace std;
    char mmm[15] = "minimum"; // строка в массиве
```

```
// некоторые системы требуют предварить int словом static,
// чтобы разрешить инициализацию массива
   char *wail = "ululate"; // ожидает указания на строку
   int ms = c_in_str(mmm, 'm');
   int us = c_in_str(wail, 'u');
   cout << ms << " m символов в " << mmm << endl;
   cout << us << " u символов в " << wail << endl;
   return 0;
}
// эта функция считает количество символов ch
// в строке str
int c_in_str(const char * str, char ch)
{
   int count = 0;
   while (*str)  // завершить, когда *str равно '\0'
   {
     if (*str == ch)
        count++;
     str++;      // переместить указатель на следующих символ
   }
   return count;
}
```

Вывод программы из листинга 7.9 выглядит следующим образом:

```
3 m символов в minimum
2 u символов в ululate
```

Замечания по программе

Поскольку функция `c_int_str()` в листинге 7.9 не должна изменять исходную строку, она использует модификатор `const` в объявлении формального параметра str. После этого, если вы ошибочно позволите функции изменить часть строки, компилятор перехватит эту ошибку. Конечно, вы можетеприменять нотацию массива для объявления `str` в заголовке функции:

```
int c_in_str(const char str[], char ch) // также нормально
```

Однако использование нотации указателя напоминает вам, что аргумент не должен быть именем массива, а должен быть некоторой другой формой указателя.

Сама функция демонстрирует стандартный способ обработки символов в строке:

```
while (*str)
{
   statements
   str++;
}
```

Изначально `str` указывает на первый символ строки, поэтому `*str` представляет сам первый символ. Например, немедленно после первого вызова функции `*str` имеет значение `m` — первый символ в `minimum`. До тех пор, пока символ не является нулевым (`\0`), `*str` не равно нулю и цикл продолжается. В конце каждого шага цикла выражение `str++` увеличивает указатель на 1 байт, так что он указывает на следую-

щий символ в строке. В конечном итоге `str` указывает на нулевой символ, что приравнивает `*str` к нулю, и цикл завершается.

Функции, возвращающие строки в стиле C

Теперь предположим, что вам нужно написать функцию, возвращающую строку. Конечно, функция может это сделать. Но она может вернуть адрес строки, и это наиболее эффективно. Например, листинг 7.10 определяет функцию `buildstr()`, возвращающую указатель. Эта функция принимает два аргумента: символ и число. Используя `new`, она создает строку, чья длина равна переданному числу, и инициализирует каждый ее элемент значением переданного символа. Затем она возвращает указатель на эту новую строку.

Листинг 7.10. strgback.cpp

```
// strgback.cpp -- функция, возвращающая указатель на char
#include <iostream>
char * buildstr(char c, int n); // прототип
int main()
{
   using namespace std;
   int times;
   char ch;
   cout << "Введите символ: ";
   cin >> ch;
   cout << "Введите число: ";
   cin >> times;
   char *ps = buildstr(ch, times);
   cout << ps << endl;
   delete [] ps;              // освободить память
   ps = buildstr('+', 20);    // повторно использовать указатель
   cout << ps << "-DONE-" << ps << endl;
   delete [] ps;             // освободить память
   return 0;
}
// эта функция строит строку из n символов c
char * buildstr(char c, int n)
{
   char * pstr = new char[n + 1];
   pstr[n] = '\0';           // ограничить строку
   while (n-- > 0)
     pstr[n] = c;            // заполнить строку
   return pstr;
}
```

Ниже показан пример запуска программы из листинга 7.10:

```
Введите символ: V
Введите число: 46
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
++++++++++++++++++++-DONE-++++++++++++++++++++
```

Замечания по программе

Чтобы создать строку в n видимых символов, вам понадобится разместить n + 1 символов, чтобы вместить нулевой символ-ограничитель. Поэтому функция в листинге 7.10 запрашивает n + 1 байт для размещения строки. Далее она устанавливает в последний байт нулевой символ. После этого наполняет массив от конца к началу. В листинге 7.10 цикл

```
while (n-- > 0)
   pstr[n] = c;
```

выполняется n раз — уменьшая n до 0 и заполняя n элементов. В начале последнего шага n имеет значение 1. Поскольку n-- означает использовать значение, а потом уменьшить его, условие цикла while проверяет 1 на равенство 0, возвращает true, и цикл продолжается. Но после выполнения этой проверки функция уменьшает n до 0, поэтому pstr[0] — последний элемент, которому присваивается c. Причина наполнения строки от конца к началу вместо того, чтоб наполнять от начала к концу, связана с желанием избежать применения дополнительной переменной. Использование противоположного порядка потребовало бы чего-то вроде такого:

```
int i = 0;
while (i < n)
   pstr[i++] = c;
```

Обратите внимание, что переменная pstr является локальной по отношению к функции buildstr(), поэтому, когда эта функция завершается, память, занятая pstr (но не самой строкой), освобождается. Но поскольку функция возвращает значение pstr, программа имеет возможность получить доступ к новой строке через указатель ps в main().

Программа из листинга 7.10 применяет delete, чтобы освободить память, когда необходимость в строке отпадает. Затем она использует повторно ps, чтобы указать на новый блок памяти, полученный для следующей строки, и снова освобождает ее. Недостаток такого дизайна (функция, возвращающая указатель на память, выделенную операцией new) состоит в том, что он возлагает на программиста ответственность за вызов delete. В главе 12 вы увидите, как классы C++, используя конструкторы и деструкторы, могут самостоятельно позаботиться об этих деталях.

Функции и структуры

Теперь давайте перейдем от массивов к структурам. Написать функцию, использующую структуры, гораздо легче, чем функцию для массивов. Хотя структурные переменные подобны массивам в том, что и те и другие могут содержать несколько элементов данных, когда речь идет об их применении в функциях, структурные переменные ведут себя как базовые переменные, имеющие единственное значение. То есть, в отличие от массивов, структуры связывают свои значения в единое целое, которое может быть воспринято как одно значение. Вспомните, что одну структуру можно присваивать другой. Аналогично вы можете передавать структуры по значению, как это делается с обычными переменными. В этом случае функция работает с копией исходной структуры. Здесь нет никаких трюков вроде того, что имя массива трактуется как указатель на его первый элемент. Имя структуры — это просто имя

структуры, и если вам нужен ее адрес, вы можете получить его с помощью операции &. (Как C, так и C++ применяют & в качестве операции взятия адреса, но C++ еще дополнительно использует этот символ для идентификации ссылочных переменных, как сказано в главе 8.)

Самый прямой способ использования структуры в программе — это трактовать их так же, как вы трактуете обычные базовые типы — то есть, передавать в виде аргументов и если нужно, использовать их в качестве возвращаемых значений. Однако существует один недостаток в передаче структур по значению. Если структура велика, пространство и усилия, необходимые для создания копии структуры, могут значительно увеличить потребности в памяти и замедлить работу системы. По этой причине (и еще потому, что в начале C не позволял передавать структуры по значению), многие программисты на C предпочитают передавать адрес структуры и затем использовать указатель для доступа к ее содержимому. C++ предлагает третью альтернативу, которая называется *передачей по ссылке* и обсуждается в главе 8. Давайте сейчас рассмотрим два первых варианта, начиная с передачи и возврата целых структур.

Передача и возврат структур

Передача структур по значению имеет наибольший смысл, когда структура относительно компактна, поэтому давайте рассмотрим несколько примеров, демонстрирующих такой подход. Первый пример имеет дело со временем путешествия (не путать с путешествиями во времени). Некоторые карты сообщают вам, что для того, чтобы доехать из Thunder Falls до Bingo City, нужно потратить 3 часа и 50 минут, а чтобы доехать от Bingo City до Grotesquo — 1 час и 25 минут. Вы можете использовать структуры, чтобы представить эти периоды времени, используя в них один член для представления часов, а другой — для минут. Сложение двух периодов будет не простым, потому что придется преобразовывать минуты в часы, когда их сумма превышает 60. Например, два периода времени на дорогу, приведенные выше, дают в сумме 4 часа и 75 минут, что должно быть преобразовано в 5 часов и 15 минут. Давайте разработаем структуру для представления значений времени, а затем функцию, которая будет принимать две таких структуры в виде аргументов и возвращать структуру, представляющую их сумму.

Определить структуру просто:

```
struct travel_time
{
   int hours;
   int mins;
};
```

Далее рассмотрим прототип для функции `sum()`, который вернет сумму двух таких структур. Возвращаемое значение должно иметь тип `travel_time`, как и два ее аргумента. Таким образом, прототип должен выглядеть следующим образом:

```
travel_time sum(travel_time t1, travel_time t2);
```

Чтобы сложить два периода времени, сначала необходимо сложить минуты. Целочисленное деление на 60 даст количество часов, в которые сложатся минуты, а операция модуля (`%`) даст оставшиеся минуты. В листинге 7.11 этот подход воплощен в функции `sum()`; еще одна функция, `show_time()`, служит для отображения содержимого структуры `travel_time`.

Листинг 7.11. travel.cpp

```
// travel.cpp -- применение структур с функциями
#include <iostream>
struct travel_time
{
   int hours;
   int mins;
};
const int Mins_per_hr = 60;
travel_time sum(travel_time t1, travel_time t2);
void show_time(travel_time t);
int main()
{
   using namespace std;
   travel_time day1 = {5, 45}; // 5 часов и 45 минут
   travel_time day2 = {4, 55}; // 4 часа и 55 минут
   travel_time trip = sum(day1, day2);
   cout << "Всего за два дня: ";
   show_time(trip);
   travel_time day3= {4, 32};
   cout << "Всего за три дня: ";
   show_time(sum(trip, day3));
   return 0;
}
travel_time sum(travel_time t1, travel_time t2)
{
   travel_time total;
   total.mins = (t1.mins + t2.mins) % Mins_per_hr;
   total.hours = t1.hours + t2.hours +
     (t1.mins + t2.mins) / Mins_per_hr;
   return total;
}
void show_time(travel_time t)
{
   using namespace std;
   cout << t.hours << " часов и "
        << t.mins << " минут\n";
}
```

Здесь `travel_time` выступает как имя обычного стандартного типа; вы можете использовать его для объявления переменных, типа возврата функций и типа аргументов функций. Поскольку такие переменные, как `total` и `t1`, являются структурами `travel_time`, вы можете применять к ним операцию точки, чтобы обращаться к членам. Обратите внимание, что поскольку функция `sum()` возвращает структуру `travel_time`, вы можете использовать ее в качестве аргумента функции `show_time()`. А так как функции C++ по умолчанию передают аргументы по значению, то вызов `show_time(sum(trip, day3))` сначала вычислит `sum(trip, day3)`, чтобы получить ее возвращаемое значение. Затем это возвращенное значение (а не сама функция) передается `show_time()`. Ниже показан вывод программы из листинга 7.11:

```
Всего за два дня: 10 часов и 40 минут
Всего за три дня: 15 часов и 12 минут
```

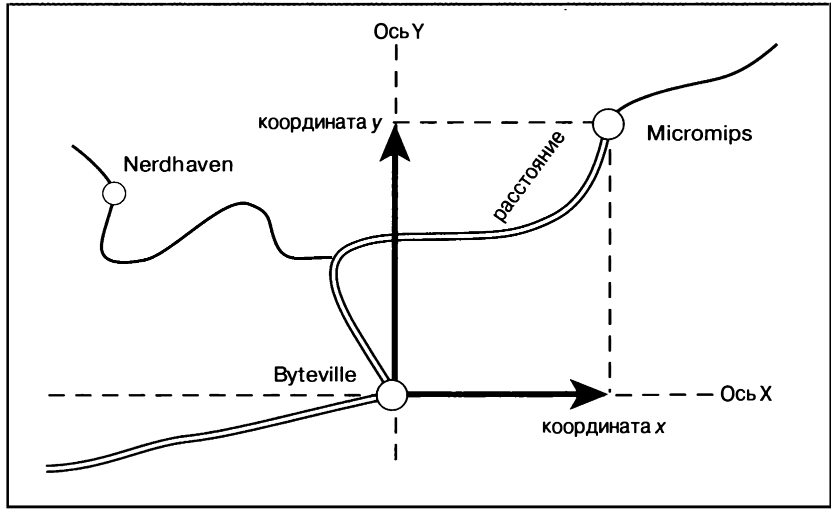
Еще один пример использования функций со структурами

Большая часть того, что вы узнали о функциях и структурах C++ в той же мере касается классов C++, поэтому нам стоит рассмотреть второй пример. На этот раз мы будем иметь дело с пространством вместо времени. В частности, в этом примере мы определим две структуры, представляющие два разных способа описания координат на плоскости, и затем разработаем функции для преобразования одной формы в другую и отображения результата. В этом примере будет больше математики, чем в предыдущем, но вам не обязательно изучать математику, постигая C++.

Предположим, что вы хотите описать положение точки на экране или местоположение на карте относительно некоторой начальной точки. Одним из способов является отсчет горизонтального и вертикального смещений точки от начала координат. Традиционно математики используют символ x для представления горизонтального смещения и y – для вертикального. (См. рис. 7.6.) Вместе x и y представляют *прямоугольные координаты*. Вы можете определить структуру, состоящую из двух координат, чтобы представить позицию:

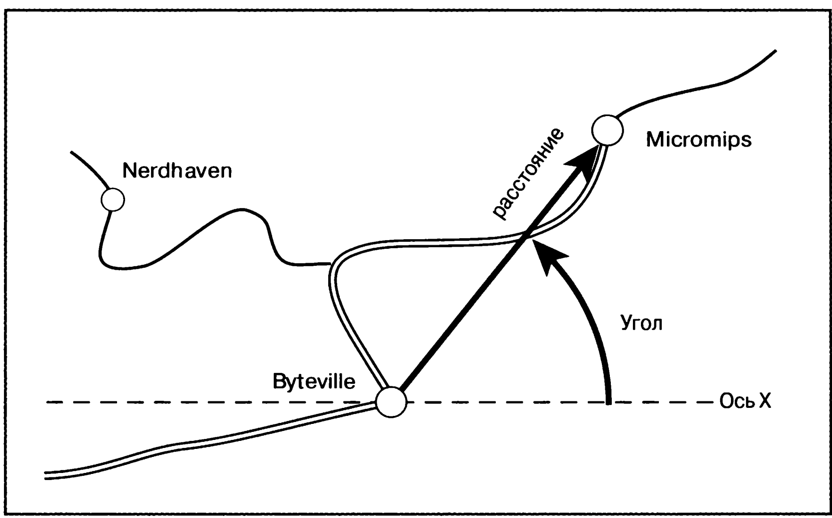
```
struct rect
{
   double x; // расстояние по горизонтали от начальной точки
   double y; // расстояние по вертикали от начальной точки
};
```

Второй способ описания позиции точки – это указать ее расстояние от начала координат и направление (например, 40 градусов на север от восточного направления). Традиционно математики измеряют угол против часовой стрелки от положительной горизонтальной оси (рис. 7.7).



Прямоугольные координаты Micromips относительно Byteville

Рис. 7.6. Прямоугольные координаты



Полярные координаты Micromips относительно Byteville

Рис. 7.7. Полярные координаты

Расстояние и угол вместе составляют полярные координаты. Вы можете определить вторую структуру для такого представления позиции:

```
struct polar
{
   double distance;    // расстояние от исходной точки
   double angle;       // направление
};
```

Теперь давайте спроектируем функцию, которая будет отображать содержимое структуры типа `polar`. Математические функции в библиотеке C++ измеряют углы в радианах, поэтому мы тоже будем измерять углы в этих величинах. Но для более удобного отображения вы можете преобразовать радианы в градусы. Это означает умножение на $180/\pi$, что примерно составляет 57.29577951. Вот эта функция:

```
// отображение полярных координат с преобразованием радиан в градусы
void show_polar (polar dapos)
{
   using namespace std;
   const double Rad_to_deg = 57.29577951;
   cout << "расстояние = " << dapos.distance;
   cout << ", угол = " << dapos.angle * Rad_to_deg;
   cout << " градусов\n";
}
```

Обратите внимание, что формальная переменная-аргумент имеет тип `polar`. Когда вы передаете структуру `polar` этой функции, ее содержимое копируется в структуру `dapos`, и функция использует эту копию в своей работе. Поскольку `dapos` — структура, функция использует операцию членства (точку) для идентификации ее членов (см. главу 4).

Далее давайте попробуем написать функцию, которая преобразует прямоугольные координаты в полярные. Эта функция должна будет принимать в виде аргумента структуру rect и возвращать вызывающей функции структуру polar. Для выполнения необходимых вычислений понадобятся функции из библиотеки math, поэтому программа должна будет включить заголовочный файл math.h. Кроме того, в некоторых системах вы должны сообщить компилятору, что он должен загрузить библиотеку math (см. главу 1). Вы можете воспользоваться теоремой Пифагора, чтобы получить расстояние по двум координатам:

```
distance = sqrt( x * x + y * y)
```

Функция atan2() из библиотеки math вычисляет угол по значениям x и y:

```
angle = atan2(y, x)
```

(Существует также функция atan(), но она не различает углы 180 градусов с разными знаками.)

Имея эти формулы, вы можете написать функцию преобразования координат следующим образом:

```
// преобразование прямоугольных координат в полярные
polar rect_to_polar(rect xypos) // тип polar
{
   polar answer;
   answer.distance =
      sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
   answer.angle = atan2(xypos.y, xypos.x);
   return answer; // возвращает структуру polar
}
```

Теперь, когда центральная функция готова, написание остальной части программы не составляет особого труда. В листинге 7.12 показан результирующий код.

Листинг 7.12. strctfun.cpp

```
// strctfun.cpp -- функции с аргументами-структурами
#include <iostream>
#include <cmath>
// объявления структур
struct polar
{
   double distance;    // расстояние от исходной точки
   double angle;       // направление
};
struct rect
{
   double x;           // расстояние по горизонтали от начальной точки
   double y;           // расстояние по вертикали от начальной точки
};
// прототипы
polar rect_to_polar(rect xypos);
void show_polar(polar dapos);
int main()
{
   using namespace std;
```

```
    rect rplace;
    polar pplace;
    cout << "Введите значения координат x и y: ";
    while (cin >> rplace.x >> rplace.y) // хитрое применение cin
    {
      pplace = rect_to_polar(rplace);
      show_polar(pplace);
      cout << "Следующие два числа (q для выхода): ";
    }
    cout << "Готово.\n";
    return 0;
}
// преобразование прямоугольных координат в полярные
polar rect_to_polar(rect xypos)   // тип polar
{
    polar answer;
    answer.distance =
        sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
    answer.angle = atan2(xypos.y, xypos.x);
    return answer;                 // возвращает структуру polar
}
// отображение полярных координат с преобразованием  радиан в градусы
void show_polar (polar dapos)
{
    using namespace std;
    const double Rad_to_deg = 57.29577951;
    cout << "расстояние = " << dapos.distance;
    cout << ", угол = " << dapos.angle * Rad_to_deg;
    cout << " градусов\n";
}
```

Замечание по совместимости

Некоторые реализации C++ используют `math.h` вместо более нового заголовочного файла `cmath`. Некоторые компиляторы требуют явных инструкций для нахождения математической библиотеки. Например, старые версии g++ требуют следующей командной строки:

```
g++ structfun.C -lm
```

Ниже показан пример выполнения программы из листинга 7.12:

```
Введите значения координат x и y: 30 40
расстояние = 50, angle = 53.1301 degrees
Следующие два числа (q для выхода): -100 100
расстояние = 141.421, угол = 135 градусов
Следующие два числа (q для выхода): q
```

Замечания по программе

Мы уже обсудили две функции из листинга 7.12, поэтому давайте разберемся, как программа использует `cin` для управления циклом `while`:

```
while (cin >> rplace.x >> rplace.y)
```

Вспомним, что cin — объект класса istream. Операция извлечения (>>) спроектирована так, что выражение cin >> rplace.x также является объектом этого типа. Как вы увидите в главе 11, операции классов реализованы функциями. Что в действительности происходит, когда вы используете cin >> rplace.x? Программа вызывает функцию, которая возвращает переменную типа istream. Если вы применяете операцию извлечения к объекту cin >> rplace.x (как в cin >> rplace.x >> rplace.y), то опять получаете объект класса istream. Таким образом, в конечном итоге проверочное условие цикла while вычисляется как cin, что, как вы помните, при использовании в контексте проверочного условия преобразуется в булевское значение true или false, в зависимости от того, успешным ли был ввод. Например, в цикле из листинга 7.12, cin ожидает от пользователя ввода двух чисел. Если же вместо этого пользователь вводит q, как показано в примере вывода программы, cin >> распознает, что q — не число. Он оставляет q во входной очереди и возвращает значение, преобразуемое в false, завершая выполнение цикла.

Посмотрите, насколько этот подход к чтению чисел проще следующего:

```
for (int i = 0; i < limit; i++)
{
   cout << "Введите значение #" << (i + 1) << ": ";
   cin >> temp;
   if (temp < 0)
      break;
   ar[i] = temp;
}
```

Чтобы прервать этот цикл, вы вводите отрицательное число. Это ограничивает диапазон допустимых для ввода значений только неотрицательными, однако чаще вы пожелаете в качестве условия прерывания цикла использовать то, что не будет исключать определенные числовые значения из списка допустимых. Применение cin>> в качестве проверочного условия исключает это ограничение, потому что принимает любые корректные числовые значения. Вам стоит запомнить этот трюк и использовать его всякий раз, когда нужно организовать в цикле ввод чисел. К тому же следует иметь в виду, что нечисловой ввод устанавливает условие ошибки, которое предотвращает чтение любого дальнейшего ввода. Если программа должна выполнять дальнейший ввод после завершения цикла, вы должны использовать cin.clear(), чтобы сбросить состояние ошибки входного потока и, кроме того, прочесть ошибочный ввод, чтобы избавиться от него. В листинге 7.7 иллюстрируется эта техника.

Передача адресов структур

Предположим, что вы хотите сэкономить время и пространство памяти за счет передачи адресов структуры вместо передачи самой структуры. Для этого потребуется переписать функции так, чтобы они использовали в качестве аргументов указатели на структуры. Во-первых, давайте посмотрим, как мы можем переписать функцию show_polar(). Для этого нужно внести три изменения:

- При вызове функции передать ей адрес структуры (&pplace) вместо самой структуры (pplace).
- Определить формальный параметр как указатель на структуру polar — то есть polar *. Поскольку функция не должна модифицировать структуру, дополнительно использовать модификатор const.

- Поскольку формальный параметр теперь будет указателем на структуру вместо самой структуры, использовать операцию -> вместо операции точки.

После внесения этих изменений функция будет выглядеть следующим образом:

```
// отображает полярные координаты, преобразуя угол в градусы
void show_polar (const polar * pda)
{
   using namespace std;
   const double Rad_to_deg = 57.29577951;
   cout << "расстояние = " << pda->distance;
   cout << ", угол = " << pda->angle * Rad_to_deg;
   cout << " градусов\n";
}
```

Теперь изменим `rect_to_polar()`. Это будет несколько сложнее, потому что исходная функция `rect_to_polar()` возвращает структуру. Чтобы воспользоваться всеми преимуществами эффективности указателей, придется также возвращать указатель вместо значения. Это можно сделать, передав функции два указателя на структуру. Первый будет указывать на преобразовываемую структуру, а второй — на структуру, содержащую результат преобразования. Вместо *возврата* новой структуры функция *модифицирует* существующую структуру в вызванной функции. Поэтому, хотя первый аргумент — константный указатель, второй — не константный. Во всем остальном применимы те же принципы, что использовались для перевода `show_polar()` к аргументам-указателям. В листинге 7.13 показана переработанная программа.

Листинг 7.13. strctptr.cpp

```
// strctptr.cpp -- функции с аргументами-указателями на структуры
#include <iostream>
#include <cmath>
// объявления структур
struct polar
{
  double distance;    // расстояние от исходной точки
  double angle;       // направление
};
struct rect
{
  double x; // расстояние по горизонтали от начальной точки
  double y; // расстояние по вертикали от начальной точки
};
// прототипы
  void rect_to_polar(const rect * pxy, polar * pda);
  void show_polar (const polar * pda);
int main()
{
  using namespace std;
  rect rplace;
  polar pplace;
  cout << "Введите значения координат x и y: ";
  while (cin >> rplace.x >> rplace.y)
  {
     rect_to_polar(&rplace, &pplace);   // передача адресов
     show_polar(&pplace);               // передача адреса
```

```
        cout << "Следующие два числа (q для выхода): ";
    }
    cout << "Готово. \n";
    return 0;
}
// отображение полярных координат с преобразованием  радиан в градусы
void show_polar (const polar * pda)
{
    using namespace std;
    const double Rad_to_deg = 57.29577951;
    cout << "расстояние = " << pda->distance;
    cout << ", угол = " << pda->angle * Rad_to_deg;
    cout << " градусов \n";
}
// преобразование прямоугольных координат в полярные
void rect_to_polar(const rect * pxy, polar * pda)
{
    using namespace std;
    pda->distance =
    sqrt(pxy->x * pxy->x + pxy->y * pxy->y);
    pda->angle = atan2(pxy->y, pxy->x);
}
```

Замечание по совместимости

Некоторые реализации C++ используют `math.h` вместо более нового заголовочного файла `cmath`. Некоторые компиляторы требуют явных инструкций для нахождения математической библиотеки.

С точки зрения пользователя программа из листинга 7.13 ведет себя точно так же, как программа из листинга 7.12. Скрытое отличие в том, что программа из листинга 7.12 работает с копиями структур, в то время, как программа из листинга 7.13 использует указатели на исходные структуры.

Функции и объекты класса string

Хотя строки в стиле C и класс `string` служат в основном одним и тем же целям, класс `string` больше похож на структуру, чем на массив. Например, вы можете присвоить структуру другой структуре и объект — другому объекту. Вы можете передать структуру как единое целое функции, и точно так же можете передать объект. Если вам требуется несколько строк, вы можете объявить одномерный массив объектов `string` вместо двумерного массива `char`.

Листинг 7.14 представляет короткий пример, который объявляет массив объектов `string` и передает его функции, отображающей их содержимое.

Листинг 7.14. topfive.cpp

```
// topfive.cpp -- обработка массива объектов string
#include <iostream>
#include <string>
using namespace std;
const int SIZE = 5;
void display(const string sa[], int n);
```

```
int main()
{
   string list[SIZE]; // массив из 5 объектов string
   cout << "Введите " << SIZE << " ваших любимых астрономических объектов:\n";
   for (int i = 0; i < SIZE; i++)
   {
     cout << i + 1 << ": ";
     getline(cin,list[i]);
   }
   cout << "Ваш список: \n";
   display(list, SIZE);
   return 0;
}
void display(const string sa[], int n)
{
   for (int i = 0; i < n; i++)
     cout << i + 1 << ": " << sa[i] << endl;
}
```

Замечание по совместимости

Некоторые старые версии Visual C++ содержат ошибку, которая приводит к рассинхронизации операторов ввода и вывода. В этих версиях может понадобиться вводить ответ до того, как компьютер отобразит запрос этого ответа.

Ниже показан пример вывода программы из листинга 7.14:

```
Введите 5 ваших любимых астрономических объектов:
1: Orion Nebula
2: M13
3: Saturn
4:  Jupiter
5:  Moon
Ваш список:
1: Orion Nebula
2: M13
3: Saturn
4: Jupiter
5: Moon
```

Главное, что нужно отметить в этом примере — это то, что если не принимать во внимание функцию `getline()`, эта программа воспринимает объекты `string` как любой встроенный тип наподобие `int`. Если вам нужен массив `string`, вы просто используете обычный формат объявления массива:

```
string list[SIZE]; // массив из 5 объектов string
```

Каждый элемент массива `list` — это объект `string`, и он может использоваться следующим образом:

```
getline(cin,list[i]);
```

Аналогично, формальный аргумент `sa` — указатель на объект `string`, поэтому `sa[i]` — объект типа `string`, и он может использоваться соответственно:

```
cout << i + 1 << ": " << sa[i] << endl;
```

Рекурсия

А теперь поговорим совсем о другом. Функции C++ обладают интересным свойством — они могут вызывать сами себя. (Однако, в отличие от C, C++ не позволяет `main()` вызывать себя.) Эта возможность называется *рекурсией*. Рекурсия — важный инструмент для некоторых областей программирования, таких как искусственный интеллект, но здесь мы дадим только поверхностные сведения о принципах ее работы.

Рекурсия с одиночным рекурсивным вызовом

Если рекурсивная функция вызывает саму себя, затем этот новый вызов снова вызывает себя и так далее, то получится бесконечная последовательность вызовов, если только код не включает в себе нечто, что позволит прервать эту цепочку вызовов. Обычный метод состоит в том, что рекурсивный вызов заключают в оператор `if`. Например, рекурсивная функция типа `void` по имени `recurs()` может иметь следующую форму:

```
void recurs(списокАргументов)
{
   операторы1
   if (проверка)
      recurs(аргументы)
   операторы2
}
```

В какой-то ситуации *проверка* возвращает `false`, и цепочка вызовов прекращается.

Рекурсивные вызовы порождают замечательную цепочку событий. До тех пор, пока условие оператора `if` остается истинным, каждый вызов `recurs()` выполняет *операторы1* и затем вызывает новое воплощение `recurs()`, не достигая конструкции *операторы2*. Когда условие оператора `if` возвращает `false`, текущий вызов переходит к *операторы2*. Затем, когда текущий вызов завершается, управление возвращается предыдущему экземпляру `recurs()`, который вызвал его. Затем этот экземпляр исполняет свой раздел *операторы2* и прекращается, возвращая управление предшествующему вызову, и так далее. Таким образом, если происходит пять вложенных вызовов `recurs()`, то первый раздел *операторы1* выполняется пять раз в том порядке, в котором произошли вызовы, а потом пять раз в обратном порядке выполняется раздел *операторы2*. После входа в пять уровней рекурсии затем программа должна пройти обратно эти же пять уровней. Код в листинге 7.15 демонстрирует описанное поведение.

Листинг 7.15. recur.cpp

```
// recur.cpp -- использование рекурсии
#include <iostream>
void countdown(int n);
int main()
{
   countdown(4); // вызов рекурсивной функции
   return 0;
}
```

```
void countdown(int n)
{
   using namespace std;
   cout << "Обратный отсчет ... " << n << endl;
   if (n > 0)
       countdown(n-1); // функция вызывает себя
   cout << n << ": Бабах!\n";
}
```

Ниже — аннотированный вывод программы из листинга 7.15:

```
Обратный отсчет ... 4  ← уровень 1; добавление уровней рекурсии
Обратный отсчет ... 3  ← уровень 2
Обратный отсчет ... 2  ← уровень 3
Обратный отсчет ... 1  ← уровень 4
Обратный отсчет ... 0  ← уровень 5; финальный рекурсивный вызов
0: Бабах!              ← уровень 5; начало обратного прохода
1: Бабах!              ← уровень 4
2: Бабах!              ← уровень 3
3: Бабах!              ← уровень 2
4: Бабах!              ← уровень 1
```

Обратите внимание, что каждый рекурсивный вызов создает свой собственный набор переменных, поэтому на момент пятого вызова она имеет пять отдельных переменных по имени n — каждая со своим собственным значением. Вы можете убедиться в этом, модифицируя листинг 7.15 таким образом, чтобы отображать адрес n наряду со значением:

```
cout << "Обратный отсчет ... " << n << " (n по адресу " << &n << ")" << endl;
...
cout << n << ": Бабах!"; << "          (n по адресу " << &n << ")" << endl;
```

Если сделать так, то вывод программы примет следующий вид:

```
Обратный отсчет ... 4  (n по адресу 0012FE0C)
Обратный отсчет ... 3  (n по адресу 0012FD34)
Обратный отсчет ... 2  (n по адресу 0012FC5C)
Обратный отсчет ... 1  (n по адресу 0012FB84)
Обратный отсчет ... 0  (n по адресу 0012FAAC)
0: Бабах!              (n по адресу 0012FAAC)
1: Бабах!              (n по адресу 0012FB84)
2: Бабах!              (n по адресу 0012FC5C)
3: Бабах!              (n по адресу 0012FD34)
4: Бабах!              (n по адресу 0012FE0C)
```

Обратите внимание, что n, имеющая значение 4, размещается в одном месте (в данном примере по адресу 0012FE0C), n, имеющая значение 3, находится в другом месте (адрес памяти 0012FD34) и так далее.

Рекурсия с множественными рекурсивными вызовами

Рекурсия, в частности, удобна в тех ситуациях, когда нужно вызывать повторяющееся разделение задачи на две разные похожие подзадачи. Например, рассмотрим

этот подход для рисования линейки. Отмечаем два конца, находим средину, помечаем ее. Затем применяем ту же процедуру для левой половины линейки и правой ее половины. Если нам нужно больше частей, применим ту же процедуру для каждой из них. Этот рекурсивный подход иногда называют *стратегией "разделяй и властвуй"*. Листинг 7.16 иллюстрирует данный подход в рекурсивной функции `subdivide()`. Она использует строку, изначально заполненную пробелами, за исключением символов | на каждом конце. Затем главная программа запускает цикл из шести вызовов `subdivide()`, каждый раз увеличивая количество уровней рекурсии и печатая результирующую строку. Таким образом, каждая строка вывода представляет дополнительный уровень рекурсии. Чтобы напомнить вам о том, что это — просто вариант, программа использует квалификатор `std::` вместо директивы `using`.

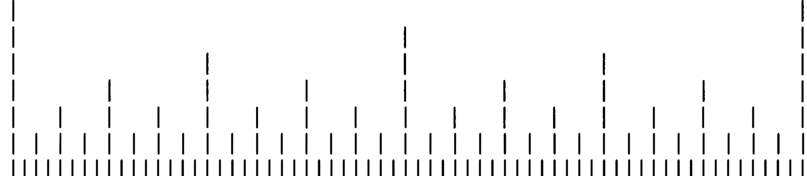
Листинг 7.16. `ruler.cpp`

```
// ruler.cpp -- использование рекурсии для деления линейки
#include <iostream>
const int Len = 66;
const int Divs = 6;

void subdivide(char ar[], int low, int high, int level);
int main()
{
   char ruler[Len];
   int i;
   for (i = 1; i < Len - 2; i++)
   ruler[i] = ' ';
   ruler[Len - 1] = '\0';
   int max = Len - 2;
   int min = 0;
   ruler[min] = ruler[max] = '|';
   std::cout << ruler << std::endl;
   for (i = 1; i <= Divs; i++)
   {
     subdivide(ruler,min,max, i);
     std::cout << ruler << std::endl;
     for (int j = 1; j < Len - 2; j++)
        ruler[j] = ' '; // очистить линейку
   }
   return 0;
}

void subdivide(char ar[], int low, int high, int level)
{
   if (level == 0)
     return;
   int mid = (high + low) / 2;
   ar[mid] = '|';
   subdivide(ar, low, mid, level - 1);
   subdivide(ar, mid, high, level - 1);
}
```

Вот как выглядит вывод программы из листинга 7.16:



Замечания по программе

Функция subdivide() из листинга 7.16 использует переменную level для управления уровнем рекурсии. Когда функция вызывает саму себя, она уменьшает level на единицу, и когда level достигает нуля, функция завершается. Обратите внимание, что subdivide() вызывает себя дважды — один раз для левой части линейки и один — для правой. Исходная средняя точка становится правым концом для одного вызова и левым — для другого. Заметьте, что количество вызовов растет в геометрической прогрессии. То есть один вызов генерирует два, которые генерируют четыре вызова, а те в свою очередь — восемь, и так далее. Вот почему уровень 6 способен заполнить 64 элемента ($2^6 = 64$). Это непрерывное удвоение количества вызовов функции (а вместе с ними и количества сохраняемых переменных) делает такую форму рекурсии плохим решением при достаточно большом числе уровней. Если же уровней не слишком много, то это — простое и элегантное решение.

Указатели на функции

Любой разговор о функциях C и C++ будет неполным, если не упомянуть указатели на функции. Рассмотрим кратко эту тему, оставив более полное ее раскрытие специализированным изданиям.

Функции, как и элементы данных, имеют адреса. Адрес функции — это адрес в памяти, где находится начало кода функции на машинном языке. Обычно пользователю ни к чему знать этот адрес, но это может быть полезно для программы. Например, можно написать функцию, которая принимает адрес другой функции в качестве аргумента. Это позволяет первой функции найти вторую и запустить ее. Такой подход сложнее, чем простой вызов второй функции из первой, но он открывает возможность передачи разных адресов функций в разные моменты времени. То есть первая функция может вызывать разные функции в разное время.

Основы указателей на функции

Проясним этот процесс на примере. Предположим, вы хотите спроектировать функцию estimate(), которая оценивает затраты времени, необходимого для написания заданного числа строк кода, и вы хотите, чтобы этой функцией пользовались разные программисты. Часть кода estimate() будет одинакова для всех пользователей, но эта функция позволит каждому программисту применить свой собственный алгоритм оценки затрат времени. Механизм, используемый для обеспечения такой возможности, будет заключаться в передаче estimate() адреса конкретной функции, реализующей алгоритм, выбранный данным программистом.

Чтобы реализовать этот план, вам понадобится сделать следующее:

- Получить адрес функции.
- Объявить указатель на функцию.
- Использовать указатель на функцию для ее вызова.

Получение адреса функции

Получить адрес функции очень просто: вы просто используете имя функции без скобок. То есть, если имеется функция `think()`, то ее адрес записывается как `think`. Чтобы передать функцию в качестве аргумента, вы просто передаете ее имя. Убедитесь в том, что вы понимаете разницу между *адресом* функции и передачей *значения ее возврата*:

```
process(think);  // передача адресе think() функции process()
thought(think());//передача возвращаемого значения think() функции thought()
```

Вызов `process()` позволяет этой функции вызвать функцию `think()` изнутри себя. Вызов `thought()` сначала вызывает функцию `think()` и затем передает ей значение, возвращенное `think()`.

Объявление указателя на функцию

Чтобы объявить указатель на тип данных, нужно явно специфицировать тип, на который будет указывать этот указатель. Аналогично, указатель на функцию должен специфицировать, на функцию какого типа он будет указывать. Это значит, что объявление должно идентифицировать тип возврата функции и ее сигнатуру (список аргументов). То есть объявление должно предоставлять ту же информацию о функции, которую предоставляет и ее прототип. Например, предположим, что программистка Пэм написала функцию для оценки затрат времени со следующим прототипом:

```
double pam(int);    // прототип
```

Вот как должно выглядеть объявление соответствующего типа указателя:

```
double (*pf)(int);  // pf указывает на функцию, которая принимает
                    // один аргумент типа int и возвращает тип double
```

Обратите внимание, что это выглядит подобно объявлению `pam()`, но роль `pam` играет `(*pf)`. Поскольку `pam` — функция, то же самое представляет собой `(*pf)`. И если `(*pf)` — функция, то `pf` — указатель на нее.

Совет

В общем случае, чтобы объявить указатель на функцию определенного рода, вы можете сначала написать прототип обычной функции требуемого вида, а затем заменить ее имя выражением в форме `(*pf)`. В этом случае `pf` представляет собой указатель на функцию этого типа.

Объявление требует скобок вокруг `*pf`, чтобы обеспечить правильный приоритет операций. Скобки имеют более высокий приоритет, чем операция `*`, поэтому `*pf(int)` означает, что `pf()` — функция, которая возвращает указатель, в то время как `(*pf)(int)` означает, что `pf` — указатель на функцию:

```
double (*pf)(int);  // pf указывает на функцию, возвращающую double
double *pf(int);    // pf() — функция, возвращающая указатель на double
```

После правильного объявления pf вы можете присваивать ему адрес подходящей функции:

```
double pam(int);
double (*pf)(int);
pf = pam;      // pf теперь указывает на функцию pam()
```

Заметьте, что pam() должна соответствовать pf как по типу возврата, так и по сигнатуре. Компилятор отвергнет несоответствующие присваивания:

```
double ned(double);
int ted(int);
double (*pf)(int);
pf = ned;      // неверно — несоответствие сигнатуры
pf = ted;      // неверно — несоответствие типа возврата
```

Вернемся к функции estimate(), упомянутой ранее. Предположим, что вы хотите передавать ей количество строк кода, который нужно написать и адрес алгоритма оценки — функции, такой как pam(). Тогда она должна иметь следующий прототип:

```
void estimate(int lines, double (*pf)(int));
```

Это объявление сообщает, что второй аргумент является указателем на функцию, принимающую аргумент int и возвращающую значение double. Чтобы заставить estimate() использовать функцию pam(), вы передаете ей адрес pam:

```
estimate(50, pam);  // вызов сообщает estimate(), что она должна
                    // использовать pam()
```

Понятно, что вся сложность использования указателей на функцию заключается в написании прототипов, в то время как передать адрес очень просто.

Использование указателя для вызова функции

Теперь обратимся к завершающей части этой техники, которая позволит использовать указатель для вызова указанной им функции. Ключ к этому — в объявлении указателя. Вспомним, что там (*pf) играет ту же роль, что имя функции. Поэтому все, что вы должны сделать — использовать (*pf), как если бы это было имя функции:

```
double pam(int);
double (*pf)(int);
pf = pam;             // pf теперь указывает на функцию pam()
double x = pam(4);    // вызвать pam(), используя ее имя
double y = (*pf)(5);  // вызвать pam(), используя указатель pf
```

В действительности C++ позволяет вам использовать pf, как если бы это было имя функции:

```
double y = pf(5);     // также вызывает pam(), используя указатель pf
```

Первая форма вызова более неуклюжа, чем эта, но она напоминает о том, что код использует указатель на функцию.

История против логики

О, великий синтаксис! Как может быть pf эквивалентно (*pf)? Одна школа утверждает, что поскольку pf — указатель на функцию, то *pf — функция, поэтому вы должны использовать для ее вызова (*pf)(). Другая школа гласит, что поскольку имя функции есть указатель на эту функцию, то и любой указатель на функцию должен вести себя как имя функции; отсюда — для вызова функции через указатель следует написать pf(). Язык C++ придерживается компромиссной точки зрения о том, что обе формы корректны, или, по крайней мере, допустимы, даже несмотря на то, что они логически несовместимы. Прежде чем вы отвергнете компромисс и выберете для себя одну форму, вспомните, что допущение несогласованных и логически несовместимых представлений — часть человеческого мышления.

Пример с указателем на функцию

Код в листинге 7.17 демонстрирует использование указателя функции в программе. Он вызывает функцию estimate() дважды — один раз передавая ей адрес функции betsy(), а второй — адрес функции pam(). В первом случае estimate() использует betsy() для вычисления необходимого количества часов, а во втором — использует для этого же pam(). Такой дизайн обеспечивает гибкость разработки программ. Когда программист Ральф разработает свой собственный алгоритм оценки затрат времени, ему не придется переписывать estimate(). Вместо этого он должен просто реализовать собственную функцию ralph(), обеспечив ей необходимую сигнатуру и тип возврата. Конечно, переписать estimate() не трудно, но те же принципы применимы и к более сложному коду. К тому же метод указателей на функции позволяет Ральфу модифицировать поведение estimate(), даже не имея доступа к ее исходному коду.

Листинг 7.17. fun_ptr.cpp

```
// fun_ptr.cpp -- указатели на функции
#include <iostream>
double betsy(int);
double pam(int);
// второй аргумент — указатель на функцию double,
// которая принимает аргумент типа int
void estimate(int lines, double (*pf)(int));
int main()
{
    using namespace std;
    int code;
    cout << "Сколько строк кода нужно написать? ";
    cin >> code;
    cout << "Вот оценка Бетси: \n";
    estimate(code, betsy);
    cout << "Вот оценка Пэм: \n";
    estimate(code, pam);
    return 0;
}
double betsy(int lns)
{
   return 0.05 * lns;
}
```

```
double pam(int lns)
{
   return 0.03 * lns + 0.0004 * lns * lns;
}
void estimate(int lines, double (*pf)(int))
{
   using namespace std;
   cout << lines << " строк потребует ";
   cout << (*pf)(lines) << " часов\n";
}
```

Ниже показан пример выполнения программы из листинга 7.17:

```
Сколько строк кода нужно написать? 30
Вот оценка Бетси:
30 строк потребует 1.5 часов
Вот оценка Пэм:
30 строк потребует 1.26 часов
```

И второй пример запуска той же программы:

```
Сколько строк кода нужно написать? 100
Вот оценка Бетси:
100 строк потребует 5 часов
Вот оценка Пэм:
100 строк потребует 7 часов
```

Резюме

Функции — это программные модули C++. Чтобы использовать функцию, вы должны предоставить ее определение и прототип, после чего ее можно вызывать. Определение функции — это код, который реализует то, что она делает. Прототип функции описывает ее интерфейс: сколько она принимает параметров и какого типа, а также каков тип возвращаемого ею значения, если оно есть. Вызов функции позволяет программе послать ей аргументы и передать поток управления коду функции.

По умолчанию функции C++ получают аргументы по значению. Это значит, что формальные параметры в определении функции — это совершенно новые переменные, которые инициализируются значениями, переданными в вызове этой функции. Таким образом, C++ защищает целостность исходных данных, работая с копиями.

C++ трактует аргумент, являющийся именем массива, как адрес первого его элемента. Технически это по-прежнему означает передачу по значению, потому что аргумент-указатель является копией исходного адреса, но функция использует его для обращения к содержимому исходного массива. Когда вы объявляете формальные параметры функции (и только в этом случае), следующие два объявления эквивалентны:

```
имяТипа arr[]
имяТипа * arr
```

Обе они означают, что `arr` — указатель на *имяТипа*. Однако, когда вы пишете код функции, вы можете использовать `arr` для доступа к его элементам, как если бы он был именем массива: `arr[i]`. Даже при передаче указателей можно предохранить целостность исходных данных, объявив формальные аргументы как указатели на кон-

стантные типы. Поскольку передача адреса массива не содержит никакой информации о его размере, обычно вы передаете размер массива в отдельном аргументе.

C++ предусматривает три способа представления строк в стиле C: в виде символьного массива, строковой константы или указателя на строку. Все они имеют тип `char*` (указатель на символ), поэтому передаются функциям как аргумент типа `char*`. C++ использует нулевой символ (`\0`) в качестве ограничителя строки, и строковые функции выполняют проверку на нулевой символ для определения конца любой обрабатываемой строки.

В C++ также определен класс `string` для представления строк. Функции могут принимать объекты `string` в качестве аргументов и использовать объекты `string` как возвращаемые значения. Метод `size()` класса `string` может применяться для определения длины хранимой в нем строки.

C++ трактует структуры точно так же, как базовые типы, в том смысле, что вы можете передавать их по значению и использовать в качестве типа возврата. Однако если структура достаточно велика, может оказаться более эффективным передавать указатель на структуру и позволить функции работать с исходными данными.

Функции C++ могут быть рекурсивными; то есть код определенной функции может включать в себя вызов ее самой.

Имя функции C++ действует как ее адрес. Используя в функциях аргументы типа указателей на функции, вы можете передавать одной функции имя второй функции, если хотите, чтобы первая функция вызвала вторую.

Вопросы для самоконтроля

1. Каковы три шага создания функции?
2. Сконструируйте прототипы, которые соответствовали бы следующим описаниям:
   а. `igor()` не принимает аргументов и не возвращает значений.
   б. `tofu()` принимает аргумент `int` и возвращает `float`.
   в. `mpg()` принимает два аргумента типа `double` и возвращает `double`.
   г. `summation()` принимает имя массива `long` и его размер и возвращает значение `long`.
   д. `doctor()` принимает строковый аргумент (строка не должна модифицироваться) и возвращает `double`.
   е. `ofcourse()` принимает структуру `boss` в качестве аргумента и не возвращает ничего.
   ж. `plot()` принимает указатель на структуру `map` в качестве аргумента и возвращает строку.
3. Напишите функцию, принимающую три аргумента: имя массива `int`, его размер и значение `int`. Пусть функция присвоит каждому элементу массива это значение `int`.
4. Напишите функцию, принимающую три аргумента: указатель на первый элемент диапазона в массиве, указатель на элемент, следующий за концом этого диапазона, и значение `int`. Пусть функция присвоит каждому элементу диапазона массива это значение `int`.

5. Напишите функцию, принимающую имя массива `double` и его размер в качестве аргументов и возвращающую наибольшее значение, содержащееся в этом массиве. Обратите внимание, что функция не должна модифицировать содержимое массива.
6. Почему вы не используете квалификатор `const` для аргументов функций, относящихся к любому из базовых типов?
7. Каковы три формы строк в стиле С могут встретиться в программах C++?
8. Напишите функцию, имеющую следующий прототип:

   ```
   int replace(char * str, char c1, char c2);
   ```

   Пусть эта функция заменяет каждое появление `c1` в строке `str` на `c2` и возвращает количество выполненных замен.
9. Что означает выражение `*"pizza"`? А как насчет `"taco"[2]`?
10. C++ позволяет передавать структуры по значению, а также передавать адрес структуры. Если `glitz` — структурная переменная, как передать ее по значению? Как передать ее адрес? Каковы преимущества и недостатки обоих подходов?
11. Функция `judge()` имеет тип возврата `int`. В качестве аргумента она принимает адрес функции. Функция, чей адрес ей передается, в свою очередь, принимает аргумент типа `const char` и возвращает `int`. Напишите прототип функции.

Упражнения по программированию

1. Напишите программу, которая многократно приглашает пользователя вводить пары чисел до тех пор, пока хотя бы одно из этой пары не будет равно 0. С каждой парой программа должна использовать функцию для вычисления среднего гармонического этих чисел. Функция должна возвращать ответ `main()` для отображения результата. Среднее гармоническое чисел — это инверсия среднего значения их инверсий; она вычисляется следующим образом:

   ```
   harmonic mean = 2.0 × x × y / (x + y)
   ```
2. Напишите программу, которая запрашивает у пользователя 10 результатов игры в гольф, сохраняя их в массиве. При этом необходимо обеспечить возможность прерывания ввода до ввода всех 10 результатов. Программа должна отобразить все результаты в одной строке и сообщить среднее их значение. Реализуйте ввод, отображение и вычисление среднего в трех отдельных функциях, работающих с массивами.
3. Пусть имеется следующее объявление структуры:

   ```
   struct box
   {
       char maker[40];
       float height;
       float width;
       float length;
       float volume;
   };
   ```

   а. Напишите функцию, принимающую структуру `box` по значению и отображающую все ее члены.

б. Напишите функцию, принимающую адрес структуры `box` и устанавливающую значение члена `volume` равным произведению остальных трех членов.

в. Напишите простую программу, использующую эти две функции.

4. Многие лотереи штатов организованы подобно той, что смоделирована в листинге 7.4. Во всех их вариациях вы должны выбрать несколько чисел из одного набора, называемого полем номеров. (Например, вы можете выбрать 5 чисел из поля 1–47.) Вы также указываете один номер (называемый мегагомером, или супершаром) из второго диапазона, такого как 1–27. Чтобы выиграть главный приз, вы должны правильно угадать все номера. Шанс выиграть вычисляется как вероятность угадывания всех номеров в поле, умноженная на вероятность угадывания меганомера. Например, вероятность выигрыша в описанном здесь примере вычисляется как вероятность угадывания 5 номеров из 47, умноженная на вероятность угадывания одного номера из 27. Модифицируйте листинг 7.4 для вычисления вероятности выигрыша в такой лотерее.

5. Определите рекурсивную функцию, принимающую целый аргумент и возвращающую его факториал. Напомним, что факториал 3 записывается, как 3! и вычисляется как 3×2! и так далее, причем 0! равно 1. Вообще, если n больше нуля, то n! = n * (n-1)!. Протестируйте функцию в программе, использующей цикл, в котором пользователь может вводить различные значения, для которых программа вычисляет и отображает факториалы.

6. Напишите программу, использующую следующие функции:

   `Fill_array()` принимает в качестве аргумента имя массива элементов типа `double` и размер этого массива. Она приглашает пользователя ввести значения `double` для помещения их в массив. Ввод прекращается при наполнении массива либо когда пользователь вводит нечисловое значение, и возвращает действительное количество элементов.

   `Show_array()` принимает в качестве аргументов имя массива значений `double` и его размер, и отображает содержимое массива.

   `Reverse_array()` принимает в качестве аргумента имя массива значений `double` и его размер, и изменяет порядок его элементов на противоположный.

   Программа должна использовать эти функции для наполнения массива, обращения порядка его элементов, кроме первого и последнего, с последующим отображением.

7. Повторите программу из листинга 7.7, заменив три функции обработки массива версиями, работающими с диапазонами значений, заданными парой указателей. Функция `fill_array()` вместо возврата действительного числа прочитанных значений должна возвращать указатель на место, следующее за последним введенным элементом; прочие функции должны использовать его в качестве второго аргумента для идентификации конца диапазона данных.

8. Следующее упражнение позволит попрактиковаться в написании функций, работающих с массивами и структурами. Ниже представлен каркас программы. Дополните его описанными функциями:

```
#include <iostream>
using namespace std;
const int SLEN = 30;
```

```
struct student {
   char fullname[SLEN];
   char hobby[SLEN];
   int ooplevel;
};
// getinfo() принимает два аргумента: указатель на первый элемент
// массива структур student и значение int, представляющее
// количество элементов в массиве. Функция запрашивает и
// сохраняет данные о студентах. Ввод прекращается либо после
// наполнения массива, либо при вводе пустой строки вместо имени
// студента. Функция возвращает действительное количество
// введенных элементов.
int getinfo(student pa[], int n);
// display1() принимает в качестве аргумента структуру student
// и отображает ее содержимое
void display1(student st);
// display2() принимает адрес структуры student в качестве аргумента
// и отображает ее содержимое
void display2(const student * ps);
// display3() принимает указатель на первый элемента массива
// структур student и количество элементов в этом массиве и
// отображает содержимое всех структур в массиве
void display3(const student pa[], int n);
int main()
{
   cout << "Введите размер класса: ";
   int class_size;
   cin >> class_size;
   while (cin.get() != '\n')
      continue;
   student * ptr_stu = new student[class_size];
   int entered = getinfo(ptr_stu, class_size);
   for (int i = 0; i < entered; i++)
   {
      display1(ptr_stu[i]);
      display2(&ptr_stu[i]);
   }
   display3(ptr_stu, entered);
   delete [] ptr_stu;
   cout << "Done\n";
   return 0;
}
```

9. **Спроектируйте функцию** `calculate()`, **принимающую два значения типа** `double` **и указатель на функцию, принимающую два аргумента** `double` **и возвращающую** `double`. **Функция** `calculate()` **также должна иметь тип** `double` **и возвращать значение, вычисленное функцией, заданной указателем, используя аргумент** `double` **функции** `calculate()`. **Например, предположим, что имеется следующее определение функции** `add()`:

```
double add(double x, double y)
{
   return x + y;
}
```

затем вызов функция

```
double q = calculate(2.5, 10.4, add);
```

должен заставить `calculate()` передать значения `2.5` и `10.4` функции `add()` и затем вернуть ее результат (`12.9`).

Используйте в программе эти функции и еще хотя бы одну, подобную `add()`. Программа должна использовать цикл, позволяющий пользователю вводить пары чисел. Для каждой пары `calculate()` должна вызвать `add()` и хотя бы еще одну функцию подобного рода. Если вы чувствуете себя уверенно, попробуйте создать массив указателей на функции, подобные `add()`, и организуйте цикл, применяя `calculate()` для вызова этих функций по их указателям. Подсказка: вот как можно объявить массив из трех таких указателей:

```
double (*pf[3])(double, double);
```

Вы можете инициализировать такой массив, используя обычный синтаксис инициализации массивов и имена функций в качестве адресов.

ГЛАВА 8

Дополнительные сведения о функциях

В этой главе:

- Встроенные функции
- Ссылочные переменные
- Передача функции аргументов по ссылке
- Аргументы, определяемые по умолчанию
- Перегрузка функций
- Шаблоны функций
- Спецификации шаблонов функций

В главе 7 представлен обширный материал по функциям, однако осталось еще рассмотреть немало вопросов. Язык C++ предоставляет много новых возможностей, связанных с функциями, что отличает его от своего предшественника – языка C. К ним относятся встроенные функции, передача переменных по ссылке, определяемые по умолчанию значения аргументов, перегрузка функций (полиморфизм) и шаблоны функций. Эта глава более всех предыдущих посвящена специфике языка C++ (но не C). Поэтому она служит отправной точкой нашего вторжения в мир "плюсов".

Встроенные функции C++

Встроенные функции являются усовершенствованием языка C++, предназначенным для ускорения работы программ. Основное различие между встроенными и обычными функциями заключается не в написании кода, а в том, как компилятор внедряет функцию в программу. Чтобы понять это различие, потребуется глубже рассмотреть внутреннее содержание программ. Именно с этого мы и начнем.

Конечным продуктом процесса компиляции является исполняемая программа, которая состоит из набора машинных команд. При запуске программы операционная система загружает эти команды в оперативную память так, что каждая команда обладает собственным адресом в памяти. Затем команды поочередно выполняются. Когда в программе встречается, например, оператор цикла или условного перехода, она "перепрыгивает" вперед или назад через ряд команд, осуществляя переход по определенному адресу. При вызове обычной функции также выполняется переход к определенному адресу (адресу функции) с последующим возвратом после завершения ее работы. Рассмотрим типичную реализацию этого процесса немного подробнее. Когда в программе встречается команда вызова функции, программа сохраняет

адрес команды, следующей сразу после вызова функции, копирует аргументы функции в стек (зарезервированный для этой цели блок памяти), переходит к ячейке памяти, обозначающей начало функции, выполняет код функции (возможно, помещая возвращаемое значение в регистр), а затем переходит к команде, адрес которой сохранен.[1] Переходы и запоминание соответствующих адресов влекут дополнительные затраты времени, связанные с использованием функций.

Встроенная функция C++ служит альтернативой. Скомпилированный код такой функции "внутритекстово" встраивается в код программы. Иначе говоря, компилятор подставляет вместо вызова функции ее код. В результате программе не нужно выполнять переход к другому адресу и возвращаться назад. Таким образом, встраиваемые функции выполняются немного быстрее, чем обычные, однако за это нужно платить дополнительным расходом памяти. Если в десяти различных местах программа выполняет вызов одной и той же встроенной функции, ее код будет содержать десять копий этой функции (рис. 8.1).

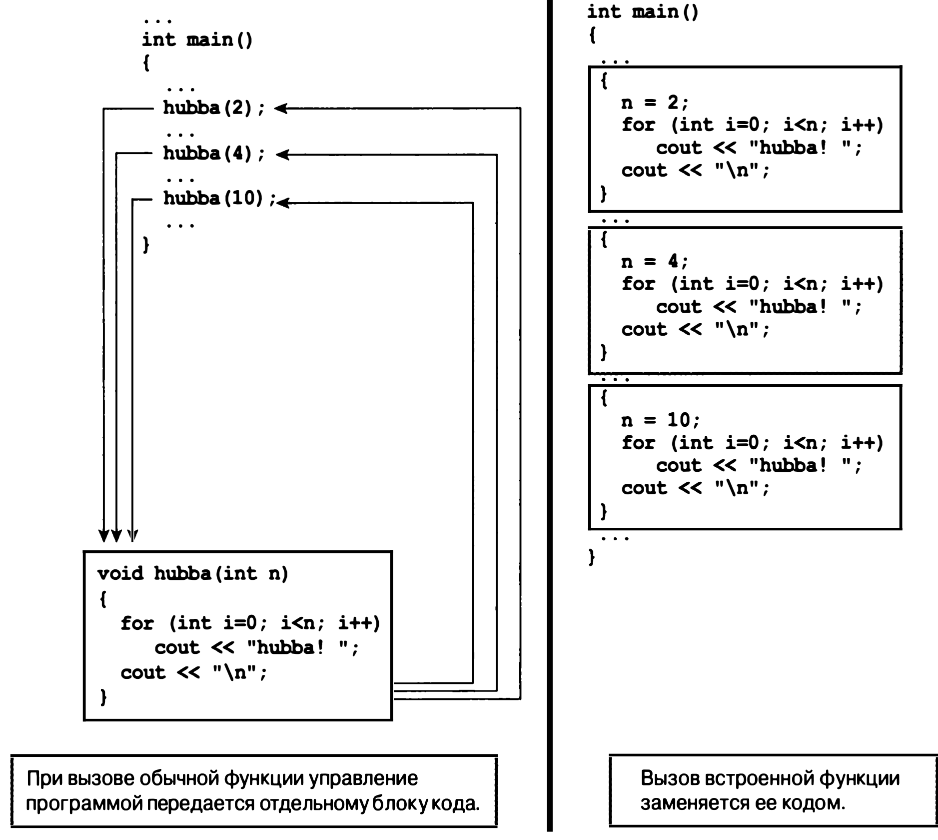


Рис. 8.1. Различия между встроенными и обычными функциями

[1] Это можно сравнить с процессом чтения некоторого текста, когда приходится отвлекаться на ознакомление с содержанием сноски, а затем возвращаться к фрагменту, где чтение было прервано.

Решение об использовании встроенной функции должно быть взвешенным. Если затраты времени на выполнение функции значительно превышают длительность реализации механизма ее вызова, экономия времени на фоне общего процесса окажется незаметной. Если же время выполнения кода невелико, то разница во времени при использовании встроенной функции по сравнению с обычной может быть значительной. С другой стороны, в этом случае ускоряется и без того сравнительно быстрый процесс, поэтому при нечастом вызове функции общая экономия времени может быть невелика.

Чтобы воспользоваться встроенной функцией, нужно выполнить хотя бы одно из следующих действий:

- Предварить объявление функции ключевым словом `inline`.
- Предварить определение функции ключевым словом `inline`.

Общепринято опускать прототип и помещать полное описание (заголовок и весь код функции) туда, где обычно находится прототип. Компилятор не обязательно удовлетворит запрос пользователя сделать функцию встроенной. Он может прийти к заключению, что функция слишком велика, или обнаружит, что она обращается сама к себе (рекурсия для встроенных функций не допускается и невозможна сама по себе). Кроме того, возможен случай, когда опция реализации встроенных функций у компилятора отключена либо он не поддерживает эту опцию вообще.

В листинге 8.1 иллюстрируется метод встраивания на примере функции `square()`, которая возводит в квадрат переданный ей аргумент. Обратите внимание, что все определение функции помещено в одной строке. Это совсем не обязательно, но, если определение не помещается в одной строке, то для функции, вероятно, статус встроенной не особенно подходит.

Листинг 8.1. `inline.cpp`

```
// inline.cpp -- использование встроенной функции
#include <iostream>
// определение встроенной функции
inline double square(double x) { return x * x; }
int main()
{
  using namespace std;
  double a, b;
  double c = 13.0;
  a = square(5.0);
  b = square(4.5 + 7.5); // допускается передача выражений
  cout << "a = " << a << ", b = " << b << "\n";
  cout << "c = " << c;
  cout << ", c в квадрате = " << square(c++) << "\n";
  cout << "Теперь c = " << c << "\n";
  return 0;
}
```

Ниже представлены результаты выполнения программы из листинга 8.1:

```
a = 25, b = 144
c = 13, c в квадрате = 169
Теперь c = 14
```

Полученные результаты показывают, что встраиваемая функция передает аргументы по значению, как это принято для обычных функций. Если аргумент представляет собой выражение вроде `4.5 + 7.5`, функция передает значение этого выражения. В рассматриваемом случае оно равно `12`. Из этого следует, что средство `inline` языка C++ обладает существенными преимуществами перед макроопределениями языка C. См. врезку "Сравнение возможностей встраивания функций и макросов" ниже в этой главе.

Хоть в программе нет отдельного прототипа, тем не менее, возможности применения прототипов C++ проявляются и в ней. Дело в том, что полное определение функции, которое дается перед тем, как она будет выполнена первый раз, служит прототипом. Это означает, что можно использовать функцию `square()` с аргументом типа `int` или `long`, и программа автоматически выполнит приведение аргумента к типу `double`, прежде чем передавать его значение функции.

Сравнение возможностей встраивания функций и макросов

Средство `inline` реализовано только в C++. В языке C используется оператор препроцессора `#define`, обеспечивающий реализацию ***макросов***, представляющих собой грубый аналог встраиваемого кода. Например, макрос, возводящий целое число в квадрат, имеет вид:

```
#define SQUARE(X) X*X
```

Этот макрос работает не по принципу передачи аргументов, а по принципу подстановки текста, при этом `X` играет роль символической метки "аргумента":

`a = SQUARE(5.0);` заменяется выражением `a = 5.0*5.0;`

`b = SQUARE(4.5 + 7.5);` заменяется выражением `b = 4.5 + 7.5 * 4.5 + 7.5;`

`d = SQUARE(c++);` заменяется выражением `d = c++*c++;`

Только первый пример выполняется должным образом. Можно исправить положение, снабдив описание макроса скобками:

```
#define SQUARE(X) ((X)*(X))
```

Но все еще остается проблема, связанная с тем, что в макросах не передаются аргументы по значению. Даже при использовании нового определения макроса (со скобками), функция `SQUARE(c++)` увеличивает значение `c` на 1 дважды, в то время как встраиваемая функция `square()`, представленная в листинге 8.1, вычисляет `c`, передает полученное значение для возведения в квадрат, а затем увеличивает значение `c` на 1.

Мы здесь не преследовали цель научить вас создавать макросы на языке C. Вместо этого мы рекомендуем вместо использования макросов для реализации средств, подобных функциям, принять во внимание возможность преобразования их во встроенные функции языка C++.

Ссылочные переменные

Язык C++ вводит в практику новый составной тип данных — ссылочную переменную. *Ссылка* представляет собой имя, которое является псевдонимом, или альтернативным именем, для ранее объявленной переменной. Например, если вы делаете `twain` ссылкой на переменную `clemens`, можно взаимозаменяемо использовать эти имена для представления данной переменной. В чем смысл использования альтернативного имени? Уж не в том ли, чтобы помочь тем программистам, которые не удовлетворены сделанным ими выбором имен переменных? Вполне возможно, однако основное назначение ссылок — их использование в качестве формальных аргумен-

тов функций. Применяя ссылку в качестве аргумента, функция работает с исходными данными, а не с их копиями. Ссылки представляют собой удобную альтернативу указателям при обработке крупных структур посредством функций. Они играют важную роль при создании классов. Однако прежде чем изучать использование ссылок при работе с функциями, рассмотрим основы определения и применения ссылок. Следует иметь в виду, что цель предстоящего обсуждения заключается в демонстрации функционирования ссылок, а не типичных методов их использования.

Создание ссылочных переменных

Как уже говорилось, в языках C и C++ символ `&` используется для обозначения адреса переменной. Язык C++ придает символу `&` дополнительный смысл и задействует его для объявления ссылок. Например, чтобы `rodents` стало альтернативным именем для переменной `rats`, необходимо написать следующее:

```
int rats;
int & rodents = rats;  // rodents становится псевдонимом имени rats
```

В этом контексте символ `&` не является операцией взятия адреса. В этом случае она воспринимается как часть идентификатора типа данных. Подобно тому, как выражение `char *` в объявлении означает указатель на значение типа `char`, выражение `int &` представляет собой ссылку на `int`. Объявление ссылки позволяет взаимозаменяемо использовать идентификаторы `rats` и `rodents`. Они ссылаются на одно и то же значение, а также на один и тот же адрес памяти. Программа, представленная в листинге 8.2, подтверждает сказанное.

Листинг 8.2. `firstref.cpp`

```
// firstref.cpp -- определение и использование ссылки
#include <iostream>
int main()
{
    using namespace std;
    int rats = 101;
    int & rodents = rats; // rodents является ссылкой
    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;
    rodents++;
    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;
    // Некоторые реализации требуют для следующих адресов
    // выполнить приведение к типу unsigned
    cout << "адрес rats = " << &rats;
    cout << ", адрес rodents = " << &rodents << endl;
    return 0;
}
```

Обратите внимание, что символ `&` в выражении

```
int & rodents = rats;
```

не обозначает операцию взятия адреса, он объявляет, что переменная `rodents` имеет тип `int &`, то есть является ссылкой на переменную типа `int`.

Однако в выражении

```
cout <<", rodents address = " << &rodents << endl;
```

этот символ является оператором адресации. Выражение &rodents представляет собой адрес переменной, на которую ссылается rodents. Вывод программы имеет следующий вид:

```
rats = 101, rodents = 101
rats = 102, rodents = 102
адрес rats = 0x0065fd48, адрес rodents = 0x0065fd48
```

Нетрудно заметить, что переменные rats и rodents имеют одно и то же значение и один и тот же адрес. Увеличение значения rodents на 1 затрагивает обе переменные. Точнее, в результате выполнения операции rodents++ увеличивается значение единственной переменной, у которой имеется два имени. (Имейте в виду, несмотря на то, что этот пример показывает, как действует ссылка, он не может служить образцом типичного ее использования. Обычно ссылка применяется в качестве параметра функции, представляющего, в частности, структуру или объект. Мы вскоре рассмотрим эти виды применения ссылок.)

На первых порах освоение ссылок программистами, которые работали в среде C и перешли на C++, не проходит гладко, поскольку ссылки очень напоминают указатели, хотя между ними существуют отличия. Например, можно создать как ссылку, так и указатель, чтобы ссылаться на переменную rats:

```
int rats = 101;
int & rodents = rats;  // rodents - это ссылка
int * prats = &rats;   // prats - это указатель
```

Затем выражения rodents и *prats могут заменять имя rats, а выражения &rodents и prats могут подменять обозначение &rats. С этой точки зрения ссылка во многом подобна указателю в замаскированной записи, где операция разыменования * предполагается неявно. И, фактически, это в какой-то степени именно то, чем является ссылка. Однако между ссылками и указателями существуют различия помимо обозначений. Одно из таких различий состоит в том, что ссылку необходимо инициализировать в момент ее объявления. Нельзя сначала объявить ссылку, а затем присвоить ей значение, как это делается для указателей:

```
int rat;
int & rodent;
rodent = rat;          // Это действие недопустимо.
```

На память!

При объявлении ссылочную переменную необходимо инициализировать.

Ссылка во многом аналогична указателю со спецификатором const. Ее следует инициализировать в момент создания, после чего ссылка остается привязанной к определенной переменной до конца программы. Таким образом, конструкция

```
int & rodents = rats;
```

по сути, является замаскированной записью выражения, подобного следующему:

```
int * const pr = &rats;
```

В данном случае ссылка rodents играет ту же роль, что и выражение *pr.

В листинге 8.3 показано, что произойдет при попытке связать ссылку с переменной bunnies вместо переменой rats.

Листинг 8.3. secref.cpp

```
// secref.cpp -- определение и использование ссылки
#include <iostream>
int main()
{
   using namespace std;
   int rats = 101;
   int & rodents = rats;      // rodents - это ссылка
   cout << "rats = " << rats;
   cout << ", rodents = " << rodents << endl;
   cout << "адрес rats = " << &rats;
   cout << ", адрес rodents = " << &rodents << endl;
   int bunnies = 50;
   rodents = bunnies;         // можно ли изменить ссылку?
   cout << "bunnies = " << bunnies;
   cout << ", rats = " << rats;
   cout << ", rodents = " << rodents << endl;
   cout << "адрес bunnies = " << &bunnies;
   cout << ", адрес rodents = " << &rodents << endl;
   return 0;
}
```

Программа генерирует следующий вывод:

```
rats = 101, rodents = 101
адрес rats = 0x0065fd44, адрес rodents = 0x0065fd44
bunnies = 50, rats = 50, rodents = 50
адрес bunnies = 0x0065fd48, адрес rodents = 0x0065fd4
```

Сначала переменная rodents ссылается на rats, но затем программа предпринимает попытку сделать rodents ссылкой на переменную bunnies:

```
rodents = bunnies;
```

В какой-то момент кажется, что эта попытка была удачной, поскольку переменная rodents вместо значения 101 принимает значение 50. Однако при ближайшем рассмотрении оказывается, что значение переменной rats также изменилось и стало равным 50. При этом переменные rats и rodents по-прежнему имеют один и тот же адрес, который отличается от адреса переменной bunnies. Поскольку rodents является псевдонимом переменной rats, оператор присваивания в действительности эквивалентен следующей конструкции:

```
rats = bunnies;
```

Эта конструкция означает следующее: "присвоить переменной rats значение переменной bunnies". Одним словом, ссылке можно присвоить значение за счет применения инициализирующего объявления, но не операции присваивания.

Для примера рассмотрим следующий фрагмент кода:

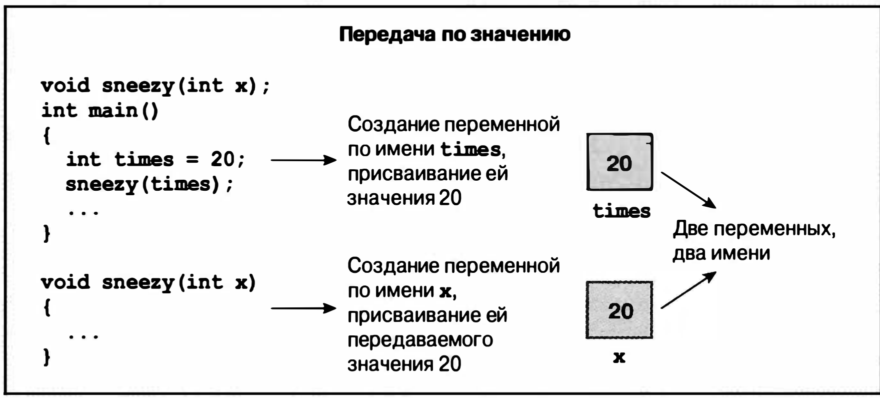
```
int rats = 101;
int * pi = &rats;
int & rodents = *pt;
```

```
int bunnies = 50;
pt = &bunnies;
```

Инициализация переменной `rodents` с присваиванием значения `*pt` приводит к тому, что `rodents` ссылается на переменную `rats`. Последующая попытка изменения переменной `pt` таким образом, чтобы она указывала на `bunnies`, не отменяет того факта, что `rodents` ссылается на `rats`.

Ссылки в роли параметров функций

Чаще всего ссылки используются в качестве параметров функции, при этом имя переменной в функции становится псевдонимом переменной из вызывающей программы. Такой метод передачи аргументов называется *передачей по ссылке*. Передача параметров по ссылке позволяет вызываемой функции получить доступ к переменным в вызывающей функции. Реализация этого средства в C++ представляет собой дальнейшее развитие основных принципов языка C, где возможна только передача по значению. Напомним, что передача по значению приводит к тому, что вызываемая функция оперирует копиями значений из вызывающей программы (рис. 8.2). Разумеется, язык C позволяет обойти ограничения, накладываемые передачей аргументов по значению, за счет применения указателей.



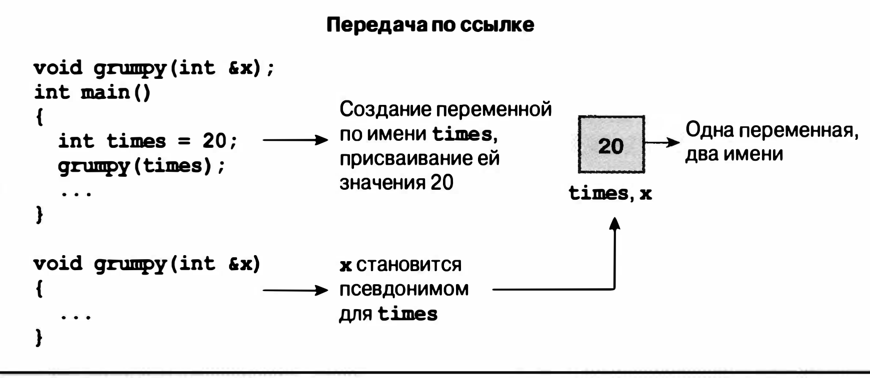


Рис. 8.2. Передача по значению и передача по ссылке

Теперь сравним, как используются ссылки и указатели при решении простой компьютерной задачи: обмен значениями двух переменных. Функция обмена должна иметь возможность изменять значения переменных в вызывающей программе. Это означает, что обычный подход, связанный с передачей переменных по значению, здесь неприемлем, поскольку функция выполнит обмен содержимым лишь копий исходных переменных, но не их самих. Однако если передавать ссылки, функция получит возможность работать с исходными данными. Вместо этого для получения доступа к исходным данным можно передавать указатели. Листинг 8.4 демонстрирует все три метода, включая и тот, который не дает желаемого результата, так что вы можете сравнить их.

Листинг 8.4. swaps.cpp

```
// swaps.cpp -- обмен значениями с помощью ссылок и указателей
#include <iostream>
void swapr(int & a, int & b);   // a, b - псевдонимы данных типа int
void swapp(int * p, int * q);   // p, q - адреса данных типа int
void swapv(int a, int b);       // a, b - новые переменные
int main()
{
   using namespace std;
   int wallet1 = 300;
   int wallet2 = 350;
   cout << "wallet1 = $" << wallet1;
   cout << " wallet2 = $" << wallet2 << endl;
   cout << "Использование ссылок для обмена содержимым:\n";
   swapr(wallet1, wallet2);    // передача переменных
   cout << "wallet1 = $" << wallet1;
   cout << " wallet2 = $" << wallet2 << endl;
   cout << "Использование указателей для повторного обмена содержимым:\n";
   swapp(&wallet1, &wallet2);  // передача адресов переменных
   cout << "wallet1 = $" << wallet1;
   cout << " wallet2 = $" << wallet2 << endl;
   cout << "Попытка использования передачи по значению:\n";
   swapv(wallet1, wallet2);    // передача значений переменных
   cout << "wallet1 = $" << wallet1;
   cout << " wallet2 = $" << wallet2 << endl;
   return 0;
}
void swapr(int & a, int & b)    // использование ссылок
{
   int temp;
   temp = a; // использование a, b для хранения значений переменных
   a = b;
   b = temp;
}
void swapp(int * p, int * q)    // использование указателей
{
   int temp;
   temp = *p; // использование *p, *q для хранения значений переменных
   *p = *q;
   *q = temp;
}
```

```
void swapv(int a, int b) // попытка использования значений
{
   int temp;
   temp = a; // использование a, b для хранения значений переменных
   a = b;
   b = temp;
}
```

Вывод программы имеет следующий вид:

```
wallet1 = $300 wallet2 = $350         ←исходные значения
Использование ссылок для обмена содержимым:
wallet1 = $350 wallet2 = $300         ←смена значений выполнена
Использование указателей для повторного обмена содержимым:
wallet1 = $300 wallet2 = $350         ←повторная смена значений
Попытка использования передачи по значению:
wallet1 = $300 wallet2 = $350         ←смена значений не удалась
```

Как и ожидалось, оба метода — метод указателей и метод ссылок — успешно реализовали обмен содержимым двух бумажников (wallet), в то время как метод передачи по значению завершился неудачей.

Замечания по программе

Прежде всего, обратите внимание на то, как вызывается каждая функция:

```
swapr(wallet1, wallet2);    // передача переменных
swapp(&wallet1, &wallet2);  // передача адресов переменных
swapv(wallet1, wallet2);    // передача значений переменных
```

Передача аргументов по ссылке (`swapr(wallet1, wallet2)`) и передача по значению (`swapv(wallet1, wallet2)`) выглядят идентичными. Единственный способ определить, что функция `swapr()` передает аргументы по ссылке — это обратиться к прототипу или к описанию функции. В то же время, наличие операции взятия адреса (`&`) явно говорит о том, что функции передается адрес значения (`swapp(&wallet1, &wallet2)`). (Напомним, что объявление типа `int *p` означает, что `p` — это указатель значения типа `int`, и поэтому аргумент, соответствующий `p`, должен быть адресом, таким как, например, `&wallet1`.)

Далее сравним программный код функций `swapr()` (передача по ссылке) и `swapv()` (передача по значению). Единственное видимое различие между ними связано с объявлением параметров:

```
void swapr(int & a, int & b)
void swapv(int a, int b)
```

Внутреннее различие между ними, естественно, состоит в том, что в функции `swapr()` переменные `a` и `b` служат псевдонимами имен `wallet1` и `wallet2`, так что обмен значениями между `a` и `b` вызывает обмен значениями между переменными `wallet1` и `wallet2`. В то же время в функции `swapv()` переменные `a` и `b` — это новые переменные, которые копируют значения переменных `wallet1` и `wallet2`. В этом случае обмен значениями между `a` и `b` никак не влияет на переменные `wallet1` и `wallet2`.

И, наконец, сравним функцию `swapr()` (передача по ссылке) и `swapp()` (передача указателей). Первое различие кроется в объявлении параметров:

```
void swapr(int & a, int & b)
void swapp(int * p, int * q)
```

Второе различие состоит в том, что вариант с указателем требует применения операции разыменования (`*`) во всех случаях, когда функция использует переменные `p` и `q`.

Как уже говорилось, ссылочную переменную необходимо инициализировать при ее определении. Можно считать, что аргументы-ссылки функции инициализируются с присваиванием значений, передаваемыми при ее вызове. Таким образом, вызов функции

```
swapr(wallet1, wallet2);
```

инициализирует формальный параметр `a` с присваиванием значения `wallet1`, а формальный параметр `b` — с присваиванием значения `wallet2`.

Свойства и особенности ссылок

Для использования ссылочных аргументов характерен ряд особенностей, о которых вам следует знать. Сначала обратимся к листингу 8.5. Представленная здесь программа использует две функции для возведения значения аргумента в куб. Одна из них принимает аргумент типа `double`, в то время как другая получает ссылку на значение типа `double`. Программный код процедуры возведения в куб выглядит несколько необычно, дабы нагляднее проиллюстрировать работу со ссылками.

Листинг 8.5. `cubes.cpp`

```
// cubes.cpp -- обычные и ссылочные аргументы
#include <iostream>
double cube(double a);
double refcube(double &ra);
int main ()
{
   using namespace std;
   double x = 3.0;
   cout << cube(x);
   cout << " = куб числа " << x << endl;
   cout << refcube(x);
   cout << " = куб числа " << x << endl;
   return 0;
}
double cube(double a)
{
   a *= a * a;
   return a;
}
double refcube(double &ra)
```

Ниже показан результат выполнения программы из листинга 8.5:

```
27 = куб числа 3
27 = куб числа 27
```

Обратите внимание, что функция refcube() изменяет значение x в функции main(), в то время как функция cube() этого не делает. Это напоминает причину, почему передача по значению является нормой. Переменная a является локальной для функции cube(). Она инициализируется значением x, однако изменения переменной a не отражаются на x. Тем не менее, поскольку функция refcube() использует в качестве аргумента ссылку, изменения, которые она вносит в переменную ra, фактически выполняются для переменной x. Если требуется, чтобы функция использовала передаваемую ей информацию, но не изменяла ее, и при этом использовать ссылку, следует воспользоваться постоянной ссылкой. В рассматриваемом примере следовало бы в прототипе и заголовке функции применить спецификатор const:

```
double refcube(const double &ra);
```

Однако в таком случае компилятор выводил бы сообщение об ошибке всякий раз, когда обнаруживал бы код, изменяющий значение переменной ra.

Если потребуется создать функцию с использованием идеи рассматриваемого примера, нужно выполнить передачу аргумента по значению, а не более экстравагантную передачу по ссылке. Ссылочные аргументы полезно применять для более крупных элементов данных, таких как структуры и классы, в чем вы вскоре убедитесь.

Функции, которые осуществляют передачу данных по значению, такие как cube() из листинга 8.5, могут использовать множество видов аргументов. Например, все приведенные ниже вызовы функции допустимы:

```
double z = cube(x + 2.0);  // вычисление выражения x + 2.0 и
                           // передача значение
z = cube(8.0);             // передача значения 8.0
int k = 10;
z = cube(k);               // приведения значения k к типу double и
                           // передача значения
double yo[3] = { 2.2, 3.3, 4.4 };
z = cube (yo[2]);          // передача значения 4.4
```

Предположим, что вы делаете попытку использовать аналогичные аргументы для функции со ссылочными параметрами. Создается впечатление, что передача ссылки сопряжена с большими ограничениями. В конце концов, если ra является альтернативным именем переменной a, то фактическим аргументом должна быть именно эта переменная. Подобное выражение

```
double z = refcube(x + 3.0);   // может привести к ошибке компиляции
```

по-видимому, не имеет смысла, поскольку выражение x + 3.0 не является переменной. Например, нельзя присвоить значение следующему выражению:

```
x + 3.0 = 5.0;  // не имеет смысла
```

Что произойдет при попытке выполнить обращение к функции наподобие такого: refcube(x + 3.0)? В современном языке C++ это ошибка, и некоторые компиляторы выведут сообщение об этом. Другие отобразят предостережение примерно такого содержания:

```
Warning: Temporary used for parameter 'ra' in call to refcube(double &)
Предупреждение: при вызове refcube(double &) в качестве параметра 'ra'
используется временная переменная.
```

Причина того, что это сообщение звучит не совсем категорично, заключается в том, что язык C++ в первые годы своего становления допускал передачу выражений в качестве ссылочных переменных. В некоторых случаях это допускается и сейчас. А происходит следующее: поскольку `x + 3.0` не является переменной типа `double`, программа создает временную переменную, не имеющую имени, инициализируя ее значением выражения `x + 3.0`. Затем `ra` становится ссылкой на эту временную переменную. Давайте рассмотрим временные переменные более подробно и выясним, в каких случаях они создаются и в каких — нет.

Временные переменные, ссылочные аргументы и спецификатор const

C++ может создавать временную переменную, если передаваемый аргумент не соответствует определению ссылочного аргумента. В настоящее время C++ допускает это только в случае, когда аргументом является ссылка со спецификатором `const`, но это тоже новое ограничение. Рассмотрим случаи, когда C++ генерирует временную переменную, и выясним, почему целесообразно вводить ограничение, требующее, чтобы ссылка использовалась со спецификатором `const`.

Прежде всего, в каких случаях создается временная переменная? При условии, что ссылочный параметр имеет спецификатор `const`, компилятор генерирует временную переменную в двух случаях:

- Когда тип фактического аргумента выбран правильно, но сам параметр не является *lvalue* (L-значением).
- Когда тип фактического параметра выбран неправильно, но может быть преобразован в правильный тип.

Аргумент, являющийся значением *lvalue* (L-значением), представляет собой объект данных, на который можно ссылаться. Примером может служить переменная, элемент массива, элемент структуры, ссылка и разыменованный указатель — все они являются L-значениями. К L-значениям не относятся литеральные константы и выражения, состоящие из нескольких элементов. Предположим, мы переопределили функцию `refcube()` так, что у нее имеется аргумент в виде ссылочной константы:

```
double refcube(const double &ra)
{
    return ra * ra * ra;
}
```

Теперь рассмотрим следующий программный код:

```
double side = 3.0;
double * pd = &side;
double & rd = side;
long edge = 5L;
double lens[4] = { 2.0, 5.0, 10.0, 12.0 };
double c1 = refcube(side);          // ra - это side
double c2 = refcube(lens[2]);       // ra - это lens[2]
```

```
double c3 = refcube(rd);               // ra - это rd и side
double c4 = refcube(*pd)               // ra - это *pd и side
double c5 = refcube(edge);             // ra - временная переменная
double c6 = refcube(7.0);              // ra - временная переменная
double c7 = refcube(side + 10.0);      // ra - временная переменная
```

Аргументы side, lens[2], rd и *pd — именованные объекты данных типа double, что позволяет генерировать ссылки на них. При этом временные переменные не нужны. (Напомним, что элемент массива ведет себя так же, как и переменная того же типа.) Однако объект edge, хоть и является переменной, имеет неверный тип. Ссылка на объект типа double не может ссылаться на данные типа long. Тип аргументов 7.0 и side + 10.0 допустим, но они не являются именованными объектами данных. В каждом из этих случаев компилятор генерирует временную анонимную переменную и превращает ra в ссылку на нее. Такие временные переменные существуют на протяжении обращения к функции, после чего компилятор может удалить их.

Почему такое поведение вполне оправдано для ссылок-констант и недопустимо в других случаях? Вернемся к функции swapr() из листинга 8.4:

```
void swapr(int & a, int & b)  // используются ссылки
{
   int temp;
   temp = a;   // a, b используются для хранения значений переменных
   a = b;
   b = temp;
}
```

Что произойдет, если выполнить представленный ниже программный код в условиях менее жестких правил ранних версий C++?

```
long a = 3, b = 5;
swapr(a, b);
```

Здесь имеет место несоответствие типов, поэтому компилятор создает две временных переменных типа int, инициализирует их значениями 3 и 5, а затем производит обмен содержимым временных переменных, оставляя при этом значения a и b неизменными.

Одним словом, если назначение функции со ссылочными аргументами состоит в том, чтобы модифицировать переменные, передаваемые в качестве аргументов, ситуация, которую создают временные переменные, препятствуют достижению этой цели. В этом случае решение заключается в том, чтобы запретить создание временных переменных, и новый стандарт C++ реализует именно этот подход. (Однако некоторые компиляторы по умолчанию выводят предупреждение вместо сообщения об ошибке. Поэтому не игнорируйте сообщения об использовании временных переменных.)

Теперь рассмотрим функцию refcube(). В ее задачу входит простое использование переданных значений без их модификации. В этом случае временные переменные не приносят никакого вреда; они придают функции более универсальный характер в отношении разнообразия аргументов, с которыми она способна работать. Следовательно, если объявление функции указывает, что ссылка имеет тип const, среда C++ при необходимости генерирует временные переменные там. По сути, функция C++, принимающая формальный ссылочный аргумент со спецификатором const и с несоответствующим фактическим аргументом имитирует традиционные

действия, выполняемые при передаче аргументов по значению. При этом гарантируется, что исходные данные не подвергнутся изменениям, а для хранения соответствующего значения применяется временная переменная.

На память!

Если передаваемый функции аргумент не является L-значением или не совместим по типу с соответствующим ссылочным `const`-параметром, C++ создает анонимную переменную требуемого типа и присваивает ей значение передаваемого функции аргумента. В результате параметр ссылается на эту переменную.

Используйте спецификатор `const` там, где это возможно

Существуют три "железных" довода в пользу объявления ссылочных аргументов в качестве ссылок на константы:

- Применение спецификатора `const` предотвращает программные ошибки, которые приводят к непреднамеренному изменению данных.
- Применение спецификатора `const` позволяет функции выполнять обработку формальных аргументов как со спецификатором `const`, так и без него. При этом функция, в прототипе которой спецификатор `const` опущен, может принимать только данные, не имеющие этого спецификатора.
- Использование ссылки со спецификатором `const` позволяет функции генерировать и использовать временные переменные по мере необходимости.

Рекомендуется объявлять формальные ссылочные аргументы со спецификатором `const` во всех случаях, когда для этого имеется возможность.

Использование ссылок при работе со структурами

Ссылки очень хорошо сочетаются со структурами и классами, то есть с типами данных C++, которые определяет пользователь. Собственно говоря, ссылки были введены, прежде всего, для использования именно с этими типами, а не с основными встроенными типами данных.

Метод использования ссылок на структуры ничем не отличается от метода применения ссылок на переменные основных типов. Достаточно воспользоваться оператором ссылки `&` при объявлении параметра структуры. Программа из листинга 8.6 именно это и делает. Кроме того, она реализует интересную идею — функция возвращает ссылку на структуру. Такое поведение несколько отличается от случая, когда функция возвращает структуру. Потребуется принять некоторые меры предосторожности. Часто лучше будет присвоить возвращаемой ссылке спецификатор `const`. Эти моменты будут разъяснены в нескольких последующих примерах. Программа содержит функцию `use()`, которая отображает два элемента структуры и выполняет приращение значения третьего элемента. Таким образом, третий элемент отслеживает количество обращений к определенной структуре со стороны функции `use()`.

Листинг 8.6. `strtref.cpp`

```
// strtref.cpp -- использование ссылок на структуру
#include <iostream>
using namespace std;
```

```
struct sysop
{
   char name[26];
   char quote[64];
   int used;
};
const sysop & use(sysop & sysopref); // функция, возвращающая ссылку
int main()
{
// ПРИМЕЧАНИЕ: некоторые реализации требуют использования
// ключевого слова static в двух объявлениях структуры,
// чтобы сделать возможной инициализацию
  sysop looper =
  {
    "Рик \"Фортран\" Цикличный",
    "Я выполняю переходы.",
         0
   };
   use(looper); // looper имеет тип sysop
   cout << "Looper: " << looper.used << " вызов(а)\n";
   sysop copycat;
   copycat = use(looper);
   cout << "Looper: " << looper.used << " вызов(а)\n";
   cout << "Copycat: " << copycat.used << " вызов(а)\n";
   cout << "use(looper): " << use(looper).used << " вызов(а)\n";
   return 0;
}
// use() возвращает передаваемую ей ссылку
const sysop & use(sysop & sysopref)
{
   cout << sysopref.name << " говорит:\n";
   cout << sysopref.quote << endl;
   sysopref.used++;
   return sysopref;
}
```

Программа из листинга 8.6 генерирует следующий вывод:

```
Рик "Фортран" Цикличный говорит:
Я выполняю переходы.
Looper: 1 вызов(а)
Рик "Фортран" Цикличный говорит:
Я выполняю переходы.
Looper: 2 вызов(а)
Copycat: 2 вызов(а)
Рик "Фортран" Цикличный говорит:
Я выполняю переходы.
use(looper): 3 вызов(а)
```

Замечания по программе

Программа охватывает три новых области. Первая заключается в использовании ссылки на структуру, что наглядно демонстрирует первый вызов функции:

```
use(looper);
```

При этом функции `use()` передается по ссылке структура `looper`, благодаря чему идентификатор `sysopref` становится синонимом структуры `looper`. Когда функция `use()` отображает элементы `name` и `quote` структуры `sysopref`, на самом деле отображаются элементы структуры `looper`. Кроме того, когда функция увеличивает значение `sysopref.used` на 1, она фактически увеличивает значение `looper.used`, как это видно из выходных данных программы:

```
Рик "Фортран" Цикличный говорит:
Я выполняю переходы.
1 вызов(а)
```

Вторым новшеством является использование ссылки в качестве возвращаемого значения. Обычно механизм возврата копирует возвращаемое значение во временную область памяти, к которой вызывающая программа затем осуществляет доступ. Однако возврат ссылки означает, что вызывающая программа выполняет прямой доступ к возвращаемому значению без использования приготовленной заранее копии. В обычных случаях такая ссылка указывает на ссылку, которая вначале была передана функции, так что вызывающая функция в результате осуществляет прямой доступ к одной из собственных переменных. Здесь, например, `sysopref` является ссылкой на `looper`, поэтому возвращаемое значение представляет собой переменную `looper`, ранее определенную в функции `main()`. Рассмотрим следующую строку кода:

```
copycat = use(looper);
```

Если бы функция `use()` просто возвращала структуру, содержимое `sysopref` копировалось бы во временную область хранения возвращаемых значений. Затем содержимое этой области копировалось бы в переменную `copycat`. Однако, поскольку функция `use()` возвращает ссылку на `looper`, содержимое этой переменной будет скопировано непосредственно в `copycat`. Это иллюстрирует одно из основных преимуществ возврата ссылки на структуру вместо самой структуры — эффективность.

На память!

Функция, которая возвращает ссылку, фактически является псевдонимом переменной, на которую ссылается.

Третья новинка, которая исследуется в программе, заключается в использовании вызова функции для доступа к элементу структуры:

```
cout << "use(looper): " << use(looper).used << " вызов(а)\n";
```

Поскольку функция `use()` возвращает ссылку на структуру `looper`; эта строка кода приводит к такому же результату, что и две следующих строки:

```
use(looper);
cout << "use(looper): " << looper.used << " вызов(а)\n";
```

Запись `use(looper).used` реализует доступ к элементу структуры `looper`. Если бы функция возвращала структуру вместо ссылки на нее, данный код осуществлял бы доступ к элементу `used` временной копии возвращаемого значения `looper`.

Будьте осмотрительными при выборе объекта, на который указывает возвращаемая ссылка

Важнее всего избегать ситуации, когда возвращаемая ссылка указывает на область памяти, которая прекращает существование после завершения работы функции. Ниже приводится пример кода, который не следует использовать:

```
const sysop & clone2(sysop & sysopref)
{
  sysop newguy;             // первый шаг к серьезной ошибке
  newguy = sysopref;        // копирование информации
  return newguy;            // возврат ссылки на копию
}
```

В результате выполнения этого кода возвращается ссылка на временную переменную (newguy), которая прекращает существование сразу после завершения работы функции. (Время жизни переменных различного вида обсуждается в главе 9.) Подобно этому следует избегать ситуаций, когда на такие временные переменные возвращаются указатели.

Проще всего избежать подобной ошибки путем реализации возврата ссылки, которая была передана функции в качестве аргумента. Такой параметр будет ссылаться на данные, используемые вызывающей функцией. Таким образом, возвращаемая ссылка будет ссылаться на те же данные.

Второй метод заключается в использовании операции new для создания новой области хранения. Мы уже рассматривали примеры, когда операция new создает область хранения для строки, а функция возвращает на эту область указатель. Вот как решается подобная задача с помощью ссылки:

```
const sysop & clone(sysop & sysopref)
{
   sysop * psysop = new sysop;
   *psysop = sysopref;      // копирование информации
   return *psysop;          // возврат ссылки на копию
}
```

Первый оператор создает безымянную структуру sysop. Указатель psysop указывает на эту структуру, поэтому выражение *psysop означает саму структуру. Создается впечатление, что приведенный выше фрагмент программы возвращает структуру. Однако объявление функции указывает, что она возвращает ссылку на эту структуру. Затем функцию можно использовать следующим образом:

```
sysop & jolly = clone(looper);
```

В результате переменная jolly становится ссылкой на новую структуру. Такое решение связано с одним затруднением: потребуется использовать оператор delete для освобождения памяти, выделенной операцией new, когда она более не будет использоваться. Вызов функции call() скрывает обращение к операции new, в результате чего будет сложнее впоследствии вспомнить о необходимости применить операцию delete. В главе 16 рассматривается шаблон auto_ptr, который помогает автоматизировать процесс освобождения памяти.

Причины использования спецификатора const при объявлении возвращаемой ссылки

В предыдущем примере функция `use()` возвращает значение `const sysop &`. Может возникнуть вопрос о назначении спецификатора `const` в данном случае. Он не означает, что структура `sysop` сама по себе является константой. Смысл спецификатора лишь в том, что возвращаемое значение нельзя напрямую использовать для модификации структуры. Например, при отсутствии спецификатора `const` был бы возможным следующий код:

```
use(looper).used = 10;
```

Поскольку функция `use()` возвращает ссылку на структуру `looper`, этот код будет иметь такой же эффект, как и следующий фрагмент:

```
use(looper);         // отображение структуры, приращение элемента used
looper.used = 10;    // повторная установка значения 10 для used
```

Рассмотрим еще один вариант:

```
sysop newgal = {"Polly Morf", "Polly's not a hacker.", 0};
use(looper) = newgal;
```

Той же цели можно добиться с помощью следующего кода:

```
sysop newgal = {"Polly Morf", "Polly's not a hacker.", 0};
use(looper);       // отображение структуры, приращение элемента used
looper = newgal;   // замена содержимого looper содержимым newgal
```

Одним словом, опуская спецификатор `const` можно создавать более краткий, но более сложный для понимания код.

Как правило, следует избегать конструкций, смысл которых неочевиден. Это повышает вероятность сложных для выявления ошибок. Когда в качестве возвращаемого значения используется ссылка со спецификатором `const`, это помогает преодолеть искушение "навести тень на плетень". Иногда имеет смысл опустить спецификатор `const`. Примером может служить перегруженная операция <<, которая рассматривается в главе 11.

Использование ссылок на объект класса

В языке C++ для передачи функциям объектов классов обычно практикуется использование ссылок. Например, ссылочные параметры используются в функциях, принимающих объекты классов `string`, `ostream`, `istream`, `ofstream` и `ifstream` в качестве аргументов.

Рассмотрим пример, где используется класс `string` и демонстрируются различные решения, в том числе и неудачные. Замысел состоит в том, чтобы создать функцию, которая присоединяет данную строку к обеим сторонам другой строки. В листинге 8.7 представлены три функции, предназначенные для решения этой задачи. Однако одно из решений настолько ошибочное, что может привести к сбою программы или даже отказу компиляции.

Листинг 8.7. strquote.cpp

```
// strquote.cpp -- различные решения
#include <iostream>
#include <string>
using namespace std;
string version1(const string & s1, const string & s2);
const string & version2(string & s1, const string & s2);  // побочный эффект
const string & version3(string & s1, const string & s2);  // неудачное решение

int main()
{
   string input;
   string copy;
   string result;

   cout << "Введите строку: ";
   getline(cin, input);
   copy = input;
   cout << "Вы ввели строку: " << input << endl;
   result = version1(input, "***");
   cout << "Измененная строка: " << result << endl;
   cout << "Исходная строка: " << input << endl;

   result = version2(input, "###");
   cout << "Измененная строка: " << result << endl;
   cout << "Исходная строка: " << input << endl;

   cout << "Восстановление исходной строки.\n";
   input = copy;
   result = version3(input, "@@@");
   cout << "Измененная строка: " << result << endl;
   cout << "Исходная строка: " << input << endl;
   return 0;
}
string version1(const string & s1, const string & s2)
{
   string temp;
   temp = s2 + s1 + s2;;
   return temp;
}
const string & version2(string & s1, const string & s2)  // побочный эффект
{
   s1 = s2 + s1 + s2;
  // возврат ссылки, переданной функции, надежен
  return s1;
}
const string & version3(string & s1, const string & s2)  // неудачное решение
{
   string temp;
   temp = s2 + s1 + s2;
   // возврат ссылки на локальную переменную ненадежен
   return temp;
}
```

Ниже показан пример выполнения программы из листинга 8.7:

```
Введите строку: Это не моя вина.
Вы ввели строку: Это не моя вина.
Измененная строка: ***Это не моя вина.***
Исходная строка: Это не моя вина.
Измененная строка: ###Это не моя вина.###
Исходная строка: ###Это не моя вина.###
Восстановление исходной строки.
```

Здесь программа дает сбой.

Замечания по программе

Первый вариант функции наиболее прямолинеен:

```
string version1(const string & s1, const string & s2)
{
  string temp;
  temp = s2 + s1 + s2;;
  return temp;
}
```

Функция принимает два аргумента типа string и использует класс string для создания новой строки, которая обладает требуемыми свойствами. Обратите внимание, что оба аргумента функции представляют собой ссылки со спецификатором const. Результат работы функции такой же, как если бы ей передавались два объекта типа string:

```
string version4(string s1, string & s2) // тот же результат
```

В этом случае аргументы s1 и s2 являются вновь созданными объектами типа string. Таким образом, применять ссылки эффективнее, поскольку в этом случае функция не будет создавать новые объекты и копировать в них данные из исходных объектов. Здесь спецификатор const указывает, что функция использует, но не изменяет исходные строки.

Объект temp является новым и локальным по отношению к функции version1(). Он прекращает существование после завершения работы функции. Таким образом, возврат объекта temp в качестве ссылки невозможен, поэтому для функции задан тип string. Это означает, что содержимое объекта temp будет скопировано во временную область хранения возвращаемых значений. Затем в функции main() содержимое области хранения копируется в строку с именем result:

```
result = version1(input, "***");
```

Передача строкового аргумента в стиле С параметру-ссылке на объект типа string

Вы могли обратить внимание на интересную особенность функции Version1(): оба формальных параметра (s1 и s2) имеют тип const string &, но фактические аргументы (input и "***") имеют тип string и const char * соответственно. Поскольку аргумент input имеет тип string, ссылка переменной s1 на него не вызывает затруднений. Но как программа воспримет передачу указателя на тип char в качестве аргумента ссылке на тип string?

Здесь имеют значение два момента. Первое, класс string определяет преобразование типа char * в string, что делает возможным инициализации объекта типа string строковым значением, соответствующим спецификации языка C (строкой C-стиля). Второй момент связан со свойством ссылочных формальных параметров, имеющих спецификатор const. Оно уже обсуждалось в этой главе. Предположим, тип фактического аргумента не соответствует типу ссылочного параметра, но может быть преобразован в него. Тогда программа создаст временную переменную необходимого типа, инициализирует ее преобразованным значением и передаст ссылку на временную переменную. Ранее приводился пример, что параметр типа const double & может подобным образом обрабатывать аргумент типа int. Аналогично этому, параметр типа const string & может обрабатывать аргумент char * или const char *.

Положительный результат заключается в том, что формальный параметр типа const string & допускает использование объекта типа string или строки C-стиля в качестве фактического аргумента, передаваемого функции. Примером строки C-стиля может служить строковый литерал в кавычках, массив символов (char) с завершающим нулем или переменная-указатель на тип char. Поэтому следующая строка кода выполняется корректно:

```
result = version1(input, "***");
```

Функция version2() не создает временную строку. Вместо этого она напрямую изменяет исходную строку:

```
const string & version2(string & s1, const string & s2)  // побочный эффект
{
   s1 = s2 + s1 + s2;
   // возврат ссылки, переданной функции, надежен
   return s1;
}
```

Эта функция может изменять значение s1, поскольку переменная s1, в отличие от s2, объявлена без спецификатора const.

Поскольку s1 является ссылкой на объект (input) функции main(), возврат этой переменной в качестве ссылки вполне допустим. По той же причине строка:

```
result = version2(input, "###");
```

эквивалентна следующему фрагменту кода:

```
version2(input, "###"); // input прямо изменен функцией version2()
result = input; // ссылка на s1 является ссылкой на input
```

Однако в результате возникает побочный эффект — изменение значения input:

```
Исходная строка: Это не моя вина.
Измененная строка: ###Это не моя вина.###
Исходная строка: ###Это не моя вина.###
```

Таким образом, если исходная строка не должна изменяться, такое решение ошибочно.

Третий вариант функции в листинге 8.7 служит напоминанием о возможной ошибке:

```
const string & version3(string & s1, const string & s2) //ошибочное решение
{
   string temp;
   temp = s2 + s1 + s2;
   // возвращать ссылку на локальную переменную ненадежно
   return temp;
}
```

Он содержит неисправимую ошибку — возврат ссылки на переменную, объявленную локально внутри функции version3().

Эта функция компилируется (с выводом предупреждающего сообщения), но при попытке выполнения программы происходит сбой. Непосредственно ошибку вызывает следующая операция присваивания:

```
result = version3(input, "@@@");
```

Осуществляется попытка ссылки на адрес памяти, который более не используется.

Еще один урок ООП: объекты, наследование и ссылки

Классы ostream и ofstream проявляют интересное свойство ссылок. Как упоминалось в главе 6, объекты типа ofstream могут использовать методы ostream, что позволяет для файловых операций ввода-вывода применять те же формы, что и для консольного ввода-вывода. Особенность языка, позволяющая передавать свойства из одного класса в другой, называется *наследованием*. Оно подробно рассматривается в главе 13. Короче говоря, ostream называется базовым классом (поскольку класс ofstream основан на нем), а класс ofstream носит название производного (так как наследует класс ostream). Производный класс наследует методы базового класса. Это означает, что объект ofstream может использовать функции базового класса, такие как методы форматирования precision() и setf().

Второй аспект наследования состоит в том, что ссылка на базовый класс может указывать на объект производного класса, не требуя приведения типов. На практике это позволяет определять функцию, обладающую параметром-ссылкой на базовый класс. Эта функция может взаимодействовать с объектами базового класса, а также с производными объектами. Например, функция с параметром типа ostream & может принимать объект ostream, такой как cout или ofstream. Их можно объявлять в функции с одинаковым успехом.

Указанные аспекты демонстрируются в листинге 8.8, где одна и та же функция используется для записи данных в файл и отображения тех же данных на экране. Изменяется только аргумент, который передается вызываемой функции. Представленная программа рассчитывает фокусное расстояние объектива телескопа (его основного зеркала или линзы) и отдельных окуляров. Затем вычисляется кратность увеличения каждого окуляра телескопа. Кратность увеличения равна фокусному расстоянию телескопа, деленному на фокусное расстояние используемого окуляра, так что вычисления здесь несложные. Кроме того, в программе используются некоторые методы форматирования, которые, как обещалось, одинаково успешно выполняются как с объектами типа cout, так и с объектами типа ofstream (в нашем примере — fout).

Листинг 8.8. filefunc.cpp

```
//filefunc.cpp -- функция с параметром типа ostream &
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

void file_it(ostream & os, double fo, const double fe[],int n);
const int LIMIT = 5;
```

```
int main(void)
{
   ofstream fout;
   const char * fn = "ep-data.txt";
   fout.open(fn);
   if (!fout.is_open())
   {
     cout << "Невозможно открыть " << fn << ". Пока.\n";
     exit(EXIT_FAILURE);
   }
   double objective;
   cout << "Введите фокусное расстояние "
   "объектива телескопа в мм: ";
   cin >> objective;
   double eps[LIMIT];
   cout << "Введите фокусные расстояния, в мм, " << LIMIT << "  окуляров:\n";
   for (int i = 0; i < LIMIT; i++)
   {
     cout << "Окуляр #" << i + 1 << ": ";
     cin >> eps[i];
   }
   file_it(fout, objective, eps, LIMIT);
   file_it(cout, objective, eps, LIMIT);
   cout << "Конец\n";
   return 0;
}
void file_it(ostream & os, double fo, const double fe[],int n)
{
   ios_base::fmtflags initial;
   initial = os.setf(ios_base::fixed);  // сохранение исходного
                                        // состояния форматирования
   os.precision(0);
   os << "Фокусное расстояние объектива: " << fo << " мм\n";
   os.setf(ios::showpoint);
   os.precision(1);
   os.width(12);
   os << "коэффициент увеличения";
   os.width(15);
   os << "окуляра" << endl;
   for (int i = 0; i < n; i++)
   {
     os.width(12);
     os << fe[i];
     os.width(15);
     os << int (fo/fe[i] + 0.5) << endl;
   }
   os.setf(initial);  // восстановление исходного состояния форматирования
}
```

Ниже показан пример выполнения программы из листинга 8.8:

```
Введите фокусное расстояние объектива телескопа в мм: 1800
Введите фокусные расстояния, в мм, 5 окуляров:
```

```
Окуляр #1: 30
Окуляр #2: 19
Окуляр #3: 14
Окуляр #4: 8.8
Окуляр #5: 7.5
Фокусное расстояние объектива: 1800 мм
коэффициент увеличения окуляра
30.0     60
19.0     95
14.0     129
8.8      205
7.5      240
Конец
```

Строка

```
file_it(fout, objective, eps, LIMIT);
```

записывает данные окуляра в файл ep-data.txt, а строка

```
file_it(cout, objective, eps, LIMIT);
```

выводит идентичную информацию в том же формате на экран.

Замечания по программе

Главная идея программы из листинга 8.8 состоит в демонстрации того факта, что параметр типа ostream & может ссылаться на объект ostream вроде cout и на объект ofstream, такой как fout. Однако программа также иллюстрирует использование методов форматирования объекта ostream для обоих типов параметров. (Более подробно эта тема рассматривается в главе 17.)

Метод setf() позволяет устанавливать различные состояния форматирования. Например, при вызове метода setf(ios_base::fixed) объект переводится в режим использования фиксированной десятичной точки. При вызове метода setf(ios_base::showpoint) объект переводится в режим отображения завершающей десятичной точки, даже если последующие цифры представляют нули. Метод precision() указывает количество цифр, отображаемых справа от десятичной точки (если объект выводится в режиме fixed). Все эти установки сохраняются до тех пор, пока не будут изменены в результате следующего вызова метода. В результате вызова метода width() устанавливается ширина поля для следующей операции вывода. Эта установка действует только для отображения единственного значения, а затем возвращается в принимаемое по умолчанию состояние. (По умолчанию ширина поля равна нулю. Затем эта величина повышается до точного соответствия отображаемому значению.)

Функция file_it() содержит два интересных вызова метода:

```
ios_base::fmtflags initial;
initial = os.setf(ios_base::fixed);  // сохранение исходного
                                     // состояния форматирования
...
os.setf(initial);    // восстановление исходного
                     // состояния форматирования
```

Метод setf() возвращает копию всех установок форматирования, которые действовали до его вызова. Обозначение ios_base::fmtflags является причудливым именем типа данных, необходимых для хранения этой информации. В результате операции присваивания в переменной initial сохраняются настройки, которые действовали на момент вызова функции file_it(). Затем переменная initial может использоваться в качестве аргумента функции setf(), чтобы восстановить исходные значения всех установок форматирования. Таким образом, эта функция восстанавливает состояние объекта, которое существовало до его передачи функции file_it().

Более полное знакомство с классами поможет вам лучше уяснить работу этих методов, а также причину, по которой в программе часто используются обозначения вроде ios_base. Однако применение рассмотренных методов не обязательно откладывать до момента прочтения главы 17.

И последнее: каждый объект хранить собственные параметры форматирования. Поэтому, когда программа передает объект cout функции file_it(), его настройки форматирования изменяются, а затем восстанавливаются. То же самое происходит с объектом fout, когда он передается функции file_it().

Когда целесообразно использовать ссылочные аргументы

Существуют две главных причины использования ссылочных аргументов:

- Чтобы сделать возможным изменение объекта данных вызывающей функции.
- Чтобы ускорить работу программы за счет передачи ссылки вместо полной копии объекта данных.

Вторая причина наиболее важна для крупных объектов данных, таких как структуры и объекты классов. По этим же двум причинам в качестве аргумента может использоваться указатель. Это оправдано, поскольку аргументы-ссылки, по сути, являются лишь альтернативным интерфейсом для кода, где применяются указатели. Итак, в каких случаях следует использовать ссылку, указатель или передачу по значению? Ниже приводятся рекомендации.

Функция использует передаваемые данные без их изменения:

- Если объект данных имеет малый размер, например, значение встроенного типа данных или небольшая структура, передавайте его по значению.
- Если объект данных представляет собой массив, используйте указатель, поскольку это единственно приемлемый вариант. Объявите указатель со спецификатором const.
- Если объект данных является структурой приемлемого размера, используйте const-указатель или const-ссылку для увеличения эффективности программы. В этом случае удастся сохранить время и пространство, необходимое для копирования структуры или строения класса. Объявите указатель или ссылку со спецификатором const.
- Если объект данных является объектом класса, используйте ссылку со спецификатором const. Семантика структуры класса часто требует применения ссылки. Эта главная причина добавления этого новшества в язык C++. Таким образом, стандартом является передача объектов класса по ссылке.

Функция изменяет данные вызывающей функции:

- Если объект данных относится к встроенным типам, используйте указатель. Если в коде встретилось выражение вида `fixit(&x)`, где `x` имеет тип `int`, это явно означает, что данная функция должна изменять значение `x`.
- Если объект данных представляет собой массив, остается один вариант — указатель.
- Если объект данных является структурой, можно использовать ссылку или указатель.
- Когда объект данных представляет собой объект класса, следует использовать ссылку.

Конечно, это лишь рекомендации, и могут существовать причины для других решений. Например, объект `cin` использует ссылки на базовые типы данных, поэтому вместо записи `cin >> &n` можно применять запись `cin >> n`.

Аргументы, определяемые по умолчанию

Теперь рассмотрим еще одно новое инструментальное средство языка C++ — *аргументы, определяемые по умолчанию.* Такие аргументы представляют собой значения, которые используются автоматически, если соответствующие фактические параметры в обращении к функции опущены. Например, если функция `void wow(int n)` описана так, что `n` по умолчанию имеет значение `1`, то обращение к функции `wow()` означает то же самое, что и вызов `wow(1)`. Это свойство позволяет более гибко использовать функции. Предположим, что функция `left()` возвращает первые `n` символов строки, при этом сама строка и число `n` являются аргументами. Точнее, функция возвращает указатель на новую строку, представляющую собой выбранный фрагмент исходной строки. Например, в результате вызова функции `left("theory", 3)` создается новая строка `"the"` и возвращается указатель на нее. Теперь предположим, что для второго аргумента по умолчанию установлено значение `1`. Вызов `left("theory", 3)` будет выполнен, как и раньше, поскольку указанная величина `3` переопределит значение, принимаемое по умолчанию. Однако вызов `left("theory")` теперь уже не будет ошибочным. Он подразумевает, что значение второго аргумента равно `1`, поэтому будет возвращен указатель на строку `"t"`. Этот вид выбора значений по умолчанию удобен для программ, которые часто извлекают строки длиной в один символ, но иногда требуется извлекать более длинные строки.

Как установить значение по умолчанию? Для этой цели следует воспользоваться прототипом функции. Поскольку компилятор использует прототип, чтобы узнать, сколько аргументов имеет функция, прототип функции также должен сообщить программе о возможности использования аргументов, заданных по умолчанию. Метод состоит в том, что значения аргументам присваиваются в прототипе. Например, пусть имеется прототип, соответствующий следующему описанию функции `left()`:

```
char * left(const char * str, int n = 1);
```

Требуется, чтобы эта функция возвращала новую строку, поэтому ее типом будет `char*`, то есть указатель на значение типа `char`. Если нужно сохранить исходную строку неизменной, следует использовать спецификатор `const` для первого аргумента. А чтобы аргумент `n` имел значение `1`, определенное по умолчанию, присвоим это

значение аргументу n. Принимаемое по умолчанию значение аргумента — это значение, задаваемое при инициализации. Таким образом, данный прототип инициализирует n значением 1. Если аргумент n опускается, для него принимается значение 1, но если данный аргумент передается, то новое значение переопределит значение 1.

Для функции со списком аргументов принимаемые по умолчанию значения присваиваются в направлении справа налево. Иначе говоря, нельзя присвоить значение по умолчанию некоторому аргументу, пока не будут присвоены значения для всех остальных аргументов, размещенных справа от него:

```
int harpo(int n, int m = 4, int j = 5);                  // ПРАВИЛЬНО
int chico(int n, int m = 6, int j);                      // НЕПРАВИЛЬНО
int groucho(int k  =  1,  int m = 2,  int n = 3);        // ПРАВИЛЬНО
```

Например, прототип функции harpo() допускает реализацию вызова функции с одним, двумя или тремя аргументами:

```
beeps = harpo(2);          // то же, что и harpo(2,4,5)
beeps = harpo(1,8);        // то же, что и (1,8,5)
beeps = harpo (8,7,6);     // аргументы, определенные по умолчанию,
                           // не используются
```

Фактические значения присваиваются соответствующим формальным аргументам в направлении слева направо. Пропуск аргументов не допускается. Таким образом, следующее выражение не допускается:

```
beeps = harpo(3, ,8);      // неправильно, аргументу m не будет
                           // присвоено значение 4
```

Аргументы, определенные по умолчанию, не являются особо выдающимся достижением в программировании. Они лишь служат удобным дополнением. Когда вы перейдете к разработке классов, то убедитесь в том, что этот прием позволяет сократить количество конструкторов, методов и перегрузок методов, которые необходимо определить.

Программа в листинге 8.9 находит применение аргументам по умолчанию. Обратите внимание, что только в прототипе указаны принимаемые по умолчанию значения. На определении функции наличие принимаемых по умолчанию аргументов не отражается.

Листинг 8.9. `left.cpp`

```
// left.cpp -- функция с определенным по умолчанию
// аргументом, предназначенная для обработки строк
#include <iostream>
const int ArSize = 80;
char * left(const char * str, int n = 1);
int main()
{
    using namespace std;
    char sample[ArSize];
    cout << "Введите строку:\n";
    cin.get(sample,ArSize);
    char *ps = left(sample, 4);
    cout << ps << endl;
    delete [] ps;     // удаление старой строки
```

```
  ps = left(sample);
  cout << ps << endl;
  delete [] ps;     // удаление новой строки
  return 0;
}
// Эта функция возвращает указатель на новую строку, состоящую
// из n первых символов строки str.
char * left(const char * str, int n)
{
  if(n < 0)
    n = 0;
  char * p = new char[n+1];
  int i;
  for (i = 0; i < n && str[i]; i++)
    p[i] = str[i];  // копирование символов
  while (i <= n)
    p[i++] = '\0';  // установка значений '\0' для оставшейся
                    // части строки
  return p;
}
```

Ниже показан пример выполнения программы из листинга 8.9:

```
Введите строку:
грядущее
гряд
г
```

Замечания по программе

Программа использует операцию `new` для создания новой строки, в которой будут храниться выбранные символы. Один из нежелательных вариантов заключается в том, что пользователь может испытать программу на прочность, указав отрицательное количество символов. В этом случае функция сбрасывает счетчик символов до 0 и в конечном итоге возвращает пустую строку. Еще одна нежелательная ситуация возникает, когда пользователь запрашивает количество символов, превышающее длину исходной строки. Функция защищена от подобных случаев за счет выполнения комбинированной проверки:

```
i < n && str[i]
```

В результате осуществления проверки `i < n` выполнение цикла прекратится после того, как будут скопированы `n` символов. Вторая часть условия проверки, выражение `str[i]`, анализирует код символа, который должен копироваться в данный момент. Если цикл достигнет нулевого символа, кодом которого является 0, выполнение цикла прекратится. Заключительный цикл `while` завершает строку нулевым символом, а затем заполняет остаток выделенного под строку пространства памяти, если таковое имеется, нулевыми символами.

Другой способ установки размера новой строки состоит в том, чтобы присвоить переменной `n` наименьшую величину из переданного значения и длины строки:

```
int len = strlen(str);
n = (n < len) ? n : len;    // наименьшее из n и len
char * p = new char[n+1];
```

Это гарантирует, что операция new не выделит объем памяти, превышающий необходимый размер для хранения строки. Это может оказаться полезным в случае примерно такого обращения к функции: left("Hi!", 32767). В случае первого варианта установки размера строка "Hi!" копируется в массив размером 32767 символов, при этом всем его элементам, за исключением первых трех, присваиваются нулевые символы. При реализации второго подхода строка "Hi!" копируется в массив, состоящий из четырех символов. Но, в то же время, добавление в программу еще одного вызова функции (strlen()) приводит к увеличению размеров программы, замедляет процесс ее выполнения и требует, чтобы разработчик не забыл включить в код заголовочный файл cstring (или string.h). Программисты, работающие на С, предпочитают иметь более компактный программный код, обеспечивающий высокое быстродействие программы, вследствие чего на них ложится ответственность за правильное использование функций. Однако в С++ основное значение традиционно придается надежности программного продукта. В конце концов, более медленная, зато правильно выполняющаяся программа лучше, нежели быстродействующая программа, которая работает неправильно. Если время, затраченное на вызов функции strlen(), неприемлемо, можно позволить непосредственно функции left() выбрать наименьшее из значений n и длины строки. Например, представленный ниже цикл прекращает свое выполнение, когда величина m станет равной значению n или длине строки, независимо от того, что из них меньше:

```
int m = 0;
while ( m <= n && str[m] != '\0')
   m++;
char * p = new char[m+1]:
// использовать m вместо n до конца программного кода
```

Перегрузка функций

Полиморфизм функций — это расширение возможностей языка С, реализованное исключительно в С++. Аргументы, определенные по умолчанию, позволяют вызывать одну и ту же функцию с указанием различного количества аргументов. А *полиморфизм функций*, также называемый *перегрузкой функций*, предоставляет возможность использовать несколько функций с одним именем. Слово *полиморфизм* означает способность иметь множество форм, следовательно, в нашем случае полиморфизм позволяет функции иметь множество форм. Подобно этому, выражение *перегрузка функций* означает возможность привязки более чем одной функции к одному и тому же имени, что позволяет говорить о перегрузке имен. Оба выражения означают одно и то же, но мы будем пользоваться выражением "перегрузка функций" — оно звучит более строго. Перегрузкой функций можно воспользоваться для разработки семейства функций, которые, по сути, выполняют одно и то же, но с применением различных списков аргументов.

Перегруженные функции подобны глаголам, имеющим несколько значений. Например, глагол "болеть" в выражениях "Сергей болеет за футбольный клуб Динамо" и "Артем болеет уже третий день" имеет различный смысл. Контекст вам подскажет (надо надеяться), какое из значений подходит для того или иного случая. Подобно этому, среда С++ использует контекст, чтобы решить, какой версией перегруженной функции следует воспользоваться.

Главную роль в перегрузке функций играет список аргументов, который еще называют *сигнатурой функции.* Если две функции используют одно и то же количество аргументов и одинаковые их типы, причем они следуют в одном и том же порядке, то функции имеют одну и ту же сигнатуру. Имена переменных не имеют значения. Язык C++ предоставляет возможность определить две функции с одним именем при условии, что эти функции обладают разными сигнатурами. Сигнатуры могут различаться по количеству аргументов или по их типам, либо по тому и другому. Например, можно определить набор функций `print()` со следующими прототипами:

```
void print(const char * str, int width);  // #1
void print(double d, int width);          // #2
void print(long l, int width);            // #3
void print(int i, int width);             // #4
void print(const char *str);              // #5
```

При последующем вызове функции `print()` компилятор сопоставит используемый вариант с прототипом, имеющим ту же сигнатуру:

```
print("Pancakes", 15);                    // используется #1
print("Syrup");                           // используется #5
print(1999.0, 10);                        // используется #2
print(1999, 12);                          // используется #4
print(1999L, 15);                         // используется #3
```

Например, функция `print("Pancakes", 15)` использует строку и целое число в качестве аргументов, а это соответствует прототипу #1.

При вызове перегруженных функций важно правильно указывать типы аргументов. Для примера рассмотрим следующие операторы:

```
unsigned int year = 3210;
print(year, 6);                           // неопределенный вызов
```

Какому прототипу соответствует в рассматриваемом случае вызов функции `print()`? Ни одному из них! Отсутствие подходящего прототипа не исключает автоматически использование одной из функций, поскольку среда C++ предпримет попытку выполнить стандартное приведение типов, чтобы достичь соответствия. Если бы, скажем, единственным прототипом функции `print()` был прототип #2, вызов функции `print(year, 6)` повлек бы за собой преобразование значения `year` к типу `double`. Однако в программном коде, приведенном выше, имеется три прототипа, которые используют число в качестве первого аргумента, при этом возникают три варианта преобразования аргумента `year`. В такой неоднозначной ситуации среда C++ отбрасывает подобный вызов функции как ошибочный.

Некоторые сигнатуры, различаясь между собой, не могут сосуществовать. Для примера рассмотрим два следующих прототипа:

```
double cube(double x);
double cube(double & x);
```

Можно предположить, что это именно тот случай, когда применима перегрузка функций, поскольку сигнатуры обеих функций вроде бы различны. Однако подойдем к этой ситуации с точки зрения компилятора. Предположим, имеется следующий программный код:

```
cout << cube(x);
```

Аргумент x соответствует как прототипу double x, так и прототипу double &x. Следовательно, компилятор не может определить, какую функцию использовать. В связи с этим, чтобы избежать такой путаницы, при проверке сигнатуры функции компилятор считает, что ссылка на тип и сам тип имеют одну и ту же сигнатуру.

В ходе сопоставления функций учитываются различия между переменными со спецификатором const и без него. Рассмотрим следующие прототипы:

```
void dribble(char * bits);           // перегружена
void dribble (const char *cbits);    // перегружена
void dabble(char * bits);            // не перегружена
void drivel(const char * bits);      // не перегружена
```

Ниже приведены вызовы различных функций с указанием соответствующих прототипов:

```
const char p1[20] = "Как погода?";
char p2[20] = "Как идут дела?";
dribble(p1);        // dribble(const char *);
dribble(p2);        // dribble(char *);
dabble(p1);         // нет соответствия
dabble(p2);         // dabble(char *);
drivel(p1);         // drivel(const char *);
drivel(p2);         // drivel(const char *);
```

У функции dribble() имеется два прототипа: один — для указателей со спецификатором const и один — для обычных указателей. Компилятор выбирает тот или другой прототип в зависимости от того, имеет фактический аргумент спецификатор const или нет. Функция dabble() допускает только вызовы с аргументом без спецификатора const, а функция drivel() разрешает вызовы с обоими видами аргументов. Причина такого различия в поведении функций drivel() и dabble() заключается в том, что значение без спецификатора const можно присвоить const-переменной, но не наоборот.

Имейте в виду, что именно сигнатура, а не тип функции делает возможным ее перегрузку. Например, два следующих объявления несовместимы:

```
long gronk(int n, float m);       // одинаковые сигнатуры,
double gronk(int n, float m);     // что недопустимо
```

Язык C++ не позволяет перегрузить функцию gronk() подобным образом. Можно создать несколько функций с одним именем и различными типами возвращаемых значений, но только при условии, что их сигнатуры различны:

```
long gronk(int n, float m);       // различные сигнатуры, поэтому
double gronk(float n, float m);   // объявления допустимы
```

Мы вернемся к теме определения соответствия функций в этой главе после обзора шаблонов.

Пример перегрузки

Ранее мы создали функцию left(), которая возвращает указатель на первые n символов в строке. Теперь разработаем еще одну функцию left(), но на этот раз она будет возвращать первые n цифр записи целого числа. Она применима, напри-

мер, для анализа первых трех цифр почтового ZIP-кода США, представляющего собой целое число, с целью сортировки по районам.

Функцию, выполняющую обработку целых чисел, несколько труднее запрограммировать, нежели строковую версию, поскольку цифры числа не хранятся в отдельных элементах массива. Один из способов решения предполагает сначала подсчет количества цифр в числе. Деление числа на 10 уменьшает его запись на одну цифру. Таким образом, можно воспользоваться делением, чтобы подсчитать количество цифр в числе. Точнее, можно выполнить эту процедуру в цикле, подобном приведенному ниже:

```
unsigned digits = 1;
while (n /= 10)
   digits++;
```

В данном цикле подсчитывается, сколько раз можно удалить цифру из числа `n`, пока не останется ни одной цифры. Вспомните, что запись `n /= 10` является сокращением от `n = n / 10`. Если, например, `n` равно `8`, то при вычислении проверяемого выражения переменной `n` присваивается значение `8 / 10`, или `0`, поскольку деление целочисленное. Выполнение цикла при этом прекращается, и значение переменной `digits` остается равным 1. Но если значение `n` равно `238`, то на первом этапе проверки условия цикла переменной `n` присваивается величина `238 / 10`, или `23`. Это значение не равно нулю, поэтому в ходе выполнения цикла значение переменной `digits` увеличивается до `2`. На следующем этапе цикла значение `n` устанавливается равным `23 / 10`, или `2`. Эта величина также не равна нулю, поэтому значение `digits` возрастает до 3. На следующем этапе цикла значение `n` устанавливается равным `2 / 10`, или `0`, и выполнение цикла прекращается, а значение переменной `digits` остается равным 3, то есть правильному значению.

Теперь предположим, что известно, что исходное число содержит пять цифр, и нужно возвратить первые три цифры. Чтобы получить данное трехзначное число, можно дважды разделить исходное число на 10. При каждом делении числа на 10 в записи числа удаляется одна цифра справа. Чтобы узнать, какое количество цифр нужно удалить, следует просто вычислить количество цифр, которые требуется отобразить, из общего количества цифр в представлении исходного числа. Например, чтобы отобразить четыре цифры числа, представленного девятью цифрами, удалите последние пять цифр. Это решение можно представить в виде следующего программного кода:

```
ct = digits - ct;
while (ct--)
   num /= 10;
return num;
```

Программа в листинге 8.8, включает этот код в новую функцию `left()`. Данная функция содержит и другие операторы, предназначенные для обработки специальных случаев, таких как запрос вывода нулевого количества цифр или количества, превышающего длину представления исходного числа. Поскольку сигнатура новой функции `left()` отличается от сигнатуры старой функции `left()`, мы получаем возможность использовать обе функции в одной и той же программе.

Листинг 8.10. leftover.cpp

```
// leftover.cpp -- перегрузка функции left()
#include <iostream>
unsigned long left(unsigned long num, unsigned ct);
char * left(const char * str, int n = 1);
int main()
{
   using namespace std;
   char * trip = "Гаваи!!!";    // тестовое значение
   unsigned long n = 12345678;  // тестовое значение
   int i;
   char * temp;
   for (i = 1; i < 10; i++)
   {
      cout << left(n, i) << endl;
      temp = left(trip,i);
      cout << temp << endl;
      delete [] temp;  // указатель на временную область хранения
   }
   return 0;
}
// Эта функция возвращает ct первых цифр числа num.
unsigned long left(unsigned long num, unsigned ct)
{
   unsigned digits = 1;
   unsigned long n = num;
   if (ct == 0 || num == 0)
      return 0;          // при отсутствии цифр возвращается 0
   while (n /= 10)
      digits++;
   if (digits > ct)
   {
      ct = digits - ct;
      while (ct--)
         num /= 10;
      return num;        // возврат ct знаков слева
   }
   else // if ct >= number of digits
      return num;        // возврат числа целиком
}
// Эта функция возвращает указатель на новую строку, состоящую
// из n первых символов строки str.
char * left(const char * str, int n)
{
   if(n < 0)
   n = 0;
   char * p = new char[n+1];
   int i;
   for (i = 0; i < n && str[i]; i++)
      p[i] = str[i];   // копирование символов
   while (i <= n)
      p[i++] = '\0';   // установка значений '\0' для остальной части строки
   return p;
}
```

Ниже показаны выходные данные программы из листинга 8.10:

```
1
Г
12
Га
123
Гав
1234
Гава
12345
Гаваи
123456
Гаваи!
1234567
Гаваи!!
12345678
Гаваи!!!
12345678
Гаваи!!!
```

Когда целесообразно использовать перегрузку функций

Возможность перегрузки функций может произвести на вас впечатление, но не следует ей злоупотреблять. Перегрузку целесообразно использовать для функций, которые выполняют в основном одни и те же действия, но с различными типами данных. Кроме того, имеет смысл оценить возможность достижения той же цели посредством аргументов, принимаемых по умолчанию. Например, можно заменить единственную функцию `left()`, предназначенную для обработки строк, двумя перегруженными функциями:

```
char * left(const char * str, unsigned n);      // два аргумента
char * left(const char * str);                  // один аргумент
```

Тем не менее, использование единственной функции с аргументами, определенными по умолчанию, проще. Прежде всего, достаточно создавать только одну функцию, а не две, к тому же программе потребуется меньше памяти. Если потребуется внести изменения, достаточно будет отредактировать только одну функцию. Тем не менее, если необходимо применять аргументы разного типа, то принимаемые по умолчанию аргументы здесь не помогут. Придется использовать перегрузку функций.

Пример из практики: что такое декорирование имен?

Как C++ различает перегруженные функции? Каждой из них присваивается скрытый идентификатор. Компилятор C++ скрыто от пользователя осуществляет *декорирование имен*. При этом имя каждой функции кодируется с учетом типов формальных параметров, указанных в прототипе. Рассмотрим следующий прототип в недекорированном виде:

```
long MyFunctionFoo(int, float);
```

Этот формат удобен для человека. Мы видим, что функция принимает два аргумента типов `int` и `float`, а возвращает значение типа `long`. Для собственных нужд компилятор использует внутреннее представление прототипа, которое имеет примерно следующий вид:

```
?MyFunctionFoo@@YAXH@Z
```

В этом невразумительном обозначении закодировано количество аргументов и их типы. Получаемый набор символов зависит от сигнатуры функции, а также от применяемого компилятора.

Шаблоны функций

Современные компиляторы языка C++ реализуют одно из его новейших средств – *шаблоны функций*. Они представляют собой обобщенное описание функций. Другими словами, шаблон определяет функцию на основе обобщенного типа, вместо которого может быть подставлен определенный тип данных, такой как int или double. Передавая шаблону тип в качестве параметра, можно заставить компилятор сгенерировать функцию для этого типа данных. Поскольку шаблоны позволяют программировать на основе обобщенного типа вместо определенного типа данных, этот процесс иногда называют *обобщенным программированием*. Поскольку типы представлены параметрами, шаблоны функций иногда называют *параметризированными типами*. Теперь ознакомимся с полезными свойствами и функционированием этого инструментального средства.

Ранее мы создали функцию, которая осуществляет обмен значениями двух переменных типа int. Предположим, что вместо этого необходимо произвести обмен значениями двух переменных типа double. Один из подходов к решению этой задачи состоит в дублировании исходного программного кода с последующей заменой каждого слова int словом double. Чтобы произвести обмен значениями двух переменных типа char, придется повторить эту процедуру. Тем не менее, чтобы выполнить такие незначительные изменения, вам придется затратить время, которого всегда не хватает, при этом еще и не исключена вероятность ошибки. Если вносить эти изменения вручную, можно пропустить переменную типа int. Если же воспользоваться функцией глобального поиска и замены, например, слова int словом double, можно строки

```
int x;
short interval;
```

преобразить следующим образом:

```
double x;              // намеренное изменение типа
short doubleerval;     // ненамеренное изменение имени переменной
```

Средство использования шаблонов функций в языке C++ автоматизирует этот процесс, обеспечивая высокую надежность и экономию времени.

Шаблоны функций предоставляют возможность давать определения функций на основе некоторого произвольного типа данных. Например, можно создать шаблон для осуществления обмена значениями, подобный приведенному ниже:

```
template <class Any>
void Swap(Any &a, Any &b)
{
    Any temp;
    temp = a;
    a = b;
    b = temp;
}
```

Первая строка указывает, что устанавливается шаблон, и произвольному типу данных присваивается имя Any. В описание обязательно входят ключевые слова template и class (вместо class можно использовать ключевое слово typename), а также угловые скобки. Имя типа выбирается любое (в нашем примере – Any), при

этом необходимо соблюдать обычные правила именования языка C++. Многие программисты применяют простые имена, например, `T`. Остальная часть программного кода описывает алгоритм обмена значениями типа `Any`. Шаблон не создает никаких функций. Вместо этого он предоставляет компилятору указания по определению функции. Если необходимо, чтобы функция осуществляла обмен значениями типа `int`, то компилятор создаст функцию, соответствующую образцу шаблона, подставляя `int` вместо `Any`. Подобно этому, для создания функции, которая производит обмен значениями типа `double`, компилятор будет руководствоваться требованиями шаблона, подставляя тип `double` вместо `Any`.

Недавно в C++ появилось новое ключевое слово `typename`, которое может быть использовано вместо ключевого слова `class` в рассматриваемом контексте. Это значит, что определение шаблона можно записать в такой форме:

```
template <typename Any>
void Swap(Any &a, Any &b)
{
    Any temp;
    temp = a;
    a = b;
    b = temp;
}
```

Ключевое слово `typename` подчеркивает, что параметр `Any` представляет тип. В то же время уже созданы большие библиотеки программных кодов, которые были разработаны с применением прежнего ключевого слова `class`. Стандарт C++ рассматривает в данном контексте оба эти ключевых слова как идентичные.

Совет

Используйте шаблоны в тех случаях, когда необходимы функции, применяющие один и тот же алгоритм к различным типам данных. Если перед вами не стоит проблема обратной совместимости и не затрудняет печать более длинного слова, используйте при объявлении типов параметров ключевое слово `typename` вместо `class`.

Чтобы сообщить компилятору о том, что необходима определенная форма функции обмена значениями, достаточно в программе вызвать функцию `Swap()`. Компилятор проанализирует типы передаваемых аргументов, а затем создаст соответствующую функцию. В листинге 8.11 показано, как это делается. Формат программы выбран по образцу для обычных функций — прототип шаблона функции располагается в верхней части файла, а описание шаблона функции следует после обозначения `main()`.

Замечание по совместимости

Более ранние версии C++ могут не поддерживать шаблоны. Новые версии рассматривают ключевое слово `typename` как альтернативу ключевому слову `class`. Старые версии C++ требуют, чтобы прототип и определение шаблона располагались в тексте программы до первого обращения к шаблону, но в более новых версиях это положение исправлено.

Листинг 8.11. `funtemp.cpp`

```
// funtemp.cpp -- использование шаблона функции
#include <iostream>
// прототип шаблона функции
template <class Any> // либо typename Any
```

```
void Swap(Any &a, Any &b);
int main()
{
  using namespace std;
  int i = 10;
  int j = 20;
  cout << "i, j = " << i << ", " << j << ".\n";
  cout << "Использование сгенерированной компилятором функции int swapper:\n";
  Swap(i,j); // генерирует функцию void Swap(int &, int &)
  cout << "Теперь i, j = " << i << ", " << j << ".\n";
  double x = 24.5;
  double y = 81.7;
  cout << "x, y = " << x << ", " << y << ".\n";
  cout << "Использование сгенерированной компилятором функции double swapper:\n";
  Swap(x,y); // генерирует функцию void Swap(double &, double &)
  cout << "Теперь x, y = " << x << ", " << y << ".\n";
  return 0;
}
// определение шаблона функции
template <class Any> // либо typename Any
void Swap(Any &a, Any &b)
{
  Any temp; // temp - переменная типа Any
  temp = a;
  a = b;
  b = temp;
}
```

В листинге 8.11 первая функция `Swap()` содержит два аргумента типа `int`, поэтому компилятор генерирует версию функции, предназначенную для обработки данных типа `int`. Другими словами, он замещает каждое вхождение слова `Any` словом `int`, создавая определение следующего вида:

```
void Swap(int &a, int &b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Этот код не отображается, но компилятор генерирует его, а затем использует в программе. Вторая функция `Swap()` содержит два аргумента типа `double`, поэтому компилятор генерирует версию функции, предназначенную для обработки данных типа `double`. Таким образом, он замещает все вхождения слова `Any` словом `double`, генерируя следующий код:

```
void Swap(double &a, double &b)
{
  double temp;
  temp = a;
  a = b;
  b = temp;
}
```

Выходные данные программы из листинга 8.11 показывают, что она работоспособна:

```
i, j = 10, 20.
Использование сгенерированной компилятором функции int swapper:
Теперь i, j = 20, 10.
x, y = 24.5, 81.7.
Использование сгенерированной компилятором функции double swapper:
Теперь x, y = 81.7, 24.5.
```

Обратите внимание, что применение шаблонов функций не сокращает размер исполняемого кода. Выполнение программы из листинга 8.11 приводит к созданию двух отдельных определений функции точно так же, как если бы программист реализовал это вручную. Получаемый в итоге код не содержит шаблонов. Он содержит реальные функции, сгенерированные для программы. Преимущество шаблонов состоит в упрощении процесса создания нескольких определений функции, а также в повышении его надежности.

Перегруженные шаблоны

Шаблоны используются, когда необходимо создать функции, которые применяют один и тот же алгоритм к различным типам данных, как показано в листинге 8.11. Однако один и тот же алгоритм может быть применимым не для всех типов данных. Для таких случаев можно воспользоваться перегрузкой шаблонов, точно так же, как и перегрузкой обычных функций. Как и в случае с функциями, перегруженные шаблоны должны иметь различные сигнатуры. Например, листинг 8.12 содержит новый шаблон, обеспечивающий обмен элементами между двумя массивами. Исходный шаблон обладает сигнатурой `(Any &, Any &)`, в то время как новый шаблон имеет сигнатуру `(Any [], Any [], int)`. Обратите внимание, что последним аргументом является определенный тип (`int`), а не обобщенный. Не все аргументы шаблонов обязательно должны иметь обобщенный тип.

Когда в файле `twotemps.cpp` компилятор встречает первый вызов функции `Swap()`, он обнаруживает, что в ней имеется два аргумента типа `int`, и сопоставляет ее с исходным шаблоном. Однако во втором случае использования этой функции в качестве аргументов выступают два массива элементов типа `int` и значение типа `int`, что соответствует новому шаблону.

Листинг 8.12. `twotemps.cpp`

```
// twotemps.cpp -- использование перегруженных шаблонов функций
#include <iostream>
template <class Any>    // исходный шаблон
void Swap(Any &a, Any &b);
template <class Any>    // новый шаблон
void Swap(Any *a, Any *b, int n);
void Show(int a[]);
const int Lim = 8;
int main()
{
  using namespace std;
  int i = 10, j = 20;
  cout << "i, j = " << i << ", " << j << ".\n";
```

```
  cout << "Использование сгенерированной компилятором функции int  swapper:\n";
  Swap(i,j); // соответствует исходному шаблону
  cout << "Теперь i, j = " << i << ", " << j << ".\n";
  int d1[Lim] = {0,7,0,4,1,7,7,6};
  int d2[Lim] = {0,6,2,0,1,9,6,9};
  cout << "Исходные массивы:\n";
  Show(d1);
  Show(d2);
  Swap(d1,d2,Lim); // соответствует новому шаблону
  cout << "Массивы после обмена значениями:\n";
  Show(d1);
  Show(d2);
  return 0;
}
template <class Any>
void Swap(Any &a, Any &b)
{
  Any temp;
  temp = a;
  a = b;
  b = temp;
}
template <class Any>
void Swap(Any a[], Any b[], int n)
{
  Any temp;
  for (int i = 0; i < n; i++)
   {
     temp = a[i];
     a[i] = b[i];
     b[i] = temp;
   }
}
void Show(int a[])
{
  using namespace std;
  cout << a[0] << a[1] << "/";
  cout << a[2] << a[3] << "/";
  for (int i = 4; i < Lim; i++)
     cout << a[i];
  cout << endl;
}
```

Замечание по совместимости

Более ранние версии C++ могут не поддерживать шаблоны. Новые версии рассматривают ключевое слово `typename` как альтернативу ключевому слову `class`. Ранние версии C++ более строги в отношении соответствия типов. Чтобы выражение `const int Lim` соответствовало требованиям шаблона, предъявляемым к обычному типу `int`, они требуют наличия в программе следующего кода:

```
Swap(d1,d2, int (Lim)); // приведение Lim к типу int, не имеющему
                        // спецификатора const
```

Более старые версии C++ требуют, чтобы все определения шаблона были размещены в программе перед описанием функции `main()`.

Ниже показаны результаты выполнения программы из листинга 8.12:

```
i, j = 10, 20.
Использование сгенерированной компилятором функции int swapper:
Теперь i, j = 20, 10.
Исходные массивы:
07/04/1776
07/20/1969
Массивы после обмена значениями:
07/20/1969
07/04/1776
```

Явные специализации

Рассмотрим следующий пример определения структуры:

```
struct job
{
    char name[40];
    double salary;
    int floor;
};
```

Предположим, что необходимо обеспечить возможность обмена содержимым этих структур. Исходный шаблон использует следующий код для осуществления такого обмена:

```
temp = a;
a = b;
b = temp;
```

Поскольку язык C++ позволяет присваивать одну структуру другой, этот код работает безупречно даже в том случае, когда тип `Any` является структурой `job`. Однако предположим, что мы хотим совершить только обмен данными между элементами `salary` и `floor`, а элементы `name` оставить без изменений. Для этого требуется другой программный код, но аргументы функции `Swap()` будут такими же, как и в первом случае (ссылки на две структуры `job`), так что нельзя воспользоваться перегрузкой шаблона, чтобы предоставить альтернативный код.

Вместо этого можно создать специализированное определение функции, называемое *явной специализацией*, которое содержит необходимый код. Если компилятор обнаруживает специализированное определение, которое точно соответствует вызову функции, он использует его без поиска шаблонов.

Механизм специализации претерпевал изменения по мере эволюции языка. Мы рассмотрим форму, действующую на текущий момент и регламентируемую стандартом C++, а затем перейдем к двум предыдущим формам, которые поддерживаются прежними компиляторами.

Специализация третьего поколения (стандарт C++ ISO/ANSI)

После экспериментов с подходами, которые будут описаны ниже, в новом стандарте C++ приняты следующие положения:

- Одно и то же имя может применяться для обычной функции, шаблона функции и явной специализации шаблона, а также всех перегруженных версий всего перечисленного.
- Прототип и определение явной специализации должно начинаться с обозначения `template <>`, а также указывать имя обобщенного типа данных.
- Специализация перекрывает обычный шаблон, а обычная функция перекрывает и специализацию и шаблон.

Ниже приводятся примеры прототипов всех трех форм функций, осуществляющих обмен данными для структур типа `job`.

```
// прототип обычной функции
void Swap(job &, job &);

// прототип шаблона
template <class Any>
void Swap(Any &, Any &);

// явная специализация для типа job
template <> void Swap<job>(job &, job &);
```

Как уже говорилось, если существует более одного из перечисленных прототипов, для компилятора прототип обычной функции обладает приоритетом перед явной специализацией и шаблоном. Явная специализация имеет приоритет перед шаблоном. В следующем примере кода первый вызов функции `Swap()` приводит к использованию обобщенного шаблона, а второй вызов обращен к явной специализации, основанной на типе `job`:

```
...
template <class Any>          // шаблон
void Swap(Any &, Any &);
// явная специализация для типа job
template <> void Swap<job>(job &, job &);
int main()
{
   double u, v;
   ...
   Swap(u,v);  // использовать шаблон
   int a, b;
   ...
   Swap(a,b);  // использовать void Swap<job>(job &, job &)
}
```

Обозначение `<job>` в выражении `Swap<job>` необязательно, поскольку типы аргументов функции указывают, что это специализация для структуры `job`. Поэтому прототип может иметь и такой вид:

```
template <> void Swap (job &, job &);    // упрощенная форма
```

Ниже будут рассмотрены правила специализации для версий компиляторов, предшествующих стандарту ISO/ANSI, но сейчас ознакомимся с практическим применением явной специализации.

Пример явной специализации

Программа, представленная в листинге 8.13, показывает, как реализуется явная специализация. Она разработана с соблюдением требований стандарта C++.

Листинг 8.13. twoswap.cpp

```
// twoswap.cpp -- специализация перекрывает шаблон
#include <iostream>
template <class Any>
void Swap(Any &a, Any &b);
struct job
{
 char name[40];
 double salary;
 int floor;
};
// явная специализация
template <> void Swap<job>(job &j1, job &j2);
void Show(job &j);
int main()
{
 using namespace std;
 cout.precision(2);
 cout.setf(ios::fixed, ios::floatfield);
 int i = 10, j = 20;
 cout << "i, j = " << i << ", " << j << ".\n";
 cout << "Использование сгенерированной компилятором функции int  swapper:\n";
 Swap(i,j); // генерирует функцию void Swap(int &, int &)
 cout << "Теперь i, j = " << i << ", " << j << ".\n";
 job sue = {"Сьюзен Яффи", 73000.60, 7};
 job sidney = {"Сидни Таффи", 78060.72, 9};
 cout << "До обмена значениями между структурами job:\n";
 Show(sue);
 Show(sidney);
 Swap(sue, sidney); // использует функцию void Swap(job &, job &)
 cout << "После обмена значениями между структурами job:\n";
 Show(sue);
 Show(sidney);
 return 0;
}
template <class Any>
void Swap(Any &a, Any &b) // обобщенная версия
{
 Any temp;
 temp = a;
 a = b;
 b = temp;
}
// обмен данных только между полями salary и floor структуры job
template <> void Swap<job>(job &j1, job &j2) // специализация
{
 double t1;
 int t2;
```

```
  t1 = j1.salary;
  j1.salary = j2.salary;
  j2.salary = t1;
  t2 = j1.floor;
  j1.floor = j2.floor;
  j2.floor = t2;
}
void Show(job &j)
{
  using namespace std;
  cout << j.name << ": $" << j.salary
       << " на этаже " << j.floor << endl;
}
```

Замечание по совместимости

Версия программы в листинге 8.13 требует поддержки спецификации ISO/ANSI C++.

Ниже представлены выходные данные программы из листинга 8.13:

```
i, j = 10, 20.
Использование сгенерированной компилятором функции int  swapper:
Теперь i, j = 20, 10.
До обмена значениями между структурами job:
Сьюзен Яффи: $73000.60 on floor 7
Сидни Таффи: $78060.72 on floor 9
После обмена значениями между структурами job:
Сьюзен Яффи: $78060.72 на этаже 9
Сидни Таффи: $73000.60 на этаже 7
```

Прежние реализации специализации

Если программа из листинга 8.13 не воспринимается вашим компилятором, возможно, придется обратиться к более ранней реализации специализации. Проще всего создать обычную функцию для определенного типа данных, подлежащих обработке. В связи с этим в листинге 8.13 строку

```
template <> void Swap<job>(job &j1, job &j2);
```

потребуется заменить строкой

```
void Swap(int & n, int & m);  // обычный прототип
```

а фрагмент

```
template <> void Swap<job>(job &j1, job &j2);  // специализация
{
...
}
```

заменить фрагментом

```
void Swap(job &j1, job &j2);  // обычная функция
{
...  // код остается без изменений
}
```

Когда компилятор обнаруживает вызов функции Swap(sue, sidney), он становится перед выбором создания определения функции посредством шаблона или с помощью обычной функции Swap(int &, int &). Первая реализация шаблонов (как и современный стандарт) предусматривала, что компилятор должен отдать предпочтение обычному описанию функции (не шаблону).

Если данное решение не дает результата в вашей системе, возможно, причина в том, что ваш компилятор использует неофициальную реализацию шаблонов, где шаблоны имеют приоритет перед обычными функциями. Руководствуясь этим правилом, компилятор будет использовать шаблон Swap() вместо версии, где указывается структура job. Поэтому, чтобы добиться желаемого результата, придется воспользоваться явной специализацией, которая не вполне соответствует современной форме. Таким образом, вместо конструкции

```
template <> void Swap<job>(job &j1, job &j2);    // соответствует
                                                 // стандарту C++ ISO/ANSI
```

нужно применить следующую форму:

```
void Swap<job>(job &, job &);   // прежняя форма специализации
```

Обратите внимание на отсутствующую запись template <>. Подобное изменение следует внести и в заголовок функции. Другими словами, код

```
template <> void Swap<job>(job &j1, job &j2);  // специализация
{
...
}
```

заменяется кодом

```
void Swap<job>(job &j1, job &j2);  // прежняя форма специализации
{
...  // код остается без изменений
}
```

Надеемся, что вы пользуетесь новейшей версией компилятора, и подобных изменений вносить не придется.

Создание экземпляров и специализация

Чтобы расширить круг представлений о шаблонах, необходимо ознакомиться с понятиями *создания экземпляров* и *специализации*. Имейте в виду, что включение шаблона функции в программный код само по себе не приводит к генерации определения функции. Это просто план построения определения функции. Когда компилятор использует шаблон при создании определения функции для определенного типа данных, результат называется *созданием экземпляра* шаблона. Например, в программе, представленной в листинге 8.13, вызов функции Swap(i,j) заставляет компилятор генерировать экземпляр Swap(), используя тип данных int. Определением функции является не шаблон, а определенный экземпляр шаблона, использующий тип int. Такой способ создания экземпляров шаблонов получил название *неявного создания экземпляров*, поскольку компилятор приходит к выводу о необходимости построения определения, обнаружив, что в программе используется функция Swap() с параметрами типа int.

Первоначально неявное создание экземпляров было единственным способом, посредством которого компилятор генерировал определения функций на основе шабло-

нов, однако сейчас C++ позволяет выполнять *явное создание экземпляров.* Это означает возможность дать компилятору прямую команду создать определенный экземпляр, например, `Swap<int>()`. Синтаксис требует указать используемый тип данных посредством обозначения <>, а также начать объявление с ключевого слова `template`:

```
template void Swap<int>(int, int);    // явное создание экземпляра
```

Компилятор, в котором реализовано это свойство, обнаружив такое объявление, использует шаблон функции `Swap()`, чтобы сгенерировать экземпляр функции, применяющей тип `int`. Другими словами, такое объявление означает "использовать шаблон функции `Swap()`, чтобы сгенерировать определение функции для типа `int`".

Сопоставим явное создание экземпляров с явной специализацией, которая использует одно из следующих эквивалентных объявлений:

```
template <> Swap<int>(int &, int &);  // явная специализация
template <> Swap(int &, int &);       // явная специализация
```

Различие состоит в том, что эти объявления означают следующее: "не применять шаблон функции `Swap()`, чтобы сгенерировать определение функции; вместо этого воспользоваться отдельным, специализированным определением функции, явно сформулированным для типа `int`". Эти прототипы должны быть ассоциированы с их собственными определениями функций. В объявлении явной специализации после ключевого слова `template` следует обозначение <>. В объявлении явного создания экземпляра это обозначение опускается.

Внимание!

Попытка одновременного использования в одном программном модуле как явного создания экземпляра, так и явной специализации для одного и того же типа (типов) данных приводит к ошибке.

Неявное и явное создание экземпляров, а также явная специализация получили общее название — *специализация.* Общим для них является то, что они представляют определение функции, в основу которого положены определенные типы данных, а не определение функции, являющееся обобщенным описанием.

Введение явного создания экземпляров привело к появлению нового синтаксиса префиксов `template` и `template <>` в объявлениях, что дает возможность провести различие между явным созданием экземпляров и явной специализацией. Чаще всего бывает так, что более широкие возможности обусловливаются увеличением количества синтаксических правил. Следующий фрагмент кода служит сводкой рассмотренных понятий:

```
...
template <class Any>
void Swap (Any &, Any &);  // прототип шаблона
template <> void Swap<int>(job &, job &);// явная специализация для job
int main(void)
{
  template void Swap<char>(char &, char &); // явное создание
                                            // экземпляра для типа char
  short a, b;
  ...
  Swap(a,b);  // неявное создание экземпляра шаблона для short
  job n, m;
  ...
```

```
    Swap(n, m); // использование явной специализации для job
    char g, h;
    ...
    Swap(g, h); // использование явного создания экземпляра шаблона
                // для типа char
    ...
}
```

Когда компилятор обнаруживает явное создание экземпляра для типа char, он использует определение шаблона, чтобы сгенерировать версию функции Swap(), предназначенную для типа данных char. В остальных случаях вызова функции Swap() компилятор соотносит шаблон с используемыми для вызова аргументами. Например, когда компилятор обнаруживает вызов функции Swap(a,b), он генерирует версию этой функции для типа данных short, поскольку оба аргумента принадлежат именно этому типу. Когда компилятор обнаруживает вызов функции Swap(n,m), он использует отдельное определение (явная специализация), созданное для типа данных job. Если же компилятор обнаруживает вызов функции Swap(g,h), он применяет специализацию шаблона, сгенерированную ранее, когда компилятор выполнял явное создание экземпляра.

Какую версию функции выбирает компилятор?

В отношении перегрузки функций, шаблонов функций и перегрузки шаблонов функций, язык C++ располагает четко выраженной методикой выбора определения функции, применимого для данного вызова функции, особенно при наличии нескольких аргументов. Этот процесс выбора называется *разрешением перегрузки*. Детальная проработка стратегии во всей ее полноте требует, по меньшей мере, отдельной небольшой главы, поэтому здесь мы рассмотрим реализацию этого процесса в общих чертах:

- **Фаза 1.** Составьте список функций-кандидатур. Таковыми являются функции и шаблоны функций с таким же именем, как у вызываемой функции.
- **Фаза 2.** Беря за основу список функций-кандидатур, составьте список подходящих функций. Таковыми являются функции с правильно установленным количеством аргументов, для которых существует неявная последовательность преобразований типов. В нее входит случай точного соответствия каждого типа фактического аргумента типу формального аргумента. Например, при вызове функции с аргументом типа float эта величина может быть приведена к типу double, что позволит достичь соответствия типу double формального параметра, а шаблон может сгенерировать экземпляр функции для типа float.
- **Фаза 3.** Проверьте наличие наиболее подходящей функции. Если она есть, воспользуйтесь ею. В противном случае подобный вызов функции является ошибочным.

Рассмотрим пример вызова функции с единственным аргументом:

```
may('B');   // фактический аргумент имеет тип char
```

Прежде всего, компилятор отмечает все "подозрительные" объекты, каковыми являются функции и шаблоны функций с именем may(). Затем он находит среди них те, которые могут быть вызваны с одним аргументом. Например, в этом случае проверку пройдут следующие объекты с одинаковыми именами:

```
void may(int);                              // #1
float may(float, float = 3);                // #2
void may(char);                             // #3
char * may(const char *);                   // #4
char may(const char &);                     // #5
template<class T> void may(const T &);      // #6
template<class T> void may(T *);            // #7
```

Обратите внимание, что при этом учитываются только сигнатуры, а не типы возвращаемых значений. Однако две кандидатуры (#4 и #7) не подходят, поскольку целочисленный тип данных не может быть преобразован неявно (то есть без явного приведения типов) в тип указателя. Оставшийся шаблон будет применен для создания специализации, где вместо обозначения T принимается тип `char`. Остается пять функций, каждая из которых может использоваться так, как если бы она была единственной объявленной функцией.

Далее компилятор должен определить, какая из функций-кандидатур в наибольшей степени соответствует критерию отбора. Он анализирует преобразования, необходимые для того, чтобы аргумент обращения к функции соответствовал аргументу наиболее подходящей кандидатуры. В общем случае порядок следования от наилучшего к наихудшему варианту можно представить следующим образом:

1. Точное совпадение, обычные функции имеют приоритет перед шаблонами.
2. Приведение с повышением (разрядности) типа (например, автоматическое приведение типов `char` и `short` к `int` и типа `float` к `double`).
3. Приведение типов посредством стандартных преобразований (например, приведение `int` к `char` или `long` к `double`).
4. Приведения типов, заданные пользователем, подобные тем, что определяются при объявлении классов.

Например, функция #1 предпочтительнее функции #2, поскольку преобразование `char` к `int` является повышением типа (см. главу 3), в то время как `char` к `float` – это стандартное преобразование. Функции #3, #5 и #6 предпочтительнее функций #1 и #2, так как они являются точными соответствиями. Функции #3 и #5 предпочтительнее варианта #6, поскольку он представляет собой шаблон. Этот анализ порождает два вопроса. Что такое точное соответствие, и что произойдет, если таких соответствий будет два? Обычно, как и в нашем примере, два точных соответствия вызывают ошибку, но из этого правила есть два исключения. Что ж, придется продолжить изучение этого вопроса!

Точные соответствия и наилучшие соответствия

Для достижения точного соответствия язык C++ допускает некоторые "тривиальные преобразования данных". В табл. 8.1 приводится список таких преобразований, при этом слово *Тип* обозначает некоторый произвольный тип данных. Например, фактический аргумент типа `int` представляет собой точное соответствие формальному параметру `int &`. Обратите внимание, что *Тип* может быть чем-то подобным `char &`, так что эти правила включают приведение типа `char &` к `const char &`. Запись *Тип* (*список-аргументов*) означает, что имя функции как фактический аргумент соответствует указателю функции, выступающему в качестве формального параметра, при условии, что оба имеют один и тот же тип возвращаемых значений

и список аргументов. (Указатели функций обсуждались в главе 7. Там же рассматривалась возможность передачи имени функции в качестве аргумента функции, которая должна принимать указатель на функцию.) Далее в этой главе мы рассмотрим ключевое слово `volatile`.

Таблица 8.1. Тривиальные преобразования, допустимые для случая точного соответствия

Из фактического аргумента	В формальный аргумент
Тип	*Тип* &
Тип &	*Тип*
Тип []	* *Тип*
Тип (*список-аргументов*)	*Тип* (*)(*список-аргументов*)
Тип	const *Тип*
Тип	volatile *Тип*
Тип *	const *Тип* *
Тип *	volatile *Тип* *

Предположим, имеется следующий код функции:

```
struct blot { int a; char b[10];} ;
blot ink = { 25, "spots"} ;
...
recycle(ink);
```

В данном случае все следующие прототипы будут точными соответствиями:

```
void recycle(blot);           // #1 blot в blot
void recycle(const blot);     // #2 blot в (const blot)
void recycle(blot &);         // #3 blot в (blot &)
void recycle(const blot &);   // #4 blot в (const blot  &)
```

Можно предположить, что в результате наличия нескольких подходящих прототипов компилятор не в состоянии завершить процесс разрешения перегрузки. Поскольку не существует наиболее подходящей функции, компилятор сгенерирует сообщение об ошибке; скорее всего, в нем будут присутствовать такие слова, как “ambiguous” (“неоднозначный”).

Тем не менее, разрешение перегрузки иногда возможно даже в случае, когда две функции дают точное соответствие. Прежде всего, указатели и ссылки на данные, не имеющие спецификатора `const`, сопоставляются преимущественно с указателями и ссылочными параметрами, также не имеющими спецификатора `const`. Таким образом, если бы в примере вызова функции `recycle()` существовали только прототипы #3 и #4, то был бы выбран прототип #3, поскольку переменная `ink` не была объявлена со спецификатором `const`. Однако такое неодинаковое отношение к типам со спецификатором `const` и без него применимо только к данным, на которые имеются ссылки и указатели. Другими словами, если бы существовали только прототипы #1 и #2, то компилятор вывел бы сообщение об ошибке, вызванной неопределенностью.

Другой пример приоритета одного точного соответствия над другим касается ситуации, когда существует обычная функция и шаблон. В этом случае обычная функция имеет приоритет перед шаблоном, включая явную специализацию.

В случае, когда имеется два точных соответствия, и оба представляют собой шаблоны, то шаблон с более высокой степенью специализации (при наличии таковой) имеет приоритет. Это означает, например, что явная специализация получает преимущество перед функцией, неявно сгенерированной по шаблону:

```
struct blot { int a; char b[10];};
template <class Type> void recycle (Type t);     // шаблон
template <> void recycle<blot> (blot & t);       // специализация для blot
...
blot ink = { 25, "spots"};
...
recycle(ink); // используется специализация
```

Термин "наиболее специализированная" не всегда означает явную специализацию. В принципе, он указывает, что, когда компилятор выбирает используемый тип данных, имеет место меньшее количество преобразований. В качестве примера рассмотрим два следующих шаблона:

```
template <class Type> void recycle (Type t);     // #1
template <class Type> void recycle (Type * t);   // #2
```

Предположим, что программа, которая содержит эти шаблоны, также включает следующий код:

```
struct blot {int a; char b[10];};
blot ink = {25, "spots"};
...
recycle(&ink);  // адрес структуры
```

Вызов `recycle(&ink)` соответствует шаблону #1, в котором `Type` интерпретируется как `blot *`. Тот же вызов соответствует и шаблону #2, но на этот раз `Type` интерпретируется как `ink`. Это сочетание передает два неявных экземпляра, `recycle<blot *>(blot *)` и `recycle<blot>(blot *)`, в область хранения подходящих функций.

Из этих двух вариантов шаблон `recycle<blot *>(blot *)` специализирован в большей степени, поскольку он предполагает меньшее количество преобразований в процессе генерирования функции. Другими словами, шаблон #2 уже явно заявил, что аргументом функции является указатель на `Type`, так что `Type` может прямо идентифицироваться именем `blot`. При этом шаблон #1 использует обозначение `Type` в качестве аргумента функции, так что `Type` должен быть интерпретирован как указатель на `blot`. Другими словами, в шаблоне #2 `Type` уже был специализирован как указатель, отсюда происходит выражение "более специализированный".

Правила поиска наиболее специализированного шаблона называются *правилами частичного упорядочивания* шаблонов функций. Как и явное создание экземпляров, они представляют собой новые средства языка C++.

Пример использования правил частичного упорядочивания

Рассмотрим полную программу, которая использует правила частичного упорядочивания для идентификации наиболее подходящего определения шаблона. Листинг 8.14 содержит два определения шаблонов функций, которые отображают содержимое массива. Первое определение (шаблон A) предполагает, что передаваемый в качестве аргумента массив содержит данные, которые следует отобразить.

Листинг 8.14. tempover.cpp

```
// tempover.cpp --- перегрузка шаблонов
#include <iostream>
template <typename T>   // шаблон A
void ShowArray(T arr[], int n);
template <typename T>   // шаблон B
void ShowArray(T * arr[], int n);
struct debts
{
  char name[50];
  double amount;
};
int main(void)
{
  using namespace std;
  int things[6] = {13, 31, 103, 301, 310, 130};
  struct debts mr_E[3] =
  {
    {"Ima Wolfe", 2400.0},
    {"Ura Foxe ", 1300.0},
    {"Iby Stout", 1800.0}
  };
  double * pd[3];
// установка указателей на элементы amount структур массива mr_E
   for (int i = 0; i < 3; i++)
      pd[i] = &mr_E[i].amount;
   cout << "Список значений количества предметов гражданина E.:\n";
// things - это массив значений типа int
   ShowArray(things, 6); // использует шаблон A
   cout << "Список долгов гражданина E.:\n";
// pd - это массив указателей на данные типа double
   ShowArray(pd, 3);
   // использует шаблон B (более специализированный)
   return 0;
}
template <typename T>
void ShowArray(T arr[], int n)
{
  using namespace std;
  cout << "шаблон A\n";
  for (int i = 0; i < n; i++)
    cout << arr[i] << ' ';
  cout << endl;
}
template <typename T>
void ShowArray(T * arr[], int n)
{
  using namespace std;
  cout << "шаблон B\n";
  for (int i = 0; i < n; i++)
    cout << *arr[i] << ' ';
  cout << endl;
}
```

Рассмотрим следующий вызов функции:

```
ShowArray(things, 6);
```

Идентификатор things представляет собой имя массива элементов типа int, поэтому вызов функции соответствует шаблону:

```
template <typename T>            // шаблон A
void ShowArray(T arr[], int n);
```

здесь вместо обозначения T подставляется тип int.

Затем рассмотрим еще один вызов функции:

```
ShowArray(pd, 3);
```

Здесь pd — это имя массива элементов типа double *. Этот вызов соответствует шаблону A:

```
template <typename T>            // шаблон A
void ShowArray(T arr[], int n);
```

Здесь вместо обозначения T подставляется тип double. В данном случае шаблонная функция отображает содержимое массива pd, а именно три адреса. Кроме того, вызов функции можно сопоставить с шаблоном B:

```
template <typename T>            // шаблон B
void ShowArray(T * arr[], int n);
```

В данном случае обозначение T подразумевает тип double, а функция отображает разыменованные элементы *arr[i] — значения типа double, на которые указывают элементы массива. Из двух шаблонов более специализированным является шаблон B, поскольку он построен исходя из предположения, что массив содержит указатели. Поэтому именно этот шаблон используется.

Вывод программы листинга 8.14 имеет следующий вид:

```
Список значений количества предметов гражданина E.:
шаблон A
13 31 103 301 310 130
Список долгов гражданина E.:
шаблон B
2400 1300 1800
```

Если удалить из программы шаблон B, компилятор будет использовать шаблон A и выводить содержимое массива pd. В результате будут отображены адреса, а не значения.

Подведем итоги. Процесс разрешения перегрузки осуществляет поиск функции, которая наилучшим образом соответствует вызову. Если существует лишь одна такая функция, она выбирается. Если вариантов несколько, но лишь одна функция не является шаблонной, выбирается обычная функция. Когда кандидатур несколько, и все являются шаблонами, выбирается наиболее специализированный шаблон. Если существует несколько в одинаковой степени соответствующих обычных функций либо шаблонов, причем ни один и шаблонов не является в наибольшей степени специализированным, вызов функции считается неоднозначным и приводит к ошибке. При отсутствии функций, соответствующих вызову, также возникает ошибка.

Функции с множеством аргументов

Когда вызов функции с множеством аргументов сопоставляется с прототипами, содержащими несколько аргументов, ситуация значительно усложняется. Компилятору приходится проверять соответствия всех аргументов. Если он обнаруживает функцию, которая наиболее полно соответствует вызову, то выбирает именно ее. Одна функция имеет приоритет перед другой, если хотя бы один ее аргумент имеет приоритет перед аргументом другой функции, а все остальные аргументы имеют, по меньшей мере, одинаковый приоритет.

Эта книга не призвана исследовать сложные случаи процесса поиска наилучшего соответствия. Рассмотренные правила охватывают все возможные сочетания прототипов и шаблонов функций.

Резюме

В языке C++ возможности функций расширены по сравнению с языком С. Ключевое слово `inline` в определении функции и вставка этого определения до первого вызова функции указывает компилятору C++ обращаться с данной функцией как со встроенной. Иначе говоря, вместо того, чтобы передавать управление отдельному фрагменту программного кода для выполнения функции, компилятор встраивает вместо каждого вызова функции соответствующий код. Этот механизм встраивания должен быть использован только в тех случаях, когда код функции достаточно краткий.

Ссылочная переменная представляет собой некую скрытую форму указателя, который позволяет создавать псевдоним (второе имя) переменной. Ссылочные переменные главным образом используются в качестве аргументов функций, которые обрабатывают структуры и объекты классов. Обычно идентификатор, объявленный в качестве ссылки на определенный тип, может указывать только на данные этого типа. Однако когда один класс является производным от другого (например, класс `ofstream` унаследован от класса `ostream`), ссылка на базовый тип может также указывать на производный тип данных.

Прототипы C++ позволяют устанавливать используемые по умолчанию значения аргументов. Если в обращении к функции пропущен соответствующий аргумент, программа использует его значение, заданное по умолчанию. Если в обращении к функции значение аргумента указано, программа использует его вместо значения, заданного по умолчанию. Аргументы, определенные по умолчанию, могут быть представлены в списке аргументов только в заданном порядке — справа налево. Таким образом, если вы указываете значение по умолчанию для определенного аргумента, то при этом следует указать используемые по умолчанию значения для всех остальных аргументов, расположенных справа от него.

Сигнатурой функции является ее список аргументов. Можно определить две функции с одним и тем же именем при условии, что они обладают различными сигнатурами. Это называется *полиморфизмом*, или *перегрузкой функций*. Как правило, перегрузка функции осуществляется для того, чтобы обеспечить единообразную обработку различных типов данных.

Шаблоны функций автоматизируют процесс перегрузки функций. Функция определяется с использованием обобщенного типа данных и некоторого алгоритма, а компилятор генерирует соответствующие определения функций для конкретных типов аргументов, которые используются в программе.

Вопросы для самоконтроля

1. Какие функции эффективно используются в качестве встроенных?
2. Предположим, что функция song() имеет следующий прототип:

   ```
   void song(char * name, int times);
   ```

   а. Как модифицировать этот прототип, чтобы для переменной times по умолчанию принималось значение 1?

   б. Какие изменения следует внести в определение функции?

   в. Можно ли переменной name присвоить используемое по умолчанию значение "О, мой папа"?
3. Создайте перегруженную версию функции iquote(), которая отображает аргументы, заключенные в двойные кавычки. Напишите три версии: одну для аргумента типа int, другую для аргумента типа double и третью для аргумента типа string.
4. Ниже приведена структура шаблона:

   ```
   struct box
   {
      char maker[40];
      float height;
      float width;
      float length;
      float volume;
   };
   ```

   а. Создайте функцию, которая ссылается на структуру box(), как на свой формальный аргумент, и отображает значение каждого элемента структуры.

   б. Создайте функцию, которая ссылается на структуру box(), как на свой формальный аргумент, и присваивает элементу volume результат произведения трех других измерений.
5. Ниже дано описание результатов, которых требуется достичь. Укажите, может ли каждый из них быть получен с помощью аргументов, заданных по умолчанию, путем перегрузки функций, тем и другим способом, или же можно обойтись без этих средств. Создайте необходимые прототипы.

   а. Функция mass(density, volume) возвращает массу объекта, имеющего плотность density и объем volume, а функция mass(density) возвращает массу тела, имеющего плотность density и объем 1.0 кубический метр. Все величины имеют тип double.

   б. В результате вызова функции repeat(10, "Я в порядке") указанная строка отображается десять раз, в то время как функция repeat("Но ты не в своем уме") отображает заданную строку пять раз.

   в. В результате вызова функции average(3,6) возвращается среднее значение типа int двух аргументов int, а в результате вызова average(3.0, 6.0) возвращается среднее значение типа double двух величин double.

   г. В результате вызова функции mangle("Рад вас видеть") возвращается символ Р или указатель на строку "Добро пожаловать", в зависимости от того, присваиваете ли вы возвращаемое значение переменной типа char или переменной типа char*.

6. Запишите шаблон функции, которая возвращает наибольший из двух ее аргументов.
7. Используя шаблон из вопроса 6 и структуру box из вопроса 4, создайте специализацию шаблона функции, которая принимает два аргумента типа box и возвращает тот из них, у которого больше объем.

Упражнения по программированию

1. Создайте функцию, которая обычно принимает один аргумент — адрес строки — и выводит эту строку один раз. Однако если задан второй аргумент типа int, не равный нулю, то эта функция выводит строку столько раз, сколько было осуществлено вызовов этой функции к моменту ее данного вызова. (Обратите внимание, что количество выводимых строк не равно значению второго аргумента, оно равно числу вызовов функции к моменту последнего вызова.) Действительно, это не слишком полезная функция, но она заставит вас применить некоторые из методов, рассмотренных в данной главе. Воспользуйтесь этой функцией в простой программе, которая способна показать, как эта функция работает.
2. Структура CandyBar содержит три элемента. Первый элемент содержит фирменное название конфеты. Второй элемент содержит вес (который может принимать дробное значение) конфеты, а третий элемент представляет число калорий (целочисленное значение) в конфете. Напишите программу, использующую функцию, которая принимает в качестве аргументов ссылку на структуру CandyBar, указатель на значение типа char, переменную типа double и переменную типа int. Программа использует три последних величины для присваивания значений соответствующим элементам структуры. Три последних аргумента по умолчанию должны иметь значения: “Millennium Munch”, 2.85 и 350. Кроме того, программа должна использовать функцию, принимающую в качестве аргумента ссылку на CandyBar, которая отображает содержимое этой структуры. Используйте спецификатор const там, где он необходим.
3. Создайте функцию, которая принимает ссылку на объект string в качестве параметра и преобразует содержимое строки string в символы верхнего регистра. Используйте функцию toupper(), описанную в табл. 6.4. Напишите программу, использующую цикл, который позволяет проверять работу функции, вводя различные данные. Ниже приводится пример выполнения программы:

```
Введите строку (для завершения введите q): go away
GO AWAY
Следующая строка (для завершения введите q): good grief!
GOOG GRIEF!
Следующая строка (для завершения введите q): q
Всего наилучшего.
```

4. Ниже представлена общая структура программы:

```
#include <iostream>
using namespace std;
#include <cstring>          // для вызова функций strlen(), strcpy()
struct stringy {
    char * str;             // указывает на строку
    int ct;                 // длина строки (не считая символа '\0')
};
```

```
// здесь размещаются прототипы функций set() и show()
int main()
{
    stringy beany;
    char testing[] = "Реальность - не то, что нам видится.";
    set(beany, testing);    // первым аргументом является ссылка,
// выделяет пространство для хранения копии
// testing, использует элемент типа str
// структуры beany как указатель на новый
// блок, копирует testing в новый блок и
// создает элемент ct структуры beany
    show(beany);            // печатает элемент строкового типа один раз
    show(beany, 2);         // печатает элемент строкового типа дважды
    testing[0] = 'D';
    testing[1] = 'u';
    show(testing);          // печатает сроку testing один раз
    show(testing, 3);       // печатает строку testing три раза
    show("Готово!");
    return 0;
}
```

Завершите написание программы, создав соответствующие функции и прототипы. Обратите внимание, что в программе должны быть две функции `show()`, и каждая из них использует аргументы, заданные по умолчанию. Используйте спецификатор `const` при объявлении аргументов там, где это оправдано. Обратите также внимание на то, что функция `set()` должна использовать операцию `new` для выделения достаточного пространства памяти под хранение заданной строки. Используемые здесь методы аналогичны методам, применяемым при задании и реализации классов. (Возможно, вам придется изменить имена файлов заголовков и удалить директиву `using`, что зависит от используемого компилятора.)

5. Создайте шаблон функции `max5()`, которая принимает в качестве аргумента массив из пяти элементов типа `T` и возвращает наибольший элемент этого массива. (Поскольку размер массива фиксирован, эту операцию можно выполнять в жестко заданном цикле, а не использовать соответствующий аргумент.) Протестируйте его в программе, которая использует массив, состоящий из 5 значений типа `int`, и массив, содержащий 5 значений типа `double`.

6. Создайте шаблон функции `maxn()`, которая принимает в качестве аргумента массив элементов типа `T` и целое число, представляющее собой количество элементов в массиве, а возвращает элемент с наибольшим значением. Протестируйте ее работу в программе, которая использует шаблон данной функции с массивом из шести значений типа `int` и массивом из четырех значений типа `double`. Программа также должна включать специализацию, которая использует в качестве аргумента массив указателей на `char` и количество указателей в качестве второго аргумента, а затем возвращает адрес самой длинной строки. Если имеется более одной строки наибольшей длины, функция возвращает адрес первой из них. Выполнить проверку специализации на примере массива, состоящего из пяти указателей на строки.

7. Измените программу из листинга 8.1 таким образом, чтобы шаблонные функции возвращали сумму содержимого массива, а не отображали это содержимое. Программа должна выводить общее количество предметов и сумму всех задолженностей.

ГЛАВА 9

Модели памяти и пространства имен

В этой главе:

- Раздельная компиляция программ
- Продолжительность существования области хранения, область видимости и компоновка
- Применение операции new
- Пространства имен

Язык C++ предлагает несколько способов хранения данных в памяти. Существует возможность выбора длительности хранения данных в памяти (продолжительность существования области хранения) и определения частей программы, имеющих доступ к данным (область видимости и связывание). Операция new позволяет динамически выделять память, а позиция этой операции определяет вариации этого процесса. Возможности пространства имен C++ обеспечивают дополнительный контроль над доступом к данным. Большие программы обычно состоят из нескольких файлов исходного кода, которые могут совместно использовать некоторые данные. Поскольку в таких программах применяется раздельная компиляция файлов, эта глава начинается с освещения данной темы.

Раздельная компиляция

Язык C++, как и С, допускает и даже поощряет размещение функций программы в отдельных файлах. Как говорилось в главе 1, файлы можно компилировать раздельно, а затем связывать их с конечным продуктом — исполняемой программой. (Как правило, компилятор C++ не только компилирует программы, но и управляет работой компоновщика.) При изменении только одного файла, можно перекомпилировать только один этот файл и затем связать его с ранее скомпилированными версиями других файлов. Этот механизм облегчает работу с крупными программами. Более того, большинство сред программирования на C++ предоставляют дополнительные средства, упрощающие процесс создания программ. Например, в системах Unix и Linux существует программа make, хранящая сведения обо всех тех файлах, от которых зависит программа, и о времени их последней модификации. После запуска make обнаруживает изменения в исходных файлах с момента последней компиляции, а затем предлагает выполнить соответствующие действия, необходимые для

воссоздания программы. Интегрированные среды разработки (integrated development environment — IDE) Borland C++, Microsoft Visual C++ и Metrowerks CodeWarrior предоставляют аналогичные средства, доступ к которым осуществляется с помощью меню Project (Проект).

Рассмотрим простой пример. Вместо того чтобы разбирать детали компиляции, которые зависят от реализации, давайте сосредоточимся на более общих аспектах, таких как проектирование.

Предположим, что нужно разбить программу из листинга 7.12 на части и поместить используемые ею функции в отдельный файл. Напомним, что эта программа преобразует прямоугольные координаты в полярные, после чего отображает результат. Нельзя просто "обрезать" исходный файл по пунктирной линии после окончания функции `main()`. Дело в том, что и функция `main()` и другие две функции используют одни и те же объявления структур, поэтому необходимо поместить эти объявления в оба файла. Простой ввод объявлений в код может привести к ошибке. Даже если объявления будут скопированы безошибочно, в случае последующих модификаций нужно будет не забыть внести изменения в оба файла. Одним словом, разделение программы на несколько файлов создает новые проблемы.

Кому нужны дополнительные трудности? Только не разработчикам C и C++. Поэтому они создали директиву `#include` для решения подобных проблем. Вместо того чтобы помещать объявления структур в каждый файл, их можно разместить в заголовочном файле, а затем включить этот заголовочный файл в каждый файл с исходным кодом. Таким образом, чтобы изменить объявление структуры, нужно будет сделать это только один раз в заголовочном файле. Кроме того, в заголовочный файл можно помещать прототипы функций. Таким образом, исходную программу можно разбить на три части:

- Заголовочный файл, содержащий объявления структур и прототипы функций, использующих эти структуры.
- Файл с исходным кодом, содержащий код функций, работающих со структурами.
- Файл с исходным кодом, содержащий код, осуществляющий вызовы этих функций.

Такая стратегия может быть успешно использована для организации программ. Если, например, создается другая программа, которая пользуется теми же самыми функциями, достаточно включить в нее заголовочный файл и добавить файл с функциями в проект или список `make`. К тому же такая организация программы соответствует принципам объектно-ориентированного программирования (ООП). Первый файл, а именно — заголовочный файл, содержит определения пользовательских типов. Второй файл содержит код функций для управления типами, определенными пользователем. В совокупности оба файла образуют пакет, который можно использовать в различных программах.

В заголовочный файл не стоит помещать определения функций или объявления переменных. В простых программах это может работать хорошо, но очень часто служит источником проблем. Например, если в заголовочном файле содержатся определения функций, и этот заголовочный файл включается в два других файла, которые являются частью одной программы, в одной программе окажется два определения одной и той же функции, что будет ошибкой, если функция не встроенная. В заголовочных файлах обычно содержится следующее:

- Прототипы функций.
- Символические константы, определенные с помощью директивы #define или const.
- Объявления структур.
- Объявления классов.
- Объявления шаблонов.
- Встроенные функции.

Помещать объявления структур в заголовочные файлы очень удобно, потому что они не создают переменных, а только указывают компилятору, как создать структурную переменную, когда она объявляется в файле исходного кода. Подобно этому объявления шаблонов — это не код, который нужно компилировать, а инструкции для компилятора, указывающие, как генерировать определения функций, чтобы они соответствовали вызовам функций, встречающимся в исходном коде. Данные, объявленные со спецификатором const, и встроенные функции обладают специальными свойствами связывания (которые скоро будут рассмотрены), которые позволяют размещать их в заголовочных файлах, не вызывая при этом каких-либо проблем.

В листингах 9.1, 9.2, и 9.3 показан результат разбиения программы, представленной в листинге 7.12, на отдельные части. Обратите внимание, что при включении в программу заголовочного файла мы используем запись "coordin.h" вместо <coordin.h>. Если имя файла заключено в угловые скобки, компилятор С++ ищет его в той части базовой файловой системы, где расположены стандартные заголовочные файлы. Но когда имя файла заключено в двойные кавычки, компилятор сначала ищет файл в текущем рабочем каталоге или в каталоге с исходным кодом (либо делает другой аналогичный выбор, в зависимости от компилятора). Если он не находит заголовочный файл там, то ищет его в стандартно используемом для этой цели каталоге. Таким образом, при включении собственных заголовочных файлов нужно использовать кавычки, а не угловые скобки.

На рис. 9.1 показаны этапы компоновки этой программы в системе Unix. Обратите внимание, что пользователь только дает команду выполнить компиляцию — CC, а остальные действия выполняются автоматически. Компиляторы g++, gpp и Borland C++ (bcc32.exe), работающие в режиме командной строки, ведут себя таким же образом. Средства разработки Borland C++, Turbo C++, Metrowerks CodeWarrior, Watcom C++ и Microsoft Visual C++, по сути, выполняют те же самые действия. Однако, как показано в главе 1, процесс инициируется по-другому, при помощи команд меню, которые позволяют создавать проект и связывать с ним файлы исходного кода. Обратите внимание, что в проекты включаются только файлы исходного кода, но не заголовочные файлы. Дело в том, что заголовочными файлами управляет директива #include. Кроме того, не следует использовать директиву #include, чтобы включать файлы исходного кода, поскольку это может привести к повторяющимся объявлениям.

Внимание!

В интегрированных средах разработки программ не включайте заголовочные файлы в список проекта и не используйте директиву #include для включения одних файлов исходного кода в другие файлы исходного кода.

Листинг 9.1. coordin.h

```
// coordin.h -- шаблоны структур и прототипы функций
// шаблоны структур
#ifndef COORDIN_H_
#define COORDIN_H_
struct polar
{
  double distance;  // расстояние от начала координат
  double angle;     // направление от начала координат
};
struct rect
{
  double x;         // расстояние от начала координат по горизонтали
  double y;         // расстояние от начала координат по вертикали
};
// прототипы
polar rect_to_polar(rect xypos);
void show_polar(polar dapos);
#endif
```

Управление заголовочными файлами

Заголовочный файл следует включать в файл только один раз. Казалось бы, что это легко запомнить, но можно включить заголовочный файл несколько раз, даже не подозревая об этом. Например, можно использовать заголовочный файл, который включает другой заголовочный файл. Стандартная методика C/C++ для избегания многократных включений заголовочных файлов, основана на использовании директивы препроцессора `#ifndef` (*if not defined* — если не определено). В таком фрагменте кода

```
#ifndef COORDIN_H_
...
#endif
```

операторы, находящиеся между директивами `#ifndef` и `#endif`, обрабатываются только в том случае, если имя `COORDIN_H_` не было определено ранее директивой препроцессора `#define`.

Обычно директива `#define` используется для создания символьных констант, как в следующем примере:

```
#define MAXIMUM 4096
```

Однако чтобы имя было определено, достаточно просто использовать директиву `#define` с этим именем, как показано ниже:

```
#define COORDIN_H_
```

Методика, которая используется в листинге 9.1, состоит в том, чтобы поместить содержимое файла внутрь оператора `#ifndef`:

```
#ifndef COORDIN_H_
#define COORDIN_H_
// сюда помещается содержимое подключаемого файла
#endif
```

Когда компилятор впервые встречает файл, имя `COORDIN_H_` не должно быть определено. (Мы используем имя подключаемого файла с несколькими символами подчеркивания, чтобы избежать совпадения с существующими именами.) В этом случае компилятор будет обрабатывать код между директивами `#ifndef` и `#endif`, что нам и нужно. При этом компилятор считывает строку, определяющую имя `COORDIN_H_`. Если после этого компилятор обнаруживает второе включение

coordin.h в том же файле, он учитывает, что имя COORDIN_H_ уже определено, и переходит к строке, идущей после обозначения #endif. Обратите внимание, что этот метод не предотвращает повторное включение файла. Вместо этого компилятор будет игнорировать содержимое всех включений, кроме первого. Данная методика защиты используется в большинстве стандартных заголовочных файлов C и C++.

Листинг 9.2. file1.cpp

```
// file1.cpp -- пример программы, состоящей из трех файлов
#include <iostream>
#include "coordin.h" // шаблоны структур, прототипы функций
using namespace std;
int main()
{
  rect rplace;
  polar pplace;
  cout << "Введите значения x и y: ";
  while (cin >> rplace.x >> rplace.y) //остроумное использование cin
  {
    pplace = rect_to_polar(rplace);
    show_polar(pplace);
    cout << "Следующие два числа (для выхода введите q): ";
  }
  cout << "Всего наилучшего!\n";
  return 0;
}
```

Листинг 9.3. file2.cpp

```
// file2.cpp -- содержит функции, вызываемые в файле file1.cpp
#include <iostream>
#include <cmath>
#include "coordin.h" // шаблоны структур, прототипы функций
// преобразование прямоугольных координат в полярные
polar rect_to_polar(rect xypos)
{
  using namespace std;
  polar answer;
  answer.distance =
     sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
  answer.angle = atan2(xypos.y, xypos.x);
  return answer; // возвращает структуру polar
}
// вывод полярных координат, преобразование радиан в градусы
void show_polar (polar dapos)
{
  using namespace std;
  const double Rad_to_deg = 57.29577951;
  cout << "радиус = " << dapos.distance;
  cout << ", угол = " << dapos.angle * Rad_to_deg;
  cout << " градусов\n";
}
```

В результате компиляции и компоновки этих двух файлов исходного кода и нового заголовочного файла получается исполняемая программа. Ниже приведен пример ее выполнения:

```
Введите значения x и y: 120 80
радиус = 144.222, угол = 33.6901 градусов
Следующие два числа (для выхода - q): 120 50
радиус = 130, угол = 22.6199 градусов
Следующие два числа (для выхода - q): q
```

Несмотря на то что мы рассмотрели раздельную компиляцию в понятиях файлов, стандарт C++ вместо термина "файл" использует термин *единица трансляции*, чтобы сохранить более высокую степень обобщенности. Для организации информации в компьютере могут применяться не только файлы.

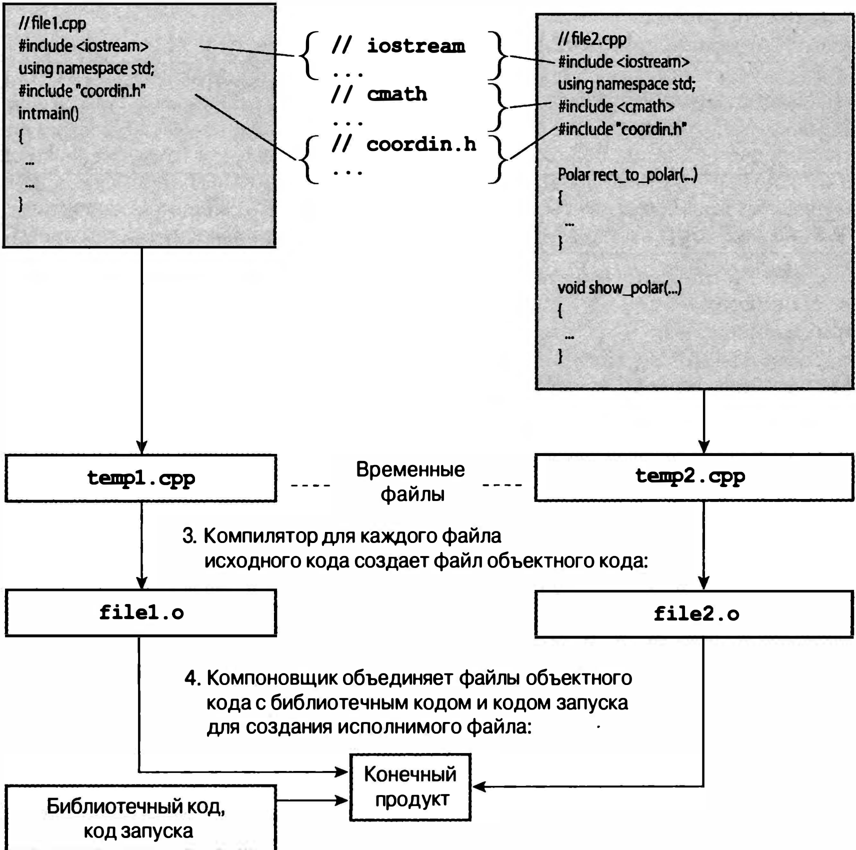


Рис. 9.1. Компиляция многофайловых программ C++ в системе Unix

Пример из практики: компоновка нескольких библиотек

Стандарт С ++ предоставляет каждому разработчику компилятора свободу реализации корректировки имен (см. врезку "Пример из практики: что такое декорирование имен?" в главе 8). Поэтому следует учитывать, что компоновка двоичных модулей (файлов объектного кода), созданных различными компиляторами, скорее всего, не будет успешной. Другими словами, для одной и той же функции два компилятора сгенерируют различные декорированные имена. Это различие имен не позволит компоновщику сопоставить вызов функции, сгенерированной одним компилятором, с определением функции, сгенерированным другим компилятором. Перед компоновкой скомпилированных модулей нужно обеспечить, чтобы каждый объектный файл или библиотека была сгенерирована одним и тем же компилятором. При наличии исходного кода ошибки компоновки обычно можно исправить путем его повторной компиляции.

Продолжительность существования области хранения, область видимости и компоновка

Теперь, после обзора многофайловых программ, пришло время продолжить рассмотрение моделей памяти, начатое в главе 4. Дело в том, что категории областей хранения определяют совместный доступ к информации со стороны файлов. Напомним, что в главе 4 говорилось о памяти. В языке С++ используются три различных модели хранения данных. Они отличаются между собой продолжительностью хранения данных в памяти:

- **Автоматическая продолжительность хранения**. Такую продолжительность хранения имеют переменные, объявленные внутри определения функции, включая параметры функции. Они создаются, когда выполняется функция или блок, где переменные определены. После выхода из блока или функции используемая переменными память освобождается. В С++ существуют два вида автоматических переменных.
- **Статическая продолжительность хранения**. Такую продолжительность хранения имеют переменные, определенные вне определения функции, либо объявленные с ключевым словом `static`. Они существуют в течение всего времени выполнения программы. В языке С++ существуют три вида статических переменных.
- **Динамическая продолжительность хранения**. Память, выделяемая операцией `new`, сохраняется до тех пор, пока она не будет освобождена с помощью операции `delete` или до завершения программы, смотря какое из событий наступит раньше. Иногда эту память называют свободной областью хранения.

Теперь продолжим изучение понятий области видимости переменных (их доступности для программы) и компоновки, которая определяет общий доступ к информации со стороны файлов.

Область видимости и связывание

Область видимости определяет доступность имени в пределах файла (единицы трансляции). Например, переменная, определенная в функции, может быть использована только в этой функции, но не в какой-либо другой, в то время как переменная, определенная в файле до определений функций, может применяться во всех функциях. Связывание определяет совместное использование имени в различных единицах трансляции. Имя с внешним связыванием может быть использовано разными файлами, а имя с внутренним связыванием — функциями в пределах одного файла. Имена автоматических переменных не имеют никакого связывания, поскольку они не допускают совместного использования в программе.

В языке C++ переменная может иметь одну из нескольких возможных областей видимости. Переменная, имеющая локальную область видимости (ее еще называют областью видимости в пределах блока), известна только в пределах того блока, в котором она определена. Напомним, что блок — это группа операторов, заключенных в фигурные скобки. Например, тело функции является блоком, однако это тело могут быть вложены и другие блоки. Переменная, имеющая глобальную область видимости (часто называемая областью видимости файла), доступна во всем файле, начиная с позиции, где была определена. Автоматические переменные имеют локальную область видимости, а статические переменные могут иметь различную область видимости в зависимости от того, как они были определены. Имена, используемые в области видимости прототипа функции, доступны только в пределах круглых скобок, которые содержат список аргументов. (Вот почему не имеет большого значения, что они собой представляют и присутствуют ли вообще.) Элементы, объявленные в классе, имеют область видимости в пределах класса (см. главу 10). Переменные, объявленные в пространстве имен, имеют область видимости в пределах пространства имен. (Теперь, после ввода в язык C++ понятия пространства имен, глобальная область видимости стала частным случаем области видимости пространства имен.)

Функции C++ могут иметь область видимости класса или область видимости пространства имен, включая глобальную область видимости, но не могут иметь локальную область видимости. (Поскольку функция не может быть определена внутри блока, если бы функция могла иметь локальный диапазон доступа, она была бы доступна только самой себе, и, следовательно, не могла бы вызываться другой функцией. Такая функция вообще не могла бы действовать.)

В языке C++ различные способы хранения данных характеризуются продолжительностью существования, областью видимости и связыванием. Рассмотрим более подробно свойства классов памяти C++. Начнем с того, что исследуем ситуацию, имевшую место до ввода в язык понятия пространства имен, и посмотрим, как они повлияли на общую картину.

Автоматическая продолжительность хранения

Параметры функции и переменные, объявленные внутри функции, по умолчанию имеют автоматическую продолжительность хранения. Они также обладают локальной областью видимости и не имеют связывания. Другими словами, если объявить переменную с именем `texas` в функции `main()`, а затем объявить еще одну переменную с тем же именем в функции `oil()`, будут созданы две независимых переменные, каждая из которых будет доступна только в той функции, в которой объявлена.

Любые операции с переменной texas в функции oil() не оказывают влияния на переменную texas функции main() и наоборот. Кроме того, каждой переменной выделяется память после того, как выполнение программы доходит до блока, содержащего определение переменной. После завершения работы функции все ее переменные прекращают существование. (Обратите внимание, что переменной выделяется память, когда программа передает управление блоку, в котором переменная определена, но область видимости начинается только после позиции объявления.)

Если определить переменную внутри блока, ее время существования и область видимости ограничиваются этим блоком. Предположим, что определена переменная teledeli в начале функции main(). Теперь предположим, что в функции main() создается новый блок, в котором определяется новая переменная websight. В таком случае переменная teledeli доступна как во внешнем, так и во внутреннем блоке, в то время как websight доступна только во внутреннем блоке и существует с момента своего определения только до тех пор, пока выполнение программы не дойдет до конца блока:

```
int main()
{
  int teledeli = 5;
  {   // переменной websight выделяется память
    cout << "Привет\n";
    int websight = -2;  // начинается область видимости
                        // переменной websight
    cout << websight << ' ' << teledeli << endl;
  }    // время существования переменной websight истекает
  cout << teledeli <<endl;
  ...
}
```

Но что произойдет, если во внутреннем блоке переменной присвоить имя teledeli вместо websight, в результате чего будут получены две переменные с одним и тем же именем, одна из которых будет находиться во внешнем блоке, а другая — во внутреннем? В этом случае программа интерпретирует имя teledeli как переменную, локальную по отношению к блоку, во время выполнения операторов этого блока. Принято говорить, что новое определение скрывает предыдущее. Новое определение попадает в область видимости, а предыдущее из него временно удаляется. Когда программа выходит за пределы блока, исходное определение возвращается в область видимости (рис. 9.2).

Код в листинге 9.4 иллюстрирует вхождение автоматических переменных в область видимости функции или блока, который их содержит.

Листинг 9.4. auto.cpp

```
// auto.cpp -- иллюстрация области видимости
// автоматических переменных
#include <iostream>
void oil(int x);
int main()
{
  using namespace std;
  int texas = 31;
  int year = 1999;
```

```
  cout << "В main(), texas = " << texas << ", &texas = ";
  cout << &texas << endl;
  cout << "В main(), year = " << year << ", &year = ";
  cout << &year << endl;
  oil(texas);
  cout << "В main(), texas = " << texas << ", &texas = ";
  cout << &texas << endl;
  cout << "В main(), year = " << year << ", &year = ";
  cout << &year << endl;
  return 0;
}

void oil(int x)
{
  using namespace std;
  int texas = 5;

  cout << "В oil(), texas = " << texas << ", &texas = ";
  cout << &texas << endl;
  cout << "В oil(), x = " << x << ", &x = ";
  cout << &x << endl;
  {                       // начало блока
    int texas = 113;
    cout << "В блоке, texas = " << texas;
    cout << ", &texas = " << &texas << endl;
    cout << "В блоке, x = " << x << ", &x = ";
    cout << &x << endl;
  }                       // конец блока
  cout << "После выхода из блока texas = " << texas;
  cout << ", &texas = " << &texas << endl;
}
```

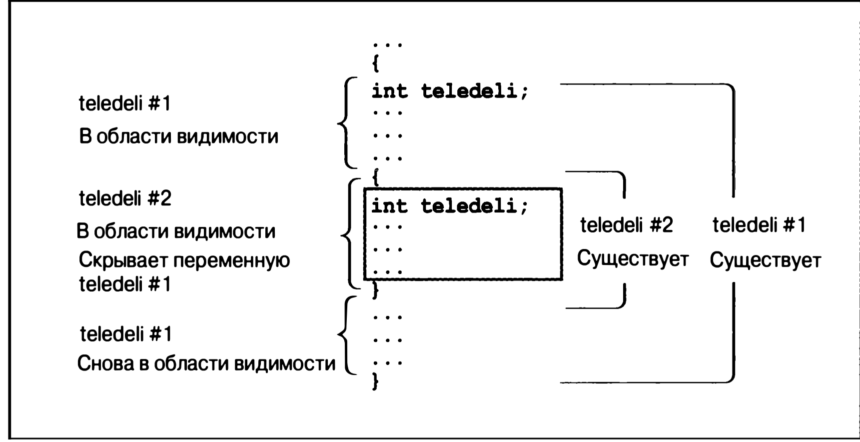


Рис. 9.2. Блоки и область видимости

Вот как выглядит вывод программы из листинга 9.4:

```
B main(), texas = 31, &texas = 0012FED4
B main(), year = 1999, &year = 0012FEC8
B oil(), texas = 5, &texas = 0012FDE4
B oil(), x = 31, &x = 0012FDF4
B блоке, texas = 113, &texas = 0012FDD8
B блоке, x = 31, &x = 0012FDF4
После выхода из блока texas = 5, &texas = 0012FDE4
B main(), texas = 31, &texas = 0012FED4
B main(), year = 1999, &year = 0012FEC8
```

Обратите внимание, что каждая из трех переменных `texas` имеет свой собственный, отличный от других адрес, и программа использует только ту переменную, которая в данное время находится в области видимости. Поэтому присвоение переменной `texas` значения `113` во внутреннем блоке функции `oil()` никак не отражается на других переменных с таким же именем. (Обычно значение и формат адресов зависят от используемой системы.)

Рассмотрим последовательность событий. В момент начала работы функции `main()` программа выделяет пространство памяти для переменных `texas` и `year`, они обе попадают в область видимости. Когда программа вызывает функцию `oil()`, эти переменные остаются в памяти, но исчезают из области видимости. Две новых переменных, `x` и `texas`, также размещаются в памяти и попадают в область видимости. Когда программа передает управление внутреннему блоку функции `oil()`, новая переменная `texas` выходит из области видимости и замещается более новым определением. Однако переменная `x` остается в области видимости, поскольку блок не определяет новую переменную с таким же именем. Когда выполнение программы выходит за пределы этого блока, освобождается память, занятая самой новой переменной `texas`. При этом переменная `texas` номер 2 возвращается в область видимости. После завершения работы функции `oil()` переменные `texas` и `x` перестают существовать, а в область видимости возвращаются исходные переменные `texas` и `year`.

Между прочим, чтобы явно указать класс переменной, можно воспользоваться ключевым словом `auto` языка C++ (и C):

```
int froob(int n)
{
    auto float ford;
    ...
}
```

Поскольку ключевое слово `auto` можно использовать только при работе с переменными, которые уже являются автоматическими по умолчанию, программисты редко прибегают к его использованию. Иногда оно применяется, чтобы программный код был более понятным пользователю. Например, можно воспользоваться этим словом, чтобы показать, что намеренно создается автоматическая переменная, которая перекрывает глобальное определение, подобное тому, что мы вскоре рассмотрим в разделе “Статические переменные, внешнее связывание”.

Инициализация автоматических переменных

Автоматическую переменную можно инициализировать, присваивая ей любое выражение, значение которого известно к моменту данного объявления. Ниже приведен пример инициализации переменных x и z:

```
int w;          // значение w не определено
int x = 5;      // инициализация с помощью выражения-константы
int y = 2 * x; // используется ранее определенное значение x
cin >> w;
int z = 3 * w; // используется новое значение w
```

Автоматические переменные и стек

Чтобы получить более полное представление об автоматических переменных, рассмотрим их реализацию обычным компилятором C++. Поскольку количество автоматических переменных растет или сокращается по мере того, как функции начинают и завершают выполнение, программа должна управлять автоматическими переменными в процессе своей работы. Стандартная методика состоит в выделении области памяти, которая будет использоваться в качестве стека, управляющего движением переменных. Термин *стек* применяется потому, что новые данные размещаются, образно говоря, поверх старых данных (то есть в смежных, а не в тех же самых ячейках памяти), а затем удаляются из стека, после того как программа завершит работу с ними. По умолчанию размер стека зависит от реализации, однако обычно компилятор предоставляет опцию изменения размера стека. Программа отслеживает состояние стека с помощью двух указателей. Один указывает на базу стека, с которой начинается выделенная область памяти, а другой — на вершину стека, которая представляет собой следующую ячейку свободной памяти. Когда происходит вызов функции, ее автоматические переменные добавляются в стек, а указатель вершины устанавливается на свободную ячейку памяти, следующую за только что размещенными переменными. После завершения функции указатель вершины снова принимает значение, которое он имел до вызова функции. В результате эффективно освобождается память, которая использовалась для хранения новых переменных.

Стек построен по принципу LIFO (last-in, first-out — последним пришел, первый обслужен). Это означает, что переменная, которая попала в стек последней, удаляется из него первой. Такой механизм упрощает передачу аргументов. Процедура вызова функции помещает значения ее аргументов в вершину стека и переустанавливает указатель вершины. Вызванная функция использует описание своих формальных параметров для определения адреса каждого аргумента. Например, на рис. 9.3 показана функция `fib()`, которая в момент вызова передает 2-байтное значение типа `int` и 4-байтное значение типа `long`. Эти значения помещаются в стек. Когда функция `fib()` начинает выполняться, она связывает имена `real` и `tell` с этими двумя значениями. После завершения работы функции `fib()` указатель вершины стека возвращается в прежнее состояние. Новые значения не удаляются, но теперь они лишаются меток, и пространство памяти, которое они занимают, будет использовано следующим процессом, в ходе которого будут размещаться значения в стеке. (На рис. 9.3 показана упрощенная картина, поскольку при вызове функции может передаваться дополнительная информация вроде адреса возврата.)

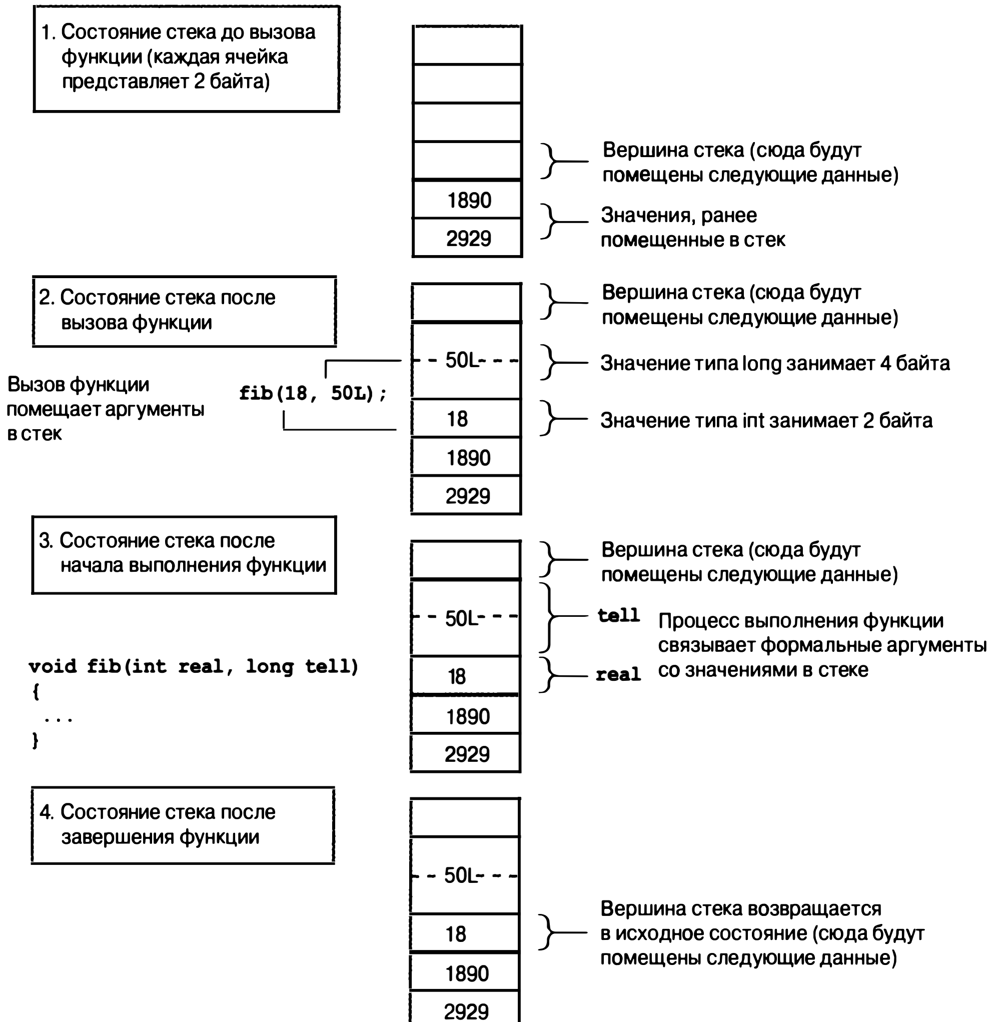


Рис. 9.3. Передача аргументов с использованием стека

Регистровые переменные

Язык C++, как и C, поддерживает ключевое слово `register`, используемое при объявлении локальных переменных. Регистровая переменная — это разновидность автоматической переменной, поэтому у нее автоматическая продолжительность хранения, локальная область видимости и отсутствие связывания. Ключевое слово `register` указывает компилятору, что к переменной желательно обеспечить быстрый доступ, возможно, за счет использования регистра центрального процессора вместо стека. Дело в том, что центрального процессор может обращаться к значениям своих регистров быстрее, чем к областям памяти в стеке. Чтобы объявить регистровую переменную, следует перед обозначением ее типа поместить ключевое слово `register`:

```
register int count_fast; // запрос создания регистровой переменной
```

Возможно, вы обратили внимание на слова "желательно" и "запрос". Компилятор не обязательно должен удовлетворять запрос. Например, регистры уже могут быть заняты либо запрашивается создание типа данных, который не подходит для регистра. Многие программисты считают, что современные компиляторы достаточно интеллектуальны, чтобы обходиться без каких-либо запросов. Например, если создается цикл for, компилятор может самостоятельно принять решение использовать регистр для отслеживания изменений индекса цикла.

Если переменная хранится в регистре, она не имеет адреса памяти, следовательно, к ней нельзя применить операцию взятия адреса. Поэтому в следующем примере извлечение адреса переменной x выполняется корректно, но недопустимо для регистровой переменной y:

```
void gromb(int *); // функция, ожидающая передачи адреса
int main()
{
  int x;
  register int y;
  gromb(&x);       // правильно
  gromb(&y);       // недопустимо
  ...
```

Использования ключевого слова register в объявлении достаточно, чтобы вызвать это ограничение, даже если компилятор на самом деле не использует регистр для хранения переменной.

Одним словом, обычная локальная переменная, локальная переменная, объявленная с использованием ключевого слова auto, и локальная переменная, объявленная с использованием ключевого слова register, характеризуются автоматической продолжительностью хранения, локальной областью видимости и отсутствием связывания. Следующий код иллюстрирует все три случая:

```
int main()
{
  short waffles;          // автоматическая переменная по умолчанию
  auto short pancakes;    // явно указанный автоматический тип переменной
  register int muffins;   // регистровая переменная
```

Объявление локальной переменной без спецификатора — то же самое, что и объявление ее со спецификатором auto. Такая переменная обычно помещается в стек памяти. Использование спецификатора register указывает на то, что переменная будет часто использоваться, и компилятор может выбрать вместо стека памяти что-либо другое для ее хранения, например, регистр центрального процессора.

Статическая продолжительность хранения

Язык C++, как и C, предоставляет три опции связывания для статических переменных: внешнее связывание, внутреннее связывание и отсутствие связывания. Переменные, которые относятся к каждому из трех типов связывания, существуют в течение всего времени выполнения программы. Они долговечнее автоматических переменных. Поскольку количество статических переменных не меняется на протяжении выполнения программы, она не нуждается в специальных механизмах, подоб-

ных стеку, чтобы управлять ими. Компилятор просто резервирует фиксированный блок памяти для хранения всех статических переменных, и эти переменные доступны программе на протяжении всего времени ее выполнения. Более того, если статическая переменная не инициализирована явно, компилятор устанавливает для нее нулевое значение. Элементы статических массивов и структур устанавливаются равными нулю по умолчанию.

Замечание по совместимости

Классический стандарт K&R языка C не позволяет инициализировать автоматические массивы и структуры, но допускает инициализацию статических массивов и структур. Спецификации ANSI C и C++ допускают инициализацию обоих видов данных. Однако некоторые ранние трансляторы C++ используют компиляторы языка C, которые не полностью совместимы со стандартом ANSI C. Если вы пользуетесь такого рода версией, для инициализации массивов и структур вам может потребоваться воспользоваться одной из трех разновидностей статических переменных.

Рассмотрим создание всех трех видов статических переменных, а затем приступим к обзору их свойств. Чтобы создать статическую переменную с внешним связыванием, нужно объявить ее вне всех блоков. Чтобы создать статическую переменную с внутренним связыванием, следует объявить ее вне всех блоков с использованием спецификатора `static`. Для создания статической переменной без связывания объявите ее внутри какого-либо блока с указанием спецификатора `static`. Следующий фрагмент кода демонстрирует все три случая:

```
...
int global = 1000;          // статическая продолжительность хранения,
                            // внешнее связывание
static int one_file = 50    // статическая продолжительность хранения,
                            // внутреннее связывание
int main()
{
...
}
void funct1(int n)
{
    static int count = 0;   // статическая продолжительность хранения,
                            // нет связывания
  int llama = 0;
...
}
void funct2(int q)
{
...
}
```

Как уже упоминалось, все статические переменные (в нашем примере `global`, `one_file` и `count`) существуют с момента начала выполнения программы и до ее завершения. Переменная `count`, объявленная вне функции `funct1()`, характеризуется локальной областью видимости и отсутствием связывания. Это означает, что она может использоваться только в функции `funct1()`, также как и автоматическая переменная `llama`. Но, в отличие от `llama`, переменная `count` остается в памяти, даже когда функция `funct1()` не выполняется. Обе переменных имеют область видимости в пределах файла. Это означает, что они могут быть использованы, начиная от точки

объявления и до конца файла. В частности, обе переменные могут быть задействованы в функциях `main()`, `funct1()` и `funct2()`. Поскольку переменная `one_file` имеет внутреннее связывание, она может быть использована только в файле, содержащем этот код. Поскольку переменная `global` имеет внешнее связывание, она может использоваться и в других файлах, которые являются частью программы.

Все статические переменные обладают следующими двумя особенностями инициализации:

- У неинициализированной статической переменной все биты установлены в 0.
- Статическая переменная может быть инициализирована только константным выражением.

Константное выражение может содержать константы-литералы, константы типа `const` и `enum`, а также операцию `sizeof`. Следующий фрагмент кода иллюстрирует эти условия:

```
int x;                          // переменная x устанавливается равной 0
int y = 49;                     // 49 — это константное выражение
int z = 2 * sizeof(int) + 1;  // также константное выражение
int m = 2 * z;                  // недопустимо, z — не константа
int main() {...}
```

В табл. 9.1 приводится сводка особенностей классов памяти, которые существовали до ввода понятия пространства имен. Рассмотрим разновидности статических переменных более подробно.

Таблица 9.1. Пять видов переменных

Описание переменной	Продолжительность хранения	Область видимости	Связывание	Способ объявления
Автоматическая	Автоматический	Блок	Нет	В блоке (необязательно с помощью ключевого слова `auto`)
Регистровая	Автоматический	Блок	Нет	В блоке с помощью ключевого слова `register`
Статическая без связывания	Статический	Блок	Нет	В блоке с помощью ключевого слова `static`
Статическая с внешним связыванием	Статический	Файл	Внешнее	Вне всех функций
Статическая с внутренним связыванием	Статический	Файл	Внутреннее	Вне всех функций с помощью ключевого слова `static`

Статические переменные, внешнее связывание

Переменные с внешним связыванием часто называются просто внешними переменными. Они обязательно имеют статическую продолжительность хранения и область видимости в пределах файла. Внешние переменные определяются вне всех функций и поэтому являются внешними по отношению к любой функции. Например, они могут быть объявлены до описания функции `main()`. Внешнюю переменную

можно использовать в любой функции, которая следует в файле после определения переменной. Поэтому внешние переменные также называются *глобальными* в отличие от автоматических переменных, которые являются локальными. Тем не менее, если определить автоматическую переменную с тем же именем, что и внешняя переменная, то именно автоматическая переменная попадает в область видимости, когда программа выполняет функцию, содержащую определение автоматической переменной. Программа, представленная в листинге 9.5, иллюстрирует сказанное. Она также демонстрирует использование ключевого слова `extern` для повторного объявления внешней переменной, определенной ранее, а также применение операции разрешения контекста для реализации доступа к скрытой до этого внешней переменной. Поскольку данный пример представляет собой однофайловую программу, он не иллюстрирует свойство внешнего связывания. Этой цели служит следующий пример настоящей главы.

Листинг 9.5. `external.cpp`

```
// external.cpp -- внешние переменные
#include <iostream>
using namespace std;
// внешняя переменная
double warming = 0.3;
// прототипы функций
void update(double dt);
void local();
int main()         // использует глобальную переменную
{
  cout << "Глобальное потепление - " << warming << " градуса.\n";
  update(0.1);     // вызов функции, изменяющей переменную warming
  cout << "Глобальное потепление - " << warming << " градуса.\n";
  local();         // вызов функции с локальной переменной warming
  cout << "Глобальное потепление - " << warming << " градуса.\n";
  return 0;
}

void update(double dt)    // изменяет глобальную переменную
{
  extern double warming;  // необязательное повторное объявление
  warming += dt;
  cout << "Изменение глобального потепления до " << warming;
  cout << " градуса.\n";
}

void local()              // использует локальную переменную
{
  double warming = 0.8;   // новая переменная скрывает внешнюю

  cout << "Локальное потепление = " << warming << " градуса.\n";
  // Доступ к глобальной переменной с помощью
  // операции разрешения контекста
  cout << "Но глобальное потепление = " << ::warming;
  cout << " градуса.\n";
}
```

Вывод программы из листинга 9.5 выглядят следующим образом:

```
Глобальное потепление - 0.3 градуса.
Изменение глобального потепления до 0.4 градуса.
Глобальное потепление - 0.4 градуса.
Локальное потепление = 0.8 градуса.
Но глобальное потепление = 0.4 градуса.
Глобальное потепление - 0.4 градуса.
```

Замечания по программе

Результаты выполнения программы показывают, что функции `main()` и `update()` имеют доступ к внешней переменной `warming`. Обратите внимание, что изменения, которые вносит функция `update()` в переменную `warming`, проявляются при последующих обращениях к этой переменной.

Функция `update()` выполняет повторное объявление переменной `warming`, используя для этой цели ключевое слово `extern`. Это ключевое слово означает "использовать переменную с данным именем, определенную ранее внешне". Поскольку функция `update()` будет работать и без этого объявления, его следует рассматривать как необязательное. Оно документирует, что данная функция разработана с целью использования внешней переменной. Исходное объявление

```
double warming = 0.3;
```

называется *определяющим объявлением* или просто *определением*. Оно приводит к выделению памяти для переменной. Повторное объявление

```
extern double warming;
```

называется *ссылочным объявлением* или просто *объявлением*. Оно не реализует выделение памяти, поскольку ссылается на существующую переменную. Ключевое слово `extern` можно применять только в объявлении, касающемся переменной (или функции, как будет показано ниже), определенной в другом месте программы. В сущности, это объявление говорит следующее: "использовать переменную `warming`, определенную в качестве внешней в другом месте программы". В ссылочном объявлении должен указываться тот же тип данных, что и в определяющем объявлении. К тому же, в ссылочном объявлении нельзя инициализировать переменную:

```
extern double warming = 0.5;  // НЕ ДОПУСКАЕТСЯ
```

Переменную в объявлении можно инициализировать только в том случае, когда объявление размещает эту переменную в памяти, другими словами — только в определяющем объявлении. В конце концов, термин *инициализация* означает присваивание значения ячейке памяти, когда она выделена для некоторой переменной.

Функция `local()` показывает, что при объявлении локальную переменную с тем же именем, что и у глобальной переменной, локальная переменная скрывает глобальную переменную. В нашем примере функция `local()` использует локальное определение переменной `warming` при отображении ее значения.

Язык C++ расширяет возможности C за счет новой *операции разрешения контекста* (`::`). Когда эта операция предшествует имени переменной, должна использоваться ее глобальная версия. Таким образом, функция `local()` отображает значение `warming`, равное `0.8`, а значение `::warming` — равное `0.4`. Эта операция еще будет встречаться при обсуждении пространств имен и классов.

Выбор между глобальными и локальными переменными

Теперь, когда у нас есть возможность выбора между глобальными и локальными переменными, возникает вопрос, каким из них отдать предпочтение? На первый взгляд глобальные переменные кажутся более привлекательными. Поскольку все функции имеют к ним доступ, не нужно беспокоиться о том, как правильно передавать аргументы. Однако такой легкий доступ достается дорогой ценой — за счет снижения надежности программ. Опыт программирования показывает, что чем эффективнее программа изолирует данные от нежелательного доступа, тем лучше будет сохраняться их целостность. В большинстве случаев следует пользоваться локальными переменными и передавать данные функциям только в случае необходимости, а не делать данные общедоступными за счет применения глобальных переменных. Как вы сможете убедиться позже, объектно-ориентированное программирование делает очередной шаг в повышении эффективности изоляции данных.

Однако и у глобальных переменных имеется своя область применения. Предположим, что существует блок данных, который используется несколькими функциями. Это может быть массив с названиями месяцев или список атомных весов химических элементов. Класс внешней памяти наилучшим образом подходит для представления постоянных данных, поскольку в этом случае для предотвращения изменения данных можно воспользоваться ключевым словом `const`:

```
const char * const months[12] =
{
  "January", "February", "March", "April", "May",
  "June", "July", "August", "September", "October",
  "November", "December"
};
```

Первое ключевое слово `const` защищает от изменений строки, а второе слово `const` гарантирует, что каждый указатель массива будет постоянно указывать на одну и ту же исходную строку.

Статические переменные, внутреннее связывание

Применение спецификатора `static` к переменной с областью видимости файла обеспечивает для нее внутреннее связывание. Различие между внутренним и внешним связыванием проявляется в многофайловых программах. В этом контексте переменная с внутренним связыванием является локальной по отношению к файлу, который ее содержит. При этом обычная внешняя переменная обладает внешним связыванием, что означает возможность использования в различных файлах. При внешнем связывании один и только один файл может содержать внешнее определение переменной. Во всех других файлах, где будет использоваться эта переменная, в ссылочном объявлении необходимо применять ключевое слово `extern` (рис. 9.4).

Если в файле отсутствует объявление со спецификатором `extern` для переменной, он не может использовать внешнюю переменную, определенную в другом файле:

```
// файл 1
int errors = 20;  // глобальное объявление
...
------------------------------------------------
// файл 2
...               // отсутствует объявление extern int errors
void froobish()
{
  cout << errors; // неудачная попытка воспользоваться переменной errors
```

```
// программа file1.cpp
#include <iostream>
using namespase std;

// прототипы функций
#include  "mystuff.h"

// определение внешней переменной
int process_status = 0;

void promise ();
int main()
{
 ...
}

void promise ()
{
 ...
}
```

В этом файле определяется переменная **process_status**, в результате чего компилятор выделяет для нее память.

```
// программа file2.cpp
#include <iostream>
using namespase std;

// прототипы функций
#include  "mystuff.h"

// ссылка на внешнюю переменную
extern int process_status;

int manipulate(int n)
{
 ...
}

char * remark(char * str)
{
 ...
}
```

В этом файле используется ключевое слово **extern**, которое указывает программе использовать переменную **process_status**, определенную в другом файле.

Рис. 9.4. Определяющее и ссылочное объявления

Если в файле дается определение второй внешней переменной с тем же именем, это приводит к ошибке:

```
// файл 1
int errors = 20;     // внешнее объявление
...
----------------------------------------------
// файл 2
int errors;          // недопустимое объявление
void froobish()
{
   cout << errors;   //неудачная попытка воспользоваться переменной errors
...
```

Правильный подход предусматривает использование ключевого слова `extern` во втором файле:

```
// файл 1
int errors = 20;     // внешнее объявление
...
----------------------------------------------
// файл 2
extern int errors;  // ссылка на переменную errors из файла 1
void froobish()
{
  cout << errors;    //использует переменную errors, определенную в файле 1
...
```

Однако если в файле объявляется статическая внешняя переменная с тем же именем, что и обычная внешняя переменная, уже объявленная в другом файле, то в область видимости первого файла попадает статическая версия:

```
// файл 1
int errors = 20;        // внешнее объявление
...
--------------------------------------------
// файл 2
static int errors = 5;  // доступна только файлу 2
void froobish()
{
  cout << errors;       // использует переменную errors,
                        // определенную в файле 2
  ...
```

На память!

В многофайловой программе можно внешнюю переменную следует объявлять в одном и только в одном файле. Все остальные файлы, использующие эту переменную, должны содержать ее объявление с ключевым словом `extern`.

Чтобы обеспечить совместное использование данных различными частями многофайловой программы, нужно использовать внешнюю переменную. Чтобы обеспечить совместное использование данных различными функциями одного файла, применяйте статическую переменную с внутренним связыванием. (Пространства имен представляют для этого альтернативный метод. В стандарте C++ указано, что использование ключевого слова `static` для реализации внутреннего связывания в будущем будет прекращено.) Кроме того, если переменная с областью видимости файла объявляется со спецификатором `static`, не возникает проблемы конфликта ее имени с переменными, содержащимися в других файлах и имеющими область видимости файла.

Программы, представленные в листингах 9.6 и 9.7, демонстрируют функционирование переменных с внешним и внутренним связыванием. В программе из листинга 9.6 (`twofile1.cpp`) определены внешние переменные `tom` и `dick`, а также статическая внешняя переменная `harry`. Функция `main()` в этом файле отображает адреса всех трех переменных, а затем вызывает функцию `remote_access()`, которая определена во втором файле. Содержимое этого файла показано в листинге 9.7 (`twofile2.cpp`). Помимо определения функции `remote_access()` этот файл использует ключевое слово `extern`, чтобы использовать переменную `tom` из первого файла. Далее в нем объявляется статическая переменная с именем `dick`. Спецификатор `static` делает эту переменную локальной по отношению к файлу и перекрывает глобальное определение. Затем этот файл определяет внешнюю переменную с именем `harry`. При этом не возникает конфликта с переменной `harry` из первого файла, поскольку она обладает только внутренним связыванием. После этого функция `remote_access()` отображает адреса всех трех переменных, так что их можно сравнить с адресами соответствующих переменных из первого файла. Не забывайте, что для получения готовой программы потребуется скомпилировать и скомпоновать оба файла.

Листинг 9.6. twofile1.cpp

```
// twofile1.cpp -- переменные с внешним и внутренним связыванием
#include <iostream>         // компилируется совместно с файлом twofile2.cpp
int tom = 3;                // определение внешней переменной
int dick = 30;              // определение внешней переменной
static int harry = 300;     // Статическая переменная,
                            // внутреннее связывание
// прототип функции
void remote_access();
int main()
{
   using namespace std;
   cout << "main() выводит следующие адреса:\n";
   cout << &tom << " = &tom, " << &dick << " = &dick, ";
   cout << &harry << " = &harry\n";
   remote_access();
   return 0;
}
```

Листинг 9.7. twofile2.cpp

```
// twofile2.cpp -- переменные с внутренним и внешним связыванием
#include <iostream>
extern int tom;             // переменная tom определена в другом месте
static int dick = 10;       // перекрывает внешнюю переменную dick
int harry = 200;            // определение внешней переменной,
                            // без конфликта с переменной harry
                            // из файла twofile1
void remote_access()
{
   using namespace std;
   cout << "remote_access() выводит следующие адреса:\n";
   cout << &tom << " = &tom, " << &dick << " = &dick, ";
   cout << &harry << " = &harry\n";
}
```

Ниже показан результат выполнения программы, представленной в листингах 9.6 и 9.7:

```
main() выводит следующие адреса:
0x0041a020 = &tom, 0x0041a024 = &dick, 0x0041a028 = &harry
remote_access() выводит следующие адреса:
0x0041a020 = &tom, 0x0041a050 = &dick, 0x0041a054 = &harry
```

На основе значений адресов видно, что в обоих файлах используется одна и та же переменная `tom`, но различные переменные `dick` и `harry`. (Значения адресов и формат вывода зависят от системы, в которой выполняется программа. Однако в выходных данных обеих функций адреса переменной `tom` должны совпадать, но адреса переменных `dick` и `harry` должны различаться.)

Статические переменные, отсутствие связывания

До сих пор мы рассматривали переменные, имеющие область видимости в пределах файла, с внешним и внутренним связыванием. Теперь обсудим третий член семейства статических переменных — локальную переменную без связывания. Такая переменная создается применением спецификатора `static` к переменной, определенной внутри блока. Если она используется внутри блока, спецификатор `static` задает статическую область видимости переменной. Это означает, что, несмотря на ограничение области видимости переменной пределами блока, она существует даже тогда, когда блок неактивен. Таким образом, статическая локальная переменная может сохранять свое значение между вызовами функции. (Статические переменные могут быть использованы для "реинкарнации" — их можно применять для передачи секретных номеров счетов швейцарского банка вашему следующему воплощению.) Кроме того, если задать инициализацию статической локальной переменной, программа будет инициализировать ее один раз, при запуске программы. Последующие вызовы функции не будут приводить к повторной инициализации переменной, как это происходит для автоматических переменных. Сказанное иллюстрируется программой в листинге 9.8.

Листинг 9.8. `static.cpp`

```
// static.cpp -- использование статической локальной переменной
#include <iostream>
// константы
const int ArSize = 10;
// прототип функции
void strcount(const char * str);
int main()
{
   using namespace std;
   char input[ArSize];
   char next;
   cout << "Введите строку: \n";
   cin.get(input, ArSize) ;
   while (cin)
   {
      cin.get(next);
      while (next != '\n')  // строка не помещается!
         cin.get(next);
      strcount(input);
      cout << "Введите следующую строку (или пустую строку для завершения):\n";
      cin.get(input, ArSize);
   }
   cout << "Всего наилучшего!\n";
   return 0;
}
void strcount(const char * str)
{
   using namespace std;
   static int total = 0;     //статическая локальная переменная
   int count = 0;            //автоматическая локальная переменная
   cout <<  "\"" << str <<"\" содержит ";
```

```
    while (*str++)      // переход к концу строки
        count++;
    total += count;
    cout << count <<  " символов\n";
    cout << "Всего символов - "<< total << \n";
}
```

Между прочим, программа демонстрирует один способ обработки вводимой строки, которая может превышать размер выделенного для нее массива. Метод ввода cin.get(input,ArSize) запрашивает строку, считывает ее до конца или до позиции ArSize — 1, в зависимости от того, что произойдет быстрее. Символ новой строки остается в очереди на ввод. Программа использует метод cin.get(next) для чтения символа, который следует после введенной строки. Если переменной next присваивается символ новой строки, значит, в результате предыдущего вызова метода cin.get(input, ArSize) строка была прочитана целиком. В противном случае строка содержит непрочитанные символы. Затем в программе следует цикл, который отбрасывает оставшуюся часть строки. Однако код можно изменить таким образом, чтобы остаток строки был задействован в следующем цикле ввода. Кроме того, в программе используется тот факт, что попытка считывания пустой строки с помощью метода get(char *, int) приводит к тому, что cin возвращает false.

Замечание по совместимости

В некоторых устаревших компиляторах не реализовано необходимое условие, чтобы при считывании функцией cin.get(char *, int) пустой строки устанавливался флаг, сигнализирующий об ошибке, который бы приводил к возврату значения false оператором cin. В этом случае проверку условия

```
while (cin)
```

можно заменить следующим кодом:

```
while (input[0])
```

Следующий вариант проверки условия применим как для старой, так и для новой реализации:

```
while (cin && input[0])
```

Ниже показан результат выполнения программы из листинга 9.8:

```
Введите строку:
nice pants
"nice pants" содержит 9 символов
Всего символов - 9
Введите следующую строку (или пустую строку для завершения)
thanks
"thanks" содержит 6 символов
Всего символов - 15
Введите следующую строку (или пустую строку для завершения)
parting in such sweet sorrow
"parting i" содержит 9 символов
Всего символов - 24
Введите следующую строку (или пустую строку для завершения)
ok
"ok" содержит 2 символа
Всего символов - 26
Введите следующую строку (или пустую строку для завершения)
Всего наилучшего!
```

Обратите внимание, что, поскольку размер массива равен `10`, программа не считывает более 9 символов для одной строки. Также заметьте, что автоматическая переменная `count` сбрасывает значение в `0` при каждом вызове функции. Однако для статической переменной `total` значение `0` устанавливается только один раз, в начале выполнения программы. После этого переменная `total` обновляет свое значение между вызовами функций, что позволяет использовать ее для вывода нарастающей суммы.

Спецификаторы и классификаторы

Некоторые ключевые слова C++, называемые *спецификаторами класса памяти* и *cv-спецификаторами*, являются носителями дополнительной информации о хранении. Ниже приводится список спецификаторов классов хранения:

```
auto
register
static
extern
mutable
```

Большинство из этих спецификаторов вам уже знакомы. В одном объявлении можно использовать не более одного из них. Напомним, что ключевое слово `auto` может использоваться в объявлении для документирования факта, что переменная является автоматической. Ключевое слово `register` применяется в объявлении для указания регистрового класса памяти. Ключевое слово `static`, когда оно используется в объявлении с областью видимости в пределах файла, определяет внутреннее связывание. При использовании в локальном объявлении оно определяет статический класс хранения для локальной переменной. Ключевое слово `extern` определяет ссылочное объявление. Это означает, что объявление относится к переменной, определенной в другом файле. Ключевое слово `mutable` задается в контексте спецификатора `const`. Поэтому мы сначала рассмотрим cv-спецификаторы, а затем вернемся к ключевому слову `mutable`.

Существуют следующие cv-спецификаторы:

```
const
volatile
```

(Как вы могли догадаться, аббревиатура *cv* означает `const` и `volatile`.) Спецификатор `const` применяется наиболее часто. Его назначение уже рассматривалось: он указывает, что переменная, будучи однажды инициализированной, не может быть изменена программой. Чуть позже мы вернемся к обсуждению спецификатора `const`.

Ключевое слово `volatile` указывает, что значение в ячейке памяти может быть изменено, даже если в программном коде нет ничего такого, что может модифицировать ее содержимое. Звучит загадочно, но все объясняется просто. Предположим, имеется указатель на аппаратную область памяти, в которой хранятся показания таймера или информация, поступающая из последовательного порта. В этом случае оборудование, а не программа, изменяют содержимое области памяти. Назначение этого ключевого слова состоит в оптимизации возможностей компилятора. Предположим, компилятор обнаружил, что программа использует значение некоторой переменной дважды на протяжении выполнения нескольких операторов. Вместо того чтобы заставить программу дважды предпринимать поиск этого значения, компилятор может

поместить его в регистр. Такая оптимизация предполагает, что значение этой переменной не изменяется между двумя случаями ее использования. Если переменная не объявлена со спецификатором volatile, компилятор вправе осуществлять оптимизацию подобного рода. Ключевое слово volatile говорит компилятору, что подобная оптимизация неприемлема.

Вернемся к спецификатору mutable. С его помощью можно указать, что некоторый элемент структуры (или класса) может быть изменен, даже если переменная типа структуры (или класса) была объявлена со спецификатором const. В качестве примера рассмотрим следующий код:

```
struct data
{
    char name[30];
    mutable int accesses;
    ...
};
const data veep = { "Claybourne Clodde", 0, ... };
strcpy(veep.name, "Joye Joux");   // недопустимо
veep.accesses++;                  // допустимо
```

Спецификатор const структуры veep предотвращает изменение ее элементов, однако спецификатор mutable, предшествующий элементу accesses, снимает с данного элемента это ограничение.

В настоящей книге спецификаторы volatile и mutable не используются, зато мы продолжим изучение спецификатора const.

Дополнительные сведения о спецификаторе const

В языке C++ (но не C) спецификатор const привносит небольшие изменения в классы памяти, используемые по умолчанию. В то время как глобальная переменная по умолчанию обладает внешним связыванием, глобальная переменная со спецификатором const по умолчанию имеет внутреннее связывание. Другими словами, в C++ глобальное определение const обрабатывается так, как будто в нем использован спецификатор static, как в следующем фрагменте кода:

```
const int fingers = 10; // то же, что и static const int fingers;
int main(void)
...
```

Чтобы упростить программирование, в C++ правила, регламентирующие использование постоянных типов данных, были несколько изменены. Предположим, существует набор констант, который требуется поместить в заголовочный файл. Этот файл будет использован в нескольких файлах одной и той же программы. После того как препроцессор включит содержимое этого заголовочного файла в каждый исходный файл, во всех исходных файлах появятся следующие определения:

```
const int fingers = 10;
const char * warning = "Wak!";
```

Если бы определения глобального спецификатора const обладали внешним связыванием, как обычные переменные, это бы вызвало ошибку, поскольку глобальную переменную можно определить только в одном файле. Другими словами, только один файл может содержать представленное выше объявление, в то время как в дру-

гих файлах должны быть предусмотрены ссылочные объявления с использованием ключевого слова extern. Более того, только объявления без ключевого слова extern допускают инициализацию значений:

```
// если бы const-переменная обладала внешним связыванием,
// потребовалось бы ключевое слово extern
extern const int fingers;     // не может быть инициализирована
extern const char * warning;
```

Итак, потребовался бы один набор определений для одного файла и другой набор для остальных файлов. Однако, поскольку const-данные с внешним определением имеют внутреннее связывание, можно использовать одни и те же объявления во всех файлах.

Внутреннее связывание также означает, что каждый файл получает собственный набор констант, а не использует их совместно с другими файлами. Каждое определение принадлежит исключительно файлу, который его содержит. Именно поэтому определения констант целесообразно помещать в заголовочный файл. Таким образом, если включить один и тот же заголовочный файл в два файла исходного кода, они оба получат один и тот же набор констант.

Если по какой-либо причине необходимо, чтобы константа имела внешнее связывание, устанавливаемое по умолчанию внутреннее связывание можно перекрыть ключевым словом extern:

```
extern const int states = 50;   // внешнее связывание
```

Чтобы объявить константу во всех файлах, которые ее используют, нужно воспользоваться ключевым словом extern. В этом состоит отличие от обычной внешней переменной, при объявлении которой ключевое слово extern не применяется, но присутствует в остальных файлах, где данная переменная используется. Кроме того, в отличие от обычных переменных, значение со спецификаторами extern const можно инициализировать. И это необходимо сделать, поскольку const-данные требуют инициализации.

При объявлении переменной со спецификатором const в пределах функции или блока она получает область видимости на уровне блока. Это означает, что эта константа может быть использована только во время выполнения кода данного блока. Это также означает возможность создания констант в функции или в блоке без риска возникновения конфликта имен с константами, определенными в других местах программы.

Функции и связывание

Подобно переменным, функции обладают свойствами связывания, хотя выбор у них более ограниченный. Язык C++, как и C, не позволяет объявить одну функцию внутри другой. Таким образом, все функции автоматически характеризуются статической продолжительностью существования. Это означает, что они существуют, пока выполняется программа. По умолчанию функции обладают внешним связыванием в том смысле, что они могут использоваться различными файлами. Фактически можно в прототипе функции применить ключевое слово extern для указания, что данная функция определена в другом файле, но это не обязательно. (Чтобы программа могла найти функцию в другом файле, этот файл должен быть одним из тех, который

компилируется как часть программы, или библиотечным файлом, поиск которого осуществляет компоновщик.) Кроме того, можно ограничить область видимости функции одним файлом, назначив для нее внутреннее связывание с помощью ключевого слова static. Это ключевое слово применимо к прототипу и к определению функции:

```
static int private(double x);
...
static int private(double x)
{
    ...
}
```

В результате функция доступна только в пределах данного файла. Таким образом, то же самое имя можно применять для некоторой функции в другом файле. Как и в случае с переменными, в файле, содержащем статическое объявление, статическая функция перекрывает внешнее определение. Таким образом, файл, содержащий статическое объявление функции, будет использовать именно эту версию функции даже при наличии внешнего объявления функции с таким же именем.

Язык C++ подчиняется "правилу уникальности объявлений". Оно гласит, что каждая программа должна содержать только одно определение каждой функции, кроме встроенных. Для функций с внешним связыванием это означает, что только один файл многофайловой программы может содержать объявление функции. Однако каждый файл, использующий функцию, должен содержать прототип функции.

Встроенные функции являются исключением из этого правила. Они допускают помещение объявлений в заголовочный файл. Таким образом, каждый файл, использующий заголовочный файл, будет иметь объявление встроенной функции. Однако язык C++ требует, чтобы все объявления встроенных функций были идентичными.

Где компилятор C++ ищет функции?

Предположим, код содержит вызов функции, которая находится в определенном файле программы. Где компилятор C++ будет искать ее определение? Если прототип функции в данном файле указывает, что функция имеет тип static, компилятор проверяет на наличие функции только текущий файл. В противном случае компилятор (а также и компоновщик) просматривает все файлы программы. Если компилятор находит два определения, он выдает сообщение об ошибке, поскольку может существовать только одно определение внешней функции. Если компилятору не удается обнаружить ни одного определения этой функции, он ведет поиск в библиотеках. Следовательно, если разработчик создал определение функции с именем, совпадающим с именем библиотечной функции, компилятор воспользуется версией разработчика, а не библиотечной версией. (Однако в языке C++ имена стандартных библиотечных функций зарезервированы, поэтому дублировать их нельзя.) Некоторые компиляторы-компоновщики для идентификации библиотек, где следует вести поиск, требуют наличия явных инструкций.

Языковое связывание

Существует другая форма связывания, именуемая *языковым связыванием*, которая касается функций. Начнем с основ. Компоновщику требуется иметь уникальное символьное имя для каждой отдельной функции. В языке C это реализуется просто, поскольку с любым именем может быть связана только одна функция. Поэтому для

внутренних потребностей компилятор языка C может, например, преобразовывать имя функции spiff в _spiff. Этот принцип называется *языковым связыванием C*. При этом в языке C++ допускается наличие нескольких функций с одним и тем же именем. Для них необходима трансляция в отдельные символические имена. Таким образом, компилятор C++ инициирует процесс декорирования имен (рассмотренный в главе 8), чтобы создать различные символические имена для перегруженных функций. Например, имя spiff(int) может быть преобразовано в _spiff_i, а имя spiff(double, double) — в _spiff_d_d. Этот принцип называется *языковым связыванием C++*.

Когда компоновщик осуществляет поиск функции C++, соответствующей определенному вызову, он использует метод просмотра, отличный от метода, применяемого для поиска функции, соответствующей вызову в языке C. Но, предположим, что требуется использовать предварительно скомпилированную функцию из библиотеки языка C в программе, написанной на C++. Например, программа содержит следующий код:

```
spiff(22); //  обращение к функции spiff(int) из C-библиотеки
```

В библиотеке C с этой функцией связано символическое имя _spiff, однако для нашего гипотетического компоновщика правило именования диктует поиск символического имени _spiff_i. Для решения этой проблемы можно воспользоваться прототипом функции, указывающим, каким протоколом следует воспользоваться:

```
extern "C" void spiff(int);     // использовать протокол языка C
                                // для поиска имени
extern void spoff(int);         // использовать протокол языка C++
                                // для поиска имени
extern "C++" void spaff(int);   // использовать протокол языка C++
                                // для поиска имени
```

Первый пример демонстрирует использование языкового связывания C. Второй и третий примеры относятся к языковому связыванию C++. Во втором примере этот вид связывания задается по умолчанию, а в третьем — явно.

Схемы хранения и динамическое распределение памяти

Ранее было рассмотрено пять схем, используемых в C++ для выделения памяти переменным (включая массивы и структуры). Они неприменимы к памяти, распределяемой с помощью операции new языка C++ (или более старой функции языка C malloc()). Этот вид памяти называется *динамической памятью*. Как говорилось в главе 4, управление динамической памятью осуществляется операциями new и delete, а не правилами, определяющими область видимости и связывание. Таким образом, динамическая память может выделяться в одной функцией и освобождаться в другой. В отличие от автоматической памяти, динамическая память не подчиняется схеме LIFO (last-in first-out — последним пришел, первым обслужен). Порядок выделения и высвобождения памяти зависит от применения операций new и delete. Как правило, компилятор использует три различных области памяти: одну для статических переменных (эта область может быть разбита на разделы), одну для автоматических переменных и одну для динамической памяти.

Несмотря на то что концепция классов памяти неприменима к динамической памяти, она применима к автоматическим и статическим переменным-указателям, используемым для отслеживания динамической памяти. Предположим, функция содержит следующий оператор:

```
float * p_fees = new float [20];
```

80 байтов памяти (предполагая, что тип `float` занимает 4 байта), выделенных операцией `new`, остаются занятыми до тех пор, пока операция `delete` не освободит их. Однако указатель `p_fees` перестает существовать, когда завершится выполнение функции, содержащей его объявление. Если требуется, чтобы 80 байтов выделенной памяти стали доступными другой функции, нужно передать или вернуть адрес области памяти этой функции. С другой стороны, если сделать объявление указателя `p_fees` внешним, то он станет доступным всем функциям, которые следуют в файле после этого объявления. Воспользовавшись конструкцией

```
extern float * p_fees;
```

во втором файле, можно сделать указатель доступным и в нем. Однако обратите внимание, что оператор, использующий ключевое слово `new` для установки указателя `p_fees`, должен находиться в функции, поскольку переменные со статическим классом памяти могут быть инициализированы только константным выражением:

```
float * p_fees;                // создание указателя на статическую
                               // переменную допустимо
// float * p2 = new float[20]; // инициализация неконстантным значением
                               // в данном случае недопустима
int main()
{
  p_fees = new float [20];
...
```

Замечание по совместимости

Выделяемая с помощью операции `new` память обычно освобождается после завершения работы программы. Однако так происходит не всегда. Например, в некоторых менее устойчивых к ошибкам операционных системах запрос крупного блока памяти может приводить к ситуации, когда после завершения программы эта память не освобождается. На практике лучше всего для освобождения памяти, выделенной операцией `new`, использовать операцию `delete`.

Операция `new` с адресацией

Обычно операция `new` отвечает за поиск в куче блока памяти, обладающего достаточным размером, чтобы удовлетворить запрос памяти. Существует разновидность операции `new` — *операция new с адресацией*. Она позволяет указывать адрес используемого блока. Программист может воспользоваться этой возможностью для создания собственных процедур управления памятью либо для операций с оборудованием, доступ к которому осуществляется по определенным адресам.

Чтобы воспользоваться операцией `new` с адресацией, сначала нужно подключить заголовочный файл `new`, который содержит прототип этой версии операции `new`. Затем можно применить операция `new` с аргументом, указывающим требуемый адрес. В остальном синтаксис операции `new` остается прежним. В частности, операция `new` с адресацией может записываться со скобками или без них.

Следующий фрагмент кода демонстрирует синтаксис всех четырех форм записи операции new:

```
#include <new>
struct chaff
{
  char dross[20];
  int slag;
};
char buffer1[50];
char buffer2[500];
int main()
{
  chaff *p1, *p2;
  int *p3, *p4;
// сначала - обычные формы операции new
  p1 = new chaff;             // помещение структуры в память кучи
  p3 = new int[20];           // помещение массива элементов типа int
                              // в память кучи
// теперь - две формы операции new с адресацией
  p2 = new (buffer1) chaff;   // помещение структуры в область buffer1
  p4 = new (buffer2) int[20]; // помещение массива элементов типа int
                              // в область buffer2
...
```

Ради простоты в этом примере для обеспечения пространством памяти операции new с адресацией используются два статических массива. Таким образом, для структуры chaff выделяется область памяти buffer1, а для массива из 20 элементов типа int — область памяти buffer2.

Теперь, когда мы ознакомились с операцией new с адресацией, рассмотрим пример программы. В листинге 9.9 оба вида операций new используются для создания динамических массивов. Программа иллюстрирует некоторые важные различия между данными разновидностями операции new. Мы их обсудим после анализа вывода этой программы.

Листинг 9.9. newplace.cpp

```
// newplace.cpp -- использование операции new с адресацией
#include <iostream>
#include <new>    // для операции new с адресацией
const int BUF = 512;
const int N = 5;
char buffer[BUF];  // блок памяти
int main()
{
  using namespace std;
  double *pd1, *pd2;
  int i;
  cout << "Вызов обычной и операции new с адресацией:\n";
  pd1 = new double[N];            // использование памяти кучи
  pd2 = new (buffer) double[N];   // использование массива buffer
  for (i = 0; i < N; i++)
   pd2[i] = pd1[i] = 1000 + 20.0 * i;
```

```
  cout << "Адреса буфера:\n" << " куча: " << pd1
       << " статический: " << (void *) buffer <<endl;
  cout << "Содержимое буфера:\n";
  for (i = 0; i < N; i++)
  {
    cout << pd1[i] << " at " << &pd1[i] << "; ";
    cout << pd2[i] << " at " << &pd2[i] << endl;
  }
  cout << "\nВторой вызов обычной и операции new с адресацией:\n";
  double *pd3, *pd4;
  pd3= new double[N];
  pd4 = new (buffer) double[N];
  for (i = 0; i < N; i++)
    pd4[i] = pd3[i] = 1000 + 20.0 * i;
  cout << "Содержимое буфера:\n";
  for (i = 0; i < N; i++)
  {
    cout << pd3[i] << " по адресу " << &pd3[i] << "; ";
    cout << pd4[i] << " по адресу " << &pd4[i] << endl;
  }
  cout << "\n Третий вызов обычной и операции new с адресацией:\n";
  delete [] pd1;
  pd1= new double[N];
  pd2 = new (buffer + N * sizeof(double)) double[N];
  for (i = 0; i < N; i++)
     pd2[i] = pd1[i] = 1000 + 20.0 * i;
  cout << "Содержимое буфера:\n";
  for (i = 0; i < N; i++)
  {
    cout << pd1[i] << " по адресу " << &pd1[i] << "; ";
    cout << pd2[i] << " по адресу " << &pd2[i] << endl;
  }
  delete [] pd1;
  delete [] pd3;
  return 0;
}
```

Ниже показан пример вывода программы из листинга 9.9:

```
Вызов обычной и операции new с адресацией:
Адреса буфера:
  куча: 0xc9d34 статический: 0x42e10
Содержимое буфера:
1000 по адресу 0xc9d34; 1000 по адресу 0x42e10
1020 по адресу 0xc9d3c; 1020 по адресу 0x42e18
1040 по адресу 0xc9d44; 1040 по адресу 0x42e20
1060 по адресу 0xc9d4c; 1060 по адресу 0x42e28
1080 по адресу 0xc9d54; 1080 по адресу 0x42e30
Второй вызов обычной и операции new с адресацией:
Содержимое буфера:
1000 по адресу 0xc9d64; 1000 по адресу 0x42e10
1020 по адресу 0xc9d6c; 1020 по адресу 0x42e18
1040 по адресу 0xc9d74; 1040 по адресу 0x42e20
```

```
1060 по адресу 0xc9d7c; 1060 по адресу 0x42e28
1080 по адресу 0xc9d84; 1080 по адресу 0x42e30
Третий вызов обычной и операции new с адресацией:
Содержимое буфера:
1000 по адресу 0xc9d34; 1000 по адресу 0x42e38
1020 по адресу 0xc9d3c; 1020 по адресу 0x42e40
1040 по адресу 0xc9d44; 1040 по адресу 0x42e48
1060 по адресу 0xc9d4c; 1060 по адресу 0x42e50
1080 по адресу 0xc9d54; 1080 по адресу 0x42e58
```

Замечания по программе

Первое, что заслуживает внимания — операция new с адресацией действительно помещает массив p2 в массив buffer. Для переменных p2 и buffer установлено значение адреса 0x42e10. При этом обычная операция new выделяет массиву p1 адрес памяти с более высоким значением — 0xc9d34, который принадлежит динамически управляемой памяти кучи.

Во-вторых, следует отметить, что при втором вызове обычной операции new он выбирает другой блок памяти — начинающийся с адреса 0xc9d64. Однако второй вызов операции new с адресацией приводит к использованию того же блока памяти, что и прежде. Этот блок начинается с адреса 0x42e10. Здесь важно учитывать, что данный вид операции new просто использует передаваемый в качестве аргумента адрес. Он не анализирует, свободна ли указываемая область памяти, а также не ведет поиск блока неиспользуемой памяти. В результате бремя управления памятью ложится на программиста. Например, третий вызов операции new с адресацией предусматривает смещение в массиве buffer, чтобы использовать новую область памяти:

```
pd2 = new (buffer + N * sizeof(double)) double[N]; //смещение на 40 байтов
```

Третий момент касается наличия или отсутствия операции delete. Для обычной операции new строка

```
delete [] pd1;
```

освобождает блок памяти, начинающийся с адреса 0xc9d34. В результате при следующем вызове операции new возможно повторное использование того же блока. Однако для освобождения памяти, выделенной операцией new с адресацией, операция delete не используется. Фактически, это невозможно. Область памяти указана переменной buffer в статической памяти, а операция delete может использоваться только для указателя на область памяти кучи, выделенной обычной операцией new. Другими словами, массив buffer не попадает в область действия операции delete, поэтому следующая строка вызывает ошибку времени выполнения:

```
delete [] pd2;  // не допускается
```

С другой стороны, если бы для создания буфера памяти использовался обычная операция new, для освобождения всего блока памяти следовало бы применить операцию delete.

Когда операция new с адресацией используется для объектов классов, ситуация усложняется. Эта тема будет продолжена в главе 12.

Пространства имен

В языке C++ имена могут быть присвоены переменным, функциям, структурам, перечислениям, классам, а также элементам классов и структур. По мере усложнения программных проектов возрастает вероятность конфликта имен. При использовании библиотек классов из нескольких источников могут возникнуть конфликты имен. Например, две библиотеки могут определять классы с именами `List`, `Tree` и `Node`, но без соблюдения правил совместимости. Возможно, потребуется использовать класс `List` из одной библиотеки и класс `Tree` из другой, и при этом для каждого из них необходимо взаимодействие с собственной версией класса `Node`. Конфликты подобного рода называются *конфликтами пространства имен*.

Стандарт C++ предусматривает средства, обеспечивающие более совершенное управление областью видимости имен. Реализация поддержки пространства имен в компиляторах потребовала времени, но сейчас эта функциональная возможность распространена повсеместно.

Традиционные пространства имен C++

Прежде чем приступать к изучению новых средств, ознакомимся со свойствами пространства имен, которые уже реализованы в языке C++, и введем некоторые терминологические понятия. Это поможет лучше освоиться с понятием пространства имен.

Для начала ознакомимся с понятием *области объявления*. Это область, в которой могут осуществляться объявления. Например, глобальную переменную можно объявить вне всех функций. Областью объявления для этой переменной является файл, в котором она объявлена. Если объявить переменную внутри функции, ее областью объявления будет ближайший блок, в котором она объявлена.

Второй термин — *потенциальная область видимости*. Потенциальная область видимости переменной начинается с точки объявления и продолжается до конца ее области объявления. Таким образом, потенциальная область видимости более ограничена, чем область объявления, поскольку нельзя использовать переменную в программе ранее позиции, в которой она впервые была определена.

Однако переменная может оказаться видимой не в каждой позиции потенциальной области видимости. Она может быть перекрыта другой переменной с тем же именем, объявленной во вложенной области объявлений. Например, локальная переменная, объявленная в функции (областью объявления служит функция) скрывает глобальную переменную, объявленную в том же файле (областью объявления служит файл). Часть программы, которой фактически доступна данная переменная, называется областью видимости. Именно в этом смысле мы использовали данный термин до сих пор. На рис. 9.5 и 9.6 иллюстрируется применимость терминов *области объявлений*, *потенциальной области видимости* и *области видимости*.

Правила языка C++, касающиеся глобальных и локальных переменных, определяют своего рода иерархию пространств имен. В каждой области определения могут быть определены имена, не зависящие от имен, объявленных в других областях определения. Локальная переменная, объявленная в одной функции, не конфликтует с локальной переменной, объявленной в другой функции.

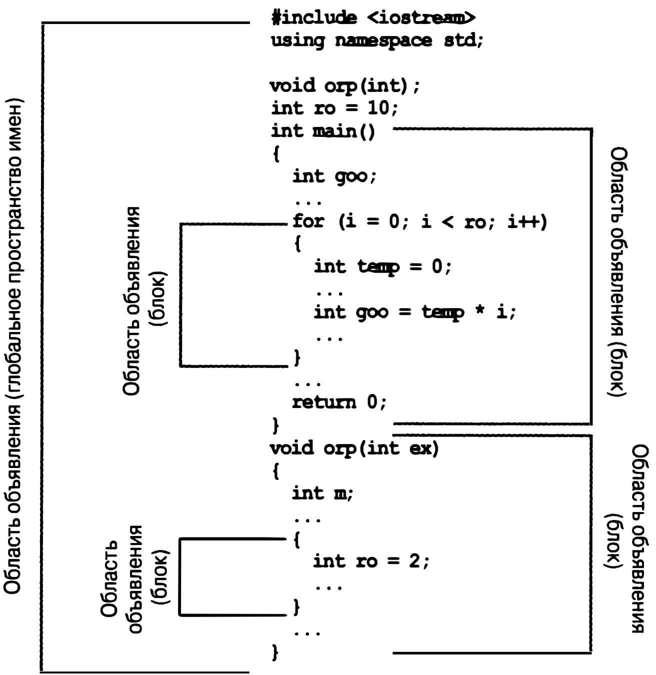


Рис. 9.5. Области объявления

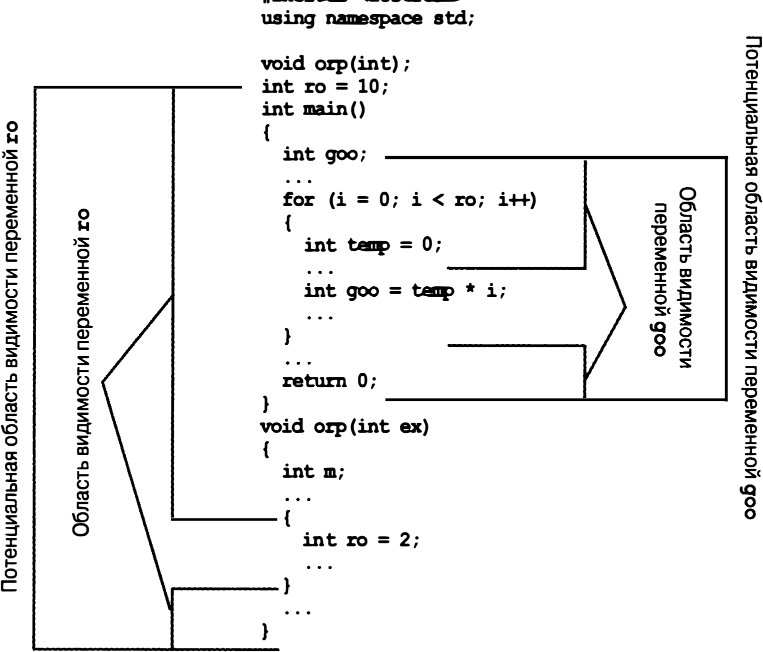


Рис. 9.6. Потенциальная область видимости и область видимости

Новые свойства пространства имен

В настоящее время в языке C++ появилась возможность создавать именованные пространства имен путем создания области определения нового вида, одно из основных назначений которой состоит в предоставлении области, где будут объявляться имена. Имена одного пространства имен не конфликтуют с теми же именами, объявленными в других пространствах имен. При этом существуют механизмы, позволяющие другим частям программы использовать элементы, объявленные в пространстве имен. Например, в приведенном ниже программном коде используется новое ключевое слово namespace для создания двух пространств имен Jack и Jill:

```
namespace Jack {
  double pail;                       // объявление переменной
  void fetch();                      // прототип функции
  int pal;                           // объявление переменной
  struct Well { ... };               // объявление структуры
}

namespace Jill {
  double bucket(double n) { ... }  // определение функции
  double fetch;                      // объявление переменной
  int pal;                           // объявление переменной
  struct Hill { ... };               // объявление структуры
}
```

Пространства имен могут находиться на глобальном уровне или внутри других пространств имен, однако они не могут быть помещены в блок. Следовательно, имя, объявленное в пространстве имен, по умолчанию обладает внешним связыванием (если оно не ссылается на константу).

Кроме пространств имен, объявленных пользователями, существует еще одно пространство имен — *глобальное*. Оно соответствует области объявления на уровне файла. Следовательно, то, что раньше подразумевалось под *глобальными переменными*, сейчас описывается как часть глобального пространства имен.

Имена любого пространства имен не конфликтуют с именами другого пространства имен. Таким образом, имя fetch в пространстве имен Jack вполне может сосуществовать с именем fetch в пространстве Jill, а имя Hill пространства Jill может мирно сосуществовать с внешним именем Hill. Правила, регламентирующие объявления и определения в пространствах имен, совпадают с правилами глобальных объявлений и определений.

Пространства имен *открыты*. Это означает, что можно включать новые имена в существующие пространства имен. Например, оператор

```
namespace Jill {
    char * goose(const char *);
}
```

добавляет имя goose к существующему списку имен пространства Jill.

Подобно этому, исходное пространство имен Jack содержит прототип функции fetch(). Код функции можно поместить далее в этом (или другом) файле, снова воспользовавшись пространством имен Jack:

```
namespace Jack {
   void fetch()
   {
      ...
   }
}
```

Разумеется, для этого необходим метод доступа к именам данного пространства имен. Простейший путь заключается в использовании операции разрешения контекста (`::`), позволяющей *уточнить* имя, указав его вместе с пространством имен:

```
Jack::pail = 12.34;    // использование переменной
Jill::Hill mole;       // создание структуры типа Hill
Jack::fetch();         // использование функции
```

Имя без добавлений, такое как, например, `pail`, называется *неквалифицированным именем*, в то время как имя с указанием пространства имен вроде `Jack::pail`, называется *квалифицированным именем*.

Объявления using и директивы using

Необходимость уточнять имена всякий раз, когда они используются, не очень-то привлекательная перспектива, поэтому язык C++ предлагает два механизма — *объявление* `using` и *директиву* `using` — для упрощения использования имен некоторого пространства. Объявление `using` обеспечивает доступ к отдельным идентификаторам, а директива `using` делает доступным пространство имен в целом.

Объявление `using` предусматривает наличие ключевого слова `using` перед квалифицированным именем:

```
using Jill::fetch;     // объявление using
```

Объявление `using` добавляет к области объявления отдельное имя. Например, объявление `using Jill::fetch` в функции `main()` добавляет имя `fetch` к области объявлений, определенной функцией `main()`. После такого объявления можно использовать имя `fetch` вместо записи `Jill::fetch`. Рассмотренные возможности иллюстрирует следующий фрагмент кода:

```
namespace Jill {
  double bucket(double n) { ... }
  double fetch;
  struct Hill { ... } ;
}
char fetch;
int main()
{
  using Jill::fetch;   // поместить fetch в локальное пространство имен
  double fetch;        // Ошибка! Уже имеется локальное имя fetch
  cin >> fetch;        // чтение значение в переменную Jill::fetch
  cin >> ::fetch;      // чтение значение в глобальную переменную fetch
  ...
}
```

Поскольку объявление `using` включает имя в локальную область объявлений, в этом примере невозможно создание другой переменной с именем `fetch`. Кроме того,

как и любая другая локальная переменная, fetch перекрывает глобальную переменную с таким же именем.

Помещение объявления using на внешний уровень приводит к добавлению соответствующего имени к глобальному пространству имен:

```
void other();
namespace Jill {
  double bucket(double n) { ... }
double fetch;
struct Hill { ... };
}
using Jill::fetch;  // переменная fetch помещается в глобальное
                    // пространство имен
int main()
{
  cin >> fetch;     // чтение значения в локальную переменную Jill::fetch
  other();
...
}
  void other()
  {
     cout << fetch; // вывод значения переменной Jill::fetch
  ...
  }
```

Таким образом, объявление using делает доступным отдельное имя. В отличие от этого, директива using делает доступными все имена. Директива using создается путем включения ключевых слов using namespace перед идентификатором пространства имен. При этом *все* имена данного пространства становятся доступными без необходимости использования операции разрешения контекста:

```
using namespace Jack;  // делает все имена пространства
                       // имен Jack доступными
```

При размещении директивы на глобальном уровне имена пространства имен становятся глобально доступными. В этой книге данный прием неоднократно демонстрировался:

```
#include <iostream>    // помещает имена в пространство имен std
using namespace std;   // делает имена глобально доступными
```

Если поместить директиву using в отдельную функцию, имена станут доступными только в этой функции. Рассмотрим пример:

```
int vorn(int m)
{
  using namespace jack; // делает имена доступными в функции vorn()
...
}
```

Подобная форма нам уже не раз встречалась применительно к пространству имен std.

Необходимо учитывать, что директивы и объявления using увеличивают вероятность конфликта имен. Иначе говоря, если доступны пространства имен jack и

jill, в случае использования операции разрешения контекста неопределенность не возникает:

```
jack::pal = 3;
jill::pal = 10;
```

Переменные jack::pal и jill::pal имеют различные идентификаторы, относящиеся к различным адресам памяти. Однако при использовании объявлений using ситуация меняется:

```
using jack::pal;
using jill::pal;
pal = 4;    // Которой из переменных присвоено значение?
            // Конфликт имен
```

Фактически, компиляторы не позволяют использовать два таких объявления using из-за возникающей неопределенности.

Сравнение директив using и объявлений using

Применение директив using для импорта всех идентификаторов из пространства имен *не равнозначно* использованию множества объявлений using. Это больше напоминает массовое применение операции разрешения контекста. Использование объявления using равносильно ситуации, когда имя объявлено в области действия объявления using. Если некоторое имя уже объявлено в функции, нельзя импортировать то же самое имя с помощью объявления using. Однако в случае использования директивы using имеет место разрешение имен, как если бы соответствующие имена были объявлены в наименьшей области объявлений, содержащей как объявление using, так и само пространство имен. В рассматриваемом ниже примере это будет глобальное пространство имен. Если воспользоваться директивой using для импорта некоторого имени, уже объявленного в функции, локальное имя будет перекрывать имя из пространства имен точно так же, как оно перерывало бы глобальную переменную с тем же именем. Однако это не мешает использовать операцию разрешения контекста, как показано в следующем примере:

```
namespace Jill {
  double bucket(double n) { ... }
  double fetch;
  struct Hill { ... };
}

char fetch;  // глобальное пространство имен
int main()
{
  using namespace Jill;       // импорт всех имен из пространства
  Hill Thrill;                // создание структуры типа Jill::Hill
  double water = bucket(2);   // использование функции Jill::bucket();
  double fetch;               // это не ошибка; скрывается имя Jill::fetch
  cin >> fetch;               // чтение значения в локальную переменную fetch
  cin >> ::fetch;             //чтение значения в глобальную переменную fetch
  cin >> Jill::fetch;         // чтение значения в переменную Jill::fetch
  ...
}
```

```
int foom()
{
  Hill top;          // ОШИБКА
  Jill::Hill crest; // допустимо
}
```

Здесь в функции main() имя Jill::fetch помещено в локальное пространство имен. Оно не имеет локальной области видимости, следовательно, не перекрывает глобального имени fetch. Однако локально объявленное имя fetch перекрывает переменную Jill::fetch и глобальную переменную fetch. Тем не менее, обе этих переменных становятся доступными, если воспользоваться операцией разрешения контекста. Имеет смысл сравнить этот пример с предыдущим, где применяется объявление using.

Еще следует отметить, что, хотя директива using в функции задает обработку имен из области имен, как объявленных вне функции, она не делает эти имена доступными в других функциях файла. Поэтому в предыдущем примере функция foom() не может использовать неквалифицированный идентификатор Hill.

На память!

Предположим, что одно и то же имя определено как в пространстве имен, так и области объявлений. При попытке применить объявление using, чтобы поместить имя из пространства имен в область объявлений, оба имени вступят в конфликт, и отобразится сообщение об ошибке. Если с помощью директивы using перенести имя из пространства имен в область объявлений, локальная версия этого имени перекроет версию из пространства имен.

Вообще говоря, применение объявления using более безопасно, поскольку оно показывает именно те имена, которые делаются доступными. И если такое имя вступит в конфликт с локальным именем, компилятор сообщит об этом. Директива using добавляет все имена без исключений, даже те, которые могут быть не нужны. Если локальное имя вступает в конфликт, оно перекрывает версию имени из пространства имен. Предупреждение об этом не выводится. Кроме того, открытый характер пространств имен означает, что полный список имен некоторого пространства может распространяться на несколько местоположений, что усложняет задачу точно выяснить, какие имена добавляются.

В большинстве примеров этой книги используется следующий прием:

```
#include <iostream> int main()
{
  using namespace std;
```

Во-первых, заголовочный файл iostream помещает все идентификаторы в пространство имен std. Затем директива using делает имена доступными внутри функции main(). В некоторых примерах используется другой подход:

```
#include <iostream>
using namespace std;
int main()
{
```

Здесь все элементы пространства имен std экспортируются в глобальное пространство имен. Основное преимущество такого подхода состоит в рациональности. Его легко реализовать. Кроме того, если используемая система не поддерживает пространства имен, первые две строки кода можно заменить исходной формой:

```
#include <iostream.h>
```

Однако надежды сторонников пространств имен основываются на том, что пользователи будут более разборчивыми и воспользуются либо операцией разрешения контекста, либо объявлением `using`. Другими словами, по их мнению, не следует применять следующее объявление:

```
using namespace std;   // конструкция слишком неразборчива, избегайте ее
```

Вместо этого рекомендуется использовать следующий подход:

```
int x;
std::cin >> x;
std::cout << x << std::endl;
```

Или такой:

```
using std::cin;
using std::cout;
using std::endl;
int x;
cin >> x;
cout << x << endl;
```

Чтобы создать пространство имен, содержащее часто используемые объявления `using`, можно воспользоваться вложением пространств имен, как показано в следующем разделе.

Другие возможности пространств имен

Объявления пространств имен могут быть вложенными, как в следующем примере:

```
namespace elements
{
   namespace fire
   {
      int flame;
      ...
   }
   float water;
}
```

В рассматриваемом случае ссылка на переменную `flame` аналогична записи `elements::fire::flame`. Подобно этому, можно с помощью директивы `using` сделать доступными внутренние имена:

```
using namespace elements::fire;
```

Кроме того, можно пользоваться директивами и объявлениями `using` внутри пространств имен, как показано в следующем примере:

```
namespace myth
{
   using Jill::fetch;
   using namespace elements;
   using std::cout;
   using std::cin;
}
```

Предположим, необходимо обеспечить доступ к переменной Jill::fetch. Поскольку переменная Jill::fetch сейчас является частью пространства имен myth, в котором на нее можно ссылаться по имени fetch, для доступа к переменной возможен следующий способ:

```
std::cin >> myth::fetch;
```

Разумеется, поскольку переменная является еще и частью пространства имен Jill, обращение Jill::fetch также допустимо:

```
std::cout << Jill:fetch;  // отображение значения, считанного
                          // в переменную myth::fetch
```

Возможен и следующий вариант при условии отсутствия конфликта имен локальных переменных:

```
using namespace myth;
cin >> fetch;  // это эквивалентно именам std::cin и Jill::fetch
```

Теперь рассмотрим возможность применения директивы using в пространстве имен myth. Директива using обладает свойством *транзитивности*. Считается, что операция *op* транзитивна, если из выражений A *op* B и B *op* C следует A *op* C. Например, операция > обладает свойством транзитивности. (Иначе говоря, из того, что A больше B и B больше C следует, что A больше C.) В данном контексте это означает, что выполнение оператора

```
using namespace myth;
```

приводит к добавлению пространства имен elements, что аналогично следующему фрагменту:

```
using namespace myth;
using namespace elements;
```

Для пространства имен можно создать псевдоним. Предположим, имеется пространство имен, определенное следующим образом:

```
namespace my_very_favorite_things { ... };
```

Ниже показано, как сделать имя mvft псевдонимом для my_very_favorite_things:

```
namespace mvft = my_very_favorite_things;
```

Этот же прием можно применить для упрощения операций с вложенными пространствами имен:

```
namespace MEF = myth::elements::fire;
using MEF::flame;
```

Неименованные пространства имен

Можно создать неименованное пространство имен, опустив его имя после ключевого слова namespace:

```
namespace    // неименованное пространство имен
{
    int ice;
    int bandycoot;
}
```

Результат этих объявлений будет таким же, как если бы за ними следовала директива using. Другими словами, имена, объявленные в этом пространстве имен, находятся в потенциальной области видимости, которая продолжается до границы области объявлений, содержащей неименованное пространство имен. В этом отношении имена неименованного пространства подобны глобальным переменным. Однако если пространство не имеет имени, нельзя явно применить директиву или объявление using, чтобы сделать имена доступными в другом месте. В частности, недопустимо использование имен из неименованного пространства в любом файле, кроме того, что содержит объявление пространства имен. Это может служить альтернативой применению статических переменных с внутренним связыванием. Действительно, стандарт C++ указывает, что использование ключевого слова static в пространствах имен и в глобальном диапазоне доступа является устаревшим. ("Устаревший" — это термин, используемый в стандарте для указания того, что данная практика еще применима, но в будущем может стать непригодной после последующего пересмотра стандарта.) Рассмотрим следующий пример кода:

```
static int counts;  // статическая переменная, внутреннее связывание
int other;
int main()
{
...
}
int other()
{
...
}
```

С точки зрения стандарта предпочтительнее следующий вариант:

```
namespace
{
   int counts;      // статическая переменная, внутреннее связывание
}
int other();
int main()
{
...
{
int other()
}
...
}
```

Пример пространства имен

Рассмотрим пример многофайловой программы, демонстрирующей некоторые возможности пространства имен. Первый файл — заголовочный (см. листинг 9.10). Он содержит типичные для таких файлов элементы — константы, определения структур и прототипы функций. В нашем случае элементы помещены в два пространства имен. Первое пространство имен, называемое pers, содержит определение структуры Person, а также прототипы функции, которая помещает в структуру имя некоторого лица, и функции, отображающей содержимое структуры. Второе пространство

имен, называемое debts, определяет структуру для хранения имени лица и суммы его задолженности. Эта структура использует структуру Person, поэтому пространство имен debts содержит директиву using, которая делает имена пространства pers доступными в пространстве имен debts. Пространство имен debts также содержит прототипы функций.

Листинг 9.10. namesp.h

```
// namesp.h
// Создание пространств имен pers и debts
namespace pers
{
  const int LEN = 40;
  struct Person
  {
    char fname[LEN];
    char lname[LEN];
  };
  void getPerson(Person &);
  void showPerson(const Person &);
}
namespace debts
{
  using namespace pers;
  struct Debt
  {
    Person name;
    double amount;
  };
  void getDebt(Debt &);
  void showDebt(const Debt &);
  double sumDebts(const Debt ar[], int n);
}
```

Второй файл этого примера (см. листинг 9.11) следует обычному шаблону, когда файл исходного кода содержит определения для прототипов функций из заголовочного файла. Имена функций, объявленные в пространстве имен, имеют область видимости в пределах пространства имен. Поэтому определения должны находиться в том же самом пространстве имен, что и объявления. Это тот случай, когда открытая природа пространства имен удобна. Исходные пространства имен подключаются с помощью файла namesp.h (см. листинг 9.10). Затем код файла добавляет определения функций в два пространства имен, как показано в листинге 9.11. Кроме того, файл namesp.cpp иллюстрирует обеспечение доступа к элементам пространства имен std с помощью объявления using и операции разрешения контекста.

Листинг 9.11. namesp.cpp

```
// namesp.cpp -- пространства имен
#include <iostream>
#include "namesp.h"
namespace pers
{
```

```
  using std::cout;
  using std::cin;
  void getPerson(Person & rp)
  {
    cout << "Введите имя: ";
    cin >> rp.fname;
    cout << "Введите фамилию: ";
    cin >> rp.lname;
  }
  void showPerson(const Person & rp)
  {
    std::cout << rp.lname << ", " << rp.fname;
  }
}
namespace debts
{
  void getDebt(Debt & rd)
  {
    getPerson(rd.name);
    std::cout << "Введите сумму задолженности: ";
    std::cin >> rd.amount;
  }
  void showDebt(const Debt & rd)
  {
    showPerson(rd.name);
    std::cout <<": $" << rd.amount << std::endl;
  }
  double sumDebts(const Debt ar[], int n)
  {
    double total = 0;
    for (int i = 0; i < n; i++)
      total += ar[i].amount;
    return total;
  }
}
```

Наконец, третий файл программы (см. листинг 9.12) представляет собой файл исходного кода, который использует структуры и функции, объявленные и определенные в пространствах имен. В листинге 9.12 показаны несколько методов обеспечения доступа к идентификаторам пространства имен.

Листинг 9.12. usenmsp.cpp

```
// usenmsp.cpp -- использование пространств имен
#include <iostream>
#include "namesp.h"
void other(void);
void another(void);
int main(void)
{
  using debts::Debt;
  using debts::showDebt;
  Debt golf = { {"Benny", "Goatsniff"}, 120.0 };
```

```
  showDebt(golf);
  other();
  another();

  return 0;
}
void other(void)
{
  using std::cout;
  using std::endl;
  using namespace debts;
  Person dg = {"Doodles", "Glister"};
  showPerson(dg);
  cout << endl;
  Debt zippy[3];
  int i;
  for (i = 0; i < 3; i++)
    getDebt(zippy[i]);
  for (i = 0; i < 3; i++)
    showDebt(zippy[i]);
  cout << "Общая задолженность: $" << sumDebts(zippy, 3) << endl;.cpp;.cpp;.cpp;

  return;
}
void another(void)
{
  using pers::Person;;
  Person collector = { "Milo", "Rightshift" };
  pers::showPerson(collector);
  std::cout << std::endl;
}
```

Функция `main()` в листинге 9.12 начинается с двух объявлений `using`:

```
using debts::Debt;        // делает доступным определение структуры Debt
using debts::showDebt;    // делает доступной функцию showDebt
```

Обратите внимание, что в объявлениях `using` используются только имена. Так, во втором примере отсутствует описание типа возвращаемого значения и сигнатуры функции `showDebt`, а приводится лишь ее имя. (Таким образом, если функция перегружена, одно объявление `using` будет импортировать все ее версии.) Кроме того, хотя функции `Debt` и `showDebt` используют тип `Person`, нет необходимости импортировать какую-либо переменную типа `Person`, поскольку в пространстве имен `debt` уже содержится директива `using`, подключающая пространство имен `pers`.

Функция `other()` использует менее удачный способ импорта всего пространства имен с помощью директивы `using`:

```
using namespace debts;    // делает доступными для функции other()
                          // все имена из пространств debts и pers
```

Поскольку директива `using` в функции `other()` импортирует пространство имен `pers`, эта функция может использовать тип данных `Person` и функцию `showPerson()`.

Наконец, в функции another() применяется объявление using и операция разрешения контекста для доступа к отдельным именам:

```
using pers::Person;
pers::showPerson(collector);
```

Результаты выполнения программы, представленной в листингах 9.10, 9.11 и 9.12, выглядят следующим образом:

```
Goatsniff, Benny: $120
Glister, Doodles
Введите имя: Arabella
Введите фамилию: Binx
Введите сумму задолженности: 100
Введите имя: Cleve
Введите фамилию: Delaproux
Введите сумму задолженности: 120
Введите имя: Eddie
Введите фамилию: Fiotox
Введите сумму задолженности: 200
Binx, Arabella: $100
Delaproux, Cleve: $120
Fiotox, Eddie: $200
Общая задолженность: $420
Rightshift, Milo
```

Будущее пространств имен

По мере освоения программистами пространств имен будет вырабатываться новый стиль программирования. Ниже представлены современные рекомендации:

- Используйте переменные именованных пространств имен вместо внешних глобальных переменных.
- Используйте переменные неименованных пространств имен вместо статических глобальных переменных.
- Если вы разработали библиотеку функций или классов, поместите ее в пространство имен. Современный язык C++ уже требует помещения стандартных библиотечных функций в пространство имен std. Это же распространяется и на функции, унаследованные из языка C. Например, заголовочный файл math.c, который совместим с языком C, не использует пространства имен, но заголовочный файл cmath языка C++ предусматривает помещение различных математических функций в пространство имен std. (Пока не все компиляторы прошли этот эволюционный путь.)
- Используйте директиву using только в качестве временного средства адаптации устаревшего кода к использованию пространств имен.
- Не используйте директивы using в заголовочных файлах. Во-первых, этот прием скрывает имена, которые становятся доступными. Кроме того, порядок следования заголовочных файлов может влиять на поведение программы. Если вы используете директиву using, помещайте ее после всех директив препроцессора #include.

- Для импорта имен отдавайте предпочтение операции разрешения контекста или объявлению `using`.
- Для объявлений `using` отдавайте предпочтение локальной области видимости вместо глобальной.

Имейте в виду, что главная цель использования пространств имен состоит в упрощении управлением крупными проектами. Для простой программы из одного файла применение директивы `using` не является большим недостатком.

Как уже говорилось, нововведение пространств имен отражается в изменении имен заголовочных файлов. Прежние заголовочные файлы, такие как `iostream.h`, не используют пространств имен, а новый заголовочный файл, `iostream`, предусматривает использование пространства имен `std`.

Резюме

Особенности языка C++ благоприятствуют разработке программ, расположенных во множестве файлов. Эффективная стратегия организации программ состоит в использовании заголовочного файла для определения пользовательских типов данных и прототипов функций, управляющих этими данными. Целесообразно помещать определения функций в отдельный файл исходного кода. Заголовочный файл и файл исходного кода совместно определяют и реализуют определяемый пользователем тип данных, а также возможности его использования. Функция `main()` и другие функции, использующие функции файла исходного кода, могут быть помещены в третий файл.

Классы хранения C++ определяют продолжительность нахождения переменных в памяти, а также части программы, имеющие к ним доступ (область видимости и связывание). Автоматическими называются переменные, которые определены в пределах блока, такого как тело функции или блок внутри него. Они существуют и доступны только тогда, когда программа выполняет операторы блока, содержащего их определения. Автоматические переменные могут быть объявлены с помощью спецификаторов класса памяти `auto` и `register`. Отсутствие спецификатора равносильно применению спецификатора `auto`. Спецификатор `register` указывает компилятору, что переменная будет часто использоваться.

Статические переменные существуют на протяжении всего периода выполнения программы. Переменная, определенная за пределами всех функций, доступна всем функциям данного файла, которые следуют за ее определением (область видимости в пределах файла). К ней возможен доступ со стороны других файлов программы (внешнее связывание). В файле, использующем такую переменную, она должна быть объявлена с ключевым словом `extern`. Если переменная используется в нескольких файлах, один из файлов должен содержать ее определяющее объявление (без слова `extern`), а остальные — ссылочные объявления (с ключевым словом `extern`). Переменная, объявленная вне функций, но снабженная ключевым словом `static`, обладает областью видимости в пределах файла, но не доступна другим файлам (внутреннее связывание). Переменная, определенная внутри блока, но сопровождаемая ключевым словом `static`, является локальной по отношению к этому блоку (локальная область видимости без связывания), однако сохраняет свое значение в течение всего времени выполнения программы.

По умолчанию функции C++ обладают внешним связыванием. Поэтому они могут совместно использоваться несколькими файлами. Однако функции со спецификатором static имеют внутреннее связывание. Использование таких функций ограничено файлом, который содержит их определения.

Пространства имен позволяют определять именованные области памяти, в которых можно объявлять идентификаторы. Они предназначены для снижения вероятности конфликта имен, что особенно важно для крупных программ, использующих программный код различных поставщиков. Для обеспечения доступа к идентификаторам пространств имен могут применяться операции разрешения контекста, объявления using или директивы using.

Вопросы для самоконтроля

1. Каким классом хранения вы воспользуетесь в следующих ситуациях?
   а. homer — это формальный аргумент (параметр) функции.
   б. Должно быть обеспечено совместное использование переменной secret двумя файлами.
   в. Переменная topsecret должна быть доступна функциям одного файла, но скрыта от других файлов.
   г. Переменная beencalled фиксирует количество вызовов функции, которая ее содержит.
2. Опишите различия между объявлением using и директивой using.
3. Перепишите следующий код таким образом, чтобы он не использовал ни объявления, ни директивы using.

```
#include <iostream>
using namespace std;
int main()
{
    double x;
    cout << "Введите значение: ";
    while (! (cin >> x) )
    {
        cout << "Недопустимый ввод. Пожалуйста, введите число: ";
        cin.clear();
        while (cin.get() != '\n')
            continue;
    }
    cout << "Значение = " << x << endl;
    return 0;
}
```

4. Перепишите следующий код таким образом, чтобы он использовал объявления вместо директив using.

```
#include <iostream>
using namespace std;
int main()
{
    double x;
    cout << "Введите значение: ";
```

```
    while (! (cin >> x) )
    {
       cout << "Недопустимый ввод. Пожалуйста, введите число: ";
       cin.clear();
       while (cin.get() != '\n')
           continue;
    }
    cout << "Значение = " << x << endl;
    return 0;
}
```

5. Предположим, что функция `average(3,6)` возвращает число типа `int`, которое является средним арифметическим двух аргументов типа `int`, когда она вызывается в одном файле, и число типа `double`, которое является средним арифметическим от двух аргументов типа `int`, когда она вызывается в одном файле одной и той же программы. Как это реализовать?
6. Какие данные будет выводить следующая двухфайловая программа?

```
// file1.cpp
#include <iostream>
using namespace std;
void other();
void another();
int x = 10;
int y;
int main()
{
cout << x << endl;
    {
         int x = 4;
         cout << x << endl;
         cout << y << endl;
    }
    other();
    another();
    return 0;
}
void other()
{
   int y=1;
   cout << "Other: " << x << ", " << y << end;
}
// file2.cpp
#include <iostream>
using namespace std;
extern int x;
namespace
{
  int y = -4;
}
void another()
{
  cout << "another(): " << x << ", " << y << endl;
}
```

7. Какие данные будет выводить следующая программа?

```
#include <iostream>
using namespace std;
void other();
namespace n1
{
   int x = 1;
}
namespace n2
{
  int x = 2;
}
int main()
{
   using namespace n1;
   cout << x << endl;
   {
         int x = 4;
         cout << x <<  ", " << n1::x << ", " << n2::x << endl;
   }
   using n2::x;
   cout << x << endl;
   other();
   return 0;
}
void other()
{
   using namespace n2;
   cout << x << endl;
   {
         int x = 4;
         cout << x << ", " << n1::x << ", " << n2::x << endl;
   }
   using n2::x;
   cout << x << endl;
}
```

Упражнения по программированию

1. Задан заголовочный файл:

```
// golf.h -- для программы pe9-1.cpp
const int Len = 40;
struct golf
{
    char fullname[Len];
    int handicap;
};
// неинтерактивная версия:
//функция присваивает структуре типа golf имя игрока и его гандикап (фору),
// используя передаваемые ей аргументы
void setgolf(golf & g, const char * name, int hc);
```

```
// интерактивная версия:
// функция предлагает пользователю ввести имя и гандикап,
// присваивает элементам структуры g введенные значения;
// возвращает 1, если введено имя, и 0, если введена пустая строка
int setgolf(golf & g);
// функция устанавливает новое значение гандикапа
void handicap(golf & g, int hc);
// функция отображает содержимое структуры типа golf
void showgolf(const golf & g);
```

Обратите внимание, что функция `setgolf()` перегружена. Вызов первой версии функции имеет следующий вид:

```
golf ann
setgolf(ann, "Ann Birdfree", 24);
```

Функция предоставляет информацию, которая содержится в структуре `ann`. Вызов второй версии функции имеет следующий вид:

```
golf andy
setgolf(andy);
```

Функция предлагает пользователю ввести имя и гандикап, а затем сохранит эти данные в структуре `andy`. Эта функция могла бы (но в этом нет необходимости) использовать первую версию внутренне.

Постройте многофайловую программу с использованием этого заголовочного файла. Один файл, с именем `golf.cpp`, должен содержать определения функций, которые соответствуют прототипам заголовочного файла. Второй файл должен содержать функцию `main()` и обеспечивать реализацию всех свойств функций, для которых определены прототипы. Например, цикл должен запрашивать ввод массива структур типа `golf` и прекращать ввод после заполнения массива, либо когда вместо имени игрока в гольф пользователь введет пустую строку. Чтобы получить доступ к структурам типа `golf`, функция `main()` должна использовать только функции, для которых определены прототипы.

2. Измените код листинга 9.8, заменив массив символов объектом типа `string`. Программа не должна проверять соответствие вводимой строки запрашиваемым данным. Для установки факта ввода пустой строки можно реализовать сравнение вводимых данных со значением `""`.

3. Начните со следующего объявления структуры:

```
struct chaff
{
   char dross[20];
   int slag;
};
```

Напишите программу, использующую операцию `new` с адресацией, чтобы поместить массив из двух подобных структур в буфер. Затем программа присваивает значения элементам структуры (не забудьте воспользоваться функцией `strcpy()` для заполнения массива элементов типа `char`) и отображает ее содержимое с помощью цикла. Вариант 1 предусматривает использование статического массива в качестве буфера памяти, как в листинге 9.9. Вариант 2 состоит в использовании обычной операции `new` для выделения буфера памяти.

4. Напишите программу, включающую три файла и использующую следующее пространство имен:

```
namespace SALES
{
  const int QUARTERS = 4;
  struct Sales
  {
     double sales[QUARTERS];
     double average;
     double max;
     double min;
  }
  // копирует 4 или n элементов, если n<4, из массива ar
  // в элемент s типа Sales, вычисляет и сохраняет
  // среднее арифметическое, максимальное и минимальное
  // значения введенных чисел;
  // оставшимся элементам структуры Sales, если таковые есть,
  // присваиваются значения 0
  void setSales(Sales & s, const double ar[], int n);

  // интерактивно подсчитывает продажи за 4 квартала,
  // сохраняет их в элементе s типа Sales, вычисляет
  // и сохраняет среднее арифметическое, а также максимальное
  // и минимальное значения введенных чисел;
  void setSales(Sales & s)

  //отображает всю информацию из структуры s
  void showSales(const Sales & s);
}
```

Первый файл является заголовочным и содержит пространство имен. Второй файл содержит исходный код. Он расширяет пространство имен, предоставляя определения трех функций, имеющих прототипы. В третьем файле объявляются два объекта `Sales`. Он должен использовать интерактивную версию функции `setSales()` для предоставления значений первой структуре, и неинтерактивную версию той же функции для предоставления значений второй структуре. Для отображения содержимого обеих структур в файле должна использоваться функция `showSales()`.

ГЛАВА 10

Объекты и классы

В этой главе:

- Процедурное и объектно-ориентированное программирование
- Концепция классов
- Как определить и реализовать класс
- Общедоступный (`public`) и приватный (`private`) доступ к классу
- Данные-члены класса
- Методы класса (также называемые функциями-членами класса)
- Создание и использование объектов класса
- Конструкторы и деструкторы классов
- Функции-члены `const`
- Указатель `this`
- Создание массивов объектов
- Области видимости классов
- Абстрактные типы данных

Объектно-ориентированное программирование (ООП) — это особый концептуальный подход к проектированию программ, и язык C++ расширяет C средствами, облегчающими применение такого подхода. Ниже перечислены наиболее важные средства ООП:

- Абстракция.
- Инкапсуляция и сокрытие данных.
- Полиморфизм.
- Наследование.
- Повторное использование кода.

Класс — это единственное наиболее важное расширение C++, предназначенное для реализации этих средств и связывающее их между собой. В настоящей главе начинается объяснение концепции классов. Здесь рассмотрена абстракция, инкапсуляция и сокрытие данных, а также показано, как в классах реализуются эти средства. Обсуждается также определение класса, приводится класс с общедоступным и приватным разделами, создаются функции-члены, которые работают с данными класса. Кроме того, настоящая глава познакомит вас с конструкторами и деструкторами, которые представляют собой специальные функции-члены, предназначенные для создания и уничтожения объектов, относящихся к классу. И, наконец, вы встретитесь с указателем `this`, важным компонентом программирования некоторых классов. Последующие главы продолжат обсуждение в части перегрузки операций (другой вариант полиморфизма) и наследования — основы повторного использования кода.

Процедурное и объектно-ориентированное программирование

Несмотря на то что настоящая книга в основном посвящена вопросам объектно-ориентированного программирования, стоит также более пристально взглянуть на стандартный процедурный подход, присущий таким языкам, как C, Pascal и BASIC. Давайте рассмотрим пример, демонстрирующий разницу между ООП и процедурным программированием.

Предположим, что вас, как нового члена софтбольной команды "Гиганты Жанра" попросили вести статистику игр команды. Естественно, вы используете для этого свой компьютер. Если вы — процедурный программист, то, вероятно, будете думать следующим образом:

> Так, посмотрим... Я хочу вводить имя, количество подач, количество попаданий, средний уровень успешных подач (для тех, что не следит за бейсболом или софтболом: средний уровень успешных подач — это количество попаданий, деленное на официальное количество подач, выполненных игроком; подачи прекращаются, когда игрок достигает базы либо выбивает мяч за пределы поля, но некоторые события, такие как получение обводки, не засчитываются в официальное количество подач), а также другую существенную статистику по каждому игроку. Минутку, но ведь предполагается, что компьютер должен мне облегчать жизнь, поэтому я хочу, чтобы он вычислял автоматически кое-что из этого, например, средний уровень успешных подач. Кроме того, я хочу, чтобы программа выдавала отчет по результатам. Как мне это организовать? Думаю, это следует делать с помощью функций. Да, я сделаю, чтобы `main()` вызывала функцию для получения ввода, затем другую функцию — для вычислений, а потом третью — для вывода результатов. Гм-м-м... А что случится, когда я получу данные о другой игре? Я не хочу начинать все сначала! Ладно, я могу добавить функцию для обновления статистики. Черт возьми, может быть, мне понадобится меню в `main()`, чтобы выбирать между вводом, вычислением, обновлением и выводом данных. Гм-м-м ... А как мне представлять данные? Можно использовать массив строк для хранения имен игроков, другой массив — для хранения числа подач каждого игрока и еще один массив — для хранения числа попаданий и так далее... Нет, это глупость! Я могу спроектировать структуру, чтобы хранить всю информацию о каждом игроке и затем использовать массив таких структур для представления всей команды.

Короче говоря, при процедурном подходе вы сначала концентрируетесь на процедурах, которым должны следовать, а только потом думаете о том, как представить данные. (На заметку: поскольку вам не нужно держать программу работающей в течение всего игрового сезона, то, вероятно, вы захотите сохранять данные в файле и читать их оттуда).

Теперь посмотрим, как изменится ваш взгляд, когда вы наденете шляпу ООП (выполненную в симпатичном полиморфном стиле). Вы начнете думать о данных. Более того, вы будете думать о данных не только в терминах их представления, но и в терминах их использования:

Посмотрим, что я должен отслеживать? Игроков, конечно. Поэтому мне нужен объект, который представляет игрока в целом, а не только его уровень успешных подач или их общее количество. Да, это будет основная единица данных — объект, представляющий имя и статистику игрока. Мне понадобятся некоторые методы для работы с этим объектом. Гм-м-м, думаю, понадобится метод для ввода базовой информации в объект. Некоторые данные должен вычислять компьютер, например, средний уровень успешных подач. Я могу добавить метод для выполнения вычислений. И программа должна выполнять вычисления автоматически, чтобы пользователю не нужно было помнить о том, что он должен просить ее об этом. Кроме того, понадобятся методы для обновления и отображения информации. То есть у пользователя должно быть три способа взаимодействия с данными: инициализация, обновление и вывод отчетов. Это и будет пользовательский интерфейс.

Короче говоря, при объектно-ориентированном подходе вы концентрируетесь на объекте, как его представляет пользователь, думая о данных, которые нужны для описания объекта и операциях, описывающих взаимодействие пользователя с данными. После разработки описания интерфейса вы перейдете к выработке решений о том, как реализовать этот интерфейс и как организовать хранение данных. И, наконец, вы соберете все это вместе в программу, соответствующую вашему новому дизайну.

Абстракции и классы

Жизнь полна сложностей, и единственный способ справится со сложностью — это ограничиться упрощенными абстракциями. Вы сами состоите из более чем октильона (10^{48}) атомов. Некоторые студенты, изучающие сознание, могут сказать, что ваше сознание представляет собой коллекцию полуавтономных агентов. Но гораздо проще думать о себе, как о едином целом. В компьютерных вычислениях абстракция — это ключевой шаг в представлении информации в терминах его интерфейса с пользователем. То есть вы абстрагируете основные операционные свойства проблемы и выражаете решение в этих терминах. В примере с софтбольной статистикой интерфейс описывает, как пользователь инициирует, обновляет и отображает данные. От абстракций легко перейти к определяемым пользователем типам, которые в C++ представлены классами, реализующими абстрактный интерфейс.

Что такое тип?

Давайте подумаем немного о том, что собой представляет тип. Например, кто такой зануда (nerd)? Если следовать популярному стереотипу, вы можете представить его в визуальных образах: толстый, в запотевших очках, карман, полный ручек и так далее. После некоторого размышления вы можете прийти к заключению, что зануду лучше описать присущим ему поведением — например, тем, как он реагирует на затруднительные ситуации. У вас подобное положение, если вы не имеете в виду “притянутые” аналогии, с процедурным языком вроде C. Во-первых, вы думаете о данных в терминах их появления — как они сохраняются в памяти. Например, `char` занимает 1 байт памяти, а `double` — обычно 8 байт. Но если немного подумать, то вы придете к заключению, что тип данных также определен в терминах операций, которые допустимо выполнять с ним. Например, к типу `int` можно применять все арифметиче-

ские операции. Целые числа можно складывать, вычитать, умножать, делить. Можно также применять операцию вычисления модуля (`%`).

С другой стороны, рассмотрим указатели. Указатель может требовать для своего хранения столько же памяти, как и `int`. Он даже может иметь внутреннее представление в виде целого числа. Но указатель не позволяет выполнять над собой те же операции, что целое. Например, вы не можете перемножать два указателя. Эта концепция не имеет смысла, поэтому в C++ она не реализована. Отсюда, когда вы объявляете переменную вроде `int` или указателя на `float`, то не просто выделяете память для нее, но также устанавливаете, какие операции допустимы с этой переменной. Короче говоря, спецификация базового типа выполняет три вещи:

- Определяет, сколько памяти нужно объекту.
- Определяет, как интерпретируются биты памяти (`long` и `float` могут занимать одинаковое число бит памяти, но транслируются в числовые значения по-разному).
- Определяет, какие операции, или методы, могут быть применены с использованием этого объекта данных.

Для встроенных типов информация об операциях встроена в компилятор. Но когда вы определяете пользовательские типы в C++, то должны предоставить эту информацию самостоятельно. В качестве вознаграждения за эту дополнительную работу вы получаете мощь и гибкость новых типов данных, соответствующих требованиям реального мира.

Классы в C++

Класс — это двигатель C++, предназначенный для трансляции абстракций в пользовательские типы. Он комбинирует представление данных и методов для манипулирования этими данными в пределах одного аккуратного пакета. Давайте взглянем на класс, представляющий акционерный капитал.

Во-первых, нужно немного подумать, как представлять акции. Вы можете взять один пакет акций в качестве базовой единицы и определить класс, представляющий этот пакет. Однако это потребует создания 100 объектов для представления 100 пакетов, что явно не практично. Вместо этого в качестве базовой единицы вы можете представить персональную долю владельца в определенном пакете. Количество акций, находящихся во владении, будут частью представления данных. Реалистичный подход должен позволять поддерживать хранение информации о таких вещах, как начальная стоимость и дата покупки; это необходимо для налогообложения. Кроме того, он должен предусматривать обработку таких событий, как разделение пакетов. Это может показаться довольно амбициозным для первой попытки определения класса, поэтому вы можете ограничиться неким идеализированным, упрощенным взглядом на предмет. В частности, можно ограничить реализуемые операции следующим перечнем:

- Приобретение пакета акций компании.
- Приобретение дополнительных акций в имеющийся пакет.
- Продажа пакета.
- Обновление объема доли в пакете акций.
- Отображения информации о пакетах, находящихся во владении.

Вы можете использовать этот список для определения общедоступного интерфейса класса (и при необходимости добавлять дополнительные средства позднее). Чтобы поддержать этот интерфейс, вам нужна некоторая информация. Опять-таки, вы можете воспользоваться упрощенным подходом. Например, вам не нужно беспокоиться о принятой в США практике оперировать пакетами акций в объемах, кратных 8 долларам. (Очевидно, что Нью-Йоркская фондовая биржа в курсе такого упрощения, имевшего место в предыдущем изданий книги, потому что уже решила внести изменения в систему, описанную здесь.) Вам нужно хранить следующую информацию:

- Наименование компании.
- Количество акций, находящихся во владении.
- Объем каждой доли.
- Общий объем всех долей.

Далее вы можете определить класс. Обычно спецификация класса состоит из двух частей:

- *Объявление класса*, описывающее компоненты данных в терминах членов данных, а также общедоступный интерфейс в терминах функций-членов, называемых *методами*.
- *Определение методов класса*, описывающее, как реализованы определенные функции-члены.

Грубо говоря, объявление класса представляет общий обзор класса, в то время как определение методов сосредоточено на его деталях.

Что такое интерфейс?

Интерфейс — это совместно используемая часть, предназначенная для взаимодействия двух систем — например, между компьютером и принтером, или между пользователем и компьютерной программой. Например, пользователем можете быть вы, а программой — текстовый процессор. Когда вы используете текстовый процессор, то не переносите слова напрямую из своего сознания в память компьютера. Вместо этого вы взаимодействуете с интерфейсом, предложенным программой. Вы нажимаете клавишу, и компьютер отображает символ на экране. Вы перемещаете мышь, и компьютер перемещает курсор на экране. Вы случайно щелкаете кнопкой мышки, и что-то происходит с абзацем, в котором вы печатаете. Программный интерфейс управляет преобразованием ваших намерений в специфическую информацию, сохраняемую в компьютере.

В отношении классов мы говорим об общедоступном интерфейсе. В этом случае потребителем его является программа, использующая класс, система взаимодействия состоит из объектов класса, а интерфейс состоит из методов, предоставленных тем, кто написал этот класс. Интерфейс позволяет вам, как программисту, написать код, взаимодействующий с объектами класса, и таким образом, позволяет программе взаимодействовать с объектами класса. Например, чтобы определить количество символов в объекте `string`, вам не нужно открывать этот объект и смотреть что у него внутри. Вы просто используете метод `size()` класса, предоставленный его разработчиком. Таким образом, метод `size()` является частью общедоступного интерфейса между пользователем и объектом класса `string`. Аналогичным образом метод `getline()` является частью общедоступного интерфейса класса `istream`. Программа, использующая `cin`, не обращается напрямую к внутренностям объекта `cin` для чтения строки ввода, вместо этого всю работу выполняет `getline()`. Если вам нужны более доверительные отношения, то вместо того, чтобы думать о программе, использующей класс, как о внешнем пользователе, вы можете думать об авторе программы, использующей этот класс, как о внешнем пользователе. Но в любом случае, чтобы использовать класс, вы должны знать его общедоступный интерфейс, а чтобы написать класс, должны создать ему такой интерфейс.

Разработка классов и программ, использующих их, требует выполнения определенных шагов. Вместо того чтобы взять на вооружение их целиком, давайте разделим процесс разработки на небольшие стадии: позднее код, из которых они состоят, будет объединен в листинге 10.3. В листинге 10.1 представлена первая стадия, пробное объявление класса под именем Stock. (Чтобы упростить идентификацию классов, в настоящей книге используется общий, но не универсальный метод — соглашение о написании имен классов с заглавной буквы.) Обратите внимание, что листинг 10.1 выглядит как объявление структуры с некоторыми небольшими дополнениями, такими как функции-члены, а также разделы public и private. Вскоре вы разберетесь в этом объявлении (поэтому не используйте его в качестве модели), но вначале давайте посмотрим, как это объявление работает.

Листинг 10.1. Первая часть stocks.cpp

```
// Начало файла stocks.cpp
#include <iostream>
#include <cstring>
class Stock // объявление класса
{
private:
   char company[30];
   int shares;
   double share_val;
   double total_val;
   void set_tot() { total_val = shares * share_val; }
public:
   void acquire(const char * co, int n, double pr);
   void buy(int num, double price);
   void sell(int num, double price);
   void update(double price);
   void show();
}; // обратите внимание на точку с запятой в конце
```

На детали реализации класса мы взглянем позднее, а сейчас проанализируем наиболее общие средства. Для начала, ключевое слово C++ class идентифицирует код в листинге 10.1 как определение дизайна класса. Этот синтаксис идентифицирует Stock как имя типа для нового класса. Это позволяет объявлять переменные, называемые *объектами*, или *экземплярами* типа Stock. Каждый индивидуальный объект этого типа представляет отдельный пакет акций, находящийся во владении. Например, следующее объявление:

```
Stock sally;
Stock solly;
```

создает два объекта с именами sally и solly. Объект sally, например, может представлять пакет акций определенной компании, принадлежащий Салли.

Далее, отметим, что информация, которую вы решили сохранять, появляется в форме данных-членов класса, таких как компания и размер доли. Переменная-член company объекта sally, например, хранит наименование компании, переменная-член share содержит количество долей общего пакета акций компании, которыми владеет Салли, переменная-член share_val хранит объем каждой доли, а перемен-

ная-член `total_val` — общий объем всех долей. Аналогично операции, которые вы хотите иметь в виде функций-членов (или методов), называются `sell()` и `update()`. Функции-члены могут быть определены на месте — как, например, `set_tot()`, либо могут быть представлены прототипами — как остальные функции-члены класса. Полные определения остальных функций-членов появятся позже, но прототип уже позволяет определить их интерфейс. Связка данных и методов в единое целое — наиболее замечательное свойство класса. При таком дизайне создание объекта типа `Stock` автоматически устанавливает правила, определяющие использование объекта.

Вы уже видели, как классы `istream` и `ostream` владеют функциями-членами вроде `get()` и `getline()`. Прототипы функций в определении класса `Stock` демонстрируют, как функции-члены устанавливаются. Заголовочный файл `iostream`, например, содержит прототип `getline()` в объявлении класса `iostream`.

Новыми также являются ключевые слова `private` и `public`. Эти метки описывают *управление доступом* к членам класса. Любая программа, которая использует объект определенного класса, может иметь непосредственный доступ к членам из раздела `public`. Доступ к членам класса из раздела `private` программа может получить *только* через общедоступные функции-члены из раздела `public` (или же, как будет показано в главе 11, через дружественные функции). Например, единственный способ изменить переменную `shares` класса `Stock` — это воспользоваться одной из функций-членов класса `Stock`. Таким образом, общедоступные функции-члены действуют в качестве посредников между программой и приватными членами объекта. Они предоставляют интерфейс между объектом и программой. Эта изоляция данных от прямого доступа со стороны программы называется *сокрытием данных* (C++ предлагает третье ключевое слово для управления доступом — `protected`, которое мы поясним, когда будем говорить о наследовании в главе 13). (См. рис. 10.1.) В то время как сокрытие данных может быть недобросовестным действием, когда мы говорим в общепринятом контексте о доступе к информации инвестиционных фондов, это является хорошей практикой в компьютерных вычислениях, поскольку предохраняет целостность данных.

Дизайн класса пытается отделить общедоступный интерфейс от специфики реализации. Общедоступный интерфейс представляет абстрактный компонент дизайна. Собрание деталей реализации в одном месте и отделение их от абстракции называется *инкапсуляцией*. *Сокрытие данных* (размещение данных в разделе `private` класса) представляет собой способ инкапсуляции, и поэтому скрывает функциональные детали реализации в разделе `private`, как это сделано в классе `Stock` с функцией `set_tot()`. Другой пример инкапсуляции — обычная практика помещения определения функций класса в файл, отдельный от объявления класса.

Объектно-ориентированное программирование и C++

Объектно-ориентированное программирование — это стиль программирования, который в определенной степени можно применять в любом языке. Безусловно, вы можете реализовать многие идеи ООП в обычной программе на языке C. Например, в главе 9 представлен пример (см. листинги 9.1, 9.2, 9.3), в котором заголовочный файл содержит прототип структуры вместе с прототипами функций, предназначенных для манипулирования этой структурой. Функция `main()` просто определяет переменные этого типа и использует ассоциированные функции для управления этими переменными. `main()` не имеет непосредственного доступа к членам структур. По сути, этот пример объявляет абстрактный тип, который помещает формат хранения и прототипы функций в заголовочный файл, скрывая реальное представление данных от `main()`.

C++ включает средства, специально предназначенные для реализации подхода ООП, что позволяет продвинуться в этом направлении на несколько шагов дальше, чем в языке C. Во-первых, размещение прототипов функций в едином объявлении класса вместо того, чтобы держать их раздельно, унифицирует описание за счет размещения его в одном месте. Во-вторых, объявление данных с приватным доступом разрешает доступ к ним только для авторизованных функций. Если в примере на C функция `main()` имеет непосредственный доступ к членам структуры, что нарушает дух ООП, это не нарушает никаких правил языка C. Однако, при попытке прямого доступа, скажем, к члену `shares` объекта `Stock` будут нарушены правила языка C++ и компилятор перехватит это.

Отметим, что сокрытие данных не только предотвращает прямой доступ к данным, но также избавит вас (в роли пользователя этого класса) от необходимости знать то, как представлены данные. Например, член `show()` отображает, помимо прочего, общую сумму пакета акций, находящегося во владении. Это значение может быть сохранено в виде части объекта, как это делает код в листинге 10.1, либо может быть при необходимости вычислено. С точки зрения пользователя нет разницы, какой подход применяется. Необходимо знать только то, что делают различные функции-члены — то есть какие аргументы они принимают, и какие типы значений возвращают. Принцип состоит в том, чтобы отделить детали реализации от дизайна интерфейса. Если позже вы найдете лучший способ реализации представления данных или деталей внутреннего устройства функций-членов, то сможете изменить их без изменения программного интерфейса, что значительно облегчает поддержку и сопровождение программ.

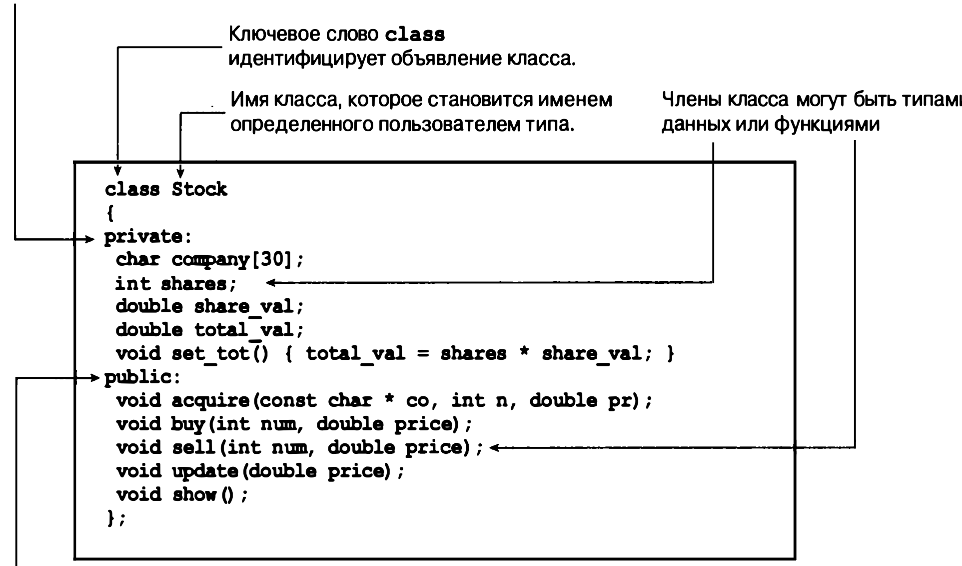


Рис. 10.1. Класс `Stock`

Управление доступом к членам: public или private?

Вы можете объявлять члены класса — будь они единицами данных или функциями-членами — как в общедоступном (public), так и в приватном (private) разделах объявления класса. Но поскольку одним из основных принципов ООП является сокрытие данных, то единицы данных обычно размещаются в разделе private. Функции-члены, которые образуют интерфейс класса, размещаются в разделе public. В противном случае вы не сможете вызывать эти функции из программы. Как показывает объявление класса Stock, вы все же можете поместить функции-члены в раздел private. Вы не сможете вызвать такие функции из программы непосредственно, но общедоступные (public) методы могут использовать их. Как правило, вы используете приватные функции-члены для управления деталями реализации, которые не формируют собой часть общедоступного интерфейса.

Вы не должны использовать ключевое слово private в объявлении класса, поскольку это спецификатор доступа к объектам класса по умолчанию:

```
class World
{
   float mass;       // приватный по умолчанию
   char name[20];    // приватный по умолчанию
public:
   void tellall(void);
   ...
}
```

Однако в этой книге используется явное указание метки private для того, чтобы подчеркнуть концепцию сокрытия данных.

Классы и структуры

Объявление класса выглядит похожим на объявление структуры с дополнениями в виде функций-членов и меток видимости private и public. Фактически C++ расширяет и на структуры свойства, которые есть у классов. Единственная разница состоит в том, что типом доступа по умолчанию у структур является public, в то время как у классов — private. Программисты на C++ обычно используют классы для реализации дескрипторов классов, тогда как ограниченные структуры применяются для простых объектов данных или же классов, не имеющих компонентов private.

Реализация функций-членов класса

Вы по-прежнему обязаны определять вторую часть спецификации класса, то есть предоставлять код для тех функций-членов, которые описаны прототипами в объявлении класса. Определения функций-членов очень похожи на обычные определения функций. Каждая из них имеет заголовок и тело. Определения функций-членов могут иметь тип возврата и аргументы. Но, кроме того, с ними связаны две специфических характеристики:

- Когда вы определяете функцию-член, то используете операцию разрешения контекста (::) для идентификации класса, которому принадлежит функция.
- Методы класса имеют доступ к private-компонентам класса.

Рассмотрим это подробнее.

Во-первых, заголовок для функции-члена использует операцию разрешения контекста (::) для идентификации класса, которому она принадлежит. Например, заголовок для функции-члена update() выглядит так:

```
void Stock::update(double price)
```

Эта нотация означает, что вы определяете функцию update(), которая является членом класса Stock. Но это означает не только то, что функция update() является функцией-членом, но также и то, что вы можете использовать то же имя для функций-членов другого класса. Например, функция update() для класса Button будет иметь следующий заголовок:

```
void Button::update(double price)
```

Таким образом, операция разрешения контекста идентифицирует класс, к которому данный метод относится. Говорят, что идентификатор update() имеет область видимости класса. Другие функции-члены класса Stock могут при необходимости использовать метод update() без операции разрешения контекста. Это связано с тем, что они принадлежат одному классу, а, следовательно, имеют общую область видимости. Использование update() за пределами объявления класса и определений методов, однако, требует применения специальных мер, которые мы рассмотрим далее.

Единственный способ однозначного разрешения имен методов — использовать полное имя, включающее имя класса. Stock::update() называется *квалифицированным именем* функции. Простое имя update(), с другой стороны, является аббревиатурой (неквалифицированным именем) полного имени и может применяться только в контексте класса.

Следующей специальной характеристикой методов является то, что метод может иметь доступ к приватным членам класса. Например, метод show() может использовать код вроде следующего:

```
cout << "Company: " << company
     << " Shares: " << shares << endl
     << " Share Price: $" << share_val
     << " Total Worth: $" << total_val << endl;
```

Здесь company, shares и так далее являются приватными членами данных класса Stock. Если вы попытаетесь воспользоваться функцией, не являющейся членом, для доступа к этим данным-членам, то компилятор остановит вас. (Однако дружественные функции, описанные в главе 11, представляют собой исключение.)

Памятуя об этих двух обстоятельствах, вы можете реализовать методы класса, как показано в листинге 10.2. Эти определения методов могут быть помещены в отдельный файл либо в тот же, где находится объявление класса. Это самый простой, хотя и не наилучший способ сделать объявление класса доступным определениям методов. (Лучший способ, который мы применим далее в настоящей главе, заключается в том, чтобы использовать объявление класса и отдельный файл исходного кода с определением функций.) Для предоставления возможности работы с областями имен, в некоторых методах используется квалификатор std::, а в других — объявления using.

Листинг 10.2. Продолжение stocks.cpp

```
// Продолжение stocks.cpp -- реализация функций-членов класса
void Stock::acquire(const char * co, int n, double pr)
{
   std::strncpy(company, co, 29); // усечь co для помещения в company
   company[29] = '\0';
   if (n < 0)
   {
     std::cerr << "Количество пакетов не может быть отрицательным; для "
               << company << " установлено в 0.\n";
     shares = 0;
   }
   else
     shares = n;
   share_val = pr;
   set_tot();
}
void Stock::buy(int num, double price)
{
   if (num < 0)
   {
     std::cerr << "Количество приобретаемых пакетов не может быть отрицательным. "
              << "Транзакция прервана.\n";
   }
   else
   {
     shares += num;
     share_val = price;
     set_tot();
   }
}

void Stock::sell(int num, double price)
{
   using std::cerr;
   if (num < 0)
   {
     cerr << "Количество продаваемых пакетов не может быть отрицательным. "
          << "Транзакция прервана.\n";
   }
   else if (num > shares)
   {
     cerr << "Вы не можете продать больше того, чем владеете! "
          << "Транзакция прервана.\n";
   }
   else
   {
     shares -= num;
     share_val = price;
     set_tot();
   }
}
```

```
void Stock::update(double price)
{
   share_val = price;
   set_tot();
}
void Stock::show()
{
   using std::cout;
   using std::endl;
   cout << "Компания: " << company
        << " Пакетов: " << shares << endl
        << " Цена пакета: $" << share_val
        << " Общая стоимость: $" << total_val << endl;
}
```

Замечания о функциях-членах

Функция acquire() управляет первоначальной покупкой пакета акций заданной компании, в то время как buy() и sell() — дополнительной покупкой и продажей акций для существующего пакета. Методы buy() и sell() гарантируют, что количество купленных или проданных акций не будет отрицательным. Кроме того, если пользователь пытается продать больше акций, чем у него есть, функция sell() отменит транзакцию. Техника объявления данных приватными (private) и ограничения доступа к общедоступным (public) функциям дает вам контроль над использованием данных. В данном случае это позволяет создать "предохранитель", защищающий от некорректных транзакций.

Четыре из функций-членов устанавливают или сбрасывают числовое значение члена класса total_val. Вместо того чтобы повторять в коде вычисление этого значения четыре раза, каждая из общедоступных функций-членов вызывает функцию set_tot(). Поскольку эта функция представляет собой просто реализацию внутреннего кода, а не является частью общедоступного интерфейса, класс объявляет set_tot() как приватную функцию-член. (То есть set_tot() представляет собой функцию-член, используемую автором класса, но не используемую теми, кто пишет код, использующий класс.) Если вычисления, выполняемые функцией, сложные, это может уменьшить общий объем исходного кода. Однако здесь главное в том, что, используя вызов этой функции вместо того, чтобы каждый раз повторять код вычислений, вы гарантируете, что всегда будет применяться абсолютно идентичный алгоритм. Кроме того, если вы захотите изменить его (что в данном конкретном случае маловероятно), то должны будете сделать это только в одном месте.

Метод acquire() использует strncopy() для копирования строк. Если вы забыли, напомним, что вызов strncopy(s2, s1, n) копирует либо s1 в s2 целиком (если длина s1 менее n символов) либо только первые n символов s1, если длина s1 превышает n. Если s1 содержит менее n символов, то функция strncopy() дополняет s2 нулевыми байтами до длины n. То есть strncopy(firstname, "Tim", 6) копирует символы T, i и m в firstname и добавляет три нулевых символа с тем, чтобы общее количество скопированных было равно 6. Но если длина s1 больше шести, то никакие нулевые символы не добавляются. То есть вызов strncopy(firstname,

"Priscilla", 4) просто копирует символы P, r, i и s в firstname, создавая из них символьный массив, но поскольку пропущен завершающий нулевой символ, это не будет являться строкой. Поэтому acquire() помещает в конец массива нулевой символ, тем самым гарантируя, что это будет строка.

Объект cerr

cerr, как и cout, является объектом класса ostream. Разница в том, что перенаправление вывода операционной системы касается только cout, но не cerr. Объект cerr применяется для вывода сообщений об ошибках. Поэтому если вы переадресуете вывод программы в файл, и при ее выполнении происходит ошибка, то вы по-прежнему получите сообщение о ней на экране. (В Unix можно переадресовывать вывод cout и cerr независимо друг от друга Операция командной строки > переадресует вывод cout, а операция 2> — cerr.)

Встроенные методы

Любая функция с определением внутри объявления класса автоматически становится встроенной. То есть Stock::set_tot() является встроенной функцией. Объявления класса часто используют встроенные функции для сокращения функций-членов и set_tot() — пример такого случая.

Если хотите, вы можете определить функцию-член вне объявления класса и, тем не менее, сделать ее встроенной. Чтобы это сделать, просто используйте квалификатор inline при определении функции в разделе реализации класса:

```
class Stock
{
   private:
   ...
   void set_tot();            // определение оставлено отдельным
   public:
   ...
};
inline void Stock::set_tot()   // использование inline в определении
{
   total_val = shares * share_val;
}
```

Специальные правила для встроенных функций требуют, чтобы они были определены в каждом файле, где они используются. Самый простой способ гарантировать, что встроенные определения доступны всем файлам в многофайловой программе — это включить определение inline в тот же заголовочный файл, где объявлен класс. (Некоторые системы разработки снабжены интеллектуальными компоновщиками, допускающими inline-определения в отдельном файле реализации.)

Кстати, согласно *правилу перезаписи*, определение метода в объявлении класса эквивалентно замене определения метода прототипом и последующей перезаписью определения в виде встроенной функции — немедленно после объявления класса. То есть исходное определение set_tot() в листинге 10.2 эквивалентно только что показанному, где определение следует за объявлением класса.

Какой объект использует метод?

Мы подошли к одному из наиболее важных аспектов применения объектов: как применяется метод класса к объекту. Код вроде следующего:

```
shares += num;
```

использует член shares в качестве объекта. Но какого объекта? Отличный вопрос! Чтобы ответить на него, сначала рассмотрим процесс создания объекта. Наиболее простой способ объявления переменных класса выглядит так:

```
Stock kate, joe;
```

Это создает два объекта класса Stock, один по имени kate, а другой — joe.

Далее рассмотрим, как использовать функцию-член с одним из этих объектов. Ответ, как и со структурами и с членами структур, состоит в применении операции членства:

```
kate.show();      // объект kate вызывает функцию-член
joe.show();       // объект joe вызывает функцию-член
```

Первый вариант вызывает show() в качестве члена объекта kate. Это значит, что метод интерпретирует shares как kate.shares и share_val как kate.share_val. Аналогично вызов joe.show() заставляет метод show() интерпретировать shares как joe.shares и share_val как joe.share_val, соответственно.

На память!

Когда вы вызываете функцию-член, она использует данные-члены конкретного объекта, примененного для ее вызова.

Подобным же образом вызов kate.sell() запускает функцию set_tot(), как если бы это была kate.set_tot(), позволяя ей получать доступ к данным объекта kate.

Каждый вновь созданный вами объект содержит хранилище для его собственных внутренних переменных — членов класса. Но все объекты одного класса разделяют общий набор методов, по одной копии каждого. Предположим, например, что kate и joe — это объекты класса Stock. В этом случае kate.shares занимает один кусок памяти, а joe.shares — другой. Но kate.show() и joe.show() — оба представляют собой один и тот же метод, то есть оба выполняют один и тот же блок кода, но применяют этот код к разным данным. Вызов функции-члена — это то, что в некоторых объектно-ориентированных языках называется *отправкой сообщения*. Таким образом, отправка сообщения двум разным объектам вызывает один и тот же метод, который применяется к двум разным объектам. (См. рис. 10.2.)

Использование классов

В настоящей главе было показано, как определять класс и их методы. Следующий шаг состоит в разработке программы, которая будет создавать и использовать объекты класса. Целью языка C++ является сделать применение классов насколько возможно простым — подобно базовым, встроенным типам вроде int и char. Вы можете создавать объект класса объявлением переменной этого класса либо использовать операцию new для размещения в памяти объекта этого класса. Вы можете передавать объекты в аргументах, возвращать их из функций, присваивать один объект другому.

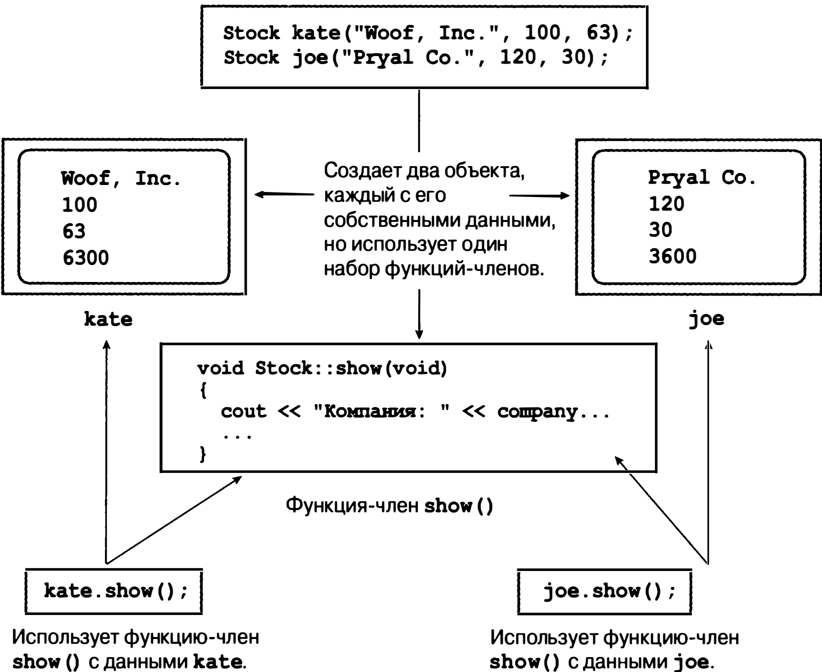


Рис. 10.2. Объекты, данные и функции-члены

Язык C++ предоставляет средства для инициализации объектов, "обучению" cin и cout распознавать объекты и даже выполнению автоматического приведения типов между объектами подобных классов. Пройдет некоторое время, прежде чем вы научитесь делать все эти вещи, но давайте начнем с наиболее простых свойств. Несомненно, вы уже видели, как объявлять объекты класса и вызывать функции-члены. В листинге 10.3 эти приемы комбинируются с объявлением класса и определением функций-членов с тем, чтобы сформировать завершенную программу. Он создает объект типа Stock по имени stock1. Программа проста, тем не менее, она проверяет все средства, которые вы встроите в класс.

Листинг 10.3. Полный текст программы stocks.cpp

```
// stocks.cpp — полный текст программы
#include <iostream>
#include <cstring>
class Stock // объявление класса
{
private:
   char company[30];
   int shares;
   double share_val;
   double total_val;
   void set_tot() { total_val = shares * share_val; }
public:
   void acquire(const char * co, int n, double pr);
   void buy(int num, double price);
```

```
   void sell(int num, double price);
   void update(double price);
   void show();
}; // обратите внимание на точку с запятой в конце
void Stock::acquire(const char * co, int n, double pr)
{
   std::strncpy(company, co, 29); // усечь co для помещения в company
   company[29] = '\0';
   if (n < 0)
   {
     std::cerr << "Количество пакетов не может быть отрицательным; для "
     << company << " установлено в 0.\n";
     shares = 0;
   }
   else
   shares = n;
   share_val = pr;
   set_tot();
}
void Stock::buy(int num, double price)
{
   if (num < 0)
   {
     std::cerr << "Количество приобретаемых пакетов не может быть
     отрицательным. "
     << "Транзакция прервана.\n";
   }
   else
   {
     shares += num;
     share_val = price;
     set_tot();
   }
}
void Stock::sell(int num, double price)
{
   using std::cerr;
   if (num < 0)
   {
     cerr << "Количество продаваемых пакетов не может быть отрицательным. "
          << "Транзакция прервана.\n";
   }
   else if (num > shares)
   {
     cerr << "Вы не можете продать больше того, чем владеете! "
          << "Транзакция прервана.\n";
   }
   else
   {
     shares -= num;
     share_val = price;
     set_tot();
   }
}
```

```
void Stock::update(double price)
{
   share_val = price;
   set_tot();
}
void Stock::show()
{
   using std::cout;
   using std::endl;
   cout << "Компания: " << company
        << " Пакетов: " << shares << endl
        << " Цена пакета: $" << share_val
        << " Общая стоимость: $" << total_val << endl;
}
int main()
{
   using std::cout;
   using std::ios_base;
   Stock stock1;
   stock1.acquire("NanoSmart", 20, 12.50);
   cout.setf(ios_base::fixed);       // формат #.##
   cout.precision(2);                // формат #.##
   cout.setf(ios_base::showpoint);   // формат #.##
   stock1.show();
   stock1.buy(15, 18.25);
   stock1.show();
   stock1.sell(400, 20.00);
   stock1.show();
   return 0;
}
```

Программа в листинге 10.3 использует три команды форматирования:

```
cout.setf(ios_base::fixed);     // использовать формат с фиксированной
                                // десятичной точкой
cout.precision(2);              // две позиции справа от десятичной точки
cout.setf(ios_base::showpoint);// показать завершающие нули
```

Кроме того, она включает в себя объявления `using`:

```
using std::ios_base;
```

Это случай вложенных пространств имен. Идентификаторы `fixed` и `showpoint` являются частью пространства имен `ios_base`, а `ios_base` — часть пространства имен `std`. Общий эффект от форматирующих операторов заключается в отображении двух разрядов справа от десятичной точки, включая завершающие нули. На самом деле в соответствии со сложившейся практикой, только первые два из этих операторов необходимы, а более старым реализациям C++ требуются только первый и третий. Использование всех трех дает одинаковый вывод — как для новых, так и для старых реализаций. Подробные сведения можно найти в главе 17. Кстати, вот как выглядит вывод программы из листинга 10.3:

```
Компания: NanoSmart Пакетов: 20
  Цена пакета: $12.50 Общая стоимость: $250.00
```

```
Компания: NanoSmart Пакетов: 35
  Цена пакета: $18.25 Общая стоимость: $638.75
Вы не можете продать больше того, чем владеете! Транзакция прервана.
Компания: NanoSmart Пакетов: 35
  Цена пакета: $18.25 Общая стоимость: $638.75
```

Обратите внимание, что `main()` — это просто механизм для тестирования класса `Stock`. Когда класс `Stock` заработает так, как вы ожидаете, его можно будет использовать в качестве пользовательского типа в других программах. Важнейшим моментом для применения нового типа является понимание того, что делает функция-член — вы не должны задумываться о деталях реализации. См. следующую врезку "Клиент-серверная модель".

Клиент-серверная модель

Программисты ООП часто обсуждают дизайн программ в терминах клиент-серверной модели. Согласно этой концепции *клиентом* является программа, которая использует класс. Объявление класса, включая его методы, образует *сервер*, который является ресурсом, доступным программе, в нем нуждающейся. Клиент использует сервер только через общедоступный (`public`) интерфейс. Это означает, что единственной заботой клиента, и, как следствие, заботой программиста, является знание интерфейса. Заботой сервера и, как следствие, заботой разработчика сервера, является обеспечение того, чтобы его реализация достоверно и точно соответствовала интерфейсу. Это позволяет программистам разрабатывать клиент и сервер независимо друг от друга, без внесения в сервер таких изменений, которые нежелательным образом отобразятся на поведении клиента.

Обзор

Первый шаг в спецификации дизайна класса — это представление объявления класса. Объявление класса смоделировано на основе объявления структуры и может включать в себя данные-члены и функции-члены. Объявление имеет приватный (`private`) раздел, и члены, объявленные в этом разделе, могут быть доступны только через функции-члены. Объявление также содержит общедоступный (`public`) раздел, и объявленные в нем члены могут быть непосредственно доступны программе, использующей объекты класса. Как правило, данные-члены попадают в приватный раздел, а функции-члены — в общедоступный, поэтому типичное объявление класса имеет следующую форму:

```
class className
{
private:
   объявления данных-членов
public:
   объявления функций-членов
};
```

Содержимое общедоступного раздела включает абстрактную часть дизайна — общедоступный интерфейс. Инкапсуляция данных в приватном разделе защищает их целостность и называется *сокрытием данных*. Таким образом, использование классов — это способ, который предлагает C++ для облегчения реализации абстракций ООП, сокрытия данных и инкапсуляции.

Второй шаг в спецификации дизайна класса — это реализация функций-членов класса. Вы можете включать в объявление полное определение функций вместо прототипов, однако общепринятая практика заключается в том, чтобы определять функции отдельно, за исключением наиболее простых. В этом случае вам понадобится операция разрешения контекста для индикации того, к какому классу данная функция-член принадлежит. Например, предположим, что класс Bozo имеет функцию-член Retort(), которая возвращает указатель на тип char. Заголовок функции должен выглядеть примерно так:

```
char * Bozo::Retort()
```

Другими словами, Retort() — не только функция типа char *, это функция типа char *, принадлежащая классу Bozo. Полное, или квалифицированное, имя функции будет выглядеть как Bozo::Retort(). Имя Retort(), с другой стороны, является аббревиатурой квалифицированного имени, и оно должно быть использовано только в определенных случаях, таких как в коде методов класса.

Другой способ описания этой ситуации — это говорить о том, что Retort имеет контекст класса, поэтому операция разрешения контекста необходима для квалификации имени, когда она используется вне объявления и вне методов класса.

Чтобы создать объект, который является частным примером класса, вы используете имя класса, как если бы оно было именем типа:

```
Bozo bozetta;
```

Это работает потому, что класс является пользовательским типом.

Вы вызываете функцию-член класса, или метод, используя объект класса. Вы делаете это с помощью операции точки:

```
cout << bozetta.Retort();
```

Это — вызов функции-члена Retort(), и всякий раз когда код этой функции обращается к определенным данным-членам, она использует значения членов объекта bozetta.

Конструкторы и деструкторы классов

Теперь нам нужно сделать с классом Stock нечто большее. Есть некоторые определенные стандартные функции, называемые *конструкторами* и *деструкторами*, которыми обычно следует снабдить класс. Поговорим о том, почему они необходимы и как их создавать.

Одной из целей C++ является сделать применение объектов классов подобным применению стандартных типов. Однако код, который был приведен до сих пор в настоящей главе, не позволяет вам инициализировать объект Stock таким же способом, как это можно сделать с int или struct. То есть применение синтаксиса инициализации не применимо к типу Stock:

```
int year = 2001; // корректная инициализация
struct thing
{
   char * pn;
   int m;
};
```

```
thing amabob = {"wodget", -23}; // корректная инициализация
Stock hot = {"Sukie's Autos, Inc.", 200, 50.25};//Нельзя! Ошибка компиляции
```

Причина, по которой вы не можете таким способом инициализировать объект Stock, состоит в том, что к данным класса разрешен только приватный доступ, а это означает, что единственный способ, с помощью которого программа может получить доступ к ним — так это через функции-члены.

Поэтому вам нужно придумать соответствующую функцию-член, если вам необходимо успешно инициализировать объект. (Вы можете инициализировать объект класса так, как это показано выше, если объявите данные-члены как public вместо private, но, поступая так, вы нарушите один из базовых принципов использования классов — сокрытие данных.)

Вообще лучше, чтобы все объекты инициализировались при их создании. Например, рассмотрим следующий код:

```
Stock gift;
gift.buy(10, 24.75);
```

При существующей реализации класса Stock объект gift не имеет установленного значения члена company. Дизайн класса предполагает, что пользователь вызовет acquire() раньше любых других функций-членов, однако нет способа гарантировать это. Единственной возможностью обойти эту трудность была бы автоматическая инициализация объектов при их создании. Чтобы достичь ее, C++ предлагает специальные функции-члены, называемые *конструкторами класса*, специально предназначенные для создания новых объектов и присваивания значений их членам-данным. Точнее выражаясь, C++ регламентирует имя для таких функций-членов, а также синтаксис их вызова, тогда как ваша задача — написать определение методов. Имя совпадает с именем класса. Например, возможный конструктор для класса Stock — это функция-член Stock(). Прототип и заголовок конструктора обладают интересным свойством: даже несмотря на то, что конструкторы не имеют возвращаемого значения, они не объявляются с типом void. Фактически конструкторы не имеют объявленного типа.

Объявление и определение конструкторов

Теперь вам понадобится написать конструктор Stock. Поскольку объект Stock имеет три значения, которые ему нужно получить из внешнего мира, вы должны передать конструктору три аргумента. (Четвертое значение — это член total_val; он вычисляется на основе shares и share_val, поэтому вы не должны передавать его конструктору.) Возможно, вы захотите передать только значение члена company и установить остальные значения равными нулю; вы можете сделать это, используя аргументы по умолчанию (см. главу 8). Таким образом, прототип будет выглядеть следующим образом:

```
// прототип конструктора с несколькими аргументами по умолчанию
Stock(const char * co, int n = 0, double pr = 0.0);
```

Первый аргумент представляет собой указатель на строку, используемую для инициализации члена класса — массива символов company. Аргументы n и pr предоставляют значения для членов shares и share_val. Обратите внимание, что тип возвращаемого значения не указан. Прототип размещен в общедоступном разделе объявления класса.

Ниже показан один из вариантов определения тела конструктора:

```
// определение конструктора
Stock:: Stock (const char * co, int n, double pr)
{
   std::strncpy(company, co, 29);
   company[29] = '\0';
   if (n < 0)
   {
      std::cerr << "Количество пакетов не может быть отрицательным; для "
                << company  << " установлено в 0.\n";
      shares = 0;
   }
   else
      shares = n;
   share_val = pr;
   set_tot();
}
```

Это тот же код, что вы использовали ранее в настоящей главе для функции `acquire()`. Разница в том, что в данном случае программа автоматически выполнит конструктор при объявлении объекта.

Внимание!

Часто пытаются использовать имена переменных-членов класса в качестве аргументов конструктора, как показано в следующем примере:

```
// Нельзя!
Stock::Stock(const char * name, int shares, double share_val)
{
...
}
```

Это неверно. Аргументы конструктора не являются переменными-членами; они представляют значения, которые присваиваются членам класса. Таким образом, они должны иметь отличающиеся имена, иначе вы столкнетесь с непонятным кодом вроде такого:

```
shares = shares;
```

Одним из часто используемых способов, призванных помочь избежать этого, является использование префикса `m_` для идентификации данных-членов:

```
Class Stock
{
private:
      string m_company;
      int m_chares;
      ...
```

Использование конструкторов

Язык C++ предлагает два способа инициализации объектов с помощью конструктора. Первый — вызывать конструктор явно:

```
Stock food = Stock("World Cabbage", 250, 1.25);
```

Это устанавливает значение члена `company` объекта `food` равным строке `"World Cabbage"`, значение `shares` равным `250` и так далее.

Второй способ — вызвать конструктор явно:

```
Stock garment("Furry Mason", 50, 2.5);
```

Вот более компактная форма явного вызова:

```
Stock garment = Stock("Furry Mason", 50, 2.5);
```

С++ обращается к конструктору класса всякий раз, когда вы создаете объект класса, даже когда вы используете операцию new для динамического выделения памяти. Вот как использовать конструктор вместе с new:

```
Stock *pstock = new Stock("Electroshock Games",18,19.0);
```

Оператор, создающий объект Stock, инициализирует его значениями, переданными в аргументах, и присваивает адрес нового объекта указателю pstock. В этом случае объект не имеет имени, но вы можете использовать указатель для управления объектом. Указатели на объекты будут обсуждаться в главе 11.

Конструкторы используются способом, отличным от всех остальных методов класса. Обычно вы пользуетесь объектом для вызова метода:

```
stock1.show(); // объект stock вызывает метод show()
```

Однако вы не можете использовать объект для вызова конструктора, поскольку до тех пор, пока конструктор не завершит создание объекта, его не существует. Вместо того чтобы быть вызванным объектом, конструктор применяется для создания объекта.

Конструкторы по умолчанию

Конструктор по умолчанию — это конструктор, применяемый для создания объекта, когда не предоставляются явные инициализирующие значения. То есть, это конструктор, который используется в объявлении следующим образом:

```
Stock stock1;  // используется конструктор по умолчанию
```

Эй, но ведь в листинге 10.3 уже делалось это! Причина, по которой этот оператор работает, состоит в том, что если вы забудете о написании конструкторов, то С++ автоматически создаст конструктор по умолчанию. Это — неявная версия конструктора по умолчанию, который ничего не делает. Для класса Stock конструктор по умолчанию будет таким:

```
Stock::Stock() { }
```

В результате мы получаем объект stock1 с инициализированными членами, как в операторе

```
int x;
```

где создается x без указания его значения. То, что конструктор по умолчанию не имеет аргументов, отображает факт, что никакие значения не присутствуют в объявлении.

Любопытным моментом, имеющим отношение к конструктору по умолчанию, является то, что компилятор создает его только в том случае, если вы не определите ни одного собственного конструктора. После того, как вы определите хотя бы один конструктор класса, компилятор перестанет создавать конструктор по умолчанию. Если вы предусмотрите конструктор не по умолчанию вроде Stock(const char *co, int n, double pr), и не предложите свою собственную версию конструктора по умолчанию, то объявление наподобие:

```
Stock stock1; // невозможно с существующим конструктором
```

вызовет ошибку. Причина такого поведения в том, что, может быть, вы хотите сделать невозможным создание объектов без их инициализации. Если же, однако, вы предпочитаете создавать объекты без явной инициализации, то должны будете определить свой собственный конструктор. Это конструктор без аргументов. Конструктор по умолчанию можно создать двумя способами. Один — предусмотреть значения по умолчанию для всех аргументов в существующем конструкторе:

```
Stock(const char *co = "Error", int n = 0, double pr = 0.0)
```

Второй — использовать возможности перегрузки функций для определения второго конструктора без аргументов:

```
Stock();
```

Допускается только один конструктор по умолчанию, поэтому убедитесь, что не создали два. (В ранних версиях C++ можно было использовать только второй способ создания конструктора по умолчанию.)

В действительности, обычно вы должны инициализировать объекты для того, чтобы иметь уверенность в том, что при создании объекта все члены получают известные, корректные значения. Таким образом, конструктор по умолчанию обычно обеспечивает явную инициализацию всех переменных-членов. Например, вот как вы можете определить его для класса Stock:

```
Stock:: Stock ()
{
   std::strcpy(company, "без имени");
   shares = 0;
   share_val = 0.0;
   total_val = 0.0;
}
```

Совет

Когда вы проектируете класс, то обычно должны предусмотреть конструктор по умолчанию, который явно инициализирует все переменные-члены класса.

После того, как вы создадите конструктор по умолчанию любым из двух способов (без аргументов или с аргументами по умолчанию), то сможете объявлять переменные-объекты без явной инициализации:

```
Stock first;                // вызывает конструктор по умолчанию неявно
Stock first = Stock();      // вызывает конструктор по умолчанию явно
Stock *prelief = new Stock; // вызывает конструктор по умолчанию явно
```

Однако вас не должна сбить с толку явная форма конструкторов не по умолчанию:

```
Stock first("Concrete Conglomerate"); // вызывает конструктор
Stock second();                       // объявляет функцию
Stock third;                          // вызывает конструктор по умолчанию
```

Первое объявление из приведенных вызывает конструкторов не по умолчанию — то есть такой, который принимает аргументы. Второе объявление устанавливает, что second() — это функция, возвращающая объект Stock. Когда вы явно вызываете конструктор по умолчанию, не указывайте скобки.

Деструкторы

Когда вы используете конструктор для создания объекта, программа отслеживает этот объект до момента его исчезновения. В этот момент программа автоматически вызывает специальную функцию-член с несколько пугающим названием *деструктор*. Деструктор призван очищать всяческий “мусор”, поэтому он служит весьма полезной цели. Например, если ваш конструктор использует операцию new для выделения памяти, то деструктор должен обратиться к delete для ее освобождения. Конструктор нашего класса Stock не делает никаких причуд, вроде вызова new, поэтому деструктору класса Stock делать нечего. В таком случае вы можете просто позволить компилятору сгенерировать неявный деструктор, который ничего не делает, что и имеет место в первой версии класса Stock. С другой стороны, полезно посмотреть, как объявляются и определяются деструкторы, поэтому давайте сделаем это для класса Stock.

Как и конструктор, деструктор имеет специальное имя. Оно формируется из имени класса и предваряющего его символа тильды (~). То есть деструктор для класса Stock называется ~Stock(). Так же как и конструктор, деструктор не имеет возвращаемого значения и не имеет объявляемого типа. В отличие от конструктора, деструктор не может иметь аргументов. Таким образом, прототип деструктора класса Stock выглядит следующим образом:

```
~Stock();
```

Поскольку деструктор Stock не имеет никаких обязанностей, его можно кодировать как функцию, которая ничего не делает.

```
Stock::~Stock()
{
}
```

Однако для того, чтобы увидеть, когда вызывается конструктор, вы можете определить его следующим образом:

```
Stock::~Stock()
{
   cout << "До встречи, " << company << "!\n";
}
```

Когда должен вызываться деструктор? Этим управляет компилятор. Обычно ваш код не должен вызывать деструктор явно. (См. главу 12.) Если вы создаете статический объект класса, то его деструктор вызывается автоматически при завершении работы программы. Если вы создаете автоматический (локальный) объект класса, как в приведенном примере, то его деструктор вызывается автоматически, когда программа выходит из блока кода, в котором определен объект. Если объект создается операцией new, он размещается в свободной памяти, и его деструктор вызывается автоматически, когда вызывается delete для ее освобождения. И, наконец, программа может создавать временные объекты для обслуживания некоторых операций; в этом случае деструктор вызывается тогда, когда программа завершает использование объекта.

Поскольку деструктор вызывается автоматически при уничтожении объекта класса, желательно, чтобы деструктор существовал. Если вы его не предусмотрите, компилятор неявно создаст конструктор по умолчанию и, если обнаружит код, который ведет к уничтожению объекта, то также неявно создаст деструктор.

Усовершенствование класса Stock

Теперь вам понадобится включить конструкторы и деструкторы в объявление класса и в определения методов. На этот раз последуем обычной практике C++ и разместим программу в нескольких файлах. Вы можете поместить объявление класса в заголовочный файл под именем `stock1.h`. (Имя предполагает возможность в будущем иметь разные версии.) Методы класса можно поместить в файл `stock1.cpp`. Как правило, заголовочный файл, содержащий объявление класса, и файл исходного кода, содержащий определения методов, должны иметь одинаковое базовое имя, чтобы было легко отслеживать взаимосвязанные файлы. Применение разных файлов для объявления класса и функций-членов позволяет отделить абстрактное определение интерфейса (объявление класса) от деталей реализации (определения функций-членов). Например, вы можете разместить объявление файла в текстовом заголовочном файле, но распространять определения функций в виде скомпилированного кода. И, наконец, вы размещаете программу, использующую ресурсы класса, в третьем файле, который можно назвать `usestock1.cpp`.

Заголовочный файл

В листинге 10.4 показан заголовочный файл программы `stock`. К исходному объявлению класса здесь добавлены прототипы функций конструктора и деструктора. Также он отличается отсутствием функции `acquire()`, которая более не нужна, поскольку класс имеет конструкторы. В файле используется также технология `#ifndef`, описанная в главе 9, для защиты от многократных включений заголовочного файла.

Листинг 10.4. `stocks.h`

```
//stock1.h -- объявление класса Stock с добавленными конструкторами и
деструкторами
#ifndef STOCK1_H_
#define STOCK1_H_

class Stock
{
private:
   char company[30];
   int shares;
   double share_val;
   double total_val;
   void set_tot() { total_val = shares * share_val; }

public:
   Stock();    // конструктор по умолчанию
   Stock(const char * co, int n = 0, double pr = 0.0);
   ~Stock();  // деструктор
   void buy(int num, double price);
   void sell(int num, double price);
   void update(double price);
   void show();
};
#endif
```

Файл реализации

Листинг 10.5 представляет определение методов для разрабатываемой программы. Он включает stock1.h для того, чтобы предоставить программе объявление класса. (Вспомните, что заключение имени файла в двойные кавычки вместо угловых скобок заставляет компилятор искать его там же, где расположен исходный файл.) Кроме того, листинг 10.5 включает в себя заголовочный файл iostream для обеспечения поддержки ввода-вывода и заголовочный файл cstring для поддержки strcpy() и strncpy(). В листинге также продемонстрировано использование объявлений и квалифицированных имен (вроде std::string) для обеспечения доступа к различным объявлениям из заголовочных файлов. В этом файле к прежним методам добавлены определения конструктора и деструктора. Чтобы помочь вам увидеть момент вызова метода, каждый из них выдает сообщение. Это не является обычным свойством конструкторов и деструкторов, однако позволяет сделать наглядным их использование классом.

Листинг 10.5. stock1.cpp

```
//stock1.cpp -- реализация класса Stock с добавленными конструкторами и
деструкторами
#include <iostream>
#include "stock1.h"
// конструкторы ("говорливая" версия)
Stock::Stock() // конструктор по умолчанию
{
   std::cout << "Вызван конструктор по умолчанию\n";
   std::strcpy(company, "без имени");
   shares = 0;
   share_val = 0.0;
   total_val = 0.0;
}
Stock::Stock(const char * co, int n, double pr)
{
   std::cout << "Вызван конструктор для " << co << "\n";
   std::strncpy(company, co, 29);
   company[29] = '\0';
   if (n < 0)
   {
     std::cerr << "Количество пакетов не может быть отрицательным; для "
                << company << " установлено в 0.\n";
     shares = 0;
   }
   else
     shares = n;
   share_val = pr;
   set_tot();
}

// деструктор класса
Stock::~Stock() // "говорливый" деструктор класса
{
   std::cout << "До встречи, " << company << "!\n";
}
```

```
// другие методы
void Stock::buy(int num, double price)
{
   if (num < 0)
   {
     std::cerr << "Количество приобретаемых пакетов не может быть отрицательным. "
               << "Транзакция прервана.\n";
   }
   else
   {
     shares += num;
     share_val = price;
     set_tot();
   }
}

void Stock::sell(int num, double price)
{
   using std::cerr;
   if (num < 0)
   {
     cerr << "Количество проданных пакетов не может быть отрицательным. "
          << "Транзакция прервана.\n";
   }
   else if (num > shares)
   {
     cerr << "Вы не можете продать больше того, чем владеете! "
          << "Транзакция прервана.\n";
   }
   else
   {
     shares -= num;
     share_val = price;
     set_tot();
   }
}

void Stock::update(double price)
{
   share_val = price;
   set_tot();
}

void Stock::show()
{
   using std::cout;
   using std::endl;
   cout << "Компания: " << company
        << " Пакетов: " << shares << endl
        << " Цена пакета: $" << share_val
        << " Общая стоимость: $" << total_val << endl;
}
```

Клиентский файл

В листинге 10.6 представлен сокращенный код для тестирования методов в разрабатываемой программе. Из-за того, что он просто использует класс `Stock`, этот листинг является клиентом класса `Stock`. Как и `stock1.cpp`, он включает файл `stock1.h` для предоставления объявления класса. Эта программа демонстрирует конструкторы и деструкторы. Она также использует те же команды форматирования, что и листинг 10.3. Для компиляции полной программы используйте технику многофайловых программ, описанную в главах 1 и 8.

Листинг 10.6. `usestock1.cpp`

```
//usestock1.cpp -- использование класса Stock
#include <iostream>
#include "stock1.h"
int main()
{
   using std::cout;
   using std::ios_base;
   cout.precision(2);                              // формат #.##
   cout.setf(ios_base::fixed, ios_base::floatfield);// формат #.##
   cout.setf(ios_base::showpoint);                 // формат #.##
   cout << "Используются конструкторы для создания новых объектов\n";
   Stock stock1("NanoSmart", 12, 20.0);            // синтаксис номер 1
   stock1.show();
   Stock stock2 = Stock ("Boffo Objects", 2, 2.0); // синтаксис номер 2
   stock2.show();
   cout << "Присвоение stock1 stock2:\n";
   stock2 = stock1;
   cout << "Листинг stock1 и stock2:\n";
   stock1.show();
   stock2.show();
   cout << "Применение конструктора для сброса объекта\n";
   stock1 = Stock("Nifty Foods", 10, 50.0);        // временный объект
   cout << "Измененный stock1:\n";
   stock1.show();
   cout << "Готово\n";
   return 0;
}
```

Замечание по совместимости

Вы можете использовать старый вариант `ios::` вместо `ios_base::`.

При компиляции исходных текстов, представленных на листингах 10.4, 10.5 и 10.6 генерируется исполнимая программа. Так выглядит выходной поток работающей скомпилированной программы:

```
Используются конструкторы для создания новых объектов
Вызван конструктор для NanoSmart
Компания: NanoSmart Пакетов: 12
Цена пакета: $20.00 Общая стоимость: $240.00
Вызван конструктор для Boffo Objects
Компания: Boffo Objects Пакетов: 2
```

```
Цена пакета: $2.00 Общая стоимость: $4.00
Присвоение stock1 stock2:
Листинг stock1 и stock2:
Компания: NanoSmart Пакетов: 12
Цена пакета: $20.00 Общая стоимость: $240.00
Компания: NanoSmart Пакетов: 12
Цена пакета: $20.00 Общая стоимость: $240.00
Применение конструктора для сброса объекта
Вызван конструктор для Nifty Foods
До встречи, Nifty Foods!
Измененный stock1:
Компания: Nifty Foods Пакетов: 10
Цена пакета: $50.00 Общая стоимость: $500.00
Готово
До встречи, NanoSmart!
До встречи, Nifty Foods!
```

Некоторые компиляторы могут генерировать программу, которая выдает вывод с дополнительной строкой:

```
Используются конструкторы для создания новых объектов
Вызван конструктор для NanoSmart
Компания: NanoSmart Пакетов: 12
Цена пакета: $20.00 Общая стоимость: $240.00
Вызван конструктор для Boffo Objects
До встречи, Boffo Objects!                <- дополнительная строка
Компания: Boffo Objects Пакетов: 2
Цена пакета: $2.00 Общая стоимость: $4.00
...
```

В приведенном ниже разделе поясняется эта дополнительная строка в выходном потоке программы.

Замечания по программе

В листинге 10.6 оператор

```
Stock stock1("NanoSmart", 12, 20.0);
```

создает объект Stock по имени stock1 и инициализирует его данные-члены указанными значениями:

```
Вызван конструктор для NanoSmart
Компания: NanoSmart Пакетов: 12
```

Оператор

```
Stock stock2 = Stock ("Boffo Objects", 2, 2.0);
```

использует другой синтаксис для создания и инициализации объекта stock2. Стандарт C++ разрешает компилятору выполнять второй синтаксис различными способами. Один из них приводит к такому же поведению, как в случае первого синтаксиса:

```
Вызван конструктор для Boffo Objects
Компания: Boffo Objects Пакетов: 2
```

Второй способ реализации этого синтаксиса вызывает конструктор для создания временного объекта, который затем копируется в stock2. После этого временный объект уничтожается. Если компилятор использует эту опцию, то для временного объекта вызывается деструктор, что приводит к следующему выводу:

```
Вызван конструктор для Boffo Objects
До встречи, Boffo Objects!
Компания: Boffo Objects Пакетов: 2
```

Компилятор, который генерирует такой вывод, уничтожает временный объект немедленно, но может быть, что он и подождет некоторое время — в этом случае сообщение деструктора появится позже.

Оператор

```
stock2 = stock1; // присваивание объекта
```

иллюстрирует возможность присвоения значения одного объекта другому, того же типа. Как и в случае присвоения структур, присваивание объектов класса по умолчанию копирует члены одного объекта в другой. В этом случае исходное содержимое stock2 перезаписывается.

На память!

Когда вы присваиваете один объект другому, того же класса, то по умолчанию C++ копирует содержимое каждого члена данных исходного объекта в соответствующий член данных другого объекта.

Вы можете использовать конструктор более чем только для инициализации нового объекта. Например, наша программа содержит такой оператор в функции main():

```
stock1 = Stock("Nifty Foods", 10, 50.0);
```

Объект stock1 уже существует. Поэтому вместо инициализации stock1 данный оператор присваивает ему новые значения. Это делается за счет создания конструктором нового временного объекта и последующего копирования его содержимого в stock1. Затем программа уничтожает временный объект, вызывая его деструктор, что и иллюстрирует следующий аннотированный вывод:

```
Применение конструктора для сброса объекта
Вызван конструктор для Nifty Foods      ← Создан временный объект
До встречи, Nifty Foods!                ← Временный объект уничтожен
Измененный stock1:
Компания: Nifty Foods Пакетов: 10       ← Данные скопированы в stock1
Цена пакета: $50.00 Общая сумма: $500.00
```

Некоторые компиляторы могут удалять временный объект позже, откладывая вызов деструктора.

В конце работы программа выдает:

```
Готово
До встречи, NanoSmart!
До встречи, Nifty Foods!
```

Когда функция main() завершает работу, ее локальные переменные (stock1 и stock2) выходят из области существования. Поскольку такие автоматические пере-

менные размещаются в стеке, последний созданный объект удаляется первым, а первый созданный — последним. (Вспомним, что "NanioSmart" было изначально в stock1, но позже было перенесено в stock2, а stock1 был сброшен в "Nifty Foods".)

Вывод программы демонстрирует принципиальную разницу между следующими двумя операторами:

```
Stock stock2 = Stock ("Boffo Objects", 2, 2.0);
stock1 = Stock("Nifty Foods", 10, 50.0); // временный объект
```

Первый из этих операторов вызывает инициализацию; создается объект с указанным значением, который может создавать, а может и не создавать временный объект. Второй оператор вызывает присваивание. Применение конструктора в операции присваивания в таком виде всегда служит причиной создания временного объекта перед выполнением собственно присваивания.

Совет

Если вы можете устанавливать значения объекта как инициализацией, так и присваиванием, отдавайте предпочтение инициализации. Обычно это более эффективно.

Функции-члены const

Рассмотрим следующий фрагмент кода:

```
const Stock land = Stock("Kludgehorn Properties");
land.show();
```

В современном языке C++ компилятор должен обратить внимание на вторую строку. Почему? Потому, что код show() не гарантирует того, что он не будет модифицировать объект, который, поскольку объявлен как const, не должен быть изменен. Вы должны предварительно позаботиться о решении этой проблемы, объявляя аргумент функции как const-ссылку или указатель на const. Однако здесь существует синтаксическая проблема. Метод show() не имеет аргументов. Вместо этого объект, который он использует, представлен неявно вызовом этого метода. Что вам надо — это новый синтаксис, который укажет на то, что функция-член не будет модифицировать объект. Решение, предлагаемое C++, заключается в том, чтобы поместить ключевое слово const после скобок функции. То есть объявление метода show() должно выглядеть так:

```
void show() const;           // обещает не изменять объект
```

Подобным образом начало определения функции должно выглядеть следующим образом:

```
void Stock::show() const     // обещает не изменять объект
```

Функция класса, объявленная и определенная подобным образом, называется константной функцией-членом. Точно так же, как вы используете константные ссылки и указатели в качестве формальных аргументов функций, где это необходимо, вы должны делать методы класса константными всегда, когда они не модифицируют объект, с которым работают. Отныне мы будем следовать этому правилу.

Обзор конструкторов и деструкторов

Теперь, когда мы ознакомились с рядом примеров конструкторов и деструкторов, сделаем паузу и подведем некоторые итоги.

Конструктор — это специальная функция-член класса, которая вызывается всякий раз при создании объекта данного класса. Конструктор класса имеет то же имя, что и класс, но благодаря возможностям перегрузки функций, существует возможность иметь более одного конструктора с одним и тем же именем и разным набором аргументов. Кроме того, конструктор не имеет объявленного типа. Обычно конструктор используется для инициализации членов объекта класса. Ваша инициализация должна соответствовать списку аргументов конструктора. Например, предположим, что класс Bozo имеет следующий прототип конструктора класса:

```
Bozo(const char * fname, const char * lname);   // прототип конструктора
```

В этом случае вы должны использовать его для инициализации объекта следующим образом:

```
Bozo bozetta = Bozo("Bozetta","Biggens");       // основная форма
Bozo fufu("Fufu","O'Dweeb");                    // сокращенная форма
Bozo *pc = new Bozo("Popo","Le Peu");           // динамический объект
```

Когда конструктор имеет только один аргумент, он вызывается, если вы инициализируете объект значением, которое имеет тот же тип, что и аргумент конструктора. Например, предположим, что существует следующий прототип конструктора:

```
Bozo(int age);
```

Затем вы можете использовать любую из следующих форм инициализации объекта:

```
Bozo dribble = Bozo(44);      // основная форма
Bozo roon(66);                // вторичная форма
Bozo tubby = 32;  // специальная форма для конструктора с одним аргументом
```

Фактически третий пример является новым, и это удобный момент, чтобы сказать о нем. В главе 11 упомянут способ отключения этого средства.

На память!

Конструктор, который принимает один аргумент, позволяет применить синтаксис присваивания для инициализации объекта значением:

```
имяКласса объект = значение;
```

Конструктор по умолчанию не имеет аргументов и используется, когда вы создаете объект без явной его инициализации. Если вы не предоставляете ни одного конструктора, то компилятор создаст конструктор по умолчанию. В противном случае вы обязаны определить свой собственный конструктор по умолчанию. Он может не иметь аргументов либо иметь значения по умолчанию для всех аргументов:

```
Bozo();                               // прототип конструктора по умолчанию
Bistro(const char *s = "Chez Zero");//правила умолчания для класса Bistro
```

Программа использует конструкторы по умолчанию для инициализации объектов:

```
Bozo bibi;             // используется конструктор по умолчанию
Bozo *pb = new Bozo;   // используется конструктор по умолчанию
```

Точно так же, как программа вызывает конструктор при создании объекта, она вызывает деструктор при его уничтожении. Вы можете иметь только один деструктор для класса. Он не имеет возвращаемого типа (даже void), не имеет аргументов, и его имя состоит из имени класса с предшествующей тильдой (~). Например, деструктор класса Bozo имеет следующий прототип:

```
~Bozo();  // деструктор класса
```

Деструктор класса, вызываемый операцией delete, становится необходимым, когда конструктор вызывается операцией new.

Изучение объектов: указатель this

Вы можете делать кое-что еще с классом Stock. До сих пор каждая функция-член класса имела дело только с одним объектом: тем, который ее вызывал. Однако иногда методу может понадобиться иметь дело с двумя объектами и для этого обращаться к любопытному указателю по имени this. Давайте посмотрим, когда может понадобиться this.

Несмотря на то что объявление класса Stock включает в себя отображение данных, все же ему недостает аналитических возможностей. Например, если взглянете на вывод функции show(), то вы сможете сказать, какой из ваших холдингов обладает наибольшим пакетом акций, но программа не сможет дать ответ на этот вопрос, поскольку не имеет прямого доступа к total_val. Самый простой способ сообщить программе о хранимых данных — это предусмотреть методы, возвращающие эти данные. Обычно вы используете для этого встроенный код, как в следующем примере:

```
class Stock
{
   private:
      ...
      double total_val;
      ...
   public:
      double total() const { return total_val; }
      ...
};
```

Это определение делает total_val доступным программе только для чтения. То есть, вы можете использовать метод total() для получения этого значения, но класс не предоставляет метода для его переустановки. (Другие методы, такие как buy(), sell() и update(), модифицируют total_val в качестве побочного эффекта от переустановки значений членов shares и share_val.)

Добавляя эту функцию к объявлению класса, вы можете позволить программе исследовать серии пакетов акций для поиска наиболее крупного из них. Однако вы можете воспользоваться другим подходом, который поможет разобраться с указателем this. Подход заключается в том, чтобы определить функцию-член, которая будет просматривать два объекта Stock и возвращать ссылку на больший из них. Попытка

реализовать эту идею вызывает некоторые интересные вопросы, которые мы сейчас рассмотрим.

Во-первых, как написать функцию, работающую с двумя объектами для сравнения? Предположим, например, что вы решили назвать ее `topval()`. Затем вызов `stock1.topval()` обращается к данным объекта `stock1`, в то время как `stock2.topval()` — к данным объекта `stock2`. Если вы хотите, чтобы метод сравнивал два объекта, то должны передать второй объект в виде аргумента. Для эффективности можете передавать его по ссылке. То есть, вы можете создать метод `topval()` так, чтобы он принимал аргумент типа `const Stock &`.

Во-вторых, как ответ метода будет передаваться в вызывающую его программу? Самый прямой путь — заставить метод возвращать ссылку на объект, который имеет большее значение `total_val`. Таким образом, метод сравнения двух объектов будет иметь следующий прототип:

```
const Stock & topval(const Stock & s) const;
```

Эта функция имеет неявный доступ к одному объекту и возвращает ссылку на один из двух объектов. Слово `const` в скобках устанавливает для функции запрет модифицировать объект, на который дается явная ссылка, а слово `const`, которое следует за скобками, устанавливает, что функция не может модифицировать объект, на который ссылается неявно. Поскольку функция возвращает ссылку на один из `const`-объектов, тип ее возврата также является константной ссылкой.

Предположим, что вы хотите сравнить два объекта `Stock` — `stock1` и `stock2` — и присвоить тот из них, что имеет большее значение, объекту `top`. Для этого вы можете использовать любой из следующих двух операторов:

```
top = stock1.topval(stock2);
top = stock2.topval(stock1);
```

Первая форма обращается к `stock1` неявно, а к `stock2` — неявно, в то время как вторая — наоборот. (См. рис. 10.3.) В любом случае метод сравнивает два объекта и возвращает ссылку на тот, который имеет большее значение `total_val`.

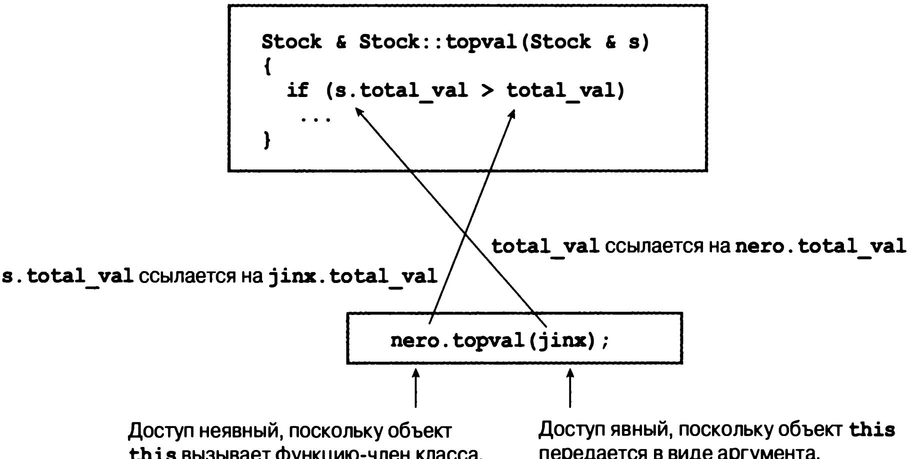


Рис. 10.3. Доступ к двум объектам из функции-члена

На самом деле такая нотация немного приводит в замешательство. Было бы понятнее, если бы можно было каким-то образом использовать операцию > для сравнения двух объектов. Вы можете сделать это с помощью перегрузки операций, которое обсуждается в главе 11.

Между тем, пока рассмотрим реализацию topval(). Она порождает небольшую проблему. Вот часть реализации, иллюстрирующая проблему:

```
const Stock & Stock::topval(const Stock & s) const
{
   if (s.total_val > total_val)
      return s; // объект-аргумент
   else
      return ?????; // вызывающий объект
}
```

Здесь s.total_val — это суммарное значение объекта, переданное в виде аргумента, а total_val — суммарное значение объекта, которому сообщение передается. Если s.total_val больше total_val, то функция возвращает s. В противном случае она возвратит объект, использованный для вызова метода. (В терминологии ООП — это объект, которому передано сообщение topval.) Существует одна проблема: чем вызывается объект? Если вызывается stock1.topval(stock2), то s — это ссылка на stock2 (то есть псевдоним для stock2), но не существует псевдонима для stock1.

Решение этой проблемы, которое предлагает C++, заключается в применении специального указателя this. Он указывает на объект, который использован для вызова функции-члена. (Обычно this передается методу в виде скрытого аргумента.) Таким образом, вызов stock1.topval(stock2) устанавливает значение this равным адресу объекта stock1 и делает его доступным методу topval(). Аналогичным образом, вызов функции stock2.topval(stock1) устанавливает значение this равным адресу объекта stock2. Вообще все методы класса получают указатель this, равный адресу объекта, который вызвал метод. Фактически total_val внутри total() является сокращенной нотацией this->total_val. (Вспомним из главы 4, что операция -> используется для доступа к членам структуры через указатель на нее. То же верно и для членов класса.) (См. рис. 10.4.)

Указатель this

Каждая функция-член, включая конструкторы и деструкторы, имеет указатель this. Специальным свойством this является то, что он указывает на вызывающий объект. Если метод нуждается в получении ссылки на вызвавший объект в целом, он может использовать выражение *this. Применение квалификатора const после скобок с аргументами заставляет трактовать this как константу; в этом случае вы не можете использовать this для изменения значений объекта.

То, что вам нужно вернуть из метода, однако, не является this, поскольку this — это адрес объекта. Вам нужно вернуть сам объект, а это символизирует выражение *this. (Вспомните, что применение операции разыменования * к указателю дает значение, на которое он указывает.) Теперь вы можете завершить определение метода, используя *this в качестве псевдонима вызвавшего объекта:

```
const Stock & Stock::topval(const Stock & s) const
{
   if (s.total_val > total_val)
      return s;     // объект-аргумент
   else
      return *this; // вызывающий объект
}
```

Тот факт, что возвращаемое значение является ссылкой, означает, что возвращаемый объект — это тот самый объект, что вызвал данный метод, а не копия, переданная механизмом возврата. В листинге 10.7 представлен новый заголовочный файл.

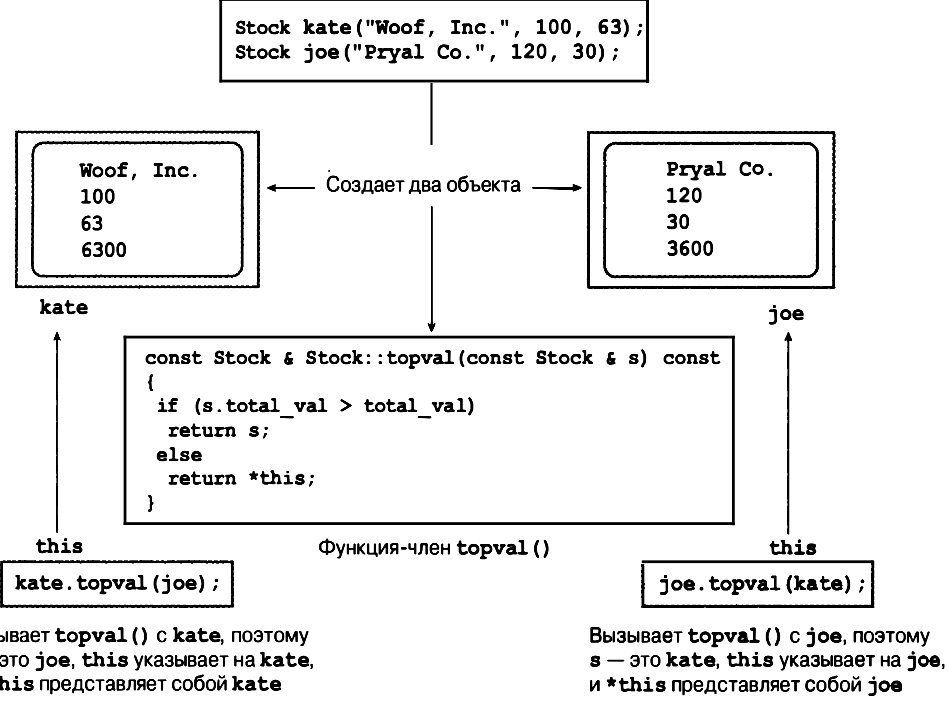


Рис. 10.4. **this** *указывает на вызвавший объект*

Листинг 10.7. stock2.h

```
//stock2.h -- следующая версия
#ifndef STOCK2_H_
#define STOCK2_H_
class Stock
{
private:
   char company[30];
   int shares;
   double share_val;
   double total_val;
   void set_tot() { total_val = shares * share_val; }
```

```
public:
   Stock(); // конструктор по умолчанию
   Stock(const char * co, int n = 0, double pr = 0.0);
   ~Stock(); // деструктор
   void buy(int num, double price);
   void sell(int num, double price);
   void update(double price);
   void show();
   const Stock & topval(const Stock & s) const;
};
#endif
```

В листинге 10.8 показан измененный файл с методами класса. Он включает в себя новый метод `topval()`. Также теперь, когда вы ознакомились с работой конструкторов и деструкторов, в листинге 10.8 они заменены "молчаливыми" версиями.

Листинг 10.8. stocks2.cpp

```
//stock2.cpp -- усовершенствованная версия
#include <iostream>
#include "stock2.h"
// конструкторы
Stock::Stock() // конструктор по умолчанию
{
   std::strcpy(company, "без имени");
   shares = 0;
   share_val = 0.0;
   total_val = 0.0;
}
Stock::Stock(const char * co, int n, double pr)
{
   std::strncpy(company, co, 29);
   company[29] = '\0';
   if (n < 0)
   {
     std::cerr << "Количество пакетов не может быть отрицательным; для "
               << company << " установлено в 0.\n";
     shares = 0;
   }
   else
     shares = n;
   share_val = pr;
   set_tot();
}
// деструктор класса
Stock::~Stock() // молчаливый деструктор класса
{
}
// другие методы
void Stock::buy(int num, double price)
{
   if (num < 0)
   {
```

```
     std::cerr << "Количество приобретаемых пакетов не может быть отрицательным. "
               << "Транзакция прервана.\n";
   }
   else
   {
     shares += num;
     share_val = price;
     set_tot();
   }
}

void Stock::sell(int num, double price)
{
   using std::cerr;
   if (num < 0)
   {
     cerr << "Количество проданных пакетов не может быть отрицательным. "
          << "Транзакция прервана.\n";
   }
   else if (num > shares)
   {
     cerr << "Вы не можете продать больше того, чем владеете! "
          << "Транзакция прервана.\n";
   }
   else
   {
     shares -= num;
     share_val = price;
     set_tot();
   }
}

void Stock::update(double price)
{
   share_val = price;
   set_tot();
}

void Stock::show()
{
   using std::cout;
   using std::endl;
   cout << "Компания: " << company
        << " Пакетов: " << shares << endl
        << " Цена пакета: $" << share_val
        << " Общая стоимость: $" << total_val << endl;
}

const Stock & Stock::topval(const Stock & s) const
{
   if (s.total_val > total_val)
     return s;
   else
     return *this;
}
```

Конечно, вы захотите проверить, как работает указатель this, и лучший способ этого — использовать новый метод в программе с массивом объектов, что и делается в следующем разделе.

Массив объектов

Часто, как и в примерах со Stock, вы захотите создать несколько объектов одного класса. Вы можете создать отдельные объектные переменные, как это делалось до сих пор в примерах настоящей главы, но, возможно, больше смысла будет в создании массива объектов. Это может выглядеть подобно прыжку в неизвестность, но фактически вы объявляете массив объектов таким же способом, как массивы любых стандартных типов:

```
Stock mystuff[4];          // создание массива из 4 объектов Stock
```

Вспомним, что программа всегда вызывает конструкторы по умолчанию, когда создает объекты класса без явной инициализации. Такое объявление требует либо того, чтобы у класса вообще не было явно определенных конструкторов — при этом используются неявные, ничего не делающие конструкторы, либо, как и в представленном случае — чтобы был явно определен конструктор по умолчанию. Каждый элемент — mystuff[0], mystuff[1] и так далее — является объектом класса Stock, а потому может быть использован с методами Stock:

```
mystuff[0].update();       // применяет update() к первому элементу
mystuff[3].show();         // применяет show() к 4-му элементу
Stock tops = mystuff[2].topval(mystuff[1]);//сравнивает 2-й и 3-й элементы
```

Вы можете использовать конструктор для инициализации элементов массива. В этом случае вы можете вызвать конструктор для каждого индивидуального элемента:

```
const int STKS = 4;
Stock stocks[STKS] = {
   Stock("NanoSmart", 12.5, 20),
   Stock("Boffo Objects", 200, 2.0),
   Stock("Monolithic Obelisks", 130, 3.25),
   Stock("Fleep Enterprises", 60, 6.5)
};
```

Этот код использует стандартную форму инициализации массива: разделенный запятой список значений, заключенный в фигурные скобки. В этом случае вызов метода-конструктора представляет каждое значение. Если класс имеет более одного конструктора, вы можете использовать разные конструкторы для разных элементов:

```
const int STKS = 10;
Stock stocks[STKS] = {
   Stock("NanoSmart", 12.5, 20),
   Stock(),
   Stock("Monolithic Obelisks", 130, 3.25),
};
```

Это инициализирует stocks[0] и stocks[2] с помощью конструктора Stock(const char * co, int n, double pr) и stock[1] — с помощью конструк-

тора Stock(). Поскольку такое объявление может инициализировать массив только частично, оставшиеся семь членов инициируются конструктором по умолчанию.

Схема инициализации массива объектов обычно изначально использует конструкторы по умолчанию для создания массива элементов. Затем конструкторы в фигурных скобках создают временные объекты, чье содержимое копируется в массив. То есть класс должен иметь конструктор по умолчанию, если вы хотите создавать массивы объектов класса.

Внимание!

Если вы хотите создать массив объектов какого-то класса, то этот класс должен иметь конструктор по умолчанию.

В листинге 10.9 этот принцип применен в короткой программе, которая инициализирует четыре элемента массива, отображает их содержимое и тестирует элементы для поиска того, который имеет наибольшее значение суммы.

Поскольку total() сравнивает только два объекта за раз, программа использует цикл for для проверки всего массива. Код в этом листинге использует заголовочный файл из листинга 10.7 и файл методов из листинга 10.8.

Листинг 10.9. usestock2.cpp

```
// usestok2.cpp -- использование класса Stock
// компилировать вместе с stock2.cpp
#include <iostream>
#include "stock2.h"

const int STKS = 4;
int main()
{
   using std::cout;
   using std::ios_base;
   // создание массива инициализированных объектов
   Stock stocks[STKS] = {
     Stock("NanoSmart", 12, 20.0),
     Stock("Boffo Objects", 200, 2.0),
     Stock("Monolithic Obelisks", 130, 3.25),
     Stock("Fleep Enterprises", 60, 6.5)
   };

   cout.precision(2);                                    // формат #.##
   cout.setf(ios_base::fixed, ios_base::floatfield);     // формат #.##
   cout.setf(ios_base::showpoint);                       // формат #.##
   cout << "Пакеты акций:\n";
   int st;
   for (st = 0; st < STKS; st++)
      stocks[st].show();
   Stock top = stocks[0];
   for (st = 1; st < STKS; st++)
      top = top.topval(stocks[st]);
   cout << "\nСамый большой пакет:\n";
   top.show();
   return 0;
}
```

Замечание по совместимости

Вы можете использовать старый вариант `ios::` вместо `ios_base::`.

Ниже приведен вывод программы из листинга 10.9:

```
Пакеты акций:
Компания: NanoSmart Пакетов: 12
Цена пакета: $20.00 Общая стоимость: $240.00
Компания: Boffo Objects Пакетов: 200
Цена пакета: $2.00 Общая стоимость: $400.00
Компания: Monolithic Obelisks Пакетов: 130
Цена пакета: $3.25 Общая стоимость: $422.50
Компания: Fleep Enterprises Пакетов: 60
Цена пакета: $6.50 Общая стоимость: $390.00
Самый большой пакет:
Компания: Monolithic Obelisks Shares: 130
Цена пакета: $3.25 Общая стоимость: $422.50
```

Следует отметить одну вещь относительно листинга 10.9 — большая часть работы приходится на дизайн класса. Когда он завершен, написание программы становится достаточно простым.

Между прочим, знание об указателе `this` позволяет заглянуть за кулисы C++. Например, утилита `cfront` преобразует программы на C++ в программы на C. Чтобы управлять определением методов, единственное, что нужно при этом сделать — это преобразовать метод C++ вроде

```
void Stock::show() const
{
   cout << "Компания: " << company
        << " Пакетов: " << shares << '\n'
        << " Цена пакета: $" << share_val
        << " Общая стоимость: $" << total_val << '\n';
}
```

в следующий код на C:

```
void show(const Stock * this)
{
   cout << "Компания: " << this->company
        << " Пакетов: " << this->shares << '\n'
        << " Цена пакета: $" << this->share_val
        << " Общая стоимость: $" << this->total_val << '\n';
}
```

То есть при этом квалификатор `Stock::` преобразуется в аргумент функции, указывающий на `Stock` и затем использующий этот указатель для доступа к членам класса.

Аналогичным образом преобразуются вызовы функций вроде

```
top.show();
```

в следующие:

```
show(&top);
```

В той же манере указатель `this` устанавливается в адрес вызывающего объекта (реальные детали этого процесса могут быть более сложными).

Возврат к интерфейсу и реализации

Используя массив символов из 30 элементов, класс Stock ограничивает длину имен компаний 29-ю символами. Например, в результате выполнения оператора

```
Stock firm ("Dunkelmeister, Dostoyevsky, and Delderfield Construction",
         8, 2.5);
```

В объекте firm будет сохранена усеченная строка "Dunkelmeister, Dostoyevsky, a".

Вы можете исключить ограничение длины, изменив реализацию класса. Один подход — увеличить размер массива, но это приведет к расходу пространства памяти. Другой подход состоит в использовании объекта string вместо массива символов, полагаясь на способность объекта string автоматически изменять свой размер. Это повлечет за собой три изменения.

Первое — добавить поддержку класса string, включив заголовочный файл поддержки string в stock2.h. Поскольку stock2.cpp включает stock2.h, поддержка класса string для stock2.cpp будет также обеспечена.

Второе изменение заключается в изменении раздела private определения класса stock2.h. Вы замените

```
class Stock
{
   private:
      char company[30];
      ...
```

следующим кодом:

```
class Stock
{
   private:
      std::string company;
      ...
```

Третьим изменением будет модификация определения конструктора в stock2.cpp. Вы измените

```
Stock::Stock(const char * co, int n, double pr)
{
   std::strncpy(company, co, 29);
   company[29] = '\0';
   ...
```

на следующее:

```
Stock::Stock(const char * co, int n, double pr)
{
   company = co; // assign C-style string to string object
   ...
```

Эти изменения являются примерами изменения реализации. Вы вносите изменения в раздел private объявления класса, а также меняете файл реализации. Однако вам не нужно изменять раздел public класса. Кто-то, кто использует ваш класс, имеет тот же список методов, из которого он может выбирать. То есть интерфейс класса не меняется. Теперь класс Stock может сохранить строку "Dunkelmeister,

Dostoyevsky, and Delderfield Construction", но внешний код останется неизменным:

```
Stock firm ("Dunkelmeister, Dostoyevsky, and Delderfield Construction",
           8, 2.5);
```

Вы можете также добавить новый конструктор:

```
Stock(const std::string & co, int n, double pr);
```

Это будет изменением интерфейса. Список доступных методов немного изменяется, и программист, использующий класс, получает некоторые новые возможности. Например, в программе может использоваться следующий код:

```
string business;
getline(cout, business);
Stock mine(business, 10, 120.5); // использовать новый конструктор
```

Короче говоря, изменения в разделе private класса и в файле реализации являются изменениями реализации, изменения в разделе public — изменениями интерфейса. Изменения реализации затрагивают внутреннюю работу класса, а изменения интерфейса изменяют набор возможностей для того, кто будет использовать ваш класс.

Возможно, вы удивитесь применению объекта класса в качестве члена другого класса. Это нормально — использовать объекты классов в качестве членов класса; в конце концов, одной из целей классов является сделать применение пользовательских типов настолько же простым, как и встроенных. Конструктор класса автоматически обнаруживает наличие членов-объектов и вызывает для них соответствующие конструкторы. Это — средство языка, называемое *списком инициализации членов*, которое может поднять эффективность дизайна конструкторов. Это средство будет рассматриваться в главе 14.

Область видимости класса

В главе 9 обсуждаются глобальные области видимости (уровня файла) и локальные области видимости (уровня блока). Вспомним, что вы можете использовать переменные из глобальной области в любом месте файла, который содержит их определение, в то время как переменные, имеющие локальную область видимости, являются локальными по отношению к блоку, содержащему их определение. Имена функций также могут иметь глобальную область видимости, но никогда — локальную. Классы C++ представляют новый тип области видимости — область видимости класса.

Область видимости класса применима к именам, определенным в классе, таким как имена переменных-членов и функций-членов класса. Сущности, имеющие область видимости класса, известны в пределах класса, но не известны вне его. То есть, вы можете без конфликтов использовать одинаковые имена членов класса в разных классах. Например, член shares класса Stock отличается от члена shares класса JobRide. Кроме того, область видимости класса означает, что вы не можете непосредственно обращаться к членам класса из внешнего мира. Это правило действует даже для общедоступных функций-членов. То есть для того, чтобы вызывать общедоступные функции-члены, вы должны использовать объект:

```
Stock sleeper("Exclusive Ore", 100, 0.25); // создать объект
sleeper.show();  // использовать объект для вызова функции-члена
show();          // неверно -- нельзя вызывать метод напрямую
```

Аналогичным образом вы должны использовать операцию разрешения контекста при определении функции-члена:

```
void Stock::update(double price)
{
   ...
}
```

Короче говоря, в пределах объявления класса или определения функции-члена вы можете использовать неквалифицированные (короткие) имена членов, как в случае, когда `sell()` вызывает функцию-член `set_tot()`.

Имя конструктора распознается при вызове потому, что оно совпадает с именем класса. В противном случае вы должны применять операцию членства (`.`), указателя на членство (`->`) либо разрешения контекста (`::`), в зависимости от контекста, когда вы используете имя члена класса. Следующий фрагмент кода иллюстрирует, как могут быть доступны идентификаторы в пределах области видимости класса:

```
class Ik
{
private:
   int fuss; // fuss имеет область видимости класса
public:
   Ik(int f = 9) {fuss = f; } // fuss в области видимости
   void ViewIk() const;      // ViewIk имеет область видимости класса
};
void Ik::ViewIk() const //Ik:: помещает ViewIk в область видимости
{
   cout << fuss << endl;//fuss в области видимости вместе с методом класса
}
...
int main()
{
   Ik * pik = new Ik;
   Ik ee = Ik(8);//конструктор в области видимости, поскольку носит имя класса
   ee.ViewIk();   // объект класса переносит ViewIk в область видимости
   pik->ViewIk(); // указатель на Ik переносит ViewIk в область видимости
   ...
```

Константы области видимости класса

Иногда хорошо бы иметь символические константы в области видимости класса. Например, объявление класса `Stock` использует литерал `30` для указания размера символьного массива `company`. Также, поскольку эта константа общая для всех объектов этого класса, было бы неплохо создать единственный экземпляр этой константы, доступ к которой разделять все объекты. Вы можете подумать, что возможно такое решение:

```
class Stock
{
private:
   const int Len = 30; // объявление константы? НЕ УДАСТСЯ
   char company[Len];
   ...
```

Но это не работает, потому что объявление класса описывает, как выглядит объект, но не создает объекта. Посему, до тех пор, пока вы не создадите объект, хранить это значение негде. Однако существует несколько других способов достичь желаемой цели.

Первый: вы можете объявить перечислимый тип внутри класса. Перечисление, заданное в объявлении класса, имеет область видимости класса, поэтому вы можете использовать его в пределах класса в качестве символического имени целочисленной константы. То есть вы можете начать объявление класса Stock следующим образом:

```
class Stock
{
private:
   enum {Len = 30}; // специфическая для класса константа
   char company[Len];
   ...
```

Обратите внимание, что такое объявление перечисления не создает переменную-член класса. То есть каждый индивидуальный объект не содержит его в себе. Вместо этого Len становится просто символическим именем, которое компилятор заменяет числом 30, когда встречает его в исходном тексте области видимости класса.

Поскольку здесь перечисление используется просто для определения константы, без намерения создавать переменные типа перечисления, то нет необходимости указывать его дескриптор. Между прочим, во многих реализациях класс ios_base объявляет нечто подобное в своем разделе public; так объявлены идентификаторы вроде ios_base::fixed. Здесь fixed — обычно перечисление, определенное в классе ios_base.

Сравнительно недавно C++ предложил второй способ задания константы в классе — с помощью ключевого слова static:

```
class Stock
{
private:
   static const int Len = 30; // объявление константы! РАБОТАЕТ
   char company[Len];
   ...
```

Это создает одиночную константу по имени Len, которая хранится вместе с остальными статическими переменными, а не в каждом объекте. То есть существует только один экземпляр константы Len, разделяемой между всеми объектами Stock. В главе 12 статические члены класса рассматриваются более подробно. Вы можете использовать эту технику только для объявления статических констант с целыми и перечислимыми значениями. Однако подобным образом невозможно хранить константу типа double.

Абстрактные типы данных

Класс Stock довольно-таки специфичен. Часто, однако, программисты определяют классы для представления более общих концепций. Например, использование классов — хороший способ реализации того, что специалисты в компьютерных науках называют абстрактными типами данных (abstract data type — ADT). Как можно предположить, ADT описывает данные в общем, без деталей реализации на конкретном языке. Рассмотрим, например, стек. Используя стек, вы можете сохранять дан-

ные так, что они всегда будут добавляться или удаляться с его вершины. Например, программы на C++ используют стек для управления автоматическими (локальными) переменными. Когда новые переменные генерируются, они добавляются на вершину стека. Когда уничтожаются, то удаляются из нее.

Давайте посмотрим на свойства стека вообще — в абстрактном смысле. Во-первых, стек содержит несколько элементов. (Это свойство делает его контейнером — то есть еще более общей абстракцией). Далее, стек характеризуется операциями, которые может выполнять:

- Вы можете создать пустой стек.
- Вы можете добавить элемент на его вершину, то есть *затолкнуть* (push) ее туда.
- Вы можете удалить его из вершины, то есть *вытолкнуть* (pop).
- Вы можете проверить, полон ли стек.
- Вы можете проверить, пуст ли стек.

Вы можете приложить это описание к объявлению класса, в котором общедоступные функции-члены представляют интерфейс, который реализует операции над стеком. Приватные данные-члены будут обеспечивать хранение информации в стеке. Концепция класса хорошо соответствует подходу ADT.

Раздел `private` должен заботиться о том, как хранить данные. Например, данные-члены могут использовать обычный массив, динамически распределенный в памяти массив либо какую-нибудь более совершенную структуру данных вроде связного списка. Однако общедоступный интерфейс класса должен скрывать детали реализации. Он должен быть выражен в общих понятиях, таких как создание стека, заталкивание элемента и так далее. В листинге 10.10 показан один из возможных подходов. Предполагается, что тип `bool` реализован. Если нет, то вы можете использовать `int` со значениями `0` и `1` вместо `bool` с `false` и `true`.

Листинг 10.10. stack.h

```
// stack.h -- определение класса для абстрактного типа данных — стека
#ifndef STACK_H_
#define STACK_H_
typedef unsigned long Item;
class Stack
{
private:
   enum {MAX = 10};     // специфическая для класса константа
   Item items[MAX];     // содержит элементы стека
   int top;             // индекс вершины стека
public:
   Stack();
   bool isempty() const;
   bool isfull() const;
   // push() возвращает false, если стек полон, иначе - true
   bool push(const Item & item);   // добавляет элемент в стек
   // pop() возвращает false, если стек пуст, иначе - true
   bool pop(Item & item);          // выталкивает элемент с вершины стека
};
#endif
```

Замечание по совместимости

Если в вашей системе не реализован тип `bool`, можете использовать `int` со значениями `0` и `1` вместо `bool` с `false` и `true`. Альтернативно ваша система может поддерживать более ранние, нестандартные формы, такие как `boolean` или `Boolean`.

В примере, представленном в листинге 10.10, раздел `private` показывает, что стек реализован с помощью массива, но раздел `public` никак не отражает этот факт. То есть вы можете заменить обычный массив, скажем, динамическим массивом без изменения интерфейса класса. Это означает, что изменение реализации стека не требует внесения изменений в код программы, которая будет его использовать. Вы просто перекомпилируете код стека и скомпонуете его с кодом вашей программы.

Представленный интерфейс несколько избыточен, так как `pop()` и `push()` возвращают информацию о состоянии стека (пуст или полон) вместо того, чтобы иметь тип `void`. Это предоставляет программисту дополнительные возможности, такие как управление переполнением стека и его очисткой. Программист может использовать `isempty()` и `isfull()` для проверки каждый раз перед модификацией стека, либо использовать возвращаемые значения `push()` и `pop()` для определения того, удалась ли соответствующая операция.

Вместо того чтобы определять стек в терминах некоторого конкретного типа, класс описывает его в терминах общего типа `Item`. В данном случае заголовочный файл использует `typedef` для задания эквивалентности типа `Item` стандартному `unsigned long`. Если вы хотите создать стек для хранения элементов типа `double` или структур, вы можете изменить `typedef` и оставить объявление класса и определения методов неизменными. Шаблоны классов (см. главу 14) предлагают более мощный метод изоляции хранимого типа от дизайна класса.

Далее вам нужно реализовать методы класса. В листинге 10.11 демонстрируется один из возможных вариантов.

Листинг 10.11. `stack.cpp`

```
// stack.cpp -- функции-члены класса Stack
#include "stack.h"
Stack::Stack() // создать пустой стек
{
   top = 0;
}
bool Stack::isempty() const
{
   return top == 0;
}
bool Stack::isfull() const
{
   return top == MAX;
}
bool Stack::push(const Item & item)
{
   if (top < MAX)
   {
     items[top++] = item;
     return true;
   }
```

```
   else
     return false;
}
bool Stack::pop(Item & item)
{
   if (top > 0)
   {
     item = items[--top];
     return true;
   }
   else
     return false;
}
```

Конструктор по умолчанию гарантирует, что все стеки будут создаваться пустыми. Код `pop()` и `push()` гарантирует, что вершина стека управляется правильно. Подобные гарантии — это одно из обстоятельств, которые обеспечивают надежность ООП. Предположим, что вы создадите отдельный массив для представления стека и независимую переменную, представляющую индекс вершины стека. В этом случае вы отвечаете за правильность кода при каждом создании нового стека. Без защиты, предоставляемой приватными данными, всегда существует возможность сделать ошибку и изменить данные нежелательным образом.

Протестируем стек. Код в листинге 10.12 моделирует жизнь клерка, обрабатывающего заказы на покупки, которые берет их из стопки на столе, используя алгоритм LIFO (last-in, first-out — последним пришел, первым обслужен), характерный для стека.

Листинг 10.12. `stacker.cpp`

```
// stacker.cpp -- тестирование класса Stack
#include <iostream>
#include <cctype> // или ctype.h
#include "stack.h"
int main()
{
   using namespace std;
   Stack st; // создать пустой стек
   char ch;
   unsigned long po;
   cout << "Пожалуйста, введите А для добавления заказа,\n "
        << " P - для обработки заказа, Q - для выхода.\n";
   while (cin >> ch && toupper(ch) != 'Q')
   {
     while (cin.get() != '\n')
        continue;
     if (!isalpha(ch))
     {
        cout << '\a';
        continue;
     }
     switch(ch)
     {
        case 'A':
```

```
        case 'a': cout << "Введите номер заказа для добавления: ";
                cin >> po;
                if (st.isfull())
                    cout << "Стек уже полон\n";
                else
                    st.push(po);
                break;
        case 'P':
        case 'p': if (st.isempty())
                    cout << "Стек уже пуст\n";
               else {
                    st.pop(po);
                    cout << "заказ No " << po << " вытолкнут\n";
               }
        break;
    }
    cout << "Пожалуйста, введите A для добавления заказа,\n"
         << "P - для обработки заказа, Q - для выхода.\n";
  }
  cout << "Всего наилучшего!\n";
  return 0;
}
```

Маленький цикл `while` в листинге 10.12, который принимает остаток строки, пока не является совершенно необходимым, но он пригодится в модифицированной версии программы, которая будет рассматриваться в главе 14.

Ниже показан пример запуска программы:

```
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода
A
Введите номер заказа для добавления: 17885
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода
P
заказ No #17885 вытолкнут
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода
A
Введите номер заказа для добавления: 17965
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода
A
Введите номер заказа для добавления: 18002
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода
P
заказ No #18002 вытолкнут
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода
P
заказ No #17965 вытолкнут
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода
P
```

```
Стек пуст
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода
Q
Всего наилучшего!
```

Пример из практики: минимизация размера класса с избирательной типизацией данных

При проектировании класса вам следует быть внимательными в обращении с типами данных, которые вы выбираете для переменных-членов класса. Необдуманное применение нестандартных или зависимых от платформы типов влияет на размер ваших классов, таким образом, увеличивая расход памяти ваших программ. Это неэффективная и плохая практика.

Классический пример, который демонстрирует это, использует нестандартный тип `BOOL typedef` вместо стандартного типа данных `bool`. Рассмотрим следующий простой класс:

```
typedef int BOOL;
class BadClassDesign
{
   BOOL m_b1;
   BOOL m_b2;
   BOOL m_b3;
   BOOL m_b4;
   BOOL m_b5;
   BOOL m_b6;
};
class GoodClassDesign
{
   bool m_b1;
   bool m_b2;
   bool m_b3;
   bool m_b4;
   bool m_b5;
   bool m_b6;
};
```

В отличие от `bool`, который обычно занимает только 1 байт на большинстве платформ, `BOOL`, как правило, занимает 4 байта. Если назначение членов класса — управлять реальными булевскими значениями, то для этого требуется только один байт. На типичной машине платформы Intel класс `BadClassDesign` займет 24 байта, а класс `GoodClassDesign` — только 6 байт. Экономия — 400%!

Вы всегда должны стараться использовать соответствующие задаче типы данных, которые минимизируют расход памяти, требуемой для программы.

Резюме

ООП акцентирует внимание на представлении данных. Первый шаг к разрешению проблем программирования с помощью объектно-ориентированного подхода – это представить данные в терминах их интерфейса с программой, описывающего, как их использовать. Далее вам нужно спроектировать класс, который реализует интерфейс. Обычно приватные (`private`) данные-члены хранят информацию, в то время, как общедоступные (`public`) функции-члены, также называемые методами, предоставляют единственный доступ к данным. Класс комбинирует данные и методы в единый узел, а приватный порядок доступа обеспечивает сокрытие данных.

Обычно объявление класса разделяется на две части, как правило, сохраняемые в разных файлах. Объявление класса попадает в заголовочный файл с методами, представленными прототипами функций. Исходный код, составляющий функции-члены, попадает в файл методов. Такой подход позволяет отделить описание интерфейса от деталей реализации. В принципе для того, чтобы использовать класс, необходимо знать только его общедоступный интерфейс. Конечно же, вы можете смотреть на реализацию (если только класс не поставляется в скомпилированном виде), но ваша программа не должна зависеть от деталей реализации класса, как и знать, что, например, какое-то значение хранится в виде `int`. До тех пор, пока программа и класс взаимодействуют только через методы, определенные в интерфейсе, вы вольны совершенствовать обе части независимо, не заботясь о нежелательном взаимодействии.

Класс — это определяемый пользователем тип, а объект — экземпляр класса. Это значит, что объект — переменная этого типа или эквивалент переменной, такой как выделенный операцией `new` участок памяти в соответствии со спецификациями класса. C++ старается сделать применение пользовательских типов настолько же простым, как и применение стандартных типов, поэтому вы можете объявлять объекты, указатели на объекты и массивы объектов. Вы можете передавать объекты в виде аргументов, возвращать их в качестве возвращаемых значений функций и присваивать один объект другому объекту того же типа. Если вы предусматриваете метод-конструктор, то можете инициализировать объект при его создании. Если предусматриваете деструктор — программа выполнит его при уничтожении объекта.

Каждый объект содержит свою собственную копию набора данных из объявления класса, но все они разделяют методы класса. Если `mr_object` — это имя определенного объекта, а `try_me()` — его функция-член, то вы можете вызывать эту функцию, используя операцию принадлежности (точку): `mr_object.try_me()`. В терминологии ООП такой вызов называется отправкой сообщения `try_me()` объекту `mr_object`. Любая ссылка на данные-члены класса в методе `try_me()` затем прилагается к данным-членам объекта `mr_object`. Аналогичным образом, вызов функции `i_object.try_me()` получает доступ к данным-членам объекта `i_object`.

Если вы хотите,, чтобы функция-член взаимодействовала с более чем одним объектом, можете передать ей дополнительные объекты в виде аргументов. Если метод нуждается в явном доступе к объекту, который его вызвал, он может сделать это через указатель `this`. Значение указателя `this` устанавливается в адрес самого объекта, поэтому выражение `*this` является псевдонимом его самого.

Классы хорошо подходят для описания абстрактных типов данных — ADT. Интерфейс общедоступных функций-членов представляет службы, описанные ADT, а приватные члены и код методов класса представляют собой реализацию, скрытую от пользователей класса.

Вопросы для самоконтроля

1. Что такое класс?
2. Как класс обеспечивает абстракцию, инкапсуляцию и сокрытие данных?
3. Каковы отношения между объектом и классом?
4. Чем отличаются функции-члены класса от данных-членов класса помимо того, что они — функции?

5. Определите класс, представляющий банковский счет. Данные-члены должны включать имя вкладчика, номер счета (используйте строку) и баланс. Функции-члены должны позволять следующее:
   - Создавать объект и инициализировать его.
   - Отображать имя вкладчика, номер счета и баланс.
   - Добавлять на счет сумму денег, переданную в аргументе.
   - Снимать сумму денег, переданную в аргументе.

   Просто дайте объявление класса, без реализации методов (упражнение 1 предоставит вам возможность написать реализацию).
6. Когда вызываются конструкторы класса? Когда вызываются деструкторы?
7. Напишите код конструктора для класса пустого счета, как описано в вопросе 5.
8. Что такое конструктор по умолчанию? Каковы выгоды его применения?
9. Модифицируйте класс `Stock` (версию из `stock2.h`) таким образом, чтобы он имел функции-члены, которые возвращают значения индивидуальных данных-членов. На заметку: член, который возвращает имя компании, не должен предоставлять возможность изменять массив. То есть он не должен просто возвращать `char *`. Он может возвращать константный указатель или указатель на копию массива, созданную операцией `new`.
10. Что такое `this` и `*this`?

Упражнения по программированию

1. Представьте определения методов для класса, описанного в вопросе 5, и напишите короткую программу, которая проиллюстрирует его возможности.
2. Имеется определение простого класса:

```
class Person {
private:
   static const LIMIT = 25;
   string lname;       // фамилия
   char fname[LIMIT];  // имя
public:
   Person() {lname = ""; fname[0] = '\0'; } // #1
   Person(const string & ln, const char * fn = "Heyyou"); // #2
   // следующие методы отображают lname и fname
   void Show() const;        // формат: имя фамилия
   void FormalShow() const;  // формат: фамилия, имя
};
```

   Напишите программу, которая дополнит реализацию за счет предоставления кода неопределенных методов. Эта программа, в которой вы используете класс, должна также применять вызовы трех возможных конструкторов (без аргументов, с одним аргументом, с двумя аргументами) и двух методов отображения. Вот пример, использующий конструкторы и методы:

```
Person one;                    // используется конструктор по умолчанию
Person two("Smythecraft");     // используется #2 с одним аргументом
                               // по умолчанию
```

```
Person three("Dimwiddy", "Sam"); // используется #2, без аргументов
                                 // по умолчанию
one.Show();
cout << endl;
one.FormalShow();
// и так далее
```

3. Выполните упражнение 1 из главы 9, но замените представленный там код соответствующим объявлением класса `goif`. Замените `setgoif(goif &, const chart *, int)` конструктором с соответствующими аргументами для выполнения инициализации. Помните об интерактивной версии `setgoif()`, но реализуйте ее с использованием конструктора. (Например, для кода `setgoif()` получите данные, передайте их конструктору для создания временного объекта и присвойте временный объект вызвавшему, представленному через `*this`.)
4. Выполните упражнение 4 из главы 9, но преобразуйте структуру `Sales` и ассоциированные с ней функции в класс и методы. Замените функцию `setSales(Sales &, double[], int)` конструктором. Реализуйте интерактивный метод `setSales(Sales &)`, используя конструктор. Поместите класс в пространство имен `SALES`.
5. Имеется следующее объявление структуры:

```
struct customer {
    char fullname[35];
    double payment;
};
```

Напишите программу, которая будет добавлять структуры заказчиков в стек и удалять из стека, представленного объявлением класса `Stack`. Всякий раз, когда заказчик удаляется, его зарплата должна добавляться к промежуточной сумме и по этой сумме выдаваться отчет. На заметку: вы должны быть готовы использовать класс `Stack` без изменений; изменить можно только объявление `typedef`, чтобы `Item` был типом `customer` вместо `unsigned long`.

6. Имеется следующее объявление класса:

```
class Move
{
private:
    double x;
    double y;
public:
    Move(double a = 0, double b = 0); // устанавливает x, y в a, b
    showmove() const; // отображает текущие значения x, y
    Move add(const Move & m) const;
    // эта функция добавляет x из m к x вызывающего объекта,
    // чтобы получить новое x,
    // добавляет y из m к y вызывающего объекта, чтобы получить новое y,
    // присваивает инициализированному объекту значения x, y
    // и возвращает его
    reset(double a = 0, double b = 0); // сбрасывает x,y в a, b
};
```

Создайте определения функций-членов и программу, которая использует этот класс.

7. Бетельгейзийский плорг обладает следующими свойствами:

   Данные:

   Плорг имеет имя не длиннее 19 символов.

   Плорг имеет индекс довольства (CI), выражаемый целым числом.

   Операции:

   Новый плорг начинает существование с именем и индексом CI равным 50.

   Индекс CI плорга может изменяться.

   Плорг может сообщать свое имя и индекс CI.

   По умолчанию плорг имеет имя "Plorga".

   Напишите объявление класса Plorg (включая данные-члены и прототипы функций-членов), представляющего плорга. Напишите определения функций-членов. Напишите короткую программу, которая демонстрирует все свойства класса Plorg.

8. Простой список можно описать следующим образом:

   - Простой список может содержать ноль или более элементов определенного типа.
   - Можно создать пустой список.
   - Можно добавлять элемент в список.
   - Можно определять, пуст ли список.
   - Можно определять, полон ли список.
   - Можно посетить каждый элемент списка и выполнить над ним определенное действие.

   Как видите, список действительно прост. Так, например, он не предусматривает вставки и удаления элементов.

   Спроектируйте класс List для представления этого абстрактного типа. Вы должны подготовить заголовочный файл list.h с объявлением класса и файл list.cpp с реализацией его методов. Вы должны также написать короткую программу, которая будет использовать ваш класс.

   Главная причина того, что спецификации списка просты, связана с попыткой упростить упражнение по программированию. Вы можете реализовать список в виде массива или же, если вы знакомы с этим типом данных — в виде связного списка. Но общедоступный интерфейс не должен зависеть от вашего выбора. То есть общедоступный интерфейс не должен иметь индексов массива, указателей на узлы и так далее. Он должен быть выражен в виде общей концепции — создание списка, добавление элемента в список и так далее. Обычный способ управления посещением каждого элемента в списке и выполнения над ним каких-то действий состоит в применении функции, которая принимает указатель на другую функцию в качестве аргумента:

   ```
   void visit(void (*pf)(Item &));
   ```

   Здесь pf указывает на функцию (не функцию-член), которая принимает ссылку на аргумент типа Item, где Item — тип элементов списка. visit() применяет эту функцию к каждому элементу списка. Можете использовать класс Stack в качестве общего руководства.

ГЛАВА 11

Работа с классами

В этой главе:

- Перегрузка операций
- Дружественные функции
- Перегрузка операции << для вывода
- Члены состояния
- Использование `rand()` для генерации случайных чисел
- Автоматическое преобразование и приведение типов для классов
- Функции преобразования классов

Классы C++ — это богатые возможностями, сложные и мощные средства. В главе 9 вы приступили к изучению объектно-ориентированного программирования, начав с разработки и применения простого класса. Вы видели, что класс определяется как тип данных, предназначенный для представления объекта, и также, посредством функций-членов — операций, которые могут выполняться над этими данными. Вы изучили также два специальных типа функций-членов — конструктор и деструктор, которые управляют процессом создания и уничтожения объектов, созданных на основе спецификаций класса. Настоящая глава продвинет вас на несколько шагов дальше в объяснении свойств классов, концентрируя основное внимание не на общих принципах, а на технике их проектирования. Возможно, некоторые из этих средств покажутся вам простыми, а некоторые — достаточно изощренными. Для лучшего понимания этих новых средств вам придется попробовать примеры и поэкспериментировать с ними. Что случится, если в функции применить обычный аргумент вместо аргумента, переданного по ссылке? Что произойдет, если что-то не будет включено в деструктор? Не бойтесь совершать ошибки — обычно вы сможете научиться большему, исправляя допущенные ошибки, чем если сразу все сделаете правильно благодаря механическому запоминанию. (Однако не следует думать, что водопад ошибок неизбежно приведет вас к невообразимой степени понимания предмета.) В конце вы будете вознаграждены более полным пониманием того, как работает C++, и что он может сделать для вас.

Настоящую главу мы начнем с перегрузки операций, которая позволит вам использовать стандартные операции C++, такие как = и +, с объектами классов. Затем рассмотрим понятие "друзей" — механизм C++, позволяющий функциям, не являющимся членами класса, получать доступ к приватным данным. И, наконец, мы рассмотрим, как можно заставить C++ выполнять автоматическое преобразование типов при работе с классами. После того, как вы ознакомитесь с настоящей главой и главой 12, вы достигнете полного понимания того, какую роль играют конструкторы и деструкторы. Кроме того, вы узнаете о некоторых дополнительных стадиях, которые нужно пройти в процессе проектирования и разработки классов.

Одной из трудностей при изучении C++, по крайней мере, на данном этапе, является огромный объем информации, которую необходимо запоминать. И, конечно, нет резона рассчитывать, что удастся запомнить все, до тех пор, пока вы не приобретете достаточный опыт. В этом смысле изучение C++ подобно изучению перегруженного разнообразными средствами сложного текстового процессора или электронной таблицы. Ни одно из средств не является таким уж сложным, но на практике выясняется, что большинство людей действительно хорошо знают только те средства, которые они используют регулярно, например, средства поиска или преобразования шрифта. Вы можете периодически вспоминать, что надо бы почитать где-нибудь, каким образом генерируются альтернативные символы или создаются таблицы, но эти знания не станут частью вашего арсенала до тех пор, пока вы не столкнетесь с ситуациями, в которых они будут востребованы часто. Возможно, лучший подход к усвоению материала данной главы — просто начать использовать хотя бы некоторые из этих новых средств в своей повседневной практике программирования на C++. По мере того, как ваш опыт будет совершенствоваться, а с ним — понимание и оценка пользы от новых возможностей, вы сможете постепенно добавлять в свой арсенал новые средства C++. Как говорил Бьерн Страуструп (Bjarne Straustrup), создатель C++, на конференции профессиональных программистов: “Упрощайте язык для себя. Не считайте себя обязанными использовать все средства языка, и уж тем более не пытайтесь использовать их все в первый день”.

Перегрузка операций

Давайте рассмотрим технику, которая придает операциям над объектами более симпатичный вид. *Перегрузка операций* — это пример полиморфизма C++. В главе 8 вы видели, как C++ позволяет определять несколько функций с одинаковыми именами и разной сигнатурой (списками аргументов). Это называлось *перегрузкой функций*. Цель такой перегрузки — позволить использовать одно и то же имя функции для некоторой базовой операции, даже несмотря на то, что она применяется к данным разных типов. (Представьте, насколько неуклюжим был бы английский язык, если бы вам пришлось применять разные словесные формы для каждого отдельного типа объектов, например, `lift_lft` — поднять левую ногу и `lift_sp` — поднять ложку). Перегрузка операций расширяет концепцию перегрузки на операции, позволяя придавать им множественные значения. На самом деле многие операции C++ (как и C) уже перегружены. Например, операция `*`, когда применяется к адресу, означает значение, хранимое по этому адресу. Но использование `*` к двум числовым величинам означает их перемножение. C++ использует количество и тип операндов, чтобы решить, какое действие нужно выполнить в каждом конкретном случае.

Язык C++ позволяет распространить перегрузку операций на пользовательские типы, разрешая, скажем, применять символ + для сложения двух объектов. Опять-таки, компилятор использует количество и тип операндов для определения того, какое именно определение данной операции следует использовать. Перегруженные операции часто могут заставить код выглядеть более естественно. Например, обычная задача в вычислениях — сложение двух массивов. Обычно это выглядит как следующий цикл `for`:

```
for (int i = 0; i < 20; i++)
   evening[i] = sam[i] + janet[i]; // добавить элемент к элементу
```

Но в C++ вы можете определить класс, который представляет массивы и перегружает операцию + таким образом, что вы можете сделать так:

```
evening = sam + janet; // сложить два объекта-массива
```

Этот пример нотации сложения скрывает механизм и подчеркивает то, что существенно, а это и является одной из целей ООП.

Для перегрузки операции используется специальная форма функции, называемая функцией операции. Функция операции выглядит следующим образом:

```
operatorop(список-аргументов)
```

здесь *op* — символ перегружаемой операции. Например, `operator+()` перегружает операцию +, а `operator*()` — операцию `*`. Операция *op* должна быть допустимой операцией C++, а не произвольным символом. Так, например, вы не можете объявить операцию `operator@()`, так как в C++ нет операции `@`. С другой стороны, функция `operator[]()` перегружает операцию `[]`, поскольку `[]` — это операция индексации массивов. Предположим, например, что у вас есть класс `Salesperson`, в котором вы определили функцию-член `operator+()` для перегрузки операции +, таким образом, что она сможет добавлять зарплату одного лица другому. Тогда, если `district2`, `sid` и `sara` — объекты класса `Salesperson`, то вы можете написать следующее выражение:

```
district2 = sid + sara;
```

Компилятор, распознав операцию как относящуюся к классу `Salesperson`, заменит ее вызовом соответствующей функции операции:

```
district2 = sid.operator+(sara);
```

Функция затем использует объект `sid` неявно (поскольку она вызывает метод) и объект `sara` — явно (поскольку он передается в виде аргумента) для вычисления суммы, возвращаемой в результате. Конечно, возможность использовать обозначение операции + вместо неуклюжего вызова функции выглядит более симпатично.

C++ накладывает некоторые ограничения на перегрузку операций, но все же их проще понимать после того, как вы разберетесь, как работает перегрузка. Поэтому давайте разработаем несколько примеров для прояснения процесса, после чего обсудим ограничения.

Время в наших руках: разработка примера перегрузки операции

Если вы работали в системе как пользователь по имени Priggs в течении 2 часов 35 минут с утра и 2 часов 40 минут после обеда, то сколько всего времени вы проработали в системе? Ниже приведен пример, в котором концепция сложения имеет смысл, несмотря на то, что единицы, которые вы складываете (смесь часов и минут) не соответствует никакому из встроенных типов. В главе 7 рассматривается подобный случай; там определяется структура `travel_time` и функция `sum()` для сложения структур упомянутого типа. Теперь давайте обобщим это в классе `Time`, используя метод для управления сложением. Начнем с простого метода, называемого `Sum()`, а затем посмотрим, как его преобразовать в перегруженную операцию. В листинге 11.1 показано объявление класса `Time`.

Листинг 11.1. mytime0.h

```
// mytime0.h -- класс Time до перегрузки операции
#ifndef MYTIME0_H_
#define MYTIME0_H_
class Time
{
   private:
   int hours;
   int minutes;
public:
   Time();
   Time(int h, int m = 0);
   void AddMin(int m);
   void AddHr(int h);
   void Reset(int h = 0, int m = 0);
   Time Sum(const Time & t) const;
   void Show() const;
};
#endif
```

Класс Time предоставляет методы для изменения и установки времени, для отображения значений времени и для сложения двух значений времени. В листинге 11.2 приведено определение методов. Обратите внимание, что методы AddMin() и Sum() используют целочисленное деление и операцию взятия модуля для корректировки значений минут и часов, когда общее количество минут превышает 59. Также поскольку единственное средство iostream, которое здесь используется — это cout, а также потому, что оно применяется только один раз, имеет смысл указать std::cout вместо того, чтобы использовать все пространство имен std.

Листинг 11.2. mytime0.cpp

```
// mytime0.cpp -- реализация методов Time
#include <iostream>
#include "mytime0.h"
Time::Time()
{
   hours = minutes = 0;
}
Time::Time(int h, int m )
{
   hours = h;
   minutes = m;
}
void Time::AddMin(int m)
{
   minutes += m;
   hours += minutes / 60;
   minutes %= 60;
}
void Time::AddHr(int h)
{
   hours += h;
}
```

```
void Time::Reset(int h, int m)
{
   hours = h;
   minutes = m;
}
Time Time::Sum(const Time & t) const
{
   Time sum;
   sum.minutes = minutes + t.minutes;
   sum.hours = hours + t.hours + sum.minutes / 60;
   sum.minutes %= 60;
   return sum;
}
void Time::Show() const
{
   std::cout << hours << " часов, " << minutes << " минут";
}
```

Рассмотрим код функции `Sum()`. Отметим, что аргумент является ссылкой, но возвращаемый тип не является ссылкой. Причина того, что аргумент передается по ссылке, в эффективности. Код должен выдавать тот же результат, если объект `Time` передается по значению, но обычно быстрее и эффективнее с точки зрения использования памяти передавать по ссылке.

Однако возвращаемое значение не может быть ссылкой. Причина состоит в том, что функция создает новый объект `Time` (`sum`), который представляет сумму двух других объектов `Time`. Возврат объекта, как это делал бы такой код, создает копию объекта, которую может использовать вызывающая функция. Если возвращаемое значение будет типа `Time &`, ссылка будет указывать на сам объект `sum`. Но объект `sum` — это локальная переменная, которая уничтожается при завершении работы функции, поэтому ссылка указывает на несуществующий объект. Использование типа возврата `Time`, однако, означает, что программа конструирует копию объекта `sum` перед тем, как разрушить его, и вызывающая функция получает корректный результат.

Внимание!

Не возвращайте ссылок на локальные переменные или другие временные объекты. Когда функция завершает работу и локальные переменные или временные объекты исчезают, то ссылка указывает на несуществующие данные.

И, наконец, листинг 11.3 тестирует суммирование времени в классе `Time`.

Листинг 11.3. usemytime0.cpp

```
// usemytime0.cpp -- использование первого наброска класса Time
#include <iostream>
#include "mytime0.h"
int main()
{
   using std::cout;
   using std::endl;
   Time planning;
   Time coding(2, 40);
   Time fixing(5, 55);
```

```
    Time total;
    cout << "плановое время = ";
    planning.Show();
    cout << endl;
    cout << "время кодирования = ";
    coding.Show();
    cout << endl;
    cout << "фиксированное время = ";
    fixing.Show();
    cout << endl;
    total = coding.Sum(fixing);
    cout << "coding.Sum(fixing) = ";
    total.Show();
    cout << endl;
    return 0;
}
```

Ниже показан вывод программы из листингов 11.1, 11.2 и 11.3.

```
плановое время = 0 часов, 0 минут
время кодирования = 2 часов, 40 минут
фиксированное время = 5 часов, 55 минут
coding.Sum(fixing) = 8 часов, 35 минут
```

Добавление операции сложения

Очень просто изменить класс Time таким образом, чтобы он использовал перегруженную операцию сложения. Вы просто должны изменить имя функции Sum() на выглядящее несколько странно имя operator+(). Все верно: вы просто добавляете символ операции (в данном случае +) в конец слова operator и используете результат как имя метода. Это — единственное место, где можно применять в имени идентификатора символ, отличный от букв, цифр и знака подчеркивания. В листингах 11.4 и 11.5 проведено это небольшое изменение.

Листинг 11.4. mytime1.h

```
// mytime1.h -- класс Time после перегрузки операции
#ifndef MYTIME1_H_
#define MYTIME1_H_
class Time
{
    private:
    int hours;
    int minutes;
public:
    Time();
    Time(int h, int m = 0);
    void AddMin(int m);
    void AddHr(int h);
    void Reset(int h = 0, int m = 0);
    Time operator+(const Time & t) const;
    void Show() const;
};
#endif
```

Листинг 11.5. mytime1.cpp

```
// mytime1.cpp -- реализация методов Time
#include <iostream>
#include "mytime0.h"
Time::Time()
{
   hours = minutes = 0;
}
Time::Time(int h, int m )
{
   hours = h;
   minutes = m;
}
void Time::AddMin(int m)
{
   minutes += m;
   hours += minutes / 60;
   minutes %= 60;
}
void Time::AddHr(int h)
{
   hours += h;
}
void Time::Reset(int h, int m)
{
   hours = h;
   minutes = m;
}
Time Time::operator+(const Time & t) const
{
   Time sum;
   sum.minutes = minutes + t.minutes;
   sum.hours = hours + t.hours + sum.minutes / 60;
   sum.minutes %= 60;
   return sum;
}
void Time::Show() const
{
   std::cout << hours << " часов, " << minutes << " минут";
}
```

Подобно Sum(), operator+() вызывается объектом Time, принимая второй объект Time в качестве аргумента, и возвращает объект Time. Таким образом, вы можете вызвать метод operator+(), используя тот же синтаксис, что и в случае с Sum():

```
total = coding.operator+(fixing);    // нотация функции
```

Однако имя метода operator+() позволяет также применить нотацию операции:

```
total = coding + fixing;             // нотация операции
```

Оба варианта вызывают метод operator+(). Отметим, что в нотации операции объект слева от операции (в данном случае coding) является вызывающим объектом,

а объект справа (в данном случае fixing) — это тот, что передается в качестве аргумента. Код в листинге 11.6 иллюстрирует это.

Листинг 11.6. usemytime1.cpp

```
// usemytime1.cpp -- использование второго наброска класса Time
   #include <iostream>
   #include "mytime1.h"
int main()
{
   using std::cout;
   using std::endl;
   Time planning;
   Time coding(2, 40);
   Time fixing(5, 55);
   Time total;
   cout << "плановое время = ";
   planning.Show();
   cout << endl;
   cout << "время кодирования = ";
   coding.Show();
   cout << endl;
   cout << "фиксированное время = ";
   fixing.Show();
   cout << endl;
   total = coding + fixing;
   // нотация операции
   cout << "coding + fixing = ";
   total.Show();
   cout << endl;
   Time morefixing(3, 28);
   cout << "Еще одно фиксированное время = ";
   morefixing.Show();
   cout << endl;
   total = morefixing.operator+(total);
   // нотация функции
   cout << "morefixing.operator+(total) = ";
   total.Show();
   cout << endl;
   return 0;
}
```

Ниже показан вывод программы из листингов 11.4, 11.5 и 11.6:

```
плановое время = 0 часов, 0 минут
время кодирования = 2 часов, 40 минут
фиксированное время = 5 часов, 55 минут
coding + fixing = 8 часов, 35 минут
Еще одно фиксированное время = 3 часов, 28 минут
morefixing.operator+(total) = 12 часов, 3 минуты
```

Короче говоря, имя функции operator+() позволяет вызывать ее как в нотации функции, так и в нотации операции. Компилятор использует тип операнда для определения, что надо делать:

```
int a, b, c;
Time A, B, C;
c = a + b;  // используется сложение целых чисел
C = A + B;  // используется сложение, определенное для объектов Time
```

Можно ли складывать более двух объектов? Например, если t1, t2, t3 и t4 являются объектами класса Time, можно ли выполнить следующее?

```
t4 = t1 + t2 + t3; // правильно ли это?
```

Чтобы ответить на этот вопрос, нужно рассмотреть, как транслируется это выражение в вызовы функций. Поскольку сложение — операция, выполняемая слева направо, первая трансляция дает:

```
t4 = t1.operator+(t2 + t3); // правильно ли это?
```

Затем аргумент функции также транслируется в вызов функции, и мы получаем:

```
t4 = t1.operator+(t2.operator+(t3)); // правильно ли это? ДА
```

Верно ли это? Да, верно. Вызов функции t2.operator+(t3) возвращает объект Time, представляющий собой сумму t2 и t3. Этот объект затем передается вызову t1.operator+() и этот вызов затем возвращает сумму t1 и объекта Time, который представляет сумму t2 и t3. Короче говоря, финальное возвращаемое значение является суммой t1, t2 и t3, как и ожидалось.

Ограничения перегрузки

Большинство операций C++ (см. табл. 11.1) могут быть перегружены таким же образом, как это описано выше. Перегруженные операции (за некоторыми исключениями) не обязательно должны быть функциями-членами. Однако по крайней мере один из операндов должен иметь тип, определенный пользователем. Посмотрим внимательно на ограничения, которые накладывает C++ на перегрузку операций, определенных пользователем:

- Перегруженные операции должны иметь как минимум один операнд пользовательского типа. Это предохраняет вас от перегрузки операций, работающих со стандартными типами. То есть, вы не сможете переопределить операцию "минус" (-) так, что она будет вычислять сумму двух действительных чисел вместо разности. Это ограничение сохраняет здравый смысл программы, хотя и несколько препятствует полету творчества.
- Вы не можете использовать операцию в такой манере, которая нарушает правила синтаксиса исходной операции. Например, вы не сможете перегрузить операцию взятия модуля (%) так, чтобы она применялась с одним операндом:

  ```
  int x;
  Time shiva;
  %x;      // не допускается для операции взятия модуля
  %shiva;  // не допускается для перегруженной операции
  ```

  Аналогично, вы не сможете изменить приоритеты операций. Поэтому, если вы перегрузите операцию сложения для класса, то новая операция будет иметь тот же приоритет, что и обычное сложение.

- Вы не можете определять новые символы операций. Например, вы не можете определить функцию `operator**()` для создания операции возведения в степень.
- Нельзя перегружать следующие операции:

Операция	Описание
`sizeof`	Операция `sizeof`.
	Операция принадлежности (членства).
`.*`	Операция указателя на член.
`::`	Операция разрешения контекста.
`:?`	Условная операция.
`typeid`	Операция RTTI (runtime type idenfication — определение типа во время выполнения).
`const_cast`	Операция приведения типа.
`dynamic_cast`	Операция приведения типа.
`reinterpret_cast`	Операция приведения типа.
`static_cast`	Операция приведения типа.

Это по-прежнему оставляет операции из табл. 11.1 доступными для перегрузки.

- Большинство операций из табл. 11.1 допускают перегрузку за счет использования как функций-членов, так и функций-не-членов. Однако вы можете использовать *только* функции-члены для перегрузки следующих операций:

Операция	Описание
`=`	Операция присваивания.
`()`	Операция вызова функции.
`[]`	Операция индексации.
`->`	Операция доступа к членам класса через указатель.

На заметку!

В настоящей главе не раскрываются все операции, упомянутые в списке ограничений или табл. 11.1. Операции, которые не рассмотрены в тексте этой главы, суммируются в приложении Д.

Таблица 11.1. Операции, которые могут быть перегружены

+	-	*	/	%	^
&	|	~=	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&
||	++	--	,	->*	->
()	[]	new	delete	new[]	delete[]

В дополнение к этим формальным ограничениям вы должны использовать осмысленные ограничения в перегруженных операциях. Например, вы не должны перегружать операцию * так, что она будет обменивать значения членов-данных двух объ-

ектов Time. В случае если вы это сделаете, ничего в нотации не скажет о том, что на самом деле операция делает, поэтому для этой цели лучше все-таки предусмотреть метод класса с осмысленным именем, например, Swap().

Дополнительные сведения о перегруженных операциях

Для класса Time имеет смысл и ряд других операций. Например, вы можете захотеть вычитать одно время из другого или умножать время на число. Представим, скажем, перегрузку операций вычитания и умножения. Техника — та же самая, что и для операции сложения. Вы создаете методы operator-() и operator*(). То есть, вы добавляете следующие прототипы к объявлению класса:

```
Time operator-(const Time & t) const;
Time operator*(double n) const;
```

Листинг 11.7. mytime2.h

```
// mytime2.h -- класс Time после перегрузки операции
#ifndef MYTIME2_H_
#define MYTIME2_H_
class Time
{
   private:
   int hours;
   int minutes;
public:
   Time();
   Time(int h, int m = 0);
   void AddMin(int m);
   void AddHr(int h);
   void Reset(int h = 0, int m = 0);
   Time operator+(const Time & t) const;
   Time operator-(const Time & t) const;
   Time operator*(const Time & t) const;
   void Show() const;
};
#endif
```

Листинг 11.8. mytime2.cpp

```
// mytime2.cpp -- реализация методов Time
#include <iostream>
#include "mytime0.h"
Time::Time()
{
   hours = minutes = 0;
}
Time::Time(int h, int m )
{
   hours = h;
   minutes = m;
}
```

```
void Time::AddMin(int m)
{
   minutes += m;
   hours += minutes / 60;
   minutes %= 60;
}
void Time::AddHr(int h)
{
   hours += h;
}
void Time::Reset(int h, int m)
{
   hours = h;
   minutes = m;
}
Time Time::operator+(const Time & t) const
{
   Time sum;
   sum.minutes = minutes + t.minutes;
   sum.hours = hours + t.hours + sum.minutes / 60;
   sum.minutes %= 60;
   return sum;
}
Time Time::operator-(const Time & t) const
{
   Time diff;
   int tot1, tot2;
   tot1 = t.minutes + 60 * t.hours;
   tot2 = minutes + 60 * hours;
   diff.minutes = (tot2 - tot1) % 60;
   diff.hours = (tot2 - tot1) / 60;
   return diff;
}
Time Time::operator*(double mult) const
{
   Time result;
   long totalminutes = hours * mult * 60 + minutes * mult;
   result.hours = totalminutes / 60;
   result.minutes = totalminutes % 60;
   return result;
}

void Time::Show() const
{
   std::cout << hours << " часов, " << minutes << " минут";
}
```

Листинг 11.9. `usemytime2.cpp`

```
// usemytime2.cpp -- использование третьего наброска класса Time
#include <iostream>
#include "mytime3.h"
```

```
int main()
{
   using std::cout;
   using std::endl;
   Time planning;
   Time weeding(4, 35);
   Time waxing(2, 47);
   Time total;
   Time diff;
   Time adjusted;
   cout << "время подготовки = ";
   weeding.Show();
   cout << endl;
   cout << "полезное время = ";
   waxing.Show();
   cout << endl;
   cout << "Общее рабочее время = ";
   total = weeding + waxing; // используется operator+()
   total.Show();
   cout << endl;
   diff = weeding — waxing; // используется operator-()
   cout << "время подготовки — полезное время = ";
   diff.Show();
   cout << endl;
   adjusted = total * 1.5; // используется operator*()
   cout << "исправленное рабочее время = ";
   adjusted.Show();
   cout << endl;
   return 0;
}
```

Вот как выглядит вывод программы из листингов 11.7, 11.8 и 11.9:

```
время подготовки = 4 hours, 35 minutes
полезное время = 2 hours, 47 minutes
Общее рабочее время = 7 hours, 22 minutes
время подготовки — полезное время = 1 hours, 48 minutes
исправленное рабочее время = 11 hours, 3 minutes
```

Что такое друзья?

Как вы уже видели, C++ управляет доступом к разделу `private` объекта класса. Обычно общедоступные (`public`) методы класса служат единственным каналом доступа, но иногда такое ограничение оказывается чересчур строгим, чтобы отвечать некоторым потребностям, возникающим в процессе программирования. Для таких случаев C++ предусматривает другую форму доступа: *друзей*. Друзья бывают в трех вариантах:

- Дружественные функции.
- Дружественные классы.
- Дружественные функции-члены.

Объявляя функцию другом класса, вы позволяете ей иметь те же привилегии доступа, что имеют функции-члены класса. Рассмотрим сейчас дружественные функции подробнее, отложив объяснение других двух типов до главы 15.

Прежде чем мы увидим, как определяются друзья, давайте посмотрим, зачем они вообще могут понадобиться. Часто перегрузка бинарной операции (то есть операции с двумя аргументами) приводит к потребности в друзьях. Умножение объекта Time на действительное число как раз представляет такую ситуацию, поэтому давайте ее изучим.

В примере класса Time перегруженная операция умножения отличается от двух других перегруженных операций тем, что комбинирует два разных типа. То есть операции сложения и вычитания работают с двумя значениями типа Time, а операция умножения комбинирует значение типа Time со значением типа double. Это ограничивает его применение. Помните, что левый операнд — это вызывающий объект. То есть

```
A = B * 2.75;
```

транслируется в следующий вызов функции-члена:

```
A = B.operator*(2.75);
```

Но как насчет приведенной ниже операции?

```
A = 2.75 * B;  // не соответствует вызову функции-члена
```

Концептуально 2.75 * B должно быть эквивалентно B * 2.75, но первое выражение не может соответствовать функции-члену, поскольку 2.75 не является объектом типа Time. Помните, что левый операнд — это вызывающий объект, но 2.75 — это НЕ объект. Поэтому компилятор не может заменить это выражение вызовом функции-члена.

Один способ обойти эту трудность — это сказать всем (и запомнить самому), что вы можете писать только B * 2.75, а не 2.75 * B. Это — дружественное к серверу решение, возлагающее ответственность на клиента, что не отвечает принципам ООП.

Однако существует другая возможность — использовать функцию, не являющуюся членом. (Вспомним, что большинство операций могут быть перегружены как функциями-членами, так и просто функциями.) Функция, не являющаяся членом, не вызывается объектом. Вместо этого все значения, которые она использует, включая объекты, являются явными аргументами. То есть компилятор может представить выражение

```
A = 2.75 * B;  // не соответствует вызову функции-члена
```

в виде следующего вызова функции-не-члена:

```
A = operator*(2.75, B);
```

Эта функция должна иметь следующий прототип:

```
Time operator*(double m, const Time & t);
```

У функции, не являющейся членом и перегружающей операцию, левый операнд соответствует первому аргументу, а правый — второму. Между тем, исходная функция-член имеет дело с операндами в обратном порядке — то есть значение типа Time умножается на значение типа double.

Применение функции, не являющейся членом, решает проблему получения операндов в нужном порядке (сначала double, затем — Time), но приводит к возникновению новой проблемы: функция-не-член не имеет непосредственного доступа к приватным данным класса. По крайней мере, обычная функция, не являющаяся членом, не имеет доступа. Однако существует специальная категория функций-не-членов, называемая *друзьями*, которая имеет доступ к приватным данным класса.

Создание друзей

Первый шаг в создании дружественных функций состоит в помещении прототипа в объявление класса с префиксом — ключевым словом friend:

```
friend Time operator*(double m, const Time & t); // размещается в
                                                 // объявлении класса
```

Этот прототип имеет два смысла:

- Несмотря на то что функция operator*() присутствует в объявлении класса, она не является функцией-членом класса. Поэтому она не вызывается через операцию принадлежности (.).
- Несмотря на то что функция operator*() не является функцией-членом класса, она имеет те же права доступа, что и функции-члены.

Второй шаг — написание определения функции. Поскольку она не функция-член, вам не нужно добавлять квалификатор Time::. Кроме того, слово friend также не используется в определении. Определение будет выглядеть следующим образом:

```
Time operator*(double m, const Time & t) // ключевое слово friend
                                         // в определении не используется
{
   Time result;
   long totalminutes = t.hours * mult * 60 +t. minutes * mult;
   result.hours = totalminutes / 60;
   result.minutes = totalminutes % 60;
   return result;
}
```

С таким объявлением и определением оператор

```
A = 2.75 * B;
```

транслируется в

```
A = operator*(2.75, B);
```

и вызывает только что определенную дружественную функцию-не-член.

Короче говоря, дружественная функция класса — это функция-не-член, имеющая те же права доступа, что и функция-член.

Нарушают ли друзья принципы ООП?

На первый взгляд может показаться, что друзья нарушают принцип сокрытия данных ООП, поскольку механизм друзей позволяет функциям-не-членам получать доступ к приватным данным. Однако так может показаться только на поверхностный взгляд. Вместо этого вы должны думать о дружественных функциях, как о части расширенного интерфейса класса. Например, с концептуальной точки зрения умножение действительного числа на объект типа Time — это то же самое,

что умножение объекта Time на действительное число. И хотя для реализации первого необходима дружественная функция, а для другого — функция-член, разница между ними выражается только в синтаксисе C++, а не в глубоком концептуальном смысле. Используя и дружественную функцию, и функцию-член, вы можете выразить обе операции в одном пользовательском интерфейсе. Кроме того, следует помнить, что только объявление класса определяет, какие функции являются дружественными, то есть объявление класса по-прежнему управляет тем, каким функциям разрешен доступ к приватным данным. Короче говоря, методы класса и друзья — это просто два разных механизма выражения интерфейса класса.

В действительности, вы можете написать конкретную дружественную функцию в виде вызова функции-члена, только изменив ее определение так, чтобы она поменяла местами операнды умножения:

```
Time operator*(double m, const Time & t)
{
   return t * m; // используется t.operator*(m)
}
```

Исходная версия имеет явный доступ к t.minutes и t.hours, поэтому должна быть другом. Приведенная последней версия только использует объект t типа Time как единое целое, позволяя функции-члену работать с приватными значениями, поэтому эта функция не обязана быть другом. Однако, несмотря на это, все же имеет смысл сделать эту версию также другом. Самое главное, что это делает ее частью официального интерфейса класса. И второе, если вы позже столкнетесь с необходимостью иметь прямой доступ к приватным данным, то должны будете изменить только определение функции, не затрагивая прототип класса.

Совет

Если вы хотите перегрузить операцию для класса и хотите использовать ее с первым операндом, не являющимся объектом класса, вы можете использовать дружественную функцию для смены порядка операндов.

Общий вид друга: перегрузка операции <<

Очень удобным свойством классов является то, что вы можете перегрузить операцию << так, чтобы использовать ее с cout для отображения содержимого объектов. В некотором роде такая перегрузка немного сложнее, чем в предыдущих примерах, поэтому мы разработаем ее за два шага вместо одного.

Предположим, что trip — это объект типа Time. Для отображения значений Time мы используем метод Show(). Однако разве не было бы лучше, если бы можно было записать так:

```
cout << trip; // научить cout распознавать класс Time?
```

Это можно сделать, поскольку << — одна из операций C++, допускающих перегрузку. Фактически, она уже в большой степени перегружена. В своей базовой реализации операция << — это одна из операций C и C++, предназначенных для манипуляции битами; она сдвигает биты значения на один влево (см. приложение Д). Но класс ostream перегружает эту операцию, превращая ее в инструмент вывода. Вспомните, что cout — объект типа ostream, и он достаточно интеллектуален, чтобы распознавать все базовые типы C++. Это потому, что объявление класса ostream включает

перегруженное определение `operator<<()` для каждого из базовых типов. То есть, одно определение использует аргумент типа `int`, одно — `double` и так далее. Поэтому одним из способов научить `cout` распознавать объекты `Time` является добавление нового определения функции операции к объявлению класса `ostream`. Но это опасная идея — изменять заголовочный файл `iostream` и вносить путаницу в стандартный интерфейс. Гораздо лучше научить класс `Time` тому, как использовать `cout`.

Первая версия перегрузки операции <<

Чтобы научить класс `Time` использовать `cout`, вы должны воспользоваться дружественной функцией. Почему? Потому, что операция вроде

```
cout << trip;
```

работает с двумя объектами, причем объект класса `ostream` (а именно — `cout`) идет первым. Если вы используете функцию-член `Time` для перегрузки `<<`, то объект `Time` должен следовать первым, как это было в примере с перегрузкой операции `*` с помощью функции-члена. Это означает, что вам пришлось бы использовать операцию `<<` следующим образом:

```
trip << cout; // если бы operator<<() была функцией-членом Time
```

что само по себе сбивает с толку. Но, используя дружественную функцию, вы можете перегрузить операцию так:

```
void operator<<(ostream & os, const Time & t)
{
   os << t.hours << " часов, " << t.minutes << " минут";
}
```

что позволит вам использовать следующий код:

```
cout << trip;
```

чтобы напечатать данные в требуемом формате:

```
4 часов, 23 минут
```

Друг или не друг?

Новое объявление класса `Time` объявляет функцию `operator<<()` дружественной функцией класса `Time`. Однако эта функция, хотя и не "враждебна" классу `ostream`, однако не является другом по отношению к нему. Функция `operator<<()` принимает аргумент `ostream`, поэтому может показаться, что эта функция должна быть другом по отношению к обоим классам. Однако если вы посмотрите на код этой функции, то увидите, что она имеет прямой доступ к индивидуальным членам объекта `Time`, но использует объект `ostream` только как единое целое. Поскольку `operator<<()` обращается к приватным членам `Time` напрямую, она должна быть другом класса `Time`. Но так как она не имеет доступа к приватным членам класса `ostream`, то другом этого класса ей быть не нужно. И это замечательно, потому что значит, что вам не нужно изменять определение `ostream`.

Обратите внимание, что новое определение `operator<<()` принимает ссылку на объект `os` типа `ostream` в качестве первого аргумента. Обычно `os` является ссылкой на объект `cout`, как это делается в выражении `cout << trip`. Но вы можете использовать операцию с другим объектом `ostream` — в этом случае `os` будет ссылаться на него.

Что? Вы не знаете других объектов ostream?

Не забудьте о cerr, представленном в главе 10. Также вспомните, что в главе 6 были упомянуты объекты ofstream, которые можно применять для перенаправления вывода в файл. С помощью магии наследования (см. главу 13) объекты ofstream могут использовать методы ostream. То есть вы можете применить определение operator<<() для вывода данных Time в файлы — точно так же, как выводите на экран. Вы просто передаете в качестве первого аргумента соответствующим образом инициализированный объект ofstream вместо cout.

Вызов cout << trip должен использовать сам объект cout, а не его копию, поэтому функция передает этот объект по ссылке, а не по значению. То есть, выражение cerr << trip делает os псевдонимом cerr. Объект Time может быть передан по значению или по ссылке, так как обе формы делают его значения доступными функции операции. Опять-таки, передача по ссылке требует меньших затрат памяти и времени, чем по значению.

Вторая версия перегрузки операции <<

С представленной выше реализацией связана одна проблему. Выражения наподобие

```
cout << trip;
```

работают хорошо, но данная реализация не позволяет вам комбинировать перегруженную операцию << так, как это обычно делается при работе с cout:

```
cout << "Время поездки: " << trip << " (вторник)\n"; // так не получится
```

Чтобы понять, почему это выражение не работает, и что надо сделать, чтобы заставить его работать, сначала вы должны узнать немного больше о том, как обращаться с cout. Предположим, что есть следующие выражения:

```
int x = 5;
int y = 8;
cout << x << y;
```

C++ читает выражение вывода слева направо, подразумевая следующий эквивалент:

```
(cout << x) << y;
```

Операция <<, как она определена в iostream, принимает слева объект ostream. Ясно, что выражение cout << x удовлетворяет этому требованию, поскольку cout представляет собой объект ostream. Но оператор вывода также требует, чтобы все выражение (cout << x) было типа ostream, поскольку оно расположено слева от << y. Таким образом, класс ostream реализует функцию operator<<() так, что она возвращает объект ostream. В частности, в данном случае возвращает вызывающий объект cout. Таким образом, выражение (cout << x) само по себе является объектом ostream и может находиться слева от операции <<.

Тот же подход можно применить с дружественной функцией. Нужно просто изменить функцию operator<<() таким образом, чтобы она возвращала ссылку на объект ostream:

```
ostream & operator<<(ostream & os, const Time & t)
{
   os << t.hours << " часов, " << t.minutes << " минут";
   return os;
}
```

Обратите внимание, что типом возврата является ostream &. Вспомните, что это означает, что функция возвращает ссылку на объект ostream. Поскольку программа передает ссылку на объект функции в первом аргументе, общий эффект состоит в том, что функция возвращает тот самый объект, который ей передан. То есть оператор

```
cout << trip;
```

превращается в следующий вызов функции:

```
operator<<(cout, trip);
```

И такой вызов возвращает объект cout. Поэтому теперь работает такой оператор:

```
cout << "Время поездки: " << trip << " (вторник)\n"; // работает
```

Давайте разобьем это на отдельные шаги, чтобы увидеть, как это работает. Вот первый шаг:

```
cout << "Время поездки: "
```

вызывает конкретное определение ostream <<, которое отображает строку и возвращает объект cout, поэтому выражение cout << "Время поездки: " отображает строку и сразу заменяется типом возврата — cout. Это превращает исходный оператор в следующий:

```
cout << trip << " (вторник)\n";
```

Далее программа использует объявление << из Time для того, чтобы отобразить значения trip и снова возвратить объект cout. Это превращает оператор в следующий:

```
cout << " (вторник)\n";
```

Программа завершается использованием определения << из ostream для строк, чтобы отобразить результирующую строку.

Интересно, что эта версия operator<<() также может быть применена для вывода в файл:

```
#include <fstream>
...
ofstream fout;
fout.open("savetime.txt");
Time trip(12, 40);
fout << trip;
```

Последний оператор становится таким:

```
operator<<(fout, trip);
```

И, как показано в главе 8, свойства наследования классов позволяют ostream-ссылке обращаться как к объектам ostream, так и ofstream.

Совет

В общем случае, для перегрузки операции << с целью отображения объекта класса `c_name` вы можете использовать дружественную функцию, определенную в следующей форме:

```
ostream & operator<<(ostream & os, const c_name & obj)
{
    os << ... ; // отобразить содержимое объекта
    return os;
}
```

В листинге 11.10 показано модифицированное определение класса с включением двух дружественных функций `operator*()` и `operator<<()`. Первая из этих функций реализована здесь как встроенная, поскольку ее код очень короткий. (Когда определение одновременно является прототипом, как в этом случае, то применяется префикс `friend`.)

На память!

Ключевое слово `friend` используется только в прототипе, представленном в объявлении класса. Вы не указываете его в определении функции, если только оно не содержится в самом прототипе.

Листинг 11.10. mytime3.h

```
// mytime3.h -- класс Time с друзьями
#ifndef MYTIME3_H_
#define MYTIME3_H_
class Time
{
   private:
   int hours;
   int minutes;
public:
   Time();
   Time(int h, int m = 0);
   void AddMin(int m);
   void AddHr(int h);
   void Reset(int h = 0, int m = 0);
   Time operator+(const Time & t) const;
   Time operator-(const Time & t) const;
   Time operator*(const Time & t) const;
   friend Time operator*(double *, const Time & t)
      {return t * m;} // встроенное определение
   friend std::ostream & operator<<(std::ostream & os, const Time & t);
};
#endif
```

В листинге 11.11 представлен пересмотренный набор определений. Отметим опять же, что методы используют квалификатор `Time::`, в то время как дружественные функции — нет. Также отметим, что поскольку `mytime3.h` включает `iostream` и предоставляет объявление `using std::ostream`, включение файла `mytime3.h` в текст `mytime3.cpp` обеспечивает поддержку использования `ostream` в файле реализации.

Листинг 11.11. `mytime3.cpp`

```
// mytime3.cpp -- класс Time с друзьями
#include "mytime3.h"
Time::Time()
{
   hours = minutes = 0;
}
Time::Time(int h, int m )
{
   hours = h;
   minutes = m;
}
void Time::AddMin(int m)
{
   minutes += m;
   hours += minutes / 60;
   minutes %= 60;
}
void Time::AddHr(int h)
{
   hours += h;
}
void Time::Reset(int h, int m)
{
   hours = h;
   minutes = m;
}
Time Time::operator+(const Time & t) const
{
   Time sum;
   sum.minutes = minutes + t.minutes;
   sum.hours = hours + t.hours + sum.minutes / 60;
   sum.minutes %= 60;
   return sum;
}
Time Time::operator-(const Time & t) const
{
   Time diff;
   int tot1, tot2;
   tot1 = t.minutes + 60 * t.hours;
   tot2 = minutes + 60 * hours;
   diff.minutes = (tot2 - tot1) % 60;
   diff.hours = (tot2 - tot1) / 60;
   return diff;
}
Time Time::operator*(double mult) const
{
   Time result;
   long totalminutes = hours * mult * 60 + minutes * mult;
   result.hours = totalminutes / 60;
   result.minutes = totalminutes % 60;
   return result;
}
```

```
std::ostream & operator<<(std::ostream & os, const Time & t)
{
   os << t.hours << " часов, " << t.minutes << " минут";
   return os;
}
```

В листинге 11.2 показана простая программа. Технически `usetime3.cpp` не должен включать заголовочный файл `iostream`, поскольку `mytime3.h` уже включает его. Однако, будучи пользователем класса `Time`, вы не обязаны знать, какие заголовочные файлы включены в код класса, поэтому в вашей ответственности включать эти файлы, если ваш код в них нуждается.

Листинг 11.12. `usetime3.cpp`

```
// usetime3.cpp -- использование третьего наброска класса Time
// компилировать usetime3.cpp и mytime3.cpp вместе
#include <iostream>
#include "mytime3.h"
int main()
{
   using std::cout;
   using std::endl;
   Time aida(3, 35);
   Time tosca(2, 48);
   Time temp;
   cout << "Aida и Tosca:\n";
   cout << aida<<"; " << tosca << endl;
   temp = aida + tosca;             // operator+()
   cout << "Aida + Tosca: " << temp << endl;
   temp = aida* 1.17;               // функция-член operator*()
   cout << "Aida * 1.17: " << temp << endl;
   cout << "10 * Tosca: " << 10 * tosca << endl;
   return 0;
}
```

Вывод программы из листингов 11.10, 11.11 и 11.12 выглядит следующим образом:

```
Aida и Tosca:
3 часов, 35 минут; 2 hours, 48 минут
Aida + Tosca: 6 часов, 23 минут
Aida * 1.17: 4 часов, 11 минут
10 * Tosca: 28 часов, 0 минут
```

Перегруженные операции: сравнение функций-членов и функций-не-членов

При реализации перегрузки многих операций у вас есть выбор между функцией-членом и функцией-не-членом. Как правило, функции-не-члены являются дружественными, чтобы иметь доступ к приватным данным класса. Например, рассмотрим дополнительную операцию класса `Time`. Она имеет следующий прототип в объявлении класса `Time`:

```
Time operator+(const Time & t) const; // вариант функции-члена
```

Вместо этого класс может использовать такой прототип:

```
friend Time operator+(const Time & t1, const Time & t2);//вариант не-члена
```

Дополнительная операция требует два операнда. Для версии функции-члена один передается неявно — через указатель `this`, а второй — явно, как аргумент функции. В версии дружественной функции оба параметра передаются в аргументах функции.

На память!

Вариант перегруженной операции с функцией-не-членом требует столько формальных параметров, сколько операндов есть у данной операции. Вариант перегрузки с использованием функции-члена требует параметров на один меньше, поскольку он передается неявно — как вызывающий объект.

Оба эти прототипы соответствуют `T2 + T3`, где `T2` и `T3` — объекты типа `Time`. То есть, компилятор может преобразовать оператор

```
T1 = T2 + T3;
```

в любой из следующих:

```
T1 = T2.operator+(T3);    // функция-член
T1 = operator+(T2, T3);   // функция-не-член
```

Помните, что вы можете выбрать одну или другую форму при определении данной операции, но не обе сразу. Поскольку обеим формам соответствует одно и то же выражение, определение обеих форм одновременно ведет к неоднозначности и ошибкам компиляции.

Итак, какую же форму стоит выбрать? Для некоторых операций, как упоминалось ранее, функция-член — единственно правильный выбор. В других случаях особой разницы между ними нет.

Иногда, в зависимости от дизайна класса, вариант с использованием функции-не-члена предпочтителен, в частности, если определили для класса преобразование типа. Эта ситуация будет обсуждаться в разделе “Преобразования и друзья” в конце настоящей главы.

Дополнительные сведения о перегрузке: класс `Vector`

Давайте еще раз взглянем на дизайн класса, который использует перегрузку операций и друзей, а именно — на класс, представляющий векторы. Этот класс также иллюстрирует некоторые дополнительные аспекты дизайна, такие как включение двух различных способов описания одной и той же вещи в объект. Даже если вам не нужны векторы, вы можете использовать многие их приведенных здесь приемов в другом контексте. *Вектор*, как термин, используемый в инженерной практике и в физике, — это сущность, которая имеет величину и направление. Например, если вы толкаете что-нибудь, то эффект от этого действия зависит от того, насколько сильно вы толкаете, и в каком направлении. Толчок в одном направлении может удержать шатающуюся вазу, а в другом — подтолкнуть ее к падению и попросту разбить. Чтобы полностью описать движение своего автомобиля, вы должны указать как его

скорость, так и направление. Если вы докажете патрульно-постовой службе, что не превышали скорости, то ваш аргумент немного будет стоить, если вы ехали против разрешенного направления движения. (Иммунологи и компьютерные ученые могут использовать термин *вектор* в другом смысле. Мы пока проигнорируем их, по крайней мере, до главы 16, в которое рассматривается шаблонный класс `vector`.) Следующий выделенный фрагмент расскажет вам больше о векторах, но полное их понимание вам пока понадобится для того, чтобы разобраться с новыми аспектами C++, представленными в примерах.

Векторы

Представьте, что вы — рабочая пчела и открыли чудесный источник нектара. Вы возвращаетесь в улей и объявляете, что нашли нектар в 100 метрах от него. "Недостаточно информации" — жужжат остальные пчелы. "Ты должна сообщить нам и направление!". Ваш ответ: "30 градусов к северу от направления на солнце". Зная расстояние (величину) и направление, другие пчелы смогут достичь это сладкого места. Пчелы знакомы с понятием вектора.

Многие вещи описываются величиной и направлением. Например, эффект от толчка зависит как от его силы, так и направления. Перемещение объекта по экрану компьютера описывается расстоянием и направлением. Вы можете описать такие вещи с помощью векторов. Например, можно описать перемещение объекта по экрану вектором, который визуализируется стрелкой, соединяющей исходное положение с конечным. Длина вектора — это величина, описывающая, насколько далеко должна быть перемещена точка. Ориентация стрелки описывает направление (рис. 11.1). Вектор, описывающий это, называется *вектором смещения*.

Теперь представьте, что вы — Лханаппа (Lhanappa), великий охотник на мамонтов. Разведчики сообщают, что видели мамонта в 14.1 километрах к северо-западу. Но поскольку дует юго-восточный ветер, вы не можете приближаться к нему с юго-востока. Поэтому вы идете 10 километров на запад, потом — 10 километров на север, приближаясь к нему с юга. Вы знаете, что два этих вектора смещения приведут вас в ту же точку, как и один 14.1-километровый вектор, указывающий на северо-восток. Лханаппа, великий охотник на мамонтов, тоже знает, как складывать векторы.

Сложение двух векторов имеет простую геометрическую интерпретацию. Сначала нарисуйте один вектор. Затем нарисуйте второй, начав его из конца стрелки первого. И, наконец, нарисуйте вектор из начальной точки первого вектора в конец второго. Этот третий вектор представляет собой сумму первых двух (рис. 11.2). Следует отметить, что длина результирующего вектора может быть меньше, чем простая сумма длин составляющих.

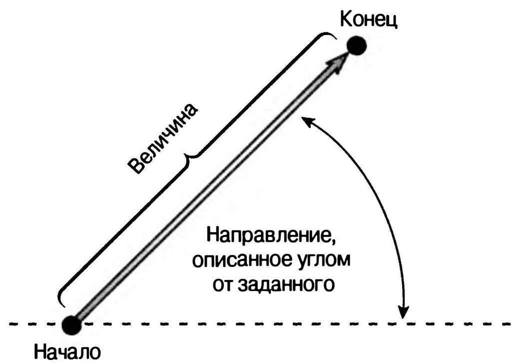


Рис. 11.1. Описание смещения с помощью вектора

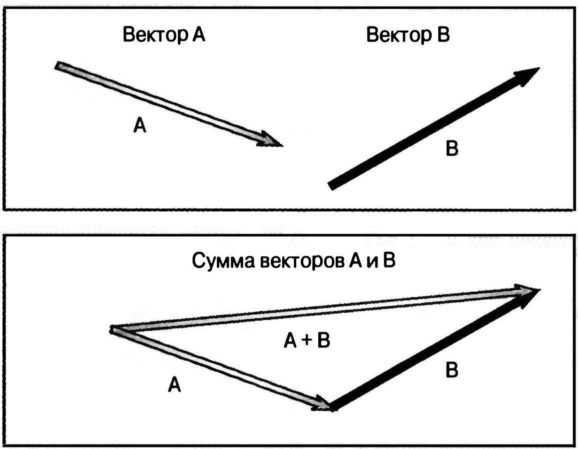


Рис. 11.2. Сложение двух векторов

Векторы — это естественный выбор для реализации перегруженных операций. Во-первых, вы не можете представить вектор единственным числом, поэтому имеет смысл создать класс, представляющий векторы. Во-вторых, векторы имеют аналоги обычных арифметических операций, таких как сложение и вычитание. Эта параллель предполагает перегрузку соответствующих операций так, чтобы их можно было использовать с векторами.

Дабы не усложнять, в настоящем разделе мы реализуем двумерный вектор вроде экранного смещения вместо трехмерного вектора — такого, которым может быть представлено движение вертолета или гимнаста в воздухе. Вам понадобится два числа для описания двумерного вектора, но у вас есть выбор — что именно будут означать эти два числа:

- Можно представить вектор длиной и направлением.
- Можно представить вектор парой координат.

Последний вариант можно рассматривать как горизонтальный вектор (компонент x) и вертикальный вектор (компонент y), который добавлен к первому. Например, вы можете описать движение перемещением на 30 единиц вправо и на 40 вверх (рис. 11.3). Это перемещение помещает точку в то же положение, как и перемещение на 50 позиций под углом 53.1 градуса. Таким образом, вектор с величиной 50 и углом 53.1 градуса эквивалентен вектору, имеющему горизонтальный компонент 30 и вертикальный компонент 40. Что существенно для вектора смещения — это знать начальную точку и конечную, а не путь перемещения из первой во вторую. Этот выбор в представлении соответствует описанному в главе 7 преобразованию прямоугольных и полярных координат.

Иногда одна форма удобнее, иногда — другая, поэтому включим в описание класса оба представления. (См. врезку “Множественные представления и классы” ниже в этой главе.) Спроектируем класс так, что если вы измените одно из представлений вектора, то второе будет изменяться автоматически. Возможность обеспечить такое интеллектуальное поведение объекта — еще одно преимущество классов C++. В листинге 11.3 показано объявление класса. Чтобы освежить ваше представление о пространствах имен, в этом листинге объявление класса помещено внутрь пространства имен `VECTOR`.

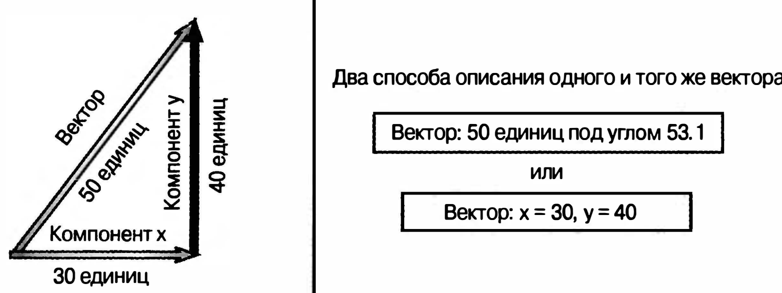


Рис. 11.3. Компоненты x и y вектора

Замечание по совместимости

Если ваша система не поддерживает пространства имен, вы можете удалить следующие строки:

```
namespace VECTOR
{
```

и

```
} // конец пространства имен VECTOR
```

Листинг 11.13. vect.h

```
// vect.h -- класс Vector с операцией << и членом состояния
#ifndef VECTOR_H_
#define VECTOR_H_
#include <iostream>
namespace VECTOR
{
   class Vector
   {
   private:
     double x;       // горизонтальное смещение
     double y;       // вертикальное смещение
     double mag;     // длина вектора
     double ang;     // направление вектора
     char mode;      // 'r' = прямоугольные, 'p' = полярные
     // приватные методы для установки значений
     void set_mag();
     void set_ang();
     void set_x();
     void set_y();
   public:
     Vector();
     Vector(double n1, double n2, char form = 'r');
     void set(double n1, double n2, char form = 'r');
     ~Vector();
     double xval() const {return x;}        // возвращает значение x
     double yval() const {return y;}        // возвращает значение y
     double magval() const {return mag;}    // возвращает величину
     double angval() const {return ang;}    // возвращает угол
```

```
    void polar_mode();   // устанавливает режим 'p'
    void rect_mode();    // устанавливает режим 'r'
    // перегрузка операций
    Vector operator+(const Vector & b) const;
    Vector operator-(const Vector & b) const;
    Vector operator-() const;
    Vector operator*(double n) const;
    // друзья
    friend Vector operator*(double n, const Vector & a);
    friend std::ostream & operator<<(std::ostream & os, const Vector & v);
  };
} // конец пространства имен VECTOR
#endif
```

Обратите внимание, что четыре функции в листинге 11.3, которые возвращают значения компонентов, определены в объявлении класса. Это автоматически делает их встроенными. Ни одна из них не должна изменять данные объекта, поэтому они объявлены с модификатором `const`. Как вы можете вспомнить из главы 10, это синтаксис для объявления функций, не модифицирующих объект, к которому они имеют непосредственный доступ.

В листинге 11.4 показаны все методы и дружественные функции, объявленные в листинге 11.3. Код в листинге использует открытую природу пространств имен для добавления объявлений методов к пространству имен `VECTOR`. Отметим, что функции-конструкторы и функция `set()` устанавливают значения как для прямоугольного, так и для полярного представления вектора. Таким образом, оба набора значений становятся доступными немедленно, без дополнительных вычислений. Кроме того, как упоминалось в главах 4 и 7, встроенные математические функции C++ работают с углами в радианах, поэтому функции, преобразующие в градусы и обратно, встроены в методы. Реализация класса `Vector` скрывает от пользователя такие вещи, как преобразования из полярных координат в прямоугольные и радианов в градусы. Все, что требуется знать пользователю, это то, что класс использует углы в градусах и то, что он представляет вектор в двух эквивалентных представлениях.

Листинг 11.14. vect.cpp

```
// vect.cpp -- методы класса Vector
#include <cmath>
#include "vect.h" // включает <iostream>
using std::sin;
using std::cos;
using std::atan2;
using std::cout;
namespace VECTOR
{
  const double Rad_to_deg = 57.2957795130823;
  // приватные методы
  // вычисляет величину от x и y
  void Vector::set_mag()
  {
    mag = sqrt(x * x + y * y);
  }
```

```
void Vector::set_ang()
{
  if (x == 0.0 && y == 0.0)
     ang = 0.0;
  else
     ang = atan2(y, x);
}
// устанавливает x по полярным координатам
void Vector::set_x()
{
  x = mag * cos(ang);
}
// устанавливает y по полярным координатам
void Vector::set_y()
{
  y = mag * sin(ang);
}
// общедоступные методы
Vector::Vector() // конструктор по умолчанию
{
  x = y = mag = ang = 0.0;
  mode = 'r';
}
//конструирует вектор по прямоугольным координатам, если установлен режим r
// (по умолчанию) или по полярным координатам, если установлен режим p
Vector::Vector(double n1, double n2, char form)
{
  mode = form;
  if (form == 'r')
  {
     x = n1;
     y = n2;
     set_mag();
     set_ang();
  }
  else if (form == 'p')
  {
     mag = n1;
     ang = n2 / Rad_to_deg;
     set_x();
     set_y();
  }
  else
  {
     cout << "Неправильный 3-й аргумент Vector() -- ";
     cout << "вектор установлен в 0\n";
     x = y = mag = ang = 0.0;
     mode = 'r';
  }
}
//устанавливает вектор по прямоугольным координатам, если установлен режим r
// (по умолчанию) или по полярным координатам, если установлен режим p
void Vector:: set(double n1, double n2, char form)
```

```
{
  mode = form;
  if (form == 'r')
  {
    x = n1;
    y = n2;
    set_mag();
    set_ang();
  }
  else if (form == 'p')
  {
    mag = n1;
    ang = n2 / Rad_to_deg;
    set_x();
    set_y();
  }
  else
  {
    cout << "Неправильный 3-й аргумент Vector() -- ";
    cout << "вектор установлен в 0\n";
    x = y = mag = ang = 0.0;
    mode = 'r';
  }
}
Vector::~Vector() // деструктор
{
}
void Vector::polar_mode() // установить полярный режим
{
  mode = 'p';
}
void Vector::rect_mode() // установить режим прямоугольных координат
{
  mode = 'r';
}
// перегрузка операций
// сложение двух векторов
Vector Vector::operator+(const Vector & b) const
{
  return Vector(x + b.x, y + b.y);
}
// вычитание вектора b из a
Vector Vector::operator-(const Vector & b) const
{
  return Vector(x - b.x, y - b.y);
}
// смена знака вектора
Vector Vector::operator-() const
{
  return Vector(-x, -y);
}
// умножение вектора на n
Vector Vector::operator*(double n) const
```

```
    {
      return Vector(n * x, n * y);
    }
    // дружественные методы
    // умножение n на вектор a
    Vector operator*(double n, const Vector & a)
    {
      return a * n;
    }
    // отображает прямоугольные координаты, если установлен режим r,
    // иначе отображает полярные координаты, если установлен режим p
    std::ostream & operator<<(std::ostream & os, const Vector & v)
    {
      if (v.mode == 'r')
         os << "(x,y) = (" << v.x << ", " << v.y << ")";
      else if (v.mode == 'p')
      {
         os << "(m,a) = (" << v.mag << ", "
         << v.ang * Rad_to_deg << ")";
      }
      else
         os << "Режим объекта вектор неверен";
      return os;
    }
} // конец пространства имен VECTOR
```

Вы можете спроектировать вектор по-разному. Например, объект может хранить прямоугольные координаты и не хранить полярные. В этом случае вычисление полярных координат может быть помещено в методы `magval()` и `angval()`. Для приложений, в которых преобразование выполняется нечасто, такой дизайн может быть более эффективным. Кроме того, метод `set()` в этом случае не нужен.

Предположим, что `shove` — это объект типа `Vector`, и есть такой код:

```
shove.set(100, 300);
```

Тот же результат можно получить, используя вместо этого следующий конструктор:

```
shove = Vector(100, 300);
```

Однако, метод `set()` изменяет содержимое `shove` непосредственно, в то время как применение конструктора добавляет несколько дополнительных шагов по созданию временного объекта и присваиванию его `shove`.

Такое проектное решение следует традиции ООП — иметь интерфейс класса, сконцентрированный на сущностях (абстрактную модель), при этом скрывая детали. Таким образом, когда вы используете класс `Vector`, то можете думать об основных свойствах вектора, таких как способность представлять смещение и возможность складывать два вектора. Когда вы описываете вектор в компонентной нотации либо в нотации величины и направления, вторая нотация синхронизируется автоматически, и вы можете в любой момент устанавливать значение вектора и опрашивать его в любом из двух форматов.

Ниже мы рассмотрим некоторые из свойств класса `Vector` подробнее.

Замечание по совместимости

Некоторые системы все еще используют `math.h` вместо `cmath`. Кроме того, некоторые системы C++ не выполняют автоматический поиск библиотеки `math`. Например, некоторые системы Unix требуют, чтобы вы поступали следующим образом:

```
$ CC source_file(s) -lm
```

Опция `-lm` инструктирует компоновщик, что нужно искать библиотеку `math`. Поэтому, когда вы компилируете программы, использующие класс `Vector`, и получаете сообщения о неразрешенных ссылках, то вы должны попробовать добавить опцию `-lm` или посмотреть, не требует ли ваша система чего-то подобного.

Как и в заголовочном файле, если ваша система не поддерживает пространств имен, вы можете удалить следующие строки:

```
namespace VECTOR
{
```

и

```
} // конец пространства имен VECTOR
```

Использование члена состояния

Класс `Vector` сохраняет и прямоугольные, и полярные координаты вектора. Он использует переменную-член по имени `mode`, указывающую, какую форму конструктора, метода `set()` и перегруженной операции `<<` следует использовать. Значение `'r'` означает представление в прямоугольных координатах, а `'p'` — в полярных. Такая переменная-член называется *членом состояния*, поскольку описывает состояние объекта. Чтобы увидеть, что это означает, давайте посмотрим на код конструктора:

```
Vector::Vector(double n1, double n2, char form)
{
   mode = form;
   if (form == 'r')
   {
      x = n1;
      y = n2;
      set_mag();
      set_ang();
   }
   else if (form == 'p')
   {
      mag = n1;
      ang = n2 / Rad_to_deg;
      set_x();
      set_y();
   }
   else
   {
      cout << "Неправильный 3-й аргумент Vector() -- ";
      cout << "вектор установлен в 0\n";
      x = y = mag = ang = 0.0;
      mode = 'r';
   }
}
```

Если третий аргумент имеет значение 'r' или же он опущен (прототип присваивает значение 'r' по умолчанию), то входные параметры интерпретируются как прямоугольные координаты, в то время как значение 'p' заставляет интерпретировать их как полярные координаты:

```
Vector folly(3.0, 4.0);            // установить x = 3, y = 4
Vector foolery(20.0, 30.0, 'p');   // установить mag = 20, ang = 30
```

Обратите внимание, что конструктор использует приватные методы set_mag() и set_ang() для установки величины и угла, если переданы значения x и y, и методы set_x() и set_y() для установки значений x и y, если переданы величина и угол. Также отметим, что конструктор выдает предупреждающее сообщение и устанавливает статус в 'r', если указано что-либо отличное от 'r' и 'p'.

Аналогично, функция operator<<() использует режим для определения того, какие значения следует отображать:

```
// отображает прямоугольные координаты, если режим установлен в r,
// иначе отображает полярные координаты, если установлен режим p
ostream & operator<<(ostream & os, const Vector & v)
{
   if (v.mode == 'r')
      os << "(x,y) = (" << v.x << ", " << v.y << ")";
   else if (v.mode == 'p')
   {
      os << "(m,a) = (" << v.mag << ", "
         << v.ang * Rad_to_deg << ")";
   }
   else
      os << "Режим объекта вектора неверен";
      return os;
}
```

Различные методы, которые могут устанавливать режим, заботятся о том, чтобы в качестве допустимых значений принимать только 'r' и 'p', поэтому последняя конструкция else в этой функции никогда не должна быть достигнута. Однако, несмотря на это, проверять все же стоит: такая проверка во многих случаях поможет перехватить некоторые неожиданные программные ошибки.

Множественные представления и классы

Величины, которые имеют различное, но эквивалентное представление, встречаются часто. Например, вы можете измерять расход топлива в милях на галлон, как это делается в Соединенных Штатах, или же в литрах на 100 километров, как это принято в Европе. Вы можете представлять число в строковой или числовой форме, и вы можете представлять уровень интеллекта в IQ или в "килотупицах". Классы хорошо приспосабливают себя для представления разных аспектов существования в одном объекте. Во-первых, вы можете сохранять множество представлений в одном объекте. Во-вторых, вы можете написать функции класса таким образом, что присваиваемые значения для одного представления автоматически присвоят значения для других представлений. Например, метод set_by_polar() в классе Vector устанавливает значения mag и ang в значения аргументов функции, но также устанавливает значения членов x и y. Управляя преобразованием внутри себя, класс поможет вам думать о величине в терминах его природы, а не в терминах его представления.

Перегрузка арифметических операций для класса Vector

Сложение двух векторов очень просто, если вы используете координаты x и y. Вы просто складываете между собой попарно два компонента x и два компонента y, чтобы получить результирующие значения x и y. Исходя из этого, вы можете предположить, что код может выглядеть так:

```
Vector Vector::operator+(const Vector & b) const
{
   Vector sum;
   sum.x = x + b.x;
   sum.y = y + b.y;
   return sum; // незавершенная версия
}
```

И все будет в порядке, если объект хранит только компоненты x и y. К сожалению, эта версия кода не поддерживает установку полярных координат. Вы можете решить эту проблему, добавив несколько строк:

```
Vector Vector::operator+(const Vector & b) const
{
   Vector sum;
   sum.x = x + b.x;
   sum.y = y + b.y;
   sum.set_ang(sum.x, sum.y);
   sum.set_mag(sum.x, sum.y);
   return sum; // ненужный дубляж
}
```

Однако проще и удобнее будет позволить выполнять эту работу конструктору:

```
Vector Vector::operator+(const Vector & b) const
{
   return Vector(x + b.x, y + b.y); // возвращает сконструированный Vector
}
```

Здесь код использует конструктор Vector для установки значений компонентов x и y. Конструктор затем создает новый безымянный объект, и функция возвращает этот объект. Таким образом, вы гарантируете, что новый объект создается в соответствии со стандартными правилами, заложенными в конструкторе.

Совет

Если метод должен вычислять новый объект класса, вы должны посмотреть, нельзя ли воспользоваться конструктором класса, чтобы выполнить эту работу. Это не только избавит вас от забот, но также гарантирует, что новый объект будет сконструирован правильным образом.

Умножение

В визуальных терминах умножение вектора на число делает его длиннее или короче в заданное число раз. Поэтому умножение вектора на 3 создает вектор втрое большей длины, чем исходный, но имеющий то же направление. Достаточно просто транслировать этот образ в то, как класс Vector представляет вектор. В терминах

полярных координат вы умножаете длину и не изменяете угол. В терминах прямоугольных координат умножение вектора на число означает раздельное умножение каждого из компонентов — x и y — на это же число. То есть, если вектор имеет эти компоненты, равные 5 и 12, умножение их на 3 дает им значения 15 и 36. И вот что перегруженная операция умножения делает:

```
Vector Vector::operator*(double n) const
{
   return Vector(n * x, n * y);
}
```

Как и в случае с перегруженным сложением, этот код позволяет конструктору создать корректный объект типа `Vector` на основе новых компонентов x и y. Приведенный вариант умножает значение `Vector` на множитель типа `double`. Так же, как и в примере с `Time`, вы можете использовать встроенную дружественную функцию для умножения `Vector` на значение типа `double`:

```
Vector operator*(double n, const Vector & a) // дружественная функция
{
   return a * n;  // преобразует умножение double на Vector
                  // в умножение Vector на double
}
```

Дополнительное усовершенствование: перегрузка перегруженной операции

В обычном C++ операция "минус" (–) уже имеет два значения. Первое — когда используется с двумя операндами как операция вычитания. Операция вычитания — пример *бинарной операции*, поскольку она имеет дело с двумя операндами. Второй — когда используется с одним операндом, как в `-x`; это операция смены знака. Такая форма называется *унарной операцией*, что означает, что она имеет только один операнд. Обе операции — вычитание и смена знака — также имеют смысл для вектора, поэтому в классе `Vector` предусмотрены они обе.

Чтобы вычесть вектор B из вектора A, вы просто вычитаете компоненты попарно, поэтому определение перегруженного вычитания достаточно похоже на сложение:

```
Vector operator-(const Vector & b) const;                // прототип
Vector Vector::operator-(const Vector & b) const         // определение
{
   return Vector(x - b.x, y - b.y); // возвращает сконструированный Vector
}
```

Здесь важно правильно указать порядок. Рассмотрим следующий оператор:

```
diff = v1 - v2;
```

Он преобразуется в вызов функции-члена:

```
diff = v1.operator-(v2);
```

Это означает, что вектор, который передается как явный аргумент, вычитается из вектора, переданного неявным аргументом, поэтому вы должны использовать `x - b.x`, а не `b.x - x`.

Далее рассмотрим операцию унарного “минуса”, которая принимает только один операнд. Применение этой операции к обычному числу, как в -x, меняет знак операнда. То есть, применение этой операции к вектору должно менять знак каждого компонента. Точнее говоря, функция должна возвращать новый вектор, противоположный исходному. (В терминах полярных координат отрицание оставляет величину вектора без изменений, но меняет направление на противоположное.) Ниже представлен прототип и определение перегруженного отрицания:

```
Vector operator-() const;
Vector Vector::operator-() const
{
   return Vector (-x, -y);
}
```

Обратите внимание, что теперь мы имеем два разных определения operator-(). И это нормально, поскольку эти два определения имеют разные сигнатуры. Вы можете создать обе версии операции “минус”, унарную и бинарную, поскольку в C++ изначально представлены две версии. Операция, которая имеет только бинарную форму (вроде деления (/)), может быть перегружена только как бинарная.

На память!

Поскольку перегрузка операций реализована в виде функций, вы можете перегружать одну и ту же операцию много раз, до тех пор, пока каждая функция операции имеет отличную от других сигнатуру, и до тех пор, пока каждая функция операции имеет то же количество операндов, что и соответствующая встроенная операция C++.

Комментарии к реализации

Реализация, описанная в предыдущих разделах, сохраняет и прямоугольные, и полярные координаты вектора в объекте Vector. Однако общедоступный интерфейс не зависит от этого факта. Все, что нужно знать об интерфейсах — это то, что оба представления могут быть отображены и индивидуальные значения возвращены. Внутренняя реализация может отличаться. Как упоминалось ранее, объект может сохранять только компоненты x и y. Тогда, скажем, метод magval(), который возвращает значение величины вектора, может вычислять это значение на основании величин x и y вместо того, чтобы возвращать величину, хранящуюся в объекте в виде отдельного компонента. Такой подход изменяет реализацию, но оставляет интерфейс неизменным. Такое отделение интерфейса от реализации — одна из целей ООП. Оно позволяет тонко настроить реализацию без изменения кода в программах, которые используют класс.

Оба варианта реализации обладают своими преимуществами и недостатками. Сохранение данных означает, что объект занимает больше места в памяти и что код должен заботиться о синхронном обновлении и прямоугольного, и полярного представлений всякий раз, когда объект Vector изменяется. Но поиск данных выполняется быстрее. Если приложение часто нуждается в доступе к обоим представлениям вектора, то предпочтительна реализация, приведенная в примере. Если же полярное представление требуется только изредка, лучше остановиться на втором представлении. Вы можете выбрать одну реализацию для одной программы и другую — для другой, используя один и тот же интерфейс для обоих.

Использование класса Vector в программе "Случайная прогулка"

В листинге 11.15 представлена короткая программа, которая использует усовершенствованный класс Vector. Она моделирует известную проблему "прогулки пьяницы". Тем не менее, сейчас пьяные воспринимаются как люди, имеющие серьезные проблемы со здоровьем, а не как предмет для насмешек, поэтому обычно теперь это называется проблемой "случайной прогулки". Идея заключается в том, что вы помещаете кого-то у фонарного столба. Он начинает двигаться, но направление с каждым шагом случайным образом изменяется по отношению к направлению предыдущего шага. Один способ сформулировать проблему: сколько шагов нужно сделать этому персонажу, чтобы удалиться, скажем, на 50 шагов от столба? В терминологии векторов это означает сложение случайно ориентированных векторов до тех пор, пока сумма не превысит 50 шагов.

Код в листинге 11.15 позволяет выбрать заданное расстояние, которое нужно преодолеть, и длину шага. Он поддерживает меняющуюся сумму, которая представляет позицию после каждого шага (в виде вектора) и сообщает количество шагов, необходимых для преодоления заданной дистанции в соответствии с текущим положением (в обоих форматах). Как вы увидите, перемещение персонажа достаточно неэффективно. Путешествие из 1000 шагов, по 2 фута каждый, может увести его всего на 50 футов от начальной точки. Программа делит общую преодоленную дистанцию (в данном случае, 50 футов) на число шагов, чтобы измерить степень неэффективности. Все случайные изменения направления делают среднюю длину перемещения значительно меньше, чем один шаг. Для случайного выбора направления программа использует стандартные библиотечные функции rand(), srand() и time(), описанные в следующем разделе "Замечания по программе". Убедитесь, что компилируете листинг 11.14 вместе с листингом 11.5.

Листинг 11.15. randomwalk.cpp

```
// randomwalk.cpp -- использование класса Vector
// компилировать вместе с файлом vect.cpp
#include <iostream>
#include <cstdlib>  // прототипы rand(), srand()
#include <ctime>    // прототип time()
#include "vect.h"
int main()
{
   using namespace std;
   using VECTOR::Vector;
   srand(time(0)); // инициировать генератор случайных чисел
   double direction;
   Vector step;
   Vector result(0.0, 0.0);
   unsigned long steps = 0;
   double target;
   double dstep;
   cout << "Введите дистанцию (q для выхода): ";
   while (cin >> target)
   {
     cout << "Введите длину шага: ";
```

```
    if (!(cin >> dstep))
      break;
    while (result.magval() < target)
    {
      direction = rand() % 360;
      step.set(dstep, direction, 'p');
      result = result + step;
      steps++;
    }
    cout << "После " << steps << " шагов субъект "
      "имеет следующее местоположение:\n";
    cout << result << endl;
    result.polar_mode();
    cout << " или\n" << result << endl;
    cout << "Средняя дистанция на один шаг = "
        << result.magval()/steps << endl;
    steps = 0;
    result.set(0.0, 0.0);
    cout << "Введите дистанцию (q для выхода): ";
  }
  cout << "Не злоупотребляйте!\n";
  return 0;
}
```

Замечание по совместимости

Вместо `cstdlib` вы можете использовать `stdlib.h`, а вместо `ctime` — `time.h`. Если ваша система не поддерживает пространства имен, опустите следующую строку:

```
using VECTOR::Vector;
```

Ниже показан пример выполнения программы из листингов 11.13, 11.14 и 11.15:

```
Введите дистанцию (q для выхода): 50
Введите длину шага: 2
После 253 шагов субъект имеет следующее местоположение:
(x,y) = (46.1512, 20.4902)
 или
(m,a) = (50.495, 23.9402)
Средняя дистанция на один шаг = 0.199587
Введите дистанцию (q для выхода): 50
Введите длину шага: 2
После 951 шагов субъект имеет следующее местоположение:
(x,y) = (-21.9577, 45.3019)
 или
(m,a) = (50.3429, 115.8593)
Средняя дистанция на один шаг = 0.0529362
Введите дистанцию (q для выхода): 50
Введите длину шага: 1
После 1716 шагов субъект имеет следующее местоположение:
(x,y) = (40.0164, 31.1244)
 или
(m,a) = (50.6956, 37.8755)
Средняя дистанция на один шаг = 0.0295429
Введите дистанцию (q для выхода): q
Не злоупотребляйте!
```

Случайная природа процесса порождает различные вариации от попытки к попытке, даже если начальные условия одинаковы. В среднем, однако, деление размера шага учетверяет число шагов, необходимых, чтобы преодолеть дистанцию. Теория вероятности предполагает, что в среднем количество шагов (N) длиной s, которое понадобится для преодоления суммарного расстояния D, вычисляется по следующей формуле:

$$N = (D/s)^2$$

Но это средняя величина, которая фактически колеблется от попытки к попытке. Например, 1000 попыток преодоления 50 футов при 2-футовом шаге в среднем дают 636 шагов (что близко к теоретической величине 625) чтобы пройти это расстояние, но оно колеблется в диапазоне от 91 до 3951. Соответственно, 1000 попыток преодолеть 50 футов при 1-футовом шаге дает в среднем 2557 шагов (близко к теоретическому числу 2500) с диапазоном от 345 до 10882. Поэтому, если вам придется двигаться случайным образом, знайте, что лучше идти большими шагами. Вы никак не можете повлиять на выбор направления, но, по крайней мере, уйдете дальше.

Замечания по программе

Во-первых, обратите внимание, насколько безболезненно применение пространства имен `VECTOR` в листинге 11.15. Использование объявления

```
using VECTOR::Vector;
```

помещает имя класса `Vector` в область видимости. Поскольку все методы класса `Vector` относятся к области видимости в пределах класса, импортирование имени класса также делает все методы класса `Vector` доступными, без необходимости применения дополнительных объявлений `using`.

Далее поговорим о случайных числах. Стандартная библиотека ANSI C, которая поставляется вместе с C++, включает в себя функцию `rand()`, возвращающую случайное целое число в диапазоне от нуля до некоего зависящего от реализации значения. Ваша программа моделирования случайной прогулки использует операцию взятия модуля для выбора угла направления шага в диапазоне от 0 до 359. Функция `rand()` работает, применяя свой алгоритм к начальному инициирующему значению, чтобы получить очередное случайное число. Это число используется как инициатор при следующем вызове функции и так далее.

На самом деле получается ряд псевдослучайных чисел, поскольку 10 последовательных вызовов обычно генерируют один и тот же набор 10 случайных чисел. (Конкретные значения зависят от реализации.) Однако функция `srand()` позволяет изменить начальное значение и, таким образом, получить другую последовательность случайных чисел. Эта программа использует значение, возвращенное `time(0)`, для инициализации генератора. Функция `time(0)` возвращает текущее календарное время, часто реализованное в виде количества секунд, прошедших с определенной специфической даты. (Наиболее часто `time()` принимает адрес переменной типа `time_t`, помещает в нее значение времени и также возвращает ее. Использование `0` в качестве аргумента-адреса исключает потребность в переменной типа `time_t`.) Таким образом, оператор

```
srand(time(0));
```

устанавливает разное начальное значение при каждом запуске программы, обеспечивая еще более случайную последовательность случайных чисел. Заголовочный файл cstdlib (бывший stdlib.h) содержит прототипы функций srand() и rand(), в то время как ctime (бывший time.h) предлагает прототип time().

Программа использует вектор result для того, чтобы отслеживать случайное движение. На каждом шаге вложенного цикла программа устанавливает для вектора step новое направление и добавляет его к текущему значению вектора result. Когда величина вектора result превысит заданную дистанцию, цикл прерывается.

Устанавливая режим вектора, программа отображает конечную позицию в терминах прямоугольных и полярных координат.

Кстати, оператор

```
result = result + step;
```

переключает result в режим 'r', независимо от начальных режимов result и step. И вот почему. Во-первых, функция операции сложения создает и возвращает новый вектор, представляющий сумму двух аргументов. Функция создает вектор, используя конструктор по умолчанию, который порождает вектор в режиме 'r'. Поэтому вектор, присваиваемый result, пребывает в режиме 'r'. По умолчанию присваивание каждой переменной-члена выполняется индивидуально, поэтому result.mode получает значение 'r'. Если вы предпочитаете какое-то другое поведение, например, чтобы result сохранял свой предыдущий режим, можете перегрузить операцию присваивания по умолчанию, определив для класса функцию операции присваивания. В главе 12 приведен пример этого.

Сохранять позицию в файле очень просто. Во-первых, вы включаете в текст <fstream>, объявляете объект ofstream и ассоциируете его с файлом:

```
#include <fstream>
...
ofstream fout;
fout.open("thewalk.txt");
```

Затем внутри цикла, вычисляющего результат, вставляете что-то вроде такого:

```
fout << result << endl;
```

Это вызывает дружественную функцию operator<<(fout, result), подставляя первым аргументом ссылку на fout, таким образом, направляя вывод в файл. Вы также можете использовать fout для того, чтобы выводить другую информацию в этот файл, например, такую как суммарную информацию, отображаемую в cout.

Автоматическое преобразование и приведение типов в классах

Следующим пунктом в нашем обсуждении классов будет преобразование типов. Мы рассмотрим, как C++ управляет преобразованием во встроенные типы и обратно. Чтобы приступить к этому, рассмотрим вначале, как C++ выполняет преобразование встроенных типов. Когда вы пишете оператор, который присваивает значение одного стандартного типа другому стандартному типу, C++ автоматически преобразует присваиваемое значение в тип принимающей переменной, предполагая, что эти два

типа совместимы. Например, все следующие операторы генерируют преобразования числовых типов:

```
long count = 8;   // int-значение 8 преобразуется в тип long
double time = 11; // int-значение 11 преобразуется в тип double
int side = 3.33;  // double-значение 3.33 преобразуется в тип int (3)
```

Все эти присваивания работают, потому что C++ распознает, что все эти разнообразные типы представляют одну базовую сущность — число, и поскольку C++ включает встроенные правила для выполнения преобразования между ними. Однако, вспомнив главу 3, мы обнаружим, что при этом преобразовании может быть потеряна точность. Например, присваивание значения `3.33` переменной типа `int` имеет побочный результат, заключающийся в потере дробной части `0.33`.

Язык C++ не преобразует между собой типы, которые не совместимы. Например, оператор

```
int * p = 10;  // конфликт типов
```

завершится сбоем, потому что слева от знака равенства находится указатель, а справа — число. И даже несмотря на то, что компьютер имеет внутреннее представление адресов в виде чисел, все-таки адреса и числа являются концептуально различными понятиями. Например, вы не можете возводить указатель в квадрат. Однако когда автоматическое преобразование не срабатывает, вы можете использовать приведение типов:

```
int * p = (int *) 10; // нормально, p и (int *) 10 — оба указатели
```

Это устанавливает указатель равным адресу `10`, выполнив приведение `10` к типу указателя на `int` (то есть, к типу `int *`).

Вы можете определить класс в достаточной мере связанным с базовым типом, чтобы имело смысл преобразовать один в другой. В этом случае вы можете указать C++, как выполнять преобразование автоматически либо, возможно, посредством явного приведения типа. Чтобы посмотреть, как это работает, вы можете переписать программу "стоуны в фунты" из главы 3 в форме классов. Во-первых, нужно спроектировать соответствующий тип. Главное, что надо представить одно и то же понятие (вес) двумя способами (в фунтах и стоунах). Класс предлагает отличный способ включить два представления одной концепции в одну сущность. Таким образом, имеет смысл поместить оба представления веса в один класс и затем предусмотреть методы для выражения веса в различной форме. В листинге 11.6 показан заголовок этого класса.

Листинг 11.16. `stonewt.h`

```
// stonewt.h -- определение класса Stonewt
#ifndef STONEWT_H_
#define STONEWT_H_
class Stonewt
{
private:
   enum {Lbs_per_stn = 14};  // фунтов на стоун
   int stone;                // всего стоунов
   double pds_left;          // дробное число фунтов
   double pounds;            // общий вес в фунтах
```

```
public:
   Stonewt(double lbs);           // конструктор в фунтах
   Stonewt(int stn, double lbs);  // конструктор в стоунах, lbs
   Stonewt();                     // конструктор по умолчанию
   ~Stonewt();
   void show_lbs() const;         // показать вес в фунтах
   void show_stn() const;         // показать вес в стоунах
};
#endif
```

Как упоминалось в главе 10, enum предоставляет удобный способ определения специфичных для класса констант, предполагая, что они будут целыми. Новые компиляторы предлагают следующую альтернативу:

```
static const int Lbs_per_stn = 14;
```

Обратите внимание, что класс Stonewt имеет три конструктора. Они позволяют инициализировать объект Stonewt дробным числом фунтов или комбинацией стоунов и фунтов. Либо же вы можете создать объект Stonewt без его инициализации:

```
Stonewt blossem(132);        // Вес = 132 фунта
Stonewt buttercup(10, 2);    // Вес = 10 стоунов, 2 фунта
Stonewt bubbles;             // Вес = значение по умолчанию
```

Кроме того, класс Stonewt предоставляет две функции отображения. Одна отображает вес в фунтах, другая — в стоунах и фунтах. В листинге 11.17 показана реализация методов класса. Отметим, что каждый конструктор присваивает значения всем трем приватным членам. Таким образом, при создании объекта Stonewt автоматически устанавливаются оба представления веса.

Листинг 11.17. stonewt.cpp

```
// stonewt.cpp -- методы класса Stonewt
#include <iostream>
using std::cout;
#include "stonewt.h"
// Конструирует объект Stonewt из значения типа double
Stonewt::Stonewt(double lbs)
{
   stone = int (lbs) / Lbs_per_stn; // целочисленное деление
   pds_left = int (lbs) % Lbs_per_stn + lbs - int(lbs);
   pounds = lbs;
}
// Конструирует объект Stonewt из стоунов и значения типа double
Stonewt::Stonewt(int stn, double lbs)
{
   stone = stn;
   pds_left = lbs;
   pounds = stn * Lbs_per_stn +lbs;
}
Stonewt::Stonewt() // Конструктор по умолчанию, wt = 0
{
   stone = pounds = pds_left = 0;
}
```

```
Stonewt::~Stonewt() // Деструктор
{
}
// Показывает вес в стоунах
void Stonewt::show_stn() const
{
   cout << stone << " стоунов, " << pds_left << " фунтов\n";
}
// Показывает вес в фунтах
void Stonewt::show_lbs() const
{
   cout << pounds << " фунтов\n";
}
```

Поскольку объект Stonewt представляет собой единственный вес, имеет смысл предусмотреть способы для преобразования целого или действительного значения в объект Stonewt. И вы уже это сделали! В C++ любой конструктор, который принимает единственный аргумент, действует как инструмент копирования для преобразования значения типа аргумента в тип класса. Следующий конструктор

```
Stonewt(double lbs); // шаблон преобразования double в Stonewt
```

служит инструкцией, преобразующей значение типа double в значение типа Stonewt. То есть, вы можете писать код следующим образом:

```
Stonewt myCat;   // создать объект Stonewt
myCat = 19.6;    // использовать Stonewt(double) для преобразования 19.6
                 // в Stonewt
```

Программа использует конструктор Stonewt(double) для конструирования временного объекта Stonewt, используя 19.6 в качестве инициализирующего значения. Затем операция присваивания копирует содержимое временного объекта в myCat. Этот процесс известен как *неявное преобразование*, поскольку происходит автоматически, без необходимости явного приведения типов.

Только конструктор с одним аргументом может быть использован в качестве преобразующей функции.

Конструктор

```
Stonewt(int stn, double lbs);
```

принимает два аргумента, поэтому не может использоваться для преобразования типов.

Возможность применения конструктора, работающего как автоматическая функция преобразования типов, кажется удобным средством. Но программисты, накопившие определенный опыт работы с C++, однако, обнаруживают, что автоматический аспект не всегда желателен, поскольку иногда ведет к нежелательным преобразованиям. Поэтому современные реализации C++ включают новое ключевое слово explicit для отключения этого автоматического поведения. То есть вы можете объявить конструктор следующим образом:

```
explicit Stonewt(double lbs); // неявное преобразование не разрешено
```

Это отключает неявное преобразование, подобное тому, что приведено в предыдущем примере, но по-прежнему позволяет использовать явное преобразование, то есть с явным приведением типов:

```
Stonewt myCat; // создать объект Stonewt
myCat = 19.6;  // недопустимо, если Stonewt(double) объявлен как explicit
mycat = Stonewt(19.6);    // так можно, явное преобразование
mycat = (Stonewt) 19.6;   // так можно, это старая форма приведения типов
```

На память!

Конструктор C++, который принимает один аргумент, задает преобразование типа аргумента в тип класса. Если конструктор квалифицирован с ключевым словом explicit, то он может использоваться только с явной формой преобразования, в противном случае допускается неявное преобразование.

Когда компилятор использует функцию Stonewt(double)? Если ключевое слово explicit присутствует в объявлении, Stonewt(double) применяется только с явным приведением типов, в противном случае его можно использовать для неявного преобразования:

- Когда вы инициализируете объект Stonewt значением типа double.
- Когда вы присваиваете значение типа double объекту Stonewt.
- Когда вы передаете значение типа double функции, ожидающей аргумент типа Stonewt.
- Когда функция, объявленная как возвращающая значение Stonewt, пытается вернуть значение double.
- Когда любая из описанных ситуаций использует встроенный тип, который может быть автоматически преобразован в double.

Взглянем более внимательно на последний пункт. Процесс идентификации аргументов, представленный прототипированием функций, позволяет конструктору Stonewt(double) выступать в качестве преобразователя для других числовых типов. То есть оба следующих оператора работают, преобразуя сначала int в double, а затем обращаясь к конструктору Stonewt(double):

```
Stonewt Jumbo(7000); //использует Stonewt(double), преобразуя int в double
Jumbo = 7300; // использует Stonewt(double), преобразуя int в double
```

Однако это двухшаговое преобразование работает только в том случае, когда выбор однозначен. То есть, если у класса также определен конструктор Stonewt(long), то компилятор отклонит эти выражения, возможно, указав, что int может быть преобразован и в double, и в long, поэтому такой вызов неоднозначен.

Код в листинге 11.18 использует два конструктора для инициализации объектов Stonewt и управления преобразованием типов. Убедитесь, что успешно скомпилирован листинг 11.17 прежде, чем работать с листинком 11.18.

Листинг 11.18. stone.cpp

```
// stone.cpp -- определенные пользователем преобразования
// компилировать вместе с stonewt.cpp
#include <iostream>
using std::cout;
```

```
#include "stonewt.h"
void display(const Stonewt & st, int n);
int main()
{
   Stonewt pavarotti = 260;  // для инициализации используется конструктор
   Stonewt wolfe(285.7);     // то же самое, что Stonewt wolfe = 285.7;
   Stonewt taft(21, 8);
   cout << "Тенор взвешен ";
   pavarotti.show_stn();
   cout << "Детектив взвешен ";
   wolfe.show_stn();
   cout << "Президент взвешен ";
   taft.show_lbs();
   pavarotti = 265.8;        // используется конструктор для преобразования
   taft = 325;               // то же самое, что taft = Stonewt(325);
   cout << "После обеда тенор весит ";
   pavarotti.show_stn();
   cout << "После обеда президент весит ";
   taft.show_lbs();
   display(taft, 2);
   cout << "Борец весит даже больше\n";
   display(422, 2);
   cout << "Не осталось больше стоунов\n";
   return 0;
}
void display(const Stonewt & st, int n)
{
   for (int i = 0; i < n; i++)
   {
     cout << "Ого! ";
     st.show_stn();
   }
}
```

Вывод программы из листинга 11.18 будет выглядеть следующим образом:

```
Тенор весит 18 стоунов, 8 фунтов
Детектив весит 20 стоунов, 5.7 фунтов
Президент весит 302 фунтов
После обеда тенор весит 18 стоунов, 13.8 фунтов
После обеда президент весит 325 фунтов
Ого! 23 стоунов, 3 фунтов
Ого! 23 стоунов, 3 фунтов
Борец весит даже больше.
Ого! 30 стоунов, 2 фунтов
Ого! 30 стоунов, 2 фунтов
Не осталось больше стоунов
```

Замечания по программе

Обратите внимание, что когда конструктор принимает единственный аргумент, вы можете использовать следующую форму инициализации объекта класса:

```
// синтаксис для инициализации объекта класса
// при использовании конструктора с одним аргументом
Stonewt pavarotti = 260;
```

Это эквивалентно следующим двум формам, которые мы уже использовали:

```
// стандартные формы синтаксиса для инициализации объектов
Stonewt pavarotti(260);
Stonewt pavarotti = Stonewt(260);
```

Однако последние две формы могут применяться с конструкторами, принимающими несколько аргументов.

Далее отметим следующие два присваивания из листинга 11.18:

```
pavarotti = 265.8;
taft = 325;
```

Первое из двух присваиваний использует конструктор с аргументом типа double для преобразования 265.8 в значение типа Stonewt. При этом члену pounds объекта pavarotti присваивается значение 265.8. Поскольку здесь используется конструктор, такое присваивание также устанавливает значение членов класса stone и pds_left. Аналогично, второе присваивание преобразует значение типа int в тип double и затем использует Stonewt(double) для установки значений всех трех членов класса.

И, наконец, обратим внимание на следующий вызов функции:

```
display(422, 2); // преобразует 422 в double, затем в Stonewt
```

Прототип display() указывает на то, что его первый аргумент должен быть объектом типа Stonewt. (Аргументу Stonewt соответствует формальный параметр Stonewt или Stonewt &.) Обнаружив аргумент типа int, компилятор ищет конструктор Stonewt(int) для преобразования аргумента int в тип Stonewt. Не найдя его, компилятор ищет конструктор с аргументом другого встроенного типа, который можно преобразовать в int. Конструктор Stonewt(double) подходит. Поэтому компилятор преобразует int в double и затем использует Stonewt(double) для преобразования результата в объект Stonewt.

Функции преобразования

Код в листинге 11.18 преобразует число в объект Stonewt. Возможен ли обратный процесс? То есть, можно ли преобразовать объект Stonewt в double, как в следующем примере:

```
Stonewt wolfe(285.7);
double host = wolfe; // ?? возможно ли это ??
```

Ответ в том, что это сделать можно, но не используя конструктор. Конструкторы применяются только для преобразования других типов в тип класса. Чтобы выполнить обратный процесс, вы должны предусмотреть специальную форму функций операций C++, называемых *функциями преобразования*.

Функции преобразования — это определенный пользователем способ приведения типов, и вы можете применять его таким же способом, как применяете обычное приведение типов. Например, если определена функция преобразования Stonewt в double, то можно выполнять следующие преобразования:

```
Stonewt wolfe(285.7);
double host = double (wolfe);     // Синтаксис #1
double thinker = (double) wolfe;  // Синтаксис #2
```

Либо вы можете предоставить компьютеру решать, что нужно сделать:

```
Stonewt wells(20, 3);
double star = wells; // неявное применение функции преобразования
```

Компилятор, зная, что правая часть выражения имеет тип `Stonewt`, а левая — `double`, смотрит, определена ли соответствующая описанию функция преобразования. (Если он не находит ее, то генерирует сообщение об ошибке, говорящее о невозможности присваивания значения типа `Stonewt` переменной типа `double`.)

Итак, как же создать функцию преобразования? Чтобы преобразовать тип *имяТипа*, вы используете функцию следующего вида:

```
operator имяТипа();
```

При этом нужно помнить следующие моменты:

- Функция преобразования должна быть методом класса.
- Функция преобразования не должна иметь возвращаемого типа.
- Функция преобразования не должна иметь аргументов.

Например, функция для преобразования в тип `double` должна иметь следующий прототип:

```
operator double();
```

Часть *имяТипа* (в данном случае — `double`) говорит о том, в какой тип нужно преобразовать, поэтому никакой тип возврата не требуется. Тот факт, что функция является методом класса, означает, что она должна вызываться конкретным объектом класса, и сообщает ей, какое значение нужно преобразовать. Поэтому эта функция не нуждается в аргументах.

Чтобы добавить функции, которые преобразуют объект `stone_wt` в тип `double`, необходимо дополнить объявление класса следующими прототипами:

```
operator int();
operator double();
```

В листинге 11.19 представлена модифицированная версия объявления класса.

Листинг 11.19. stonewt1.h

```
// stonewt1.h -- усовершенствованное определение класса Stonewt
#ifndef STONEWT1_H_
#define STONEWT1_H_
class Stonewt
{
private:
   enum {Lbs_per_stn = 14};  // фунтов на стоун
   int stone;                // всего стоунов
   double pds_left;          // дробное число фунтов
   double pounds;            // общий вес в фунтах
public:
   Stonewt(double lbs);      // конструктор в фунтах
```

```
   Stonewt(int stn, double lbs);  // конструктор в стоунах, lbs
   Stonewt();                     // конструктор по умолчанию
   ~Stonewt();
   void show_lbs() const;         // показать вес в фунтах
   void show_stn() const;         // показать вес в стоунах
// функции преобразования
   operator int() const;
   operator double() const;
};
#endif
```

Листинг 11.20 представляет собой модифицированную версию листинга 11.17 с включением функций преобразования типа. Обратите внимание, что обе функции возвращают значение нужного типа, даже несмотря на то, что не имеют объявленного типа возврата. Также отметим, что определение преобразования к `int` округляет возвращаемое значение к ближайшему целому, а не усекает его. Например, если `pounds` равно `114.4`, то `pounds+0.5` равно `114.9`, а `int(114.9)` — `114`. Но если `pounds` равно `114.6`, то `pounds+0.5` равно `115.1`, а `int(115.1)` — `115`.

Листинг 11.20. stonewt1.cpp

```
// stonewt1.cpp -- методы класса Stonewt + функции преобразования
#include <iostream>
using std::cout;
#include "stonewt1.h"
// Конструирует объект Stonewt из значения типа double
Stonewt::Stonewt(double lbs)
{
   stone = int (lbs) / Lbs_per_stn; // целочисленное деление
   pds_left = int (lbs) % Lbs_per_stn + lbs - int(lbs);
   pounds = lbs;
}
// Конструирует объект Stonewt из стоунов и значения типа double
Stonewt::Stonewt(int stn, double lbs)
{
   stone = stn;
   pds_left = lbs;
   pounds = stn * Lbs_per_stn +lbs;
}
Stonewt::Stonewt() // Конструктор по умолчанию, wt = 0
{
   stone = pounds = pds_left = 0;
}
Stonewt::~Stonewt() // Деструктор
{
}
// Показывает вес в стоунах
void Stonewt::show_stn() const
{
   cout << stone << " стоунов, " << pds_left << " фунтов\n";
}
// Показывает вес в фунтах
void Stonewt::show_lbs() const
```

```
{
   cout << pounds << " фунтов\n";
}

// conversion functions
Stonewt::operator int() const
{
   return int (pounds + 0.5);
}
Stonewt::operator double()const
{
   return pounds;
}
```

Код в листинге 11.21 тестирует новые функции преобразования. Операция присваивания в программе использует неявное преобразование, в то время как последний оператор cout использует явное приведение типа. Убедитесь, что компилируете листинг 11.20 вместе с листингом 11.21.

Листинг 11.21. stone1.cpp

```
// stone1.cpp -- определенные пользователем функции преобразования
// компилировать вместе с stonewt1.cpp
#include <iostream>
#include "stonewt1.h"
int main()
{
   using std::cout;
   Stonewt poppins(9,2.8);   // 9 стоунов, 2.8 фунта
   double p_wt = poppins;    // неявное преобразование
   cout << "Приведение к double => ";
   cout << "Poppins: " << p_wt << " фунтов.\n";
   cout << "Приведение к int => ";
   cout << "Poppins: " << int (poppins) << "фунтов.\n";
   return 0;
}
```

Ниже приведен вывод программы из листингов 11.19, 11.20 и 11.21, который показывает результат преобразования объекта Stonewt в типы int и double:

```
Приведение к double => Poppins: 128.8 фунтов.
Приведение к int => Poppins: 129 фунтов.
```

Автоматическое применение преобразования типов

Код в листинге 11.21 использует int(poppins) с cout. Предположим, что явное приведение типов пропущено:

```
cout << "Poppins: " << poppins << " фунтов.\n";
```

Будет ли программа использовать неявное преобразование, как в следующем операторе?

```
double p_wt = poppins;
```

Ответ — нет, не будет. В примере с p_wt контекст указывает на то, что poppins должно быть приведено к типу double. Но в примере с cout ничего не указывает на то, должно ли выполняться приведение к int или double. Столкнувшись с таким недостатком информации, компилятор предполагает, что вы используете неоднозначное преобразование. Ничего в этом операторе не указывает на то, какой тип использовать.

Интересно то, что если бы в классе была определена только одна функция приведения к double, в этом случае компилятор принял бы такой оператор. Это потому, что если доступна только одна функция преобразования, то никакой двусмысленности нет.

Та же ситуация возникает и с присваиванием. При существующем объявлении класса компилятор отклоняет следующий оператор как двусмысленный:

```
long gone = poppins;   // неоднозначность
```

В C++ вы можете присваивать как int, так и double значения переменной типа long, поэтому компилятор на законном основании может использовать обе функции преобразования. Но компилятор не отвечает за выбор, какую именно. Но если вы исключите одну из этих двух функций, то компилятор обработает этот оператор. Например, допустим, что удалено определение double(). В этом случае компилятор преобразует значение int в long в процессе присваивания gone.

Когда в классе определены два или более преобразований, вы по-прежнему можете использовать явное приведение типов, чтобы указать, какую именно функцию преобразования применять в конкретном случае. Можно использовать любую из следующих нотаций:

```
long gone = (double) poppins;   // использовать преобразование в double
long gone = int (poppins);      // использовать преобразование в int
```

Первый из этих операторов преобразует вес poppins в значение типа double, а второй преобразует значение double в long.

Как и преобразующие конструкторы, функции преобразования могут успешно применяться в смешанном виде. Проблема с применением функций, выполняющих автоматическое, неявное преобразование, состоит в том, что такое преобразование может происходить тогда, когда вы не ожидаете этого. Предположим, например, что вам случится по причине недосыпа написать следующий код:

```
int ar[20];
...
Stonewt temp(14, 4);
...
int Temp = 1;
...
cout << ar[temp] << "!\n"; // используется temp вместо Temp
```

Обычно вы рассчитываете, что компилятор перехватит такие ошибки, как использование объекта вместо целого числа в индексе массива. Но в классе Stonewt определена операция int(), поэтому объект temp преобразуется в int 200 и может быть использован в качестве индекса массива. Мораль в том, что часто лучше использовать явное преобразование и исключить возможность неявного. Ключевое слово explicit не работает с функциями преобразования, но все, что вам следует

сделать — это заменить преобразующие функции на непреобразующие, которые выполняют ту же работу, но только при их явном вызове. То есть, вы можете заменить

```
Stonewt::operator int() { return int (pounds + 0.5); }
```

на

```
int Stonewt::Stone_to_Int() { return int (pounds + 0.5); }
```

Это сделает невозможным

```
int plb = poppins;
```

но если вам действительно нужно преобразование, позволит следующее:

```
int plb = poppins.Stone_to_Int();
```

Внимание!

Используйте функции неявного преобразования с осторожностью. Часто лучшим выбором будет функция, вызываемая явно.

Подведем итоги. C++ предлагает следующие типы преобразования для классов:

- Конструктор класса, имеющий один аргумент, служит инструкцией по преобразованию значения типа аргумента в тип класса. Например, конструктор класса `Stonewt` с аргументом типа `int` вызывается автоматичеcки, когда вы присваиваете аргумент типа `int` объекту типа `Stonewt`. Однако использование ключевого слова `explicit` в объявлении конструктора исключает неявное преобразование и разрешает только явное.
- Специальная функция операции, называемая функцией преобразования, служит инструкцией для преобразования объекта класса в некоторый другой тип. Функция преобразования является членом класса, не имеет типа возврата, не имеет аргументов и носит имя `operator` *имяТипа*`()`, где *имяТипа* — тип, в который преобразуется объект. Эта функция преобразования вызывается автоматически, когда вы присваиваете объект класса переменной соответствующего типа или используете в выражениях приведение к этому типу.

Преобразования и друзья

Давайте добавим операцию сложения к классу `Stonewt`. Как упоминалось при обсуждении класса `Time`, вы можете либо использовать функцию-член, либо дружественную функцию, чтобы переопределить сложение. (Для простоты предположим, что никакой функции преобразования в форме `operator double()` не определено.) Вы можете реализовать сложение с помощью следующей функции-члена:

```
Stonewt Stonewt::operator+(const Stonewt & st) const
{
   double pds = pounds + st.pounds;
   Stonewt sum(pds);
   return sum;
}
```

Либо вы можете реализовать сложение в виде дружественной функции следующего вида:

```
Stonewt operator+(const Stonewt & st1, const Stonewt & st2)
{
   double pds = st1.pounds + st2.pounds;
   Stonewt sum(pds);
   return sum;
}
```

Помните, что вы можете представить либо определение метода, либо дружественной функции, но не оба сразу. Любая из этих форм позволит сделать следующее:

```
Stonewt jennySt(9, 12);
Stonewt bennySt(12, 8);
Stonewt total;
total = jennySt + bennySt;
```

Также при определении конструктора `Stonewt(double)` каждая из форм позволит применять следующие операторы:

```
Stonewt jennySt(9, 12);
double kennyD = 176.0;
Stonewt total;
total = jennySt + kennyD;
```

Но только дружественная функция разрешит такое:

```
Stonewt jennySt(9, 12);
double pennyD = 146.0;
Stonewt total;
total = pennyD + jennySt;
```

Чтобы увидеть, почему это так, вы можете транслировать каждое сложение в соответствующий вызов функции. Во-первых:

```
total = jennySt + bennySt;
```

становится

```
total = jennySt.operator+(bennySt);  // функция-член
```

или, в противном случае:

```
total = operator+(jennySt, bennySt); // дружественная функция
```

В каждом случае тип реального аргумента соответствует формальному аргументу. Также функция-член вызывается, как это и требуется, объектом `Stonewt`.

Далее, оператор

```
total = jennySt + kennyD;
```

принимает следующий вид:

```
total = jennySt.operator+(kennyD); // функция-член
```

или, в противном случае:

```
total = operator+(jennySt, kennyD); // дружественная функция
```

Опять-таки, функция-член вызывается, как положено — объектом `Stonewt`. На этот раз в каждом случае один аргумент (`kennyD`) имеет тип `double`, что вызывает конструктор `Stonewt(double)` для преобразования `double` в объект `Stonewt`.

Кстати, наличие функции-члена operator double() может здесь привести к путанице, поскольку создает дополнительный вариант интерпретации. Вместо преобразования kennyD в double и выполнения сложения с Stonewt, компилятор может преобразовать jennySt в double и выполнить сложение с double. Наличие слишком большого числа функций преобразования ведет к неоднозначности.

Наконец, оператор:

```
total = pennyD + jennySt;
```

превращается в оператор:

```
total = operator+(pennyD, jennySt); // дружественная функций
```

Здесь оба аргумента имеют тип double, что вызывает конструктор Stonewt(double) для преобразования их в объекты Stonewt. Однако функция-член не может быть вызвана.

```
total = pennyD.operator+(jennySt); // бессмыслица
```

Причина в том, что только объект класса может вызывать функцию-член. C++ не пытается преобразовать pennyD в объект Stonewt. Преобразование имеет место только для аргументов функций-членов, а не для вызовов этих функций-членов.

Вывод из сказанного: определение сложения как дружественной функции упрощает для программы задачу автоматического преобразования типов. Причина состоит в том, что оба операнда становятся аргументами функции, поэтому прототипирование работает для обоих операндов.

Выбор реализации сложения

Предположим, что вы хотите складывать величины типа double со Stonewt. У вас есть несколько вариантов. Первый, как вы уже видели, состоит в том, чтобы определить

```
operator+(const Stonewt &, const Stonewt &)
```

в качестве дружественной функции и иметь конструктор Stonewt(double), выполняющий преобразование аргументов типа double в аргументы типа Stonewt.

Второй выбор — для перегрузки операции сложения использовать функцию, которая явно принимает один аргумент типа double:

```
Stonewt operator+(double x);   // функция-член
friend Stonewt operator+(double x, Stonewt & s);
```

Таким образом, оператор

```
total = jennySt + kennyD;       // Stonewt + double
```

явно соответствует функции-члену operator+(double x), а оператор

```
total = pennyD + jennySt;       // double + Stonewt
```

явно соответствует функции-члену operator+(double x, Stonewt & s). Ранее вы делали нечто подобное с умножением для класса Vector.

Каждый вариант имеет свои преимущества. Первый (полагающийся на неявное преобразование) порождает более короткие программы, поскольку вы определяете меньше функций. Это также означает, что вам придется прилагать меньше усилий,

и снижает количество возможных ошибок. Недостаток же в том, что растет расход времени и памяти на вызов конструкторов всякий раз, когда требуется преобразование. Второй вариант (с дополнительными функциями, явно соответствующими типу) – это зеркальное отражение первого. Он приводит к удлинению программ и требует дополнительных усилий с вашей стороны для непосредственного управления преобразованием, но работает несколько быстрее.

Если ваша программа интенсивно применяет сложение значений `double` с объектами `Stonewt`, то, может быть, стоит перегрузить операцию сложения для эффективной реализации этой операции. Если же программа применяет такое сложение изредка, то проще положиться на автоматическое преобразование типов, либо, если вы хотите быть более аккуратными – на явное преобразование.

Пример из практики: вызов загрузочных функций перед `main()`

Несмотря на то что первая функция, вызываемая в любой исполняемой программе — это точка входа `main()`, есть несколько трюков, которыми можно воспользоваться, чтобы изменить это правило. Например, рассмотрим программу планирования, координирующую работу гольф-клубов. Обычно, когда программа стартует, требуется информация из множества источников, чтобы правильно составить расписание работы гольф-клуба. Поэтому вам может понадобиться вызвать некоторые “загрузочные” функции с целью подготовки почвы для запуска `main()`.

Глобальный объект (то есть объект, находящийся в области видимости в пределах файла) — это то, что вы будете искать в первую очередь, поскольку глобальные объекты гарантированно создаются перед вызовом функции `main()`. Что вы можете сделать — так это создать класс с конструктором по умолчанию, который вызовет все ваши загрузочные функции. Это может быть, например, инициализация различных компонентов данных объекта. Затем вы можете создать глобальный объект. Следующий код иллюстрирует такую технику.

```
class CompileRequirements
{
private:
    // существенная информация
public:
    CompileRequirements()           // конструктор по умолчанию
    {
       GetDataFromSales();          // вариации
       GetDataFromManufacturing();  // загрузка
       GetDataFromFinance();        // функции
    }
};
//Экземпляр класса Req, имеющий глобальную область видимости
CompileRequirements Req; // использует конструктор по умолчанию
int main(void)
{
    // Читает Req и составляет расписание
    BuildScheduleFromReq();
    //
    // остальной программный код
    //
}
```

Резюме

В настоящей главе было раскрыто много важных аспектов определения и использования классов. Некоторые из материалов главы могут показаться неясными, до тех пор, пока ваш собственный опыт не углубит понимание.

Обычно единственным способом доступа к приватным членам класса является использование методов класса. C++ ослабляет это ограничение за счет технологии дружественных функций. Чтобы сделать функцию другом класса, вы объявляете ее внутри объявления класса и предваряете объявление ключевым словом `friend`.

C++ расширяет перегрузку операций возможностью определять специальные функции операций, которые описывают, как конкретная операция применяется с конкретным классом. Функция операции может быть функцией-членом класса либо дружественной функцией. (Некоторые операции могут быть только членами класса.) C++ позволяет вам вызывать функцию операции как непосредственным обращением к этой функции, так и применением операции в обычном синтаксисе. Функция операции для операции *op* имеет следующую форму:

```
operatorop(список-аргументов)
```

список-аргументов представляет операнды операции. Если функция операции является функцией-членом класса, то первый операнд — это вызывающий объект, не являющийся частью *список-аргументов*. Например, в настоящей главе вы перегружали операцию сложения, определив функцию `operator+()` в классе `Vector`. Если `up`, `right` и `result` — три вектора, вы можете использовать любой из следующих операторов для сложения векторов:

```
result = up.operator+(right);
result = up + right;
```

Во втором варианте тот факт, что операнды `up` и `right` имеют тип `Vector`, заставляет C++ применять определение сложения, объявленное в классе `Vector`.

Когда функция операции является функцией-членом, первый операнд является вызывающим эту функцию объектом. Так, например, в приведенных выражениях объект `up` является вызывающим объектом. Если вы хотите определить функцию операции так, чтобы первый операнд не был объектом класса, то вы должны объявить дружественную функцию. Тогда вы сможете передавать ей операнды в таком порядке, в каком хотите.

Одна из наиболее часто применяемых задач перегрузки — определение операции << для применения ее с объектом `cout` с целью отображения содержимого объектов. Чтобы позволить объекту `ostream` быть первым операндом, вы определяете функцию операции как дружественную. Чтобы позволить перегруженной операции сцепляться с самим собой, вы указываете возвращаемым типом этой функции `ostream &`. Ниже показана общая форма, удовлетворяющая этим требованиям:

```
ostream & operator<<(ostream & os, const c_name & obj)
{
   os << ... ; // отображение содержимого объекта
   return os;
}
```

Однако если класс имеет методы, возвращающие значения членов, которые нужно отобразить, вы можете использовать их вместо прямого доступа в `operator<<()`.

В этом случае функция не обязана быть дружественной.

C++ позволяет вам устанавливать преобразование типов — из класса в заданный тип и обратно. Во-первых, любой конструктор класса, принимающий единственный аргумент, может служить функцией преобразования, преобразуя тип аргумента в тип класса. C++ вызывает конструктор автоматически, если вы присваиваете значение типа аргумента объекту. Например, предположим, что имеется класс `String` с конструктором, который принимает один аргумент типа `char *`. Затем, если `bean` — объект типа `String`, то вы можете использовать следующий оператор:

```
bean = "pinto"; // преобразует тип char * в тип String
```

Однако если объявлению конструктора предшествует ключевое слово `explicit`, то конструктор может быть использован только для явного преобразования. Функция преобразования должна быть функцией-членом. Если она преобразует к типу *имяТипа*, то она должна иметь следующий прототип:

```
operator имяТипа();
```

Обратите внимание, что она не должна иметь объявленного типа возврата, не должна иметь аргументов и должна (несмотря на отсутствие возвращаемого типа) возвращать преобразованное значение. Например, функция для преобразования типа `Vector` в тип `double` должна иметь следующую форму:

```
Vector::operator double()
{
   ...
   return a_double_value;
}
```

Опыт показывает, что часто все-таки лучше не полагаться на функции неявного преобразования. Как, возможно, вы уже обратили внимание, классы требуют гораздо более пристального внимания к деталям, нежели простые структуры в стиле С. Взамен они могут принести гораздо большую пользу.

Вопросы для самоконтроля

1. Используйте функцию-член для перегрузки операции умножения класса `Stonewt`. Разработайте операцию умножения членов данных на значение типа `double`. Имейте в виду, что нужно будет позаботиться о представлении “стоун-фунт”. То есть, удвоение 10 стоунов и 8 фунтов должно давать 21 стоун и 2 фунта.
2. В чем разница между функцией-другом и функцией-членом?
3. Должна ли функция, не являющаяся членом, быть дружественной для того, чтобы иметь доступ к членам класса.
4. Используйте дружественную функцию для перегрузки операции умножения класса `Stonewt`. Определите операцию умножения значения `double` на `Stone`.
5. Какие операции не могут быть перегружены?
6. Какие ограничения накладываются на перегрузку следующих операций: =, `()`, `[]` и ->?
7. Определите функцию преобразования для класса `Vector`, которая будет приводить объект `Vector` к значению типа `double`, представляющему длину вектора.

Упражнения по программированию

1. Модифицируйте листинг 11.15 таким образом, чтобы он записывал успешные перемещения случайного гуляки в файл. Отмечайте каждую позицию номером шага. Также заставьте программу записывать начальные условия (целевую дистанцию и длину шага) и суммировать результаты в файле. Содержимое файла должно выглядеть так:

```
Целевая дистанция: 100, Размер шага: 20
0: (x,y) = (0, 0)
1: (x,y) = (-11.4715, 16.383)
2: (x,y) = (-8.68807, -3.42232)
...
26: (x,y) = (42.2919, -78.2594)
27: (x,y) = (58.6749, -89.7309)
После 27 шагов субъект находится в следующем положении:
 (x,y) = (58.6749, -89.7309)
или
(m,a) = (107.212, -56.8194)
Средняя дистанция, преодоленная за один шаг = 3.97081
```

2. Модифицируйте заголовок класса `Vector` и файлы реализации (листинги 11.13 и 11.14) таким образом, чтобы длина и направление вектора не хранились в виде компонентов данных. Вместо этого они должны вычисляться по требованию при вызове методов `magval()` и `angval()`. Вы должны оставить общедоступный интерфейс без изменений (те же общедоступные методы с теми же аргументами), но изменить приватную часть, включая некоторые из приватных методов и их реализации. Протестируйте модифицированную версию с помощью программы из листинга 11.15, которая должна остаться неизменной, поскольку общедоступный интерфейс класса `Vector` не изменился.
3. Модифицируйте листинг 11.15 таким образом, чтобы вместо сообщений о результатах отдельной попытки при конкретной комбинации расстояние/шаг, он сообщал максимальное, минимальное и среднее число шагов для *N* попыток, где *N* — целое число, вводимое пользователем.
4. Перепишите финальный пример класса `Time` (листинги 11.10, 11.11 и 11.12) таким образом, чтобы все перегруженные операции были реализованы дружественными функциями.
5. Перепишите класс `Stonewt` (листинги 11.16 и 11.17) так, чтобы была переменная-член состояния, управляющая тем, что объект интерпретируется в форме стоунов, целых фунтов либо числа фунтов с плавающей точкой. Перегрузите операцию << для замены методов `show_stn()` и `show_lbs()`. Перегрузите операции сложения, вычитания и умножения значений `Stonewt`. Протестируйте полученный класс с помощью короткой программы, которая использует все методы и друзья класса.
6. Перепишите класс `Stonewt` (листинги 11.16 и 11.17) таким образом, чтобы перегрузить все шесть операций сравнения. Операции должны сравнивать члены `pounds` и возвращать значение типа `bool`. Напишите программу, которая объявляет массив из шести объектов `Stonewt` и инициализирует первые три из них при объявлении массива. Затем она должна использовать цикл для ввода

значений инициализации остальных трех элементов массива. После этого она должна вывести самый маленький элемент, самый большой, и сколько элементов больше или равны 11 стоунам. (Простейший подход состоит в том, чтобы создать объект Stonewt, инициализированный 11 стоунами, и сравнивать с ним другие объекты.)

7. Комплексное число имеет две части: действительную и мнимую. Один из способов записи такого числа выглядит как (3.0, 4.0). Здесь 3.0 — действительная часть, а 4.0 — мнимая. Предположим, a = (A, Bi) и c = (C, Di). Ниже представлены некоторые операции с комплексными числами:

   - Сложение: a + c = (A + C, (B + D)i)
   - Вычитание: a – c = (A – C, (B – D)i)
   - Умножение: a*c = (A*C – B*D, (A*D + B*C)i)
   - Умножение: (x — действительное число): x*c = (x*C, x*Di)
   - Сопряжение: ~a = (A, – Bi)

   Определите класс complex так, чтобы следующая программа могла использовать его с корректными результатами:

```
#include <iostream>
using namespace std;
#include "complex0.h" // во избежание конфликта с complex.h
int main()
{
    complex a(3.0, 4.0); // инициировать значениями (3,4i)
    complex c;
    cout << "Введите комплексное число (q для выхода):\n";
    while (cin >> c)
    {
        cout << "c is " << c << '\n';
        cout << "комплексное слияние равно " << ~c << '\n';
        cout << "a равно " << a << '\n";
        cout << "a + c равно " << a + c << '\n';
        cout << "a - c равно " << a - c << '\n';
        cout << "a * c равно " << a * c << '\n';
        cout << "2 * c равно " << 2 * c << '\n';
        cout << "Введите комплексное число (q для выхода):\n";
    }
    cout << "Готово!\n";
    return 0;
}
```

   Помните, что вы должны перегрузить операции << и >>. Многие системы уже имеют поддержку комплексных чисел в файле заголовков complex.h, поэтому применяйте complex0.h во избежание конфликтов. Используйте const там, где это оправдано.

   Ниже показан пример выполнения этой программы:

```
Введите комплексное число (q для выхода):
действительная часть: 10
мнимая часть: 12
c равно (10,12i)
```

```
комплексное сопряжение равно (10,-12i)
a равно (3,4i)
a + c равно (13,16i)
a - c равно (-7,-8i)
a * c равно (-18,76i)
2 * c равно (20,24i)
Введите комплексное число (q для выхода):
действительная часть: q
Готово!
```

Обратите внимание, что `cin >> c` через перегрузку теперь приглашает вводить действительную и мнимую часть по отдельности.

ГЛАВА 12

Классы и динамическое распределение памяти

В этой главе:

- Использование динамического распределения памяти для членов класса
- Явные и неявные конструкторы копирования
- Явные и неявные перегруженные операции присваивания
- Что необходимо делать при использовании операции `new` в конструкторе
- Использование статических членов класса
- Применение операции `new` с адресацией с объектами
- Использование указателей на объекты
- Реализация абстрактного типа данных очереди

В данной главе рассматриваются вопросы применения операций `new` и `delete` с классами, а также решение некоторых хитрых проблем, которые может вызывать использование динамической памяти. Все это может показаться слишком кратким перечнем вопросов, однако они затрагивают и разработку конструкторов, и разработку деструкторов, и перегрузку операций.

Давайте рассмотрим специальный пример, показывающий, как C++ помогает управлять загрузкой памяти. Допустим, что вы хотите создать класс с членом, представляющим чью-либо фамилию. Самым простым способом для хранения имени является использование элементов символьного массива. Однако этот способ имеет несколько недостатков. Вполне возможна ситуация, что вы начнете с использования 14-символьного массива, а затем натолкнетесь на имя вроде Бартоломей Смидсбери-Крафтховингем. Ясно, что для надежности лучше использовать 40-символьный массив. Если же будет создан массив из 2000 подобных объектов, то значительное количество памяти будет потрачено на элементы, которые заняты только частично. (При этом вы также увеличиваете загрузку памяти компьютера.) Существует неплохая альтернатива.

Некоторые вопросы (например, сколько памяти использовать) часто удобнее решать во время выполнения программы, нежели во время ее компиляции. Стандартным методом C++ для хранения имени в объекте является использование операции `new` в конструкторе класса. Это позволяет распределять надлежащий объем памяти при работе программы. Однако введение `new` в конструктор класса может привести к нескольким новым проблемам, если забыть о ряде дополнительных ша-

гов, таких как развертывание деструктора класса, согласование всех конструкторов с деструктором и создание дополнительных методов класса для облегчения правильной инициализации и распределения. (В данной главе, разумеется, объясняются все эти шаги.) Если вы только начинаете изучение C++, то, возможно, будет лучше начать движение с более простого подхода символьных массивов. Затем, когда разработанный класс уже будет нормально работать, вы можете вернуться к принципам объектно-ориентированного программирования (ООП) и усовершенствовать объявление класса за счет применения `new`. В общем, мы рекомендуем постепенно привыкать к C++.

Динамическая память и классы

Что вы предпочитаете на завтрак, ленч и обед в следующем месяце? Сколько унций молока на обед на третий день? А сколько изюма в каше на завтрак на пятнадцатый день? Большинство людей отложит принятие этих решений до фактического времени приема пищи. Частью стратегии создания программ в C++ является аналогичное отношение к распределению памяти. Лучше позволить программе принимать решения относительно памяти во время выполнения, а не во время компиляции. При этом использование памяти зависит только от нужд программы, а не от набора жестких правил групп памяти. Помните, что для усиления динамического управления памятью в C++ используются операции `new` и `delete`. К сожалению, применение данных операций с классами может породить новые проблемы программирования. Как вы увидите, деструкторы из просто декоративных могут стать жизненно необходимыми. И иногда даже понадобится перегружать операцию присваивания для того, чтобы программа вела себя должным образом. Мы изучим все эти вопросы прямо сейчас.

Простой пример и статические члены класса

Мы пока еще не пользовались операциями `new` и `delete`, поэтому давайте рассмотрим их на примере короткой программы. При этом можно также изучить новый класс хранения — статический член класса. Посредником является класс `StringBad`, который впоследствии будет заменен чуть более функциональным классом `String`. (Вы уже встречали стандартный класс C++ `string`. Он будет рассмотрен более подробно в главе 16. Тем временем, простые классы `StringBad` и `String` в данной главе дают представление о том, что лежит в основе классов подобного рода. Множество программных технологий ориентируются на предоставление такого дружественного интерфейса.)

Объекты классов `StringBad` и `String` хранят указатель на строку и значение, представляющее длину строки. Классы `StringBad` и `String` используются, прежде всего, для того, чтобы посмотреть изнутри, как работают члены классов `new`, `delete` и `static`. По этой причине конструкторы и деструкторы во время вызова отображают сообщения, дабы вы могли следить за операциями. Также для упрощения интерфейса класса можно опустить некоторые полезные функции членов и друзей класса (вроде перегруженных операций ++ и >>) и функцию преобразования. (Не переживайте! Вопросы для самоконтроля в конце данной главы предоставят вам возможность добавить указанные полезные функции поддержки.) В листинге 12.1 показано объявление класса.

Листинг 12.1. strngbad.h

```
// strngbad.h -- незаконченное определение класса строки
#include <iostream>
#ifndef STRNGBAD_H_
#define STRNGBAD_H_
class StringBad
{
private:
   char * str;                  // указатель на строку
   int len;                     // длина строки
   static int num_strings;      // количество объектов
public:
   StringBad(const char * s);   // конструктор
   StringBad();                 // конструктор по умолчанию
   ~StringBad();                // деструктор
// дружественная функция
   friend std::ostream & operator<<(std::ostream & os,
               const StringBad & st);
};
#endif
```

Почему класс называется StringBad? Просто для того чтобы напоминать вам, что StringBad является примером на стадии разработки. На этой стадии выполняется разработка класса с использованием распределения динамической памяти, и очевидные вещи класс выполняет корректно. Например, он правильно использует операции new и delete в конструкторах и деструкторах. Он фактически не делает "нехороших" вещей, но разработка не допускает реализацию некоторых дополнительных "хороших" вещей, которые являются необходимыми, но совсем не очевидными. Рассмотрение проблем, которые содержит класс, поможет понять и запомнить те неочевидные изменения, которые нужно будет сделать впоследствии, когда вы будете преобразовывать его в более функциональный класс String.

Вы должны обратить внимание на два момента в данном объявлении. Во-первых, в нем для представления имени используется указатель на char вместо массива char. Это означает, что объявление класса не распределяет пространство памяти непосредственно под строку. Для этого применяется операция new в конструкторах. Такая схема позволяет избежать привязки объявления класса к предопределенной границе размера строки.

Во-вторых, в определении объявляется член num_strings как принадлежащий к классу хранения static. *Статический член класса* обладает особым свойством: программа создает только одну копию статической переменной класса независимо от количества создаваемых объектов. Другими словами, статический член совместно используется между всеми объектами данного класса по аналогии с номером телефона, который может использоваться всеми членами семьи. Если, к примеру, вы создаете десять объектов StringBad, то в них будут содержаться десять членов str и десять членов len, но всего лишь один общий член num_strings (рис. 12.1). Это удобно для данных, которые являются приватными для класса, но при этом должны иметь одно общее значение для всех объектов класса. Например, член num_strings предназначен для отслеживания количества создаваемых объектов.

```
class StringBad
{
private:
  char * str;
  int len;
  static int num_strings;
public:
  ...
};
```

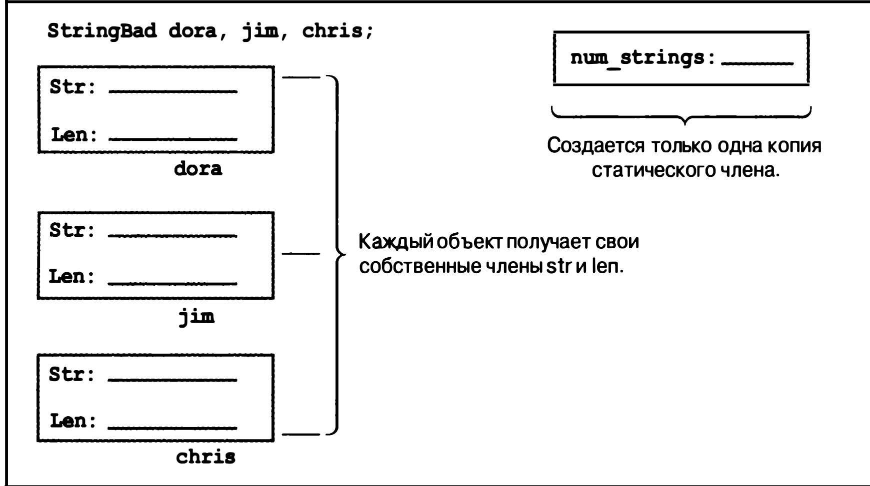


Рис. 12.1. Статические данные-члены

Кстати, в листинге 12.1 член `num_strings` служит удобным средством для иллюстрации статических данных-членов, а также для того, чтобы вы обратили внимание на возможные проблемы программирования. В общем случае строковый класс не нуждается в подобном члене.

Взгляните на реализацию методов класса в листинге 12.2. Обратите внимание на то, как в нем обрабатывается использование указателя и статического члена.

Листинг 12.2. strngbad.cpp

```
// strngbad.cpp -- методы класса StringBad
#include <cstring> // string.h для некоторых систем
#include "strngbad.h"

using std::cout;

// инициализация статического члена класса
int StringBad::num_strings = 0;

// методы класса
```

```
// конструирование StringBad из строки C-стиля
StringBad::StringBad(const char * s)
{
   len = std::strlen(s);      // установить размер
   str = new char[len + 1];   // распределить память
   std::strcpy(str, s);       // инициализировать указатель
   num_strings++;             // установить счетчик объектов
   cout << num_strings << ": объект \"" << str
        << "\" создан\n";     // информация для вас
}

StringBad::StringBad()        // конструктор по умолчанию
{
   len = 4;
   str = new char[4];
   std::strcpy(str, "C++");  // строка по умолчанию
   num_strings++;
   cout << num_strings << ": объект по умолчанию \"" << str
        << "\" создан\n";     // информация для вас
}

StringBad::~StringBad()       // необходимый деструктор
{
   cout << "Объект \"" << str << "\" удален, "; // информация для вас
   --num_strings;             // обязательно
   cout << num_strings << " осталось\n"; // информация для вас
   delete [] str;             // обязательно
}

std::ostream & operator<<(std::ostream & os, const StringBad & st)
{
   os << st.str;
   return os;
}
```

Прежде всего, обратите внимание на следующий оператор в листинге 12.2:

```
int StringBad::num_strings = 0;
```

Этот оператор устанавливает первоначальное значение `0` для статического члена `num_strings`. Заметьте, что внутри объявления класса нельзя инициализировать переменную статического члена. Это связано с тем, что объявление является описанием того, как распределяется память, но оно фактически не распределяет ее. Вы можете распределить и инициализировать память путем создания объекта с использованием данного формата. В случае статического члена класса происходит независимая инициализация статического члена, с помощью отдельного оператора вне объявления класса. Это объясняется тем, что статический член класса хранится отдельно, а не как часть объекта. Обратите внимание на то, что оператор инициализации задает тип и использует операцию разрешения контекста.

Данная инициализация выполняется в файле методов, а не в файле объявления класса, поскольку объявление класса содержится в заголовочном файле, а программа

может включать заголовочный файл в несколько других файлов. При этом оператор инициализации будет повторяться несколько раз, что является ошибкой.

Исключением, при котором статические данные-члены не нужно инициализировать внутри объявления класса (см. главу 10), является случай, когда статические данные-члены определяются как const составного или перечислимого типа.

На память!

Статические данные-члены объявляются в объявлении класса и инициализируются в файле, содержащем методы класса. При инициализации используется операция разрешения контекста, чтобы указать, какому классу принадлежит статический член. Однако если статический член определяется как const составного или перечислимого типа, то его можно инициализировать непосредственно в объявлении класса.

Далее следует обратить внимание на то, что каждый конструктор содержит выражение num_strings++. Этим обеспечивается то, что каждый раз, когда программа создает новый объект, общая переменная num_strings увеличивается на единицу, отображая итоговое количество объектов String. Кроме того, деструктор содержит выражение --num_strings. Таким образом, класс String также следит за количеством удаленных объектов, храня текущее значение члена num_strings.

Теперь давайте посмотрим на первый конструктор в листинге 12.2, который задает исходное значение объекта String с помощью стандартной строки С:

```
StringBad::StringBad(const char * s)
{
   len = std::strlen(s);     // установить размер
   str = new char[len + 1];  // распределить память
   std::strcpy(str, s);      // инициализировать указатель
   num_strings++;            // установить счетчик объектов
   cout << num_strings << ": объект \"" << str
        << "\" создан\n";    // информация для вас
}
```

Вспомните, что член класса str является всего лишь указателем, поэтому конструктор должен предусматривать память для хранения строки. Указатель строки вы можете передать в конструктор во время инициализации объекта:

```
String boston("Boston");
```

Затем конструктор должен распределить достаточный объем памяти для хранения строки, после чего скопировать строку в указанное расположение. Давайте пройдем через этот процесс шаг за шагом.

Во-первых, функция инициализирует член len, используя функцию strlen() для вычисления длины строки. Далее применяется операция new, чтобы выделить соответствующее пространство для хранения строки. После этого адрес новой памяти присваивается члену str. (Вспомните, что strlen() возвращает длину строки, не учитывая завершающий нулевой символ, поэтому конструктор добавляет единицу к величине len для того, дабы учитывать нулевой символ при распределении пространства под строку.)

После этого в конструкторе используется функция strcpy() для копирования передаваемой строки в новую память. Затем обновляется счетчик объектов. В завершение, чтобы помочь вам контролировать все происходящее, конструктор отображает текущее количество объектов и строку, хранящуюся в объекте. Данное свойство

окажется весьма полезным позднее, когда вы будете сознательно вводить нарушения в класс String.

Для того чтобы понять данный принцип, вы должны осознавать, что строка не хранится в объекте. Строка хранится отдельно, в куче, а сам объект просто содержит информацию о том, где найти строку.

Обратите внимание на то, что нельзя использовать:

```
str = s; // не делайте так
```

Данная команда приведет к сохранению адреса без создания копии строки.

Конструктор по умолчанию ведет себя аналогично, за исключением того, что он обеспечивает строку по умолчанию "C++".

В деструкторе данного примера содержится самое важное дополнение к обработке классов:

```
StringBad::~StringBad()                           // необходимый деструктор
{
   cout << "Объект \"" << str << "\" удален, ";  // информация для вас
   --num_strings; // обязательно
   cout << num_strings << " осталось\n";          // информация для вас
   delete [] str;                                 // обязательно
}
```

Деструктор начинается с уведомления того, что он вызван. Данная часть является информационной, но не существенной. При этом операция delete является необходимой. Вспомните о том, что член str указывает на память, выделенную операцией new. Когда завершается время жизни объекта StringBad, исчезает и указатель str. Однако память, указываемая str, остается распределенной до тех пор, пока вы не примените операцию delete для того, чтобы ее очистить. Удаление объекта освобождает память, занимаемую непосредственно объектом, но при этом не освобождается автоматически память, занятая указателями, которые были членами объекта. Для этого нужно использовать деструктор. Размещая операцию delete в деструкторе, вы обеспечиваете освобождение памяти, которая выделялась в конструкторе операцией new, после удаления объекта.

На память!

Всякий раз, когда в конструкторе для распределения памяти используется операция new, в соответствующем деструкторе необходимо применять операцию delete для освобождения указанной памяти. Если использовалась операция new [] (со скобками), то после нее нужно применять операцию delete [] (со скобками).

В листинге 12.3 (который взят из программы разрабатываемой программы ежедневных новостей из мира овощей) демонстрируется, когда и как работают конструкторы и деструкторы StringBad. Не забудьте компилировать листинг 12.2 вместе с листингом 12.3.

Листинг 12.3. vegnews.cpp

```
// vegnews.cpp -- использование операций new и delete с классами
// компилировать вместе с strngbad.cpp
#include <iostream>
using std::cout;
#include "strngbad.h"
```

```
void callme1(StringBad &);  // передача по ссылке
void callme2(StringBad);    // передача по значению
int main()
{
   using std::endl;
   StringBad headline1("Сельдерей подкрался в полночь");
   StringBad headline2("Салат-латук вышел на охоту");
   StringBad sports("Шпинат поселился в трехлитровой банке с долларами");
   cout << "headline1: " << headline1 << endl;
   cout << "headline2: " << headline2 << endl;
   cout << "sports: " << sports << endl;
   callme1(headline1);
   cout << "headline1: " << headline1 << endl;
   callme2(headline2);
   cout << "headline2: " << headline2 << endl;
   cout << "Инициализация одного объекта другим:\n";
   StringBad sailor = sports;
   cout << "sailor: " << sailor << endl;
   cout << "Присваивание одного объекта другому:\n";
   StringBad knot;
   knot = headline1;
   cout << "knot: " << knot << endl;
   cout << "Конец функции main()\n";
   return 0;
}
void callme1(StringBad & rsb)
{
   cout << "Строка, переданная по ссылке:\n";
   cout << " \"" << rsb << "\"\n";
}
void callme2(StringBad sb)
{
   cout << "Строка, переданная по значению:\n";
   cout << " \"" << sb << "\"\n";
}
```

Замечание по совместимости

Данный первый черновик дизайна `StringBad` содержит несколько умышленных дефектов, из-за которых точные выходные данные остаются неопределенными. Некоторые компиляторы генерировали, например, исполняемый код, который аварийно прерывал программу перед ее завершением. Однако, несмотря на то, что детали выходных данных могут отличаться, основные проблемы и их решения (которые вскоре будут показаны!) остаются теми же.

Ниже показаны выходные данные, полученные после компиляции программы из листинга 12.3, с использованием компилятора командной строки Borland C++ 5.5:

```
1: объект "Сельдерей подкрался в полночь" создан
2: объект "Салат-латук вышел на охоту" создан
3: объект "Шпинат поселился в трехлитровой банке с долларами" создан
headline1: Сельдерей подкрался в полночь
headline2: Салат-латук вышел на охоту
sports: Шпинат поселился в трехлитровой банке с долларами
```

```
Строка, переданная по ссылке:
   "Сельдерей подкрался в полночь"
headline1: Сельдерей подкрался в полночь
Строка, переданная по значению:
   "Салат-латук вышел на охоту"
Объект "Салат-латук вышел на охоту" удален, 2 осталось
headline2: DÛ¤
Инициализация одного объекта другим:
sailor: Шпинат поселился в трехлитровой банке с долларами
Присваивание одного объекта другому:
3: объект по умолчанию "C++" создан
knot: Сельдерей подкрался в полночь
Конец функции main()
Объект "Сельдерей подкрался в полночь" удален, 2 осталось
Объект "Шпинат поселился в трехлитровой банке с долларами" удален, 1 осталось
Объект "Шпинат поселился в трехлитро8" удален, 0 осталось
Объект "@g" удален, -1 осталось
Объект "-|" удален, -2 осталось
```

Различные нестандартные символы, которые появляются в выходных данных, будут отличаться в зависимости от системы. Они являются одной из причин, по которой объект `StringBad` заслуживает определения "неудачный". Еще одной причиной можно назвать отрицательные значения счетчика объектов. Более новые комбинации компиляторов и операционных систем, как правило, аварийно прерывают программу непосредственно перед отображением строки, в которой остается –1 объект. Некоторые из них выдают сообщение General Protection Fault (GPF) (общее нарушение защиты). Появление GPF означает, что программа пытается получить доступ по запрещенному адресу памяти. Это еще один "плохой" признак.

Замечания по программе

Программа в листинге 12.3 начинает работать в нормальном режиме, но постепенно лишается устойчивости и приходит к странному и в конечном итоге опасному завершению. Давайте для начала рассмотрим "хорошие" части. Конструктор объявляет, что он создал три объекта `StringBad`, а также нумерует их. Программа перечисляет объекты, используя перегруженную операцию >>:

```
1: объект "Сельдерей подкрался в полночь" создан
2: объект "Салат-латук вышел на охоту" создан
3: объект "Шпинат поселился в трехлитровой банке с долларами" создан
headline1: Сельдерей подкрался в полночь
headline2: Салат-латук вышел на охоту
sports: Шпинат поселился в трехлитровой банке с долларами
```

После этого программа передает переменную `headline1` в функцию `callme1()` и после вызова отображает заново `headline1`. Ниже показан код:

```
callme1(headline1);
cout << "headline1: " << headline1 << endl;
А вот и результат:
Строка, переданная по ссылке:
   "Сельдерей подкрался в полночь"
headline1: Сельдерей подкрался в полночь
```

Этот раздел кода также работает нормально.

Но далее программа выполняет следующее:

```
callme2(headline2);
cout << "headline2: " << headline2 << endl;
```

Здесь функция callme2() передает headline2 по значению вместо передачи по ссылке, и полученный результат указывает на серьезную проблему:

```
Строка, переданная по значению:
   "Салат-латук вышел на охоту"
Объект "Салат-латук вышел на охоту" удален, 2 осталось
headline2: DÛ¤
```

Во-первых, передача headline2 в качестве аргумента функции каким-то образом послужила причиной вызова деструктора. Во-вторых, несмотря на то, что передача по значению подразумевает защиту исходного аргумента от изменения, функция испортила исходную строку до неузнаваемости. При этом отобразились некоторые непечатаемые символы. (Точное отображение зависит от того, что в данный момент находится в памяти.)

Ситуация становится еще хуже при завершении выходных данных, когда деструктор вызывается автоматически для каждого из созданных ранее объектов:

```
Конец функции main()
Объект "Сельдерей подкрался в полночь" удален, 2 осталось
Объект "Шпинат поселился в трехлитровой банке с долларами" удален, 1
осталось
Объект "Шпинат поселился в трехлитро8" удален, 0 осталось
Объект "@g" удален, -1 осталось
Объект "-|" удален, -2 осталось
```

Поскольку объекты с автоматическим хранением удаляются в порядке, обратном тому, в котором они создавались, то первыми тремя удаленными объектами становятся knots, sailor и sport. Удаление объектов knots и sailor проходит нормально, но для в sport вместо трехлитровой появляется трехлитро8. Единственное обращение программы к объекту sport использует его для инициализации sailor, но выглядит все так, что оно изменяет sport. А последние два удаленных объекта, headline2 и headline1, становятся неузнаваемыми. Что-то запортило данные строки, прежде чем они были удалены. Также странным выглядит и подсчет: как может остаться –2 объекта?

Действительно, странный подсчет дает ключ к разгадке. Каждый объект конструируется один раз и удаляется один раз, таким образом, количество вызовов конструктора должно равняться количеству вызовов деструкторов. Поскольку счетчик объектов (num_strings) уменьшается на два больше, чем увеличивается, то тот конструктор, который не увеличил значение num_strings, должен создавать два объекта. В описании класса объявляются и определяются два конструктора (причем оба увеличивают num_strings), но при этом оказывается, что программа использует три конструктора. Например, посмотрите на эту строку:

```
StringBad sailor = sports;
```

Какой конструктор здесь используется? Не конструктор по умолчанию и не конструктор с параметром const char *. Вспомните, что инициализация, применяющая данную форму, должна иметь другой синтаксис:

```
StringBad sailor = StringBad(sports); // конструктор, использующий sports
```

Поскольку типом объекта sports является StringBad, то соответствующий конструктор должен иметь следующий прототип:

```
StringBad(const StringBad &);
```

Оказывается, что компилятор автоматически генерирует данный конструктор (называемый *конструктором копирования*, поскольку он создает копию объекта), если в качестве первоначального значения одного объекта вы указываете другой объект. Автоматически сгенерированная версия не знает про обновление статической переменной num_strings, поэтому и нарушает схему подсчета. В самом деле, все проблемы, показанные в данном примере, возникают из-за функций-членов, которые компилятор генерирует автоматически. В связи с этим давайте обратимся к следующей теме.

Неявные функции-члены

Проблемы, связанные с классом StringBad, возникают из-за неявных функций-членов, которые определяются автоматически и чье поведение не соответствует индивидуальному дизайну класса. В частности, C++ автоматически обеспечивает следующие функции-члены:

- Конструктор по умолчанию (если не было определено ни одного конструктора).
- Конструктор копирования (если он не был определен).
- Операция присваивания (если она не была определена).
- Деструктор по умолчанию (если он не был определен).
- Операция взятия адреса (если она не была определена).

Если говорить более точно, то компилятор генерирует определения для четырех последних элементов, если программа использует объекты так, как они того требуют. Например, если в качестве значения одного объекта присваивается другой объект, то программа предоставляет определение для операции присваивания.

При этом оказывается, что неявный конструктор копирования и неявная операция присваивания вызывают проблемы с классом StringBad.

Неявная адресная операция возвращает адрес вызывающего объекта (то есть значение указателя this). Эта функция удобна для наших целей, поэтому мы не будем больше обсуждать данную функцию-член. Деструктор по умолчанию ничего не делает, поэтому он обсуждаться не будет, тем более, что в классе уже предусмотрена замена для него. Однако остальное должно быть рассмотрено более подробно.

Конструкторы по умолчанию

Если вы вообще не предусмотрели конструкторы, то C++ обеспечит конструктор по умолчанию. Например, предположим, что вы определили класс Klunk и пропустили конструкторы.

В такой ситуации компилятор снабжает код следующим стандартным оператором:

```
Klunk::Klunk() { }  // явный конструктор по умолчанию
```

Другими словами, он предоставляет конструктор, который не принимает аргументы и который ничего не делает. Он необходим, поскольку создание объекта всегда активизирует конструктор:

```
Klunk lunk;          // активизирует конструктор по умолчанию
```

Конструктор по умолчанию создает `lunk` как обычную автоматическую переменную, то есть ее значение при инициализации неизвестно.

После того как вы определили какой-либо конструктор, C++ нет необходимости определять конструктор по умолчанию. Если же вы хотите создать объекты, которые не инициализируются явно, либо массив объектов, то тогда вы должны однозначно определить конструктор по умолчанию. Хотя это конструктор без аргументов, его можно использовать для установки отдельных значений:

```
Klunk::Klunk()      // явный конструктор по умолчанию
{
   klunk_ct = 0;
   ...
}
```

Конструктор с аргументами все же может быть конструктором по умолчанию, если все его аргументы имеют стандартные значения. Например, класс `Klunk` может содержать следующий встроенный конструктор:

```
Klunk(int n = 0) { klunk_ct = n; }
```

Тем не менее, можно использовать только один конструктор по умолчанию. То есть нельзя делать следующее:

```
Klunk() { klunk_ct = 0 }              // конструктор #1
Klunk(int n = 0) { klunk_ct = n; } // неоднозначный конструктор #2
```

Почему последний конструктор неоднозначный? Рассмотрите следующие два объявления:

```
Klunk kar(10);    // в точности соответствует Klunt(int n)
Klunk bus;        // может соответствовать любому конструктору
```

Второе объявление соответствует конструктору #1 (без аргументов), но оно также соответствует конструктору #2 (который использует один аргумент со значением по умолчанию 0). Это приводит к тому, что компилятор выдает сообщение об ошибке.

Конструкторы копирования

Конструктор копирования служит для копирования объекта в заново созданный объект. Другими словами, он используется во время инициализации, а не во время обычного присваивания. Конструктор копирования для класса, как правило, имеет следующий прототип:

```
Имя_класса(const Имя_класса &);
```

Обратите внимание, что в качестве аргумента он принимает константную ссылку на объект класса. Например, конструктор копирования для класса String будет выглядеть так:

```
StringBad(const StringBad &);
```

О конструкторе копирования нужно знать два момента: когда он используется и что он делает.

Когда используется конструктор копирования

Конструктор копирования активизируется всякий раз, когда создается новый объект и в качестве его первоначального значения выбирается существующий объект того же самого типа. Это происходит в нескольких ситуациях. Наиболее очевидным является случай, когда вы явно инициализируете новый объект существующим объектом. Например, пусть motto является объектом StringBad. Следующие четыре определяющие объявления активизируют конструктор копирования:

```
StringBad ditto(motto);   // вызывает StringBad(const StringBad &)
StringBad metoo = motto;  // вызывает StringBad(const StringBad &)
StringBad also = StringBad(motto); // вызывает StringBad(const StringBad &)
StringBad * pStringBad = new StringBad(motto);
                          // вызывает StringBad(const StringBad &)
```

В зависимости от реализации, два средних объявления могут использовать конструктор копирования либо для создания непосредственно объектов metoo и also, либо для генерирования временных объектов, содержимое которых затем присваивается объектам metoo и also. Последний пример инициализирует безымянный объект для motto и присваивает адрес нового объекта указателю pstring.

Менее явный пример — компилятор использует конструктор копирования всякий раз, когда программа генерирует копии объекта. В частности, он применяется, когда функция передает объект по значению (как это делает функция callme2() в листинге 12.3) или когда функция возвращает объект. Вспомните, что передача по значению подразумевает создание копии исходной переменной. Компилятор также использует конструктор копирования при генерировании временных объектов. Например, компилятор может генерировать временный объект Vector для хранения промежуточного результата при сложении трех объектов Vector. Различные компиляторы могут вести себя по-разному при создании временных объектов, но все они активизируют конструктор копирования при передаче объектов по значению и возврате их. В частности, следующий вызов функции в листинге 12.3 запускает конструктор копирования:

```
callme2(headline2);
```

Программа использует конструктор копирования для инициализации sb (формальный параметр типа StringBad для функции callme2()).

Кстати, тот факт, что передача объекта по значению влечет за собой активизацию конструктора копирования, является хорошей причиной для того, чтобы использовать вместо этого передачу по ссылке. Это позволит сэкономить время вызова конструктора и пространство для хранения нового объекта.

Что делает конструктор копирования

Конструктор копирования по умолчанию производит *почленное копирование* нестатических членов, также иногда называемое *поверхностным копированием.* Каждый член копируется по значению. Например, в листинге 12.3 оператор

```
StringBad sailor = sports;
```

означает следующее (помимо того факта, что он не компилируется, поскольку доступ к приватным членам не разрешен):

```
StringBad sailor;
sailor.str = sports.str;
sailor.len = sports.len;
```

Если член сам является объектом класса, для копирования одного объекта-члена в другой используется конструктор копирования этого класса. Статические члены, такие как num_strings, при этом не затрагиваются, поскольку они принадлежат классу как единое целое, а не как индивидуальные объекты. На рис. 12.2 показано действие неявного конструктора копирования.

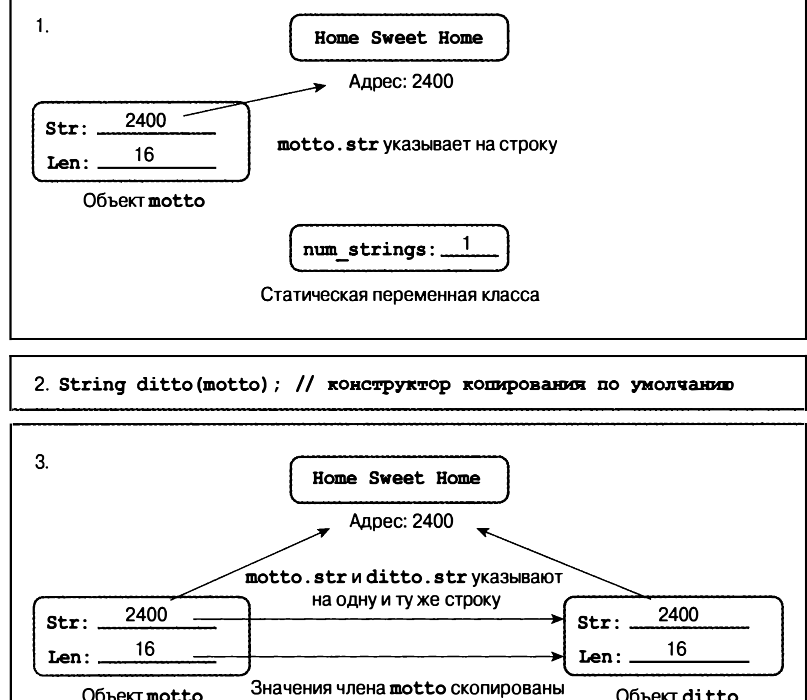


Рис. 12.2. Почленное копирование изнутри

Где конструктор копирования работает неправильно

Теперь вы готовы понять двойную странность листинга 12.3. (Предполагается, что выходные данные выглядят так, как показано сразу после листинга.) Первая странность состоит в том, что выходные данные программы указывают на то, что удалено на два объекта больше, нежели создано. Объясняется это тем, что программа создает два дополнительных объекта, используя конструктор копирования по умолчанию. Конструктор копирования применяется для инициализации формального параметра функции `callme2()` во время вызова данной функции, а также для того, чтобы присвоить объекту `sailor` первоначальное значение `sports`. Конструктор копирования по умолчанию не озвучивает свою деятельность, поэтому не объявляет о своих созданиях и не увеличивает счетчик `num_strings`. Однако деструктор обновляет счетчик и он вызывается вплоть до уничтожения всех объектов, независимо от того, как они были сконструированы. Это недоразумение порождает проблему, поскольку оно означает, что программа не поддерживает точный подсчет объектов. Для решения этой проблемы нужно предусмотреть явный конструктор копирования, который обновляет счетчик:

```
String::String(const String & s)
{
    num_strings++;
    ...// остальной существенный код
}
```

Совет

Если в вашем классе содержатся статические данные-члены, чье значение изменяется при создании новых объектов, то необходимо предусмотреть явный конструктор копирования, который обеспечивает ведение учета.

Вторая странность более хитрая и опасная. Одним симптомом является бессмысленное содержимое строки:

```
headline2: Dû¤
```

Еще одним сигналом для беспокойства является то, что множество скомпилированных версий программы аварийно прерываются. Например, в Microsoft Visual C++ 7.1 (режим отладки) отображается сообщение об ошибке, в котором говорится, что утверждение отладки потерпело неудачу, а компилятор gpp выдает ошибку General Protection Fault (общее нарушение защиты). В других системах могут появляться различные сообщения или даже не появляться вообще, но внутри программы будут скрываться те же самые недостатки.

Причина состоит в том, что неявный конструктор копирования копирует по значению. Рассмотрим, к примеру, листинг 12.3. Результат, напомним, является следующим:

```
sailor.str = sport.str;
```

Данная команда не копирует строку, она копирует указатель на строку. Другими словами, после того, как для объекта `sailor` присвоено первоначальное значение `sports`, вы оказываетесь с двумя указателями на одну и ту же строку. Это не является проблемой, когда функция `operator<<()` использует указатель для отображения строки. Но это *становится* проблемой, когда вызывается деструктор. Вспомните,

что деструктор StringBad освобождает память, на которую указывает указатель str. Результат уничтожения sailor следующий:

```
delete [] sailor.str; // удаляется объект, на который указывает ditto.str
```

Указатель sailor.str направляет на строку "Шпинат поселился в трехлитровой банке с долларами", поскольку ему присвоено значение sports.str, которое указывает на данную строку. Таким образом, операция delete освобождает память, занимаемую строкой "Шпинат поселился в трехлитровой банке с долларами".

Далее, результат уничтожения sports следующий:

```
delete [] sports.str; // результат не определен
```

Здесь sports.str указывает на ту самую ячейку памяти, которая уже очищена деструктором для sailor. Из этого вытекает неопределенное, даже пагубное, поведение. В случае листинга 12.3 программа выводит запорченные строки, что является признаком неправильного управления памятью.

Решение проблемы с помощью явного конструктора копирования

Своего рода лекарством от проблем в дизайне класса может служить *глубокое копирование*. Другими словами, вместо того, чтобы просто копировать адрес строки, конструктор копирования должен дублировать строку и присваивать адрес дубликата члену str. Таким образом, каждый объект получает свою собственную строку вместо ссылки на строку другого объекта. При каждом вызове деструктора освобождаются различные строки, и не происходит повторных попыток освобождения одной и той же строки. Ниже показан пример, как может выглядеть конструктор копирования StringBad:

```
StringBad::StringBad(const StringBad & st)
{
   num_strings++;             // обработать обновление статического члена
   len = st.len;              // та же самая длина
   str = new char [len + 1]; // распределить память
   std::strcpy(str, st.str); // копировать строку в новое расположение
   cout << num_strings << ": объект \"" << str
        << "\" создан\n";     // информация для вас
}
```

Определение конструкторов копирования становится необходимым из-за того, что некоторые члены класса являются всего лишь указателями на данные, инициализированными операцией new, а не самими данными непосредственно. На рис. 12.3 иллюстрируется глубокое копирование.

Внимание!

Если класс содержит члены-указатели, инициализированные операцией new, потребуется определить конструктор копирования, который копирует указываемые данные, а не сами указатели. Это называется *глубоким копированием*. Альтернативная форма копирования (*почленное*, или *поверхностное копирование*) всего лишь копирует значения указателей. Поверхностная копия — это только "наружное соскабливание" информации указателя для копирования, а не глубокая "добыча", которая требует копирования конструкций, указываемых указателями.

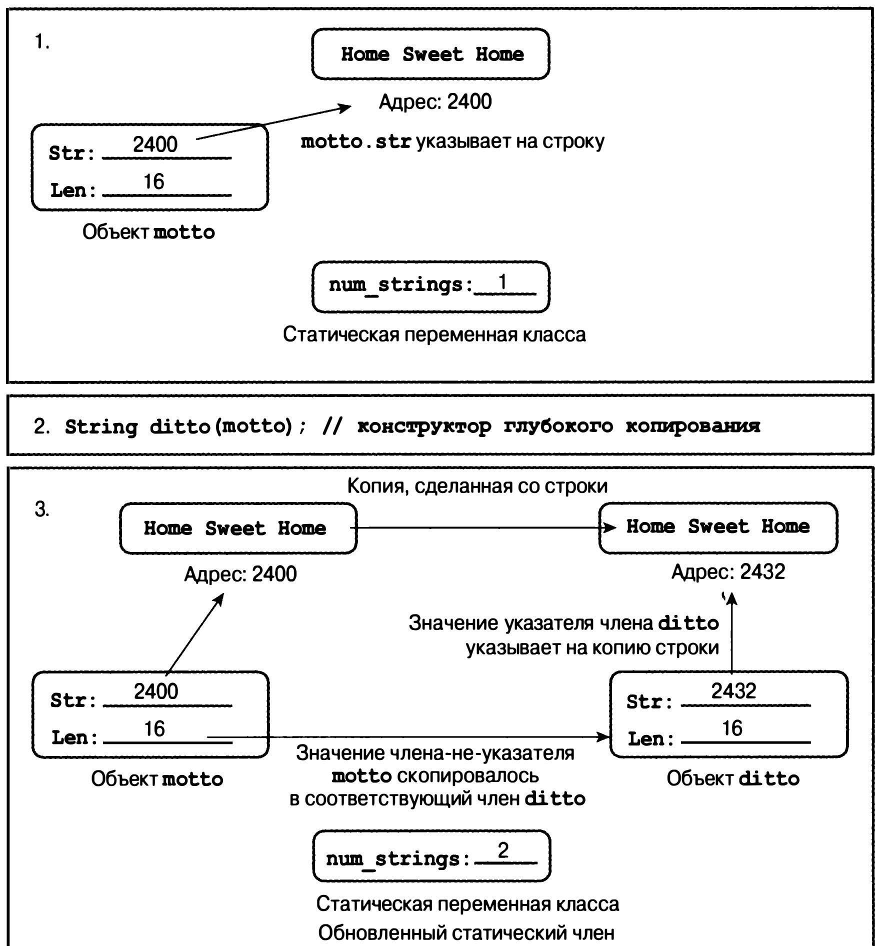


Рис. 12.3. Глубокое копирование изнутри

Операции присваивания

Не все проблемы в листинге 12.3 являются виной конструктора копирования по умолчанию. Стоит также рассмотреть операцию присваивания по умолчанию. Так же как ANSI C допускает присваивание структур, C++ допускает присваивание объектов класса. Это происходит путем автоматической перегрузки операции присваивания для класса. Данная операция имеет следующий прототип:

```
Имя_класса & Имя_класса::operator=(const Имя_класса &);
```

Другими словами, она принимает и возвращает ссылку на объект для класса. Например, прототип для класса StringBad выглядит так:

```
StringBad & StringBad::operator=(const StringBad &);
```

Когда используется операция присваивания

Перегруженная операция присваивания используется тогда, когда вы присваиваете один объект другому существующему объекту:

```
StringBad headline1("Сельдерей подкрался в полночь");
...
StringBad knot;
knot = headline1;         // вызывается операция присваивания
```

Во время инициализации объекта нет необходимости применять операцию присваивания:

```
StringBad metoo = knot;   // использовать конструктор копирования,
                          // возможно, также присваивание
```

Здесь metoo — это вновь созданный объект, который инициализирован значениями knot. Значит, используется конструктор копирования. Однако, как упоминалось ранее, реализации имеют возможность обработки данного операции в два этапа: использование конструктора копирования для создания временного объекта и затем использование присваивания для копирования значений в новый объект. Другими словами, инициализация всегда активизирует конструктор копирования, и формы, использующие операцию =, могут также обратиться к операции присваивания.

Что делает операция присваивания

По аналогии с конструктором копирования неявная реализация операции присваивания выполняет почленное копирование. Если сам член является объектом некоторого класса, то программа использует операцию присваивания, определенную для данного класса, для того, чтобы скопировать отдельный член. Статические данные-члены при этом не затрагиваются.

Где присваивание работает неправильно

В листинге 12.3 объекту knot присваивается значение headline1:

```
knot = headline1;
```

Когда для knot вызывается деструктор, он отображает следующее сообщение:

```
Объект "Сельдерей подкрался в полночь" удален, 2 осталось
```

Когда деструктор вызывается для headline1, он отображает такое сообщение:

```
Объект "-|" удален, -2 осталось
```

(Некоторые реализации аварийно завершают работу еще до вывода такого сообщения.)

Здесь встречается та же проблема, которую вызывал конструктор копирования: запорченные данные. И снова проблема заключается в почленном копировании, при котором и headline1.str, и knot.str указывают на один и тот же адрес. Таким образом, при вызове деструктора для knot строка "Сельдерей подкрался в полночь" удаляется, а при вызове деструктора для headline1 предпринимается попытка удалить уже удаленную строку. Как отмечалось ранее, результат попытки удалить ранее удаленные данные не определен, поэтому это может изменить содержимое памяти и привести к аварийному завершению программы. Если результат отдельной операции

не определен, то компилятор может делать все, что ему заблагорассудится, включая вывод на экран "Декларации о независимости" или удаление с жесткого диска не понравившихся ему файлов.

Исправление присваивания

Решением проблем, созданных несоответствующими операциями присваивания по умолчанию, может стать определение собственной операции присваивания, которая выполняет глубокое копирование. Реализация аналогична конструктору копирования, кроме нескольких отличий:

- Поскольку целевой объект может уже ссылаться на данные, для которых ранее была распределена память, то функция должна использовать операцию `delete []` для ее освобождения.
- Функция должна защищать от присваивания объекта самому себе; иначе освобождение памяти, описанное выше, может стереть содержимое объекта до того, как оно переназначено.
- Функция возвращает ссылку на вызывающий объект.

Возвращая объект, функция может эмулировать цепочку обычных присваиваний для встроенных типов. То есть, если `S0`, `S1` и `S2` являются объектами `StringBad`, то можно записать следующее:

```
S0 = S1 = S2;
```

В представлении функции это выглядит следующим образом:

```
S0.operator=(S1.operator=(S2));
```

Таким образом, возвращаемое значение функции `S1.operator=(S2)` становится аргументом функции `S0.operator=()`. Поскольку возвращаемое значение является ссылкой на объект `String`, то это корректный тип аргумента.

А вот как можно реализовать операцию присваивания для класса `StringBad`:

```
StringBad & StringBad::operator=(const StringBad & st)
{
   if (this == &st)  // объект присвоен сам себе
      return *this;  // все сделано
   delete [] str;    // освободить старую строку
   len = st.len;
   str = new char [len + 1];    // получить пространство для новой строки
   std::strcpy(str, st.str);    // копировать строку
   return *this;                // возвратить ссылку на вызывающий объект
}
```

Сначала код проверяет наличие присваивания самому себе. Это происходит путем сравнения адреса в правой части присваивания (`&s`) и адреса принимающего объекта (`this`). Если они совпадают, функция возвращает `*this` и завершается. Вы можете вспомнить из главы 10, что операция присваивания может быть перегружена только с помощью функции-члена класса.

В противном случае функция переходит к освобождению памяти, на которую указывает `str`. Это необходимо из-за того, что впоследствии указателю `str` будет присвоен адрес новой строки. Если вы не обратитесь сначала к операции `delete`,

то предыдущая строка останется в памяти. Поскольку функция больше не содержит указатель на старую строку, память будет тратиться напрасно.

Далее функция ведет себя как конструктор копирования: распределяет достаточное пространство для новой строки и затем копирует строку из правого объекта в новое место.

После завершения функция возвращает `*this` и завершается.

Присваивание не создает новый объект, поэтому вам не нужно корректировать значение статических данных-членов `num_strings`.

Добавление конструктора копирования и операции присваивания, как описывалось выше, в класс `StringBad` устраняет все проблемы. Ниже показано несколько последних строк выходных данных, которые получены после всех указанных изменений:

```
Конец функции main()
Объект "Сельдерей подкрался в полночь" удален, 4 осталось
Объект "Шпинат поселился в трехлитровой банке с долларами" удален, 3 осталось
Объект "Шпинат поселился в трехлитровой банке с долларами" удален, 2 осталось
Объект "Салат-латук вышел на охоту" удален, 1 осталось
Объект "Сельдерей подкрался в полночь" удален, 0 осталось
```

Теперь подсчет объектов ведется правильно и ни одна из строк не запорчена.

Новый усовершенствованный класс String

Теперь, когда вы уже знаете немного больше, вы можете модифицировать класс `StringBad`, переименовав его в `String`. Во-первых, нужно добавить конструктор копирования и операцию присваивания, как обсуждалось чуть выше, для того, чтобы класс корректно управлял памятью, используемой объектами класса. Во-вторых, вы уже увидели, когда создаются и уничтожаются объекты. Вы можете отключить конструкторы и деструкторы, чтобы они не больше уведомляли о своем использовании. Также теперь вы не наблюдаете за конструкторами во время работы, поэтому конструктор по умолчанию можно упростить: в качестве его содержимого вместо `"C++"` указать пустую строку.

После этого вы можете добавить некоторые новые возможности в класс. Полезный класс `String` может объединять в себе все функциональные возможности стандартной библиотеки `cstring` строковых функций. Однако имеет смысл добавить лишь столько из них, чтобы их оказалось достаточно, дабы увидеть, что происходит. (Помните, что рассматриваемый класс `String` служит учебным примером, и что стандартный C++ класс `string` значительно шире.) В частности, можно добавить следующие методы:

```
int length () const { return len; }
friend bool operator<(const String &st, const String &st2);
friend bool operator>(const String &st1, const String &st2);
friend bool operator==(const String &st, const String &st2);
friend operator>>(istream & is, String & st);
char & operator[](int i);
const char & operator[](int i) const;
static int HowMany();
```

Первый из новых методов возвращает длину хранимой строки. Следующие три дружественных функции позволяют сравнивать строки. Функция `operator>>()` обеспечивает возможности простого ввода. Две функции `operator[]()` предоставляют доступ в нотации массива к отдельным символам в строке. Статический метод класса `HowMany()` дополняет статический член класса данных `num_strings`. Давайте обсудим все это подробнее.

Пересмотренный конструктор по умолчанию

Новый конструктор по умолчанию заслуживает внимания. Он выглядит следующим образом:

```
String::String()
{
   len = 0;
   str = new char[1];
   str[0] = '\0'; // строка по умолчанию
}
```

Вас может заинтересовать, почему в коде применяется

```
str = new char[1];
```

а не

```
str = new char;
```

Обе формы распределяют одинаковый объем памяти. Различие состоит в том, что первая форма совместима с деструктором класса, а вторая нет. Вспомните, что деструктор содержит следующий код:

```
delete [] str;
```

Использование операции `delete []` совместимо с указателями, инициализированными операцией `new []`, и с нулевым указателем. Поэтому еще одной возможностью является замена кода

```
str = new char[1];
str[0] = '\0'; // строка по умолчанию
```

таким кодом:

```
str = 0; // устанавливает str на нулевой указатель
```

Результат использования `delete []` с любыми указателями, инициализированными другим способом, не определен:

```
char words[15] = "плохая идея";
char * p1= words;
char * p2 = new char;
char * p3;
delete [] p1; // не определено, поэтому не делайте так
delete [] p2; // не определено, поэтому не делайте так
delete [] p3; // не определено, поэтому не делайте так
```

Члены сравнения

Три метода в классе String производят сравнения. Функция operator<() возвращает значение true, если первая строка идет раньше второй в алфавитном порядке (или, более строго, в сортирующей последовательности). Наиболее простой способ реализации функций сравнения строк предусматривает использование стандартной функции strcmp(). Она возвращает отрицательное значение, если первый аргумент предшествует второму по алфавиту, 0, если строки одинаковые, и положительное значение, если первая строка по алфавиту следует за второй. Вы можете применять strcmp() следующим образом:

```
bool operator<(const String &st1, const String &st2)
{
   if (std::strcmp(st1.str, st2.str) > 0)
      return true;
   else
      return false;
}
```

Поскольку встроенная операция > уже возвращает значение типа bool, вы можете далее упростить код:

```
bool operator<(const String &st1, const String &st2)
{
   return (std::strcmp(st1.str, st2.str) < 0);
}
```

По аналогии можно закодировать оставшиеся две функции сравнения:

```
bool operator>(const String &st1, const String &st2)
{
   return st2.str < st1.str;
}
bool operator==(const String &st1, const String &st2)
{
   return (std::strcmp(st1.str, st2.str) == 0);
}
```

Первое определение выражает операцию > в терминах операции < и может служить хорошим кандидатом на встроенную функцию.

Создание дружественных функций сравнения облегчает сравнение объектов String и стандартных строк С. Например, предположим, что answer — это объект String, и что имеется следующий код:

```
if ("love" == answer)
```

Он транслируется в такой код:

```
if (operator==("love", answer))
```

Затем компилятор использует один из конструкторов для преобразования кода г следующему виду:

```
if (operator==(String("love"), answer))
```

Это соответствует прототипу.

Доступ к символам с помощью скобочной нотации

В стандартных строках С-стиля можно использовать скобки для доступа к отдельным символам:

```
char city[40] = "Amsterdam";
cout << city[0] << endl; // отобразить букву А
```

В C++ два символа скобок образуют одну операцию — операцию скобок. Вы можете перегрузить упомянутую операцию с помощью метода operator[](). Как правило, бинарная операция C++ (с двумя операндами) предусматривает наличие знака операции между двумя операндами, например, 2 + 5. А в случае операции скобок один операнд располагается перед первой скобкой, а второй — между двумя скобками. Соответственно, для выражения city[0] справедливо утверждать, что city — это первый операнд, [] — операция, а 0 — второй операнд.

Предположим, что opera является объектом String:

```
String opera("The Magic Flute");
```

Если вы используете выражение opera[4], C++ ищет метод со следующим именем и сигнатурой:

```
operator[](int i)
```

Если он находит соответствующий прототип, то компилятор заменяет выражение opera[4] вызовом данной функции:

```
opera.operator[](4)
```

Объект opera активизирует метод, а индекс массива 4 становится аргументом функции.

Ниже показан пример простой реализации операции скобок:

```
char & String::operator[](int i)
{
   return str[i];
}
```

При таком определении оператор

```
cout << opera[4];
```

становится следующим:

```
cout << opera.operator[](4);
```

Возвращается значение opera.str[4], или символ 'M'. Таким образом, общедоступный метод предоставляет доступ к приватным данным.

Если объявить возвращаемый тип как char &, то это позволит присваивать значения отдельным элементам. Например, можно использовать следующий код:

```
String means("might");
means[0] = 'r';
```

Второй оператор преобразуется в вызов функции перегруженной операции:

```
means.operator[][0] = 'r';
```

Это код присваивает 'r' возвращаемому значению метода. Но функция возвращает ссылку на means.str[0], делая код эквивалентным следующему:

```
means.str[0] = 'r';
```

Последняя строка кода нарушает приватный доступ, но, поскольку операция operator[]() является методом класса, она допускает изменения содержимого массива. Практическим результатом кода является то, что "might" становится "right".

Предположим, что имеется константный объект:

```
const String answer("futile");
```

Затем, если единственным доступным определением operator[]() является то, которое вы только что видели, то следующий код помечен как ошибочный:

```
cout << answer[1]; // ошибка компиляции
```

Причина в том, что answer является const, а метод не гарантирует отсутствие изменений в данных. (Фактически, изменение данных — это работа метода, поэтому он и не может гарантировать этого.)

Однако при перегрузке C++ различает сигнатуры функций типа const и не-const, поэтому вы можете предусмотреть вторую версию operator[](), которая будет использоваться только объектами const String:

```
// для использования с объектами const String
const char & String::operator[](int i) const
{
   return str[i];
}
```

При таких определениях вы будете иметь доступ для чтения/записи к стандартным объектам String и доступ только для чтения к данным const String:

```
String text("Once upon a time");
const String answer("futile");
cout << text[1];//нормально, используется версия operator[]() типа не-const
cout << answer[1];//нормально, используется версия operator[]() типа const
cin >> text[1];//нормально, используется версия operator[]() типа не-const
cin >> answer[1]; // ошибка компиляции
```

Статические функции-члены класса

Функцию-член можно объявить как статическую. (Ключевое слово static должно присутствовать в объявлении, а не в определении функции, если последнее размещается отдельно.) Это влечет за собой два важных следствия.

Во-первых, статическую функцию-член не обязательно активизировать через объект, фактически, она даже не получает указатель this. Если статическая функция-член объявляется в разделе public, то ее можно активизировать, используя имя класса и операцию разрешения контекста. Для класса String можно реализовать статическую функцию-член под именем HowMany() с помощью следующего прототипа в объявлении класса:

```
static int HowMany() { return num_strings; }
```

Вызвать ее можно так:

```
int count = String::HowMany(); // вызов статической функции-члена
```

Вторым следствием является то, что поскольку статическая функция-член не связана ни с одним объектом, то она может использовать только статические данные-члены. Например, статический метод `HowMany()` может получить доступ к статическому члену `num_strings` и не может — к членам `str` или `len`.

Аналогично статическая функция-член может применяться для установки флага, глобального для класса, который управляет поведением некоторых аспектов интерфейса класса. Например, он может управлять форматированием, которое использует метод, отображающий содержимое класса.

Последующая перегрузка операции присваивания

Прежде чем рассматривать новые листинги для примеров класса `String`, давайте обсудим еще один вопрос. Допустим, вы хотите скопировать обычную строку в объект `String`. Например, вы используете функцию `getline()` для того, чтобы прочитать строку, и хотите поместить ее в объекте `String`. Методы класса уже позволяют сделать следующее:

```
String name;
char temp[40];
cin.getline(temp, 40);
name = temp; // использовать конструктор для преобразования типа
```

Однако если вы должны делать это часто, то данное решение может оказаться неудовлетворительным. Чтобы понять почему, давайте проанализируем, как работают эти операторы:

1. Программа использует конструктор `String(const char *)` для создания временного объекта `String`, содержащего копию строки, которая хранится в `temp`. Вспомните из главы 11, что конструктор с одним аргументом служит в качестве функции преобразования.
2. В листинге 12.6 (ниже в данной главе) программа использует функцию `String & String::operator=(const String &)` для копирования информации из временного объекта в объект `name`.
3. Программа вызывает деструктор `~String()` для удаления временного объекта.

Самым простым способом увеличения эффективности процесса является перегрузка операции присваивания таким образом, чтобы она работала непосредственно с обычными строками. Это устранит дополнительные шаги по созданию и удалению временного объекта. Ниже показана одна из возможных реализаций:

```
String & String::operator=(const char * s)
{
   delete [] str;
   len = std::strlen(s);
   str = new char[len + 1];
   std::strcpy(str, s);
   return *this;
}
```

Как обычно, необходимо освободить память, ранее управляемую указателем `str`, и выделить достаточный объем памяти для новой строки.

В листинге 12.4 представлено пересмотренное объявление класса. В дополнение к уже упомянутым изменениям, в нем определяется константа `CINLIM`, которая используется при реализации `operator>>()`.

Листинг 12.4. string1.h

```
// string1.h -- исправленное и расширенное объявление строкового класса
#include <iostream>
using std::ostream;
using std::istream;

#ifndef STRING1_H_
#define STRING1_H_
class String
{
private:
   char * str; // указатель на строку
   int len; // длина строки
   static int num_strings; // число объектов
   static const int CINLIM = 80; // предел ввода для cin
public:
// конструкторы и другие методы
   String(const char * s); // конструктор
   String(); // конструктор по умолчанию
   String(const String &); // конструктор копирования
   ~String(); // деструктор
   int length () const { return len; }
// методы перегруженных операций
   String & operator=(const String &);
   String & operator=(const char *);
   char & operator[](int i);
   const char & operator[](int i) const;
// дружественные функции перегруженных операций
   friend bool operator<(const String &st, const String &st2);
   friend bool operator>(const String &st1, const String &st2);
   friend bool operator==(const String &st, const String &st2);
   friend ostream & operator<<(ostream & os, const String & st);
   friend istream & operator>>(istream & is, String & st);
// статическая функция
   static int HowMany();
};
#endif
```

Замечание по совместимости

Вы можете иметь дело с компилятором, который не воспринимает тип `bool`. В таком случае можно использовать `int` вместо `bool`, `0` вместо `false` и `1` вместо `true`. Если ваш компилятор не поддерживает статические константы класса, можете определить `CINLIM` как перечисление:

```
enum {CINLIM = 90};
```

В листинге 12.5 представлены пересмотренные определения методов.

Листинг 12.5. string1.cpp

```
// string1.cpp -- методы класса String
#include <cstring> // string.h для некоторых компиляторов
#include "string1.h" // включить <iostream>
using std::cin;
using std::cout;
// инициализация статического члена класса
int String::num_strings = 0;
// статический метод
int String::HowMany()
{
    return num_strings;
}

// методы класса
String::String(const char * s) // конструировать String из строки C
{
    len = std::strlen(s);        // установить размер
    str = new char[len + 1];     // распределить память
    std::strcpy(str, s);         // инициализировать указатель
    num_strings++;               // установить счетчик объектов
}

String::String()                 // конструктор по умолчанию
{
    len = 4;
    str = new char[1];
    str[0] = '\0';               // строка по умолчанию
    num_strings++;
}

String::String(const String & st)
{
    num_strings++;               // обработать обновление статического члена
    len = st.len;                // та же самая длина
    str = new char [len + 1];    // распределить пространство
    std::strcpy(str, st.str);    // копировать строку в новое расположение
}

String::~String()                // необходимый деструктор
{
    --num_strings;               // требуется
    delete [] str;               // требуется
}

// методы перегруженных операций

// присвоить объекту String объект String
String & String::operator=(const String & st)
{
    if (this == &st)
        return *this;
```

```
    delete [] str;
    len = st.len;
    str = new char[len + 1];
    std::strcpy(str, st.str);
    return *this;
}

// присвоить строку С объекту String
String & String::operator=(const char * s)
{
    delete [] str;
    len = std::strlen(s);
    str = new char[len + 1];
    std::strcpy(str, s);
    return *this;
}

// символьный доступ для чтения/записи к объекту String типа не-const
char & String::operator[](int i)
{
    return str[i];
}

// символьный доступ только для чтения к объекту String типа const
const char & String::operator[](int i) const
{
    return str[i];
}

// дружественные функции перегруженных операций

bool operator<(const String &st1, const String &st2)
{
    return (std::strcmp(st1.str, st2.str) < 0);
}

bool operator>(const String &st1, const String &st2)
{
    return st2.str < st1.str;
}

bool operator==(const String &st1, const String &st2)
{
    return (std::strcmp(st1.str, st2.str) == 0);
}

// простой вывод String
ostream & operator<<(ostream & os, const String & st)
{
    os << st.str;
    return os;
}
```

```
// быстрый и неформатированный ввод String
istream & operator>>(istream & is, String & st)
{
   char temp[String::CINLIM];
   is.get(temp, String::CINLIM);
   if (is)
        st = temp;
   while (is && is.get() != '\n')
        continue;
   return is;
}
```

Перегруженная операция >> предусматривает простой способ для чтения строки, введенной с клавиатуры, в объект `String`. Она принимает введенную строку из `String::CINLIM` или менее символов и отбрасывает все символы сверх установленного предела. Помните, что значение объекта `istream` в условии `if` оценивается как `false`, если ввод данных по каким-то причинам аварийно прерывается (например, случайное появление условия конца файла или, в случае `get(char *, int)`, чтение пустой строки.)

Короткая программа в листинге 12.6 тестирует класс `String`, позволяя ввести несколько строк. Программа запрашивает у пользователя ввод пословиц, помещает строки в объекты `String`, отображает их и выдает отчет о том, какая строка самая короткая, и какая идет первой в алфавитном порядке.

Листинг 12.6. sayings1.cpp

```
// sayings1.cpp -- использование расширенного класса String
// компилировать вместе с string1.cpp
#include <iostream>
#include "string1.h"
const int ArSize = 10;
const int MaxLen =81;
int main()
{
   using std::cout;
   using std::cin;
   using std::endl;
   String name;
   cout <<"Как вас зовут?\n>> ";
   cin >> name;

   cout << name << ", пожалуйста, введите до " << ArSize
        << " коротких пословиц <или пустую строку для завершения>:\n";
   String sayings[ArSize]; // массив объектов
   char temp[MaxLen];      // временная память строк
   int i;
   for (i = 0; i < ArSize; i++)
   {
      cout << i+1 << ": ";
      cin.get(temp, MaxLen);
      while (cin && cin.get() != '\n')
         continue;
```

```
    if (!cin || temp[0] == '\0') // пустая строка?
        break; // i не инкрементируется
    else
        sayings[i] = temp; // перегруженное присваивание
  }
  int total = i; //общее количество прочитанных строк

  cout << "Вы ввели следующие пословицы:\n";
  for (i = 0; i < total; i++)
    cout << sayings[i][0] << ": " << sayings[i] << endl;

  int shortest = 0;
  int first = 0;
  for (i = 1; i < total; i++)
  {
    if (sayings[i].length() < sayings[shortest].length())
        shortest = i;
    if (sayings[i] < sayings[first])
        first = i;
  }
  cout << "Кратчайшая пословица:\n" << sayings[shortest] << endl;;
  cout << "Первая пословица по алфавиту:\n" << sayings[first] << endl;
  cout << "Эта программа использовала "<< String::HowMany()
       << " объектов String. Всего наилучшего.\n";
  return 0;
}
```

Замечание по совместимости

Более старые версии `get(char *, int)` не устанавливают значение `false` при чтении пустой строки. В таких версиях при вводе пустой строки первым символом в строке считается нулевой символ. В данном примере используется следующий код:

```
if (!cin || temp[0] == '\0')  // пустая строка?
   break;  // i не инкрементируется
```

Если реализация поддерживает текущий стандарт C++, то первая проверка в операторе `if` обнаруживает пустую строку, тогда как для более старых реализаций пустая строка обнаруживается при второй проверке.

Программа из листинга 12.6 предлагает пользователю ввести до 10 пословиц. Каждая пословица считывается во временный символьный массив и затем копируется в объект `String`. Если пользователь вводит пустую строку, то оператор `break` прерывает цикл ввода. После отображения введенных данных программа использует функции-члены `length()` и `operator<()` для нахождения самой короткой и первой в алфавитном порядке строки. Программа также применяет индексную операцию (`[]`) для того, чтобы разместить перед каждой пословицей ее начальный символ. Рассмотрим пример выполнения этой программы:

```
Как вас зовут?
>> Misty Gutz
Misty Gutz, пожалуйста, введите до 10 коротких пословиц <или пустую
строку для завершения>:
1: a fool and his money are soon parted
2: penny wise, pound foolish
```

```
3: the love of money is the root of much evil
4: out of sight, out of mind
5: absence makes the heart grow fonder
6: absinthe makes the hart grow fonder
7:
Вы ввели следующие пословицы:
a: a fool and his money are soon parted
p: penny wise, pound foolish
t: the love of money is the root of much evil
o: out of sight, out of mind
a: absence makes the heart grow fonder
a: absinthe makes the hart grow fonder
Кратчайшая пословица:
penny wise, pound foolish
Первая пословица по алфавиту:
a fool and his money are soon parted
Эта программа использовала 11 объектов String. Всего наилучшего.
```

О чем следует помнить при использовании операции new в конструкторах

Теперь вы уже должны осознавать, что нужно быть особенно внимательными при использовании операции `new` для инициализации членов-указателей объекта. В частности, необходимо делать следующее:

- Если вы применяете операцию `new` для инициализации члена-указателя в конструкторе, то нужно использовать операцию `delete` в деструкторе.
- Использование операций `new` и `delete` должно быть согласованным. Нужно сочетать операцию `new` с операцией `delete`, а `new []` — с `delete []`.
- Если применяется несколько конструкторов, то все они должны использовать операцию `new` с одним и тем же синтаксисом — либо все со скобками, либо все без скобок. Поскольку существует только один деструктор, то все конструкторы должны быть совместимыми с ним. При этом допустимо инициализировать указатель с помощью операции `new` в одном конструкторе и с помощью нулевого указателя (`NULL` или `0`) — в другом, поскольку к нулевому указателю можно применять операцию `delete` (со скобками или без них).

NULL или 0?

Нулевой указатель может быть представлен как `0` или как `NULL` (символьная константа, определенная как `0` в нескольких заголовочных файлах). Программисты, работающие на языке С, часто используют `NULL` вместо `0` в качестве зрительного напоминания о том, что значение является указателем, также как они используют `'\0'` вместо `0` для символа пробела в качестве зрительного напоминания о том, что значение является символом. Однако в С++ традиционно отдают предпочтение простому `0` вместо эквивалентного ему `NULL`.

- Необходимо определить конструктор копирования, который инициализирует один объект другим с использованием глубокого копирования. Как правило, конструктор должен выглядеть подобно следующему коду:

```
String::String(const String & st)
{
   num_strings++;   // обработать обновление статического члена,
                    // если необходимо
   len = st.len;    // та же самая длина
   str = new char [len + 1]; // распределить пространство
   std::strcpy(str, st.str);// копировать строку в новое расположение
}
```

  В частности, конструктор копирования должен распределять пространство под хранение копируемых данных, и также копировать эти данные, а не только их адрес. При этом он должен обновлять все статические члены класса, чьи значения затрагиваются данных процессом.

- Необходимо определить операцию присваивания, которая копирует один объект в другой с использованием глубокого копирования. Как правило, метод класса должен выглядеть подобно следующему коду:

```
String & String::operator=(const String & st)
{
   if (this == &st) // объект присвоен сам себе
      return *this; // все сделано
   delete [] str;   // освободить старую строку
   len = st.len;
   str = new char [len + 1]; // получить пространство для новой строки
   std::strcpy(str, st.str);// копировать строку
   return *this;            // возвратить ссылку на вызывающий объект
}
```

  В частности, метод проверяет наличие присваивания самому себе, освобождает память, на которую ранее указывал член-указатель, копирует данные, а не только их адрес, и возвращает ссылку на вызывающий объект.

В следующем фрагменте кода представлены два примера, показывающие чего делать не стоит, и один пример хорошего конструктора:

```
String::String()
{
   str = "default string";   // стоп, не хватает new []
   len = std::strlen(str);
}

String::String(const char * s)
{
   len = std::strlen(s);
   str = new char;           // стоп, не хватает []
   std::strcpy(str, s);      // стоп, нет места
}

String::String(const String & st)
{
   len = st.len;
```

```
    str = new char[len + 1];  // годится, распределить пространство
    std::strcpy(str, st.str);// годится, копировать значение
}
```

В первом конструкторе недостает вызова new для инициализации str. Деструктор, вызываемый для объекта по умолчанию, применяет операцию delete к str. Результат использования операции delete с указателем, который не был инициализирован с помощью new, не определен, но, вероятно, окажется плохим. Подойдет один из следующих вариантов:

```
String::String()
{
    len = 0;
    str = new char[1];  // использует new с []
    str[0] = '\0';
}

String::String()
{
    len = 0;
    str = NULL;         // или эквивалентное выражение str = 0;
}

String::String()
{
    static const char * s = "C++";  // инициализируется только однажды
    len = std::strlen(s);
    str = new char[len + 1];        // использует new с []
    std::strcpy(str, s);
}
```

Второй конструктор в исходном отрывке применяет операцию new, но терпит неудачу при запросе соответствующего объема памяти. Поэтому операция new возвращает блок, содержащий пространство только для одного символа. При попытке скопировать более длинную строку в данную ячейку появляется сообщение о проблемах с памятью. К тому же использование new без скобок несовместимо с соответствующей формой других конструкторов.

Третий конструктор работает без ошибок.

В завершение рассмотрим пример деструктора, который *не будет* работать соответствующим образом с предыдущими конструкторами:

```
String::~String()
{
    delete str; // стоп, нужно использовать delete [] str;
}
```

В деструкторе неправильно используется delete. В связи с тем, что конструкторы запрашивают массив символов, деструктор должен удалять массив.

Замечания о возвращаемых объектах

При возврате объекта функцией-членом или автономной функцией возможны следующие варианты. Функция может возвращать ссылку на объект, константную

ссылку на объект, объект или константный объект. К данному моменту вы уже видели все примеры, кроме последнего, поэтому сейчас самое время рассмотреть данные опции.

Возврат ссылки на объект const

Основной причиной использования ссылки const является производительность, но на применение этого варианта существует несколько ограничений. Если функция возвращает объект, который ей передается (либо путем вызова объекта, либо в качестве аргумента метода), то вы имеете возможность увеличить производительность метода, заставляя его передавать ссылку. Например, предположим, что вам нужно написать функцию Max(), которая возвращает больший из двух объектов Vector, где Vector — это класс, разработанный в главе 11. Функция используется следующим образом:

```
Vector force1(50,60);
Vector force2(10,70);
Vector max;
max = Max(force1, force2);
```

Обе следующие реализации будут работать:

```
// версия 1
Vector Max(const Vector & v1, const Vector & v2)
{
   if (v1.magval() > v2.magval())
      return v1;
   else
      return v2;
}

// версия 2
const Vector & Max(const Vector & v1, const Vector & v2)
{
   if (v1.magval() > v2.magval())
      return v1;
   else
      return v2;
}
```

Здесь необходимо отметить три важных момента. Во-первых, вспомните о том, что возврат объекта активизирует конструктор копирования, тогда как возврат ссылки — нет. Соответственно, версия 2 выполняет меньше работы и является более эффективной. Во-вторых, ссылка должна указывать на объект, который существует, когда выполняется вызываемая функция. В данном примере содержится ссылка либо на force1, либо на force2, причем оба объекта определены в вызываемой функции, поэтому указанное требование выполняется. В-третьих, и v1, и v2 заявлены как ссылки const, поэтому для соответствия возвращаемый тип должен быть const.

Возврат ссылки на объект не-const

Двумя общими примерами возврата объекта не-const могут служить перегрузка операции присваивания и перегрузка операции << для использования с cout. Первое делается с целью повышения производительности, а второе — по необходимости.

Значение, возвращаемое operator=(), используется для присваивания в виде цепочки:

```
String s1("Good stuff");
String s2, s3;
s3 = s2 = s1;
```

В данном коде значение, возвращаемое s2.operator=(s1), присваивается s3. При этом можно использовать как объект String, так и ссылку на объект String. Однако, как в примере с объектом Vector, применение ссылки позволяет функции избежать вызова конструктора копирования String для создания нового объекта String. В таком случае возвращаемый тип не является const, поскольку метод operator=() возвращает ссылку на s2, который модифицируется.

Значение, возвращаемое operator<<(), используется для цепного присваивания в виде цепочки:

```
String s1("Good stuff");
cout << s1 << "is coming!";
```

Здесь значение, возвращаемое методом operator<<(cout, s1), становится объектом, который применяется для отображения строки "is coming!". Возвращаемый тип должен быть ostream &, а не только ostream. Использование типа ostream потребует вызова конструктора копирования ostream, но, как оказывается, класс ostream не имеет общедоступного конструктора копирования. К счастью, возврат ссылки на cout не вызывает проблем, поскольку cout уже содержится в области действия вызываемой функции.

Возврат объекта

Если объект, возвращаемый функцией, является локальным для вызвавшей функции, он не должен возвращаться по ссылке, поскольку локальный объект имеет собственный деструктор, вызываемый при завершении функции. Таким образом, когда управление возвращается в вызвавшую функцию, тот объект, на который может указывать ссылка, уже не существует. В таком случае вы должны возвращать объект, а не ссылку. Как правило, перегруженные арифметические операции попадают в эту категорию. Давайте рассмотрим пример, в котором снова используется класс Vector:

```
Vector force1(50,60);
Vector force2(10,70);
Vector net;
net = force1 + force2;
```

Возвращаемое значение не является ни force1, ни force2, которые должны остаться неизменными после обработки. Соответственно, возвращаемое значение не может быть ссылкой на объект, который уже представлен в вызывающей функции. При этом сумма является новым временным объектом, вычисляемым в Vector::operator+(), и функция также не должна возвращать ссылку на временный объект. Она должна возвращать фактический векторный объект, а не ссылку на него:

```
Vector Vector::operator+(const Vector & b) const
{
   return Vector(x + b.x, y + b.y);
}
```

Здесь дополнительные затраты приходятся на вызов конструктора копирования для создания возвращаемого объекта, но это неизбежно.

Еще одно наблюдение: в примере `Vector::operator+()` вызов конструктора `Vector(x+ b.x, y + b.y)` создает объект, который доступен для метода `operator+()`. Неявный вызов конструктора копирования, порождаемый оператором `return`, при этом создает объект, который доступен вызывающей программе.

Возврат объекта const

В предыдущем определении `Vector::operator+()` содержится странное свойство. Планируется следующее его использование:

```
net = force1 + force2; // 1: три объекта Vector
```

Однако определение также позволяет использовать его так:

```
force1 + force2 = net; // 2: неудобоваримое программирование
cout << (force1 + force2 = net).magval() << endl;
                       // 3: безумное программирование
```

Немедленно возникают три вопроса. Для чего кто-то может писать подобные операторы? Почему они допустимы? Что они делают?

Во-первых, нет ни одной здравой причины для написания подобного кода, но далеко не все коды создаются с осмысленными целями. Люди, даже программисты, допускают ошибки. Например, если для класса `Vector` была определена операция `operator==()`, то вы можете ошибочно напечатать

```
if (force1 + force2 = net)
```

вместо такого:

```
if (force1 + force2 == net)
```

Также программисты часто стремятся прослыть оригинальными, а это может привести к опасным ошибкам.

Во-вторых, данный код допустим, поскольку конструктор копирования создает временный объект для представления возвращаемого объекта. Так, в предыдущем коде выражение `force1 + force2` символизирует данный временный объект. В операторе 1 временный объект присваивается переменной `net`. В операторах 2 и 3 переменная `net` присваивается временному объекту.

В-третьих, временный объект используется и затем отбрасывается. Например, в операторе 2 вычисляется сумма переменных `force1` и `force2`, ответ копируется во временный объект возврата, переписывается содержимое `net`, затем временный объект удаляется. Все исходные векторы остаются без изменений. В операторе 3 значение временного объекта отображается до его удаления.

Если вас беспокоят возможные проблемы, обусловленные таким поведением, вы можете воспользоваться простым способом спасения. Объявите возвращаемый тип как объект `const`. Например, если `Vector::operator+()` объявлен с возвращаемым типом `const Vector`, то оператор 1 остается допустимым, тогда как операторы 2 и 3 — нет.

Подведем итоги. Если метод или функция возвращает локальный объект, то должен возвращаться сам объект, а не ссылка. В данном примере программа использует конструктор копирования для создания возвращаемого объекта. Если метод или функция возвращает объект класса, для которого нет общедоступного конструктора копирования (например, класс ostream), то должна возвращаться ссылка на объект. В заключение некоторые методы и функции (такие как перегруженная операция присваивания) могут возвращать как объект, так и ссылку на объект. В данном примере ссылка предпочтительнее по причинам, связанным с производительностью.

Использование указателей на объекты

В программах на C++ часто применяются указатели на объекты, поэтому давайте немного попрактикуемся в этом вопросе. В листинге 12.6 используются значения индексов массива для отслеживания самой короткой строки и первой строки в алфавитном порядке. Другим примером может послужить использование указателей для указания на текущих лидеров в данных категориях. В листинге 12.7 реализуется этот подход с использованием двух указателей на объекты String. Первоначально указатель shortest указывает на первый объект в массиве. Всякий раз, когда программа находит объект с более короткой строкой, она устанавливает указатель shortest на данный объект. По аналогии указатель first следит за предшествующей по алфавитному порядку строкой. Обратите внимание, что эти два указателя не создают новых объектов, они просто указывают на существующие объекты. Соответственно, они не требуют применения операции new для выделения дополнительной памяти.

Для разнообразия программа в листинге 12.7 использует указатель, который отслеживает новые объекты:

```
String * favorite = new String(sayings[choice]);
```

Здесь указатель favorite обеспечивает доступ к безымянному объекту, созданному операцией new. Данный особый синтаксис обозначает инициализацию нового объекта String с использованием объекта sayings[choice]. Последний активизирует конструктор копирования, поскольку тип аргумента для конструктора копирования (const String &) соответствует значению инициализации (sayings[choice]). Программа использует функции srand(), rand() и time() для случайного выбора значения.

Листинг 12.7. sayings2.cpp

```
// sayings2.cpp -- использование указателей на объекты
// компилировать вместе с string1.cpp
#include <iostream>
#include <cstdlib> // (или stdlib.h) для rand(), srand()
#include <ctime>   // (или time.h) для time()
#include "string1.h"
const int ArSize = 10;
const int MaxLen = 81;
int main()
{
   using namespace std;
   String name;
   cout <<"Как вас зовут?\n>> ";
   cin >> name;
```

```
    cout << name << ", пожалуйста, введите до " << ArSize
         << " коротких пословиц <или пустую строку для завершения>:\n";
    String sayings[ArSize];
    char temp[MaxLen]; // временная строковая память
    int i;
    for (i = 0; i < ArSize; i++)
    {
       cout << i+1 << ": ";
       cin.get(temp, MaxLen);
       while (cin && cin.get() != '\n')
          continue;
       if (!cin || temp[0] == '\0') // пустая строка?
          break; // i не инкрементируется
       else
          sayings[i] = temp; // перегруженное присваивание
    }
    int total = i; // общее количество прочитанных строк

    if (total > 0)
    {
       cout << "Вы ввели следующие пословицы:\n";
       for (i = 0; i < total; i++)
          cout << sayings[i] << "\n";

    // использовать указатели для отслеживания кратчайших и первых строк
       String * shortest = &sayings[0];    // установить в качестве
       первоначального значения первый объект
       String * first = &sayings[0];
       for (i = 1; i < total; i++)
       {
          if (sayings[i].length() < shortest->length())
               shortest = &sayings[i];
          if (sayings[i] < *first)         // сравнить значения
               first = &sayings[i];        // присвоить адрес
       }
       cout << "Кратчайшая пословица:\n" << * shortest << endl;
       cout << "Первая пословица по алфавиту:\n" << * first << endl;

       srand(time(0));
       int choice = rand() % total; // выбрать индекс случайным образом
    // использовать new для создания и инициализации объекта String
       String * favorite = new String(sayings[choice]);
       cout << "Моя любимая пословица:\n" << *favorite << endl;
       delete favorite;
    }
    else
       cout << "Что, нечего сказать, да?\n";
    cout << "Всего наилучшего.\n";
    return 0;
}
```

Инициализация объекта с помощью операции new

В общем случае, если *Имя_класса* — это класс, а *значение* имеет тип *Имя_типа*, то оператор

```
Имя_класса * pclass = new Имя_класса(значение);
```

вызывает конструктор:

```
Имя_класса(Имя_типа);
```

Некоторые преобразования могут быть тривиальными, например:

```
Имя_класса(const Имя_типа &);
```

Стандартные преобразования, вызванные соответствием прототипу, такие как из `int` в `double`, имеют место до тех пор, пока отсутствует неоднозначность. Инициализация в виде

```
Имя_класса * ptr = new Имя_класса;
```

вызывает конструктор по умолчанию.

Замечание по совместимости

Более старые реализации C++ могут требовать включения `stdlib.h` вместо `cstdlib` и `time.h` вместо `ctime`.

Ниже показан пример выполнения программы из листинга 12.7:

```
Как вас зовут?
>> Kirt Rood
Kirt Rood, пожалуйста, введите до 10 коротких пословиц <или пустую строку
для завершения>:
1: a friend in need is a friend indeed
2: neither a borrower nor a lender be
3: a stitch in time saves nine
4: a niche in time saves stine
5: it takes a crook to catch a crook
6: cold hands, warm heart
7:
Вы ввели следующие пословицы:
a friend in need is a friend indeed
neither a borrower nor a lender be
a stitch in time saves nine
a niche in time saves stine
it takes a crook to catch a crook
cold hands, warm heart
Кратчайшая пословица:
cold hands, warm heart
Первая пословица по алфавиту:
a friend in need is a friend indeed
Моя любимая пословица:
a stitch in time saves nine
Всего наилучшего.
```

Поскольку программа выбирает любимую пословицу случайным образом, то различные примеры выполнения программы демонстрируют различный выбор даже при идентичных входных данных.

Повторный взгляд на операции new и delete

Обратите внимание, что программа в листингах 12.4, 12.5 и 12.7 использует операции new и delete на двух уровнях. Во-первых, она применяет new для распределения объема памяти под хранение строк имен каждого создаваемого объекта. Это происходит в функциях конструктора, в связи с чем функция-деструктор применяет операцию delete для освобождения данной памяти. Поскольку каждая строка представляет собой массив символов, то деструктор использует операцию delete со скобками. Таким образом, память, используемая для хранения содержимого строк, автоматически освобождается при уничтожении объекта. Во-вторых, код в листинге 12.7 использует операцию new для размещения целого объекта:

```
String * favorite = new String(sayings[choice]);
```

Данный код выделяет пространство для хранения не строки, а объекта — другими словами, для указателя str, который хранит адрес строки и для члена len. (При этом для члена num_strings пространство не выделяется, поскольку он является статическим и хранится отдельно от объектов.) Создание объекта, в свою очередь, вызывает конструктор, который распределяет пространство для хранения строки и присваивает адрес строки указателю str. Программа затем использует операцию delete для удаления объекта после завершения работы с ним. Объект является одиночным, поэтому в программе применяется операция delete без скобок. И снова при этом освобождается только то пространство, которое использовалось для хранения указателя str и члена len. Память, используемая для хранения строки, на которую указывает str, при этом не освобождается. Эту завершающую задачу берет на себя деструктор (рис. 12.4).

```
class Act { ... };
...
Act nice; // внешний объект
...
int main()
{
  Act *pt = new Act; //динамический объект
  {
    Act up; // автоматический объект
    ...
  }
  delete pt;
  ...
}
```


Рис. 12.4. Вызов деструкторов

Деструкторы вызываются в следующих ситуациях (см. рис. 12.4):

- Если объект является автоматической переменной, то деструктор объекта вызывается, когда программа завершает блок, в котором определен объект. Таким образом, в листинге 12.3 деструктор вызывается для headlines[0] и headlines[1], когда выполнение покидает main(), а деструктор для grub вызывается, когда программа выходит из callme1().

- Если объект является статической переменной (внешней, статической, внешней статической или из пространства имен), то его деструктор вызывается при завершении программы. Это происходит с объектом `sports` в листинге 12.3.
- Если объект создается с помощью `new`, его деструктор вызывается только тогда, когда вы явно используете операцию `delete` для данного объекта.

Резюме по указателям и объектам

Следует обратить внимание на несколько моментов, касающихся использования указателей на объекты (обратитесь за сводной информацией к рис. 12.5):

- Вы можете объявить указатель на объект с помощью обычной нотации:

```
String * glamour;
```

- Вы можете инициализировать указатель для указания на существующий объект:

```
String * first = &sayings[0];
```

Объявление указателя на объект класса:	`String * glamour;`
Инициализация указателя для существующего объекта:	`String * first = &sayings[0];` (`sayings[0]` — Объект **String**)
Инициализация указателя с использованием операции **new** и конструктора класса по умолчанию:	`String * gleep = new String;`
Инициализация указателя с использованием операции **new** и конструктора класса **String(const char*)**:	`String * glop = new String("my my my");`
Инициализация указателя с использованием операции **new** и конструктора класса **String(const String &)**:	`String * favorite = new String(sayings[choice]);` (`sayings[choice]` — Объект **String**)
Использование операции **->** для доступа к методу класса через указатель:	`if (sayings[i].length() < shortest->length())` (`sayings[i]` — Объект; `shortest` — Указатель на объект)
Использование операции разыменования (*****) для получения объекта через указатель:	`if (sayings[i] < *first)` (`sayings[i]` — Объект; `*first` — Объект; `first` — Указатель на объект)

Рис. 12.5. Указатели и объекты

- Вы можете инициализировать указатель с помощью операции `new`; при этом создается новый объект:

```
String * favorite = new String(sayings[choice]);
```

Взгляните на рис. 12.6 для более подробного изучения примера инициализации объекта через операцию `new`.

```
String * pveg = new String("Cabbage Heads Home");
```

1. Распределить память под объект — Str: ___ Len: ___ Адрес: 2400

2. Вызвать конструктор класса для:
   - распределения памяти под строку "Cabbage Heads Home" — Cabbage Heads Home\0 Адрес: 2000
   - копирования строки "Cabbage Heads Home" в распределенное пространство
   - присваивания адреса строки "Cabbage Heads Home" указателю **str** — Str: 2000
   - присваивания значения 19 **len** — Len: 19 Адрес: 2400
   - обновления **num_strings** (не показано)

3. Создать переменную **pveg**: — **pveg** — Адрес: 2800

4. Присвоить адрес нового объекта переменной **pveg**: — 2400 **pveg** — Адрес: 2800

***Рис. 12.6. Создание объекта с помощью операции* `new`**

- Использование операции `new` с классом вызывает соответствующий конструктор класса для инициализации вновь созданного объекта:

```
// вызывает конструктор по умолчанию
String * gleep = new String;

// вызывает конструктор String(const char *)
String * glop = new String("my my my");

// вызывает конструктор String(const String &)
String * favorite = new String(sayings[choice]);
```

- Вы можете применять операцию `->` для получения доступа к методу класса через указатель:

```
if (sayings[i].length() < shortest->length())
```

- Вы можете применять операцию разыменования (`*`) к указателю для извлечения объекта:

```
if (sayings[i] < *first)   // сравнить значения объектов
    first = &sayings[i];   // присвоить адрес объекта
```

Повторный взгляд на операцию new с адресацией

Вспомните о том, что операция new с адресацией позволяет задавать ячейку памяти, используемую для распределения памяти. Операция new с адресацией в контексте встроенных типов обсуждалась в главе 9. Использование new с адресацией с объектами добавляет некоторые новые тонкости. В листинге 12.8 new с адресацией используется вместе со стандартным размещением для распределения памяти под объекты. При этом определяется класс с "говорливыми" конструктором и деструктором. Таким образом, вы имеете возможность отслеживать хронологию создания и уничтожения объектов.

Листинг 12.8. placenew1.cpp

```
//placenew1.cpp--операция new, операция new с адресацией, без операции delete
#include <iostream>
#include <string>
#include <new>
using namespace std;
const int BUF = 512;

class JustTesting
{
private:
   string words;
   int number;
public:
   JustTesting(const string & s = "Just Testing", int n = 0)
   {words = s; number = n; cout << words << " создан\n"; }
   ~JustTesting() { cout << words << " уничтожен\n";}
   void Show() const { cout << words << ", " << number << endl;}
};
int main()
{
   char * buffer = new char[BUF];       // получить блок памяти

   JustTesting *pc1, *pc2;

   pc1 = new (buffer) JustTesting;      // разместить объект в буфер
   pc2 = new JustTesting("Heap1", 20);  // разместить объект в «кучу»

   cout << "Адреса блоков памяти:\n" << "буфер: "
        << (void *) buffer << " куча: " << pc2 <<endl;
   cout << "Содержимое памяти:\n";
   cout << pc1 << ": ";
   pc1->Show();
   cout << pc2 << ": ";
   pc2->Show();

   JustTesting *pc3, *pc4;
   pc3 = new (buffer) JustTesting("Bad Idea", 6);
   pc4 = new JustTesting("Heap2", 10);

   cout << "Содержимое памяти:\n";
```

```
    cout << pc3 << ": ";
    pc3->Show();
    cout << pc4 << ": ";
    pc4->Show();

    delete pc2; // освободить Heap1
    delete pc4; // освободить Heap2
    delete [] buffer; // освободить буфер
    cout << "Готово\n";
    return 0;
}
```

Программа в листинге 12.8 использует операцию new для создания буфера памяти объемом 512 байт. Затем new применяется для создания в куче двух объектов типа JustTesting и предпринимается попытка использования размещения new для создания двух объектов типа JustTesting в буфере памяти. В завершение программа применяет операцию delete для освобождения памяти, выделенной операцией new. Ниже показан вывод программы:

```
Just Testing создан
Heap1 создан
Адреса блоков памяти:
буфер: 00320AB0 куча: 00320CE0
Содержимое памяти:
00320AB0: Just Testing, 0
00320CE0: Heap1, 20
Bad Idea создан
Heap2 создан
Содержимое памяти:
00320AB0: Bad Idea, 6
00320EC8: Heap2, 10
Heap1 уничтожен
Heap2 уничтожен
Готово
```

Как обычно, форматирование и точные значения адресов памяти могут отличаться от системы к системе.

В листинге 12.8 существует несколько проблем с операцией new с адресацией. Во-первых, во время создания второго объекта new с адресацией просто перезаписывает новый объект в ту же самую ячейку, которая использовалась для первого объекта. Это не только грубая ошибка, это также означает, что для первого объекта никогда не вызывается деструктор. Это, конечно, создаст реальные проблемы, если, например, класс использует динамическое распределение памяти для своих членов.

Во-вторых, применение операции delete для указателей pc2 и pc4 автоматически активизирует деструкторы для тех двух объектов, на которые указывают pc2 и pc4. Но использование операции delete [] для buffer не приводит к вызову деструкторов для объектов, созданных с помощью new с адресацией.

Первый урок, который извлечь из этого примера — тот же, что и в главе 9. От вас зависит управление ячейками памяти в буфере, которые заполняются new с адресацией. Для того чтобы использовать две различные ячейки, необходимо обеспечить два различных адреса внутри буфера, убедившись в том, что ячейки не перекрываются.

Например, вы можете иметь следующие операторы:

```
pc1 = new (buffer) JustTesting;
pc3 = new (buffer + sizeof (JustTesting)) JustTesting("Better Idea", 6);
```

Здесь указатель pc3 смещен относительно pc1 на размер объекта JustTesting.

Вторым уроком, который необходимо выучить на данном этапе, является то, что если вы применяете new для хранения объектов, то для них нужно организовать вызов деструкторов. Но как? Для объектов, созданных в куче, это можно сделать так:

```
delete pc2; // удалить объект, на который указывает pc2
```

При этом нельзя использовать следующие операторы:

```
delete pc1; // удалить объект, на который указывает pc1? НЕТ! Причина #1
delete pc3; // удалить объект, на который указывает pc2? НЕТ! Причина #2
```

Причина состоит в том, что операция delete работает в связке с операцией new, но не с new с адресацией. Указатель pc3, например, не получает адрес, возвращаемый операцией new, поэтому delete pc3 приводит к ошибке времени выполнения. С другой стороны, указатель pc1 имеет то же самое числовое значение, что и buffer, но buffer инициализируется с помощью операции new [], поэтому он может быть освобожден операцией delete [], а не delete. Даже если buffer был инициализирован посредством new вместо new [], delete pc1 очистит buffer, но не pc1. Это объясняется тем, что система new/delete знает о распределении 256-байтного блока, но не знает ничего о том, что new с адресацией делает этим с блоком.

Обратите внимание на то, что программа освобождает буфер:

```
delete [] buffer; // освободить буфер
```

Как гласит комментарий, код delete [] buffer; удаляет целый блок памяти, распределенный операцией new. Но он не вызывает деструкторы ни для одного из объектов, которые new с адресацией конструирует в блоке. Это можно утверждать потому, что данная программа использует "говорливые" деструкторы, которые сообщают об уничтожении "Heap1" и "Heap2", но умалчивают о "Just Testing" и "Bad Idea".

Выход из этого затруднения заключается в том, что вы должны явно вызвать деструктор для каждого объекта, созданного размещением new. Как правило, деструкторы вызываются автоматически; это один из редких случаев, которые требуют явного вызова. Явный вызов деструктора требует идентификации объекта, который нужно удалить. Поскольку существуют указатели на объекты, вы можете их использовать:

```
pc3->~JustTesting(); // уничтожить объект, на который указывает pc3
pc1->~JustTesting(); // уничтожить объект, на который указывает pc1
```

Код в листинге 12.9 усовершенствован по сравнению с листингом 12.8 за счет управления ячейками памяти, используемыми new с адресацией, а также путем добавления соответствующего обращения к операции delete и явных вызовов деструкторов. Важным моментом является соответствующий порядок удаления. Объекты, сконструированные с помощью операции new с адресацией, должны удаляться в порядке, обратном тому, в котором они были созданы. Причина, в принципе, состоит в том, что более поздний объект может зависеть от более ранних. А буфер, используемый для хранения объектов, можно очистить только после того, как все содержащиеся в нем объекты уничтожены.

Листинг 12.9. placenew2.cpp

```
// placenew2.cpp -- операция new, операция new с адресацией, без операции
delete
#include <iostream>
#include <string>
#include <new>
using namespace std;
const int BUF = 512;

class JustTesting
{
private:
   string words;
   int number;
public:
   JustTesting(const string & s = "Just Testing", int n = 0)
   {words = s; number = n; cout << words << " создан\n"; }
   ~JustTesting() { cout << words << " уничтожен\n";}
   void Show() const { cout << words << ", " << number << endl;}
};
int main()
{
   char * buffer = new char[BUF]; // получить блок памяти

   JustTesting *pc1, *pc2;

   pc1 = new (buffer) JustTesting; // разместить объект в буфер
   pc2 = new JustTesting("Heap1", 20); // разместить объект в буфер

   cout << "Адреса блоков памяти:\n" << "буфер: "
        << (void *) buffer << " куча: " << pc2 <<endl;
   cout << "Содержимое памяти:\n";
   cout << pc1 << ": ";
   pc1->Show();
   cout << pc2 << ": ";
   pc2->Show();

   JustTesting *pc3, *pc4;
// исправление ячейки, с которой работает new с адресацией
   pc3 = new (buffer + sizeof (JustTesting))
           JustTesting("Better Idea", 6);
   pc4 = new JustTesting("Heap2", 10);

   cout << "Содержимое памяти:\n";
   cout << pc3 << ": ";
   pc3->Show();
   cout << pc4 << ": ";
   pc4->Show();

   delete pc2; // освободить Heap1
   delete pc4; // освободить Heap2
// явно уничтожить объекты new с адресацией
```

```
    pc3->~JustTesting(); // уничтожить объект, на который указывает pc3
    pc1->~JustTesting(); // уничтожить объект, на который указывает pc1
    delete [] buffer; // освободить буфер
    cout << "Готово\n";
    return 0;
}
```

Ниже приведен вывод программы из листинга 12.9:

```
Just Testing создан
Heap1 создан
Адреса блоков памяти:
buffer: 00320AB0 heap: 00320CE0
Содержимое памяти:
00320AB0: Just Testing, 0
00320CE0: Heap1, 20
Better Idea создан
Heap2 создан
Содержимое памяти:
00320AD0: Better Idea, 6
00320EC8: Heap2, 10
Heap1 уничтожен
Heap2 уничтожен
Better Idea уничтожен
Just Testing уничтожен
Готово
```

Программа в листинге 12.9 располагает два объекта `new` с адресацией в соседних ячейках и вызывает соответствующие деструкторы.

Обзор технических приемов

К настоящему моменту вы уже сталкивались с некоторыми техническими приемами программирования, связанными с различными проблемами классов. Возможно, уследить за всеми описанными приемами уже трудно. В следующих разделах подводятся итоги по нескольким приемам, а также указывается, когда они применяются.

Перегрузка операции <<

Для того чтобы перегрузить операцию << и применить ее к `cout` для отображения содержимого объекта, необходимо определить дружественную функцию операции, которая имеет следующую форму:

```
ostream & operator<<(ostream & os, const c_name & obj)
{
    os << ... ; // отобразить содержимое объекта
    return os;
}
```

Здесь `c_name` представляет собой имя класса. Если класс предусматривает общедоступные методы, которые возвращают требуемое содержимое, то вы можете применить данные методы в функции операции и обойтись без дружественной функции.

Функции преобразования

Для преобразования одной величины в тип класса необходимо создать конструктор класса, который имеет следующий прототип:

```
c_name(type_name value);
```

Здесь `c_name` представляет собой имя класса, а `type_name` является именем типа, который нужно преобразовать.

Для преобразования типа класса в какой-либо другой тип вы должны создать функцию-член класса с таким прототипом:

```
operator type_name();
```

Хотя данная функция не имеет объявленного типа возврата, она должна возвращать значение требуемого типа.

Помните, что функции преобразования нужно использовать осторожно. Во время объявления конструктора можно применить ключевое слово `explicit` для того, чтобы предотвратить его использование для неявных преобразований.

Классы, конструкторы которых используют операцию new

Вы должны предпринять некоторые меры предосторожности во время разработки классов, которые используют операцию `new` для распределения памяти, на которую указывает член класса. Да, мы уже резюмировали данные меры предосторожности ранее, но эти правила очень важно запомнить, в частности, потому что компилятор их не знает и, соответственно, не заметит ваших ошибок:

- К любому члену класса, который указывает на память, выделенную операцией `new`, необходимо применить операцию `delete` в деструкторе класса. Это освободит занимаемую память.
- Если деструктор освобождает память за счет применения операции `delete` к указателю, который является членом класса, то каждый конструктор для этого класса должен инициализировать данный указатель. Это можно сделать либо с помощью операции `new`, либо делая указатель нулевым.
- Конструкторы должны основываться на использовании либо `new []`, либо `new`, но не смешивать их. Деструктор должен использовать `delete []`, если в конструкторе присутствует `new []`, и `delete` — если `new`.
- Вы должны определить конструктор копирования, который распределяет новую память, вместо того, чтобы копировать указатель на существующую память. Это дает программе возможность инициализировать объект класса другим объектом. Конструктор, как правило, должен иметь следующий прототип:
  ```
  className(const className &)
  ```
- Необходимо определить функцию-член класса, которая перегружает операцию присваивания и имеет следующий прототип (здесь `c_pointer` является членом класса *c_name* и имеет тип указатель на *type_name*). В следующем примере предполагается, что конструктор инициализирует переменную `c_pointer` через `new []`:

```
c_name & c_name::operator=(const c_name & cn)
{
   if (this == & cn_)
      return *this; //сделано, если обнаружено присваивание самому себе
   delete [] c_pointer;
   //установить количество единиц type_name, которые нужно скопировать
   c_pointer = new type_name[size];
   // затем скопировать данные, на которые указывает cn.c_pointer,
   // в ячейку, на которую указывает c_pointer
   ...
   return *this;
}
```

Моделирование очереди

Давайте применим ваше усовершенствованное понимание классов к конкретной проблеме программирования. Банк под названием Bank of Heather хочет открыть банкомат (ATM) в супермаркете Food Heap. Управление Food Heap переживает насчет очередей на ATM, которые могут помешать прохождению потока покупателей, существующего в магазине, и хотят установить предел на количество людей, допустимое в очереди на ATM. Соответственно, работникам банка Bank of Heather необходимо оценить время ожидания покупателей в очереди. Вашей задачей является подготовка программы, которая моделирует ситуацию таким образом, чтобы управление магазина могло увидеть возможный эффект от использования ATM.

Достаточно естественный путь изображения данной проблемы предусматривает использование очереди посетителей. Очередь представляет собой абстрактный тип данных (abstract data type — ATD), который хранит упорядоченную последовательность элементов. Новые элементы добавляются в конец очереди, а удаляются из начала. Очередь подобна стеку, за исключением того, что в стеке добавления и удаления производятся с одного и того же конца. Другими словами, стек является структурой LIFO (последним зашел — первым обслужен), тогда как очередь — это структура FIFO (первым зашел — первым обслужен). Концепция очереди подобна очереди в кассу или в ATM, то есть идеально подходит к данной задаче. Таким образом, одной частью проекта является определение класса Queue. (В главе 16 вы ознакомитесь с классом queue из стандартной библиотеки шаблонов, но гораздо полезнее разработать собственный класс, чем просто прочитать о таком классе.)

Элементами очереди являются покупатели. Банк Bank of Heather сообщает вам, что, в среднем, треть покупателей тратит одну минуту на обслуживание, треть — две минуты и еще одна треть — три. Кроме того, покупатели появляются через произвольные промежутки времени, но среднее количество покупателей в час является примерно постоянным. Две оставшиеся части вашего проекта будут посвящены разработке класса, представляющего покупателей, и компиляции программы, которая моделирует взаимодействия между покупателями и очередью (рис. 12.7).

Класс Queue

На первом месте стоит разработка класса Queue. Во-первых, нужно перечислить атрибуты, которыми должна обладать требуемая разновидность очереди:

- Очередь содержит упорядоченную последовательность элементов.
- Количество элементов, которые может содержать очередь, ограничено.
- Необходима возможность создания пустой очереди.
- Необходима возможность проверки, является ли очередь пустой.
- Необходима возможность проверки, является ли очередь полной.
- Должна быть возможность добавления элемента в конец очереди.
- Должна быть возможность удаления элемента из начала очереди.
- Нужна возможность определения количества элементов в очереди.

Как обычно при проектировании класса, необходимо предусмотреть общедоступный интерфейс и приватную реализацию.

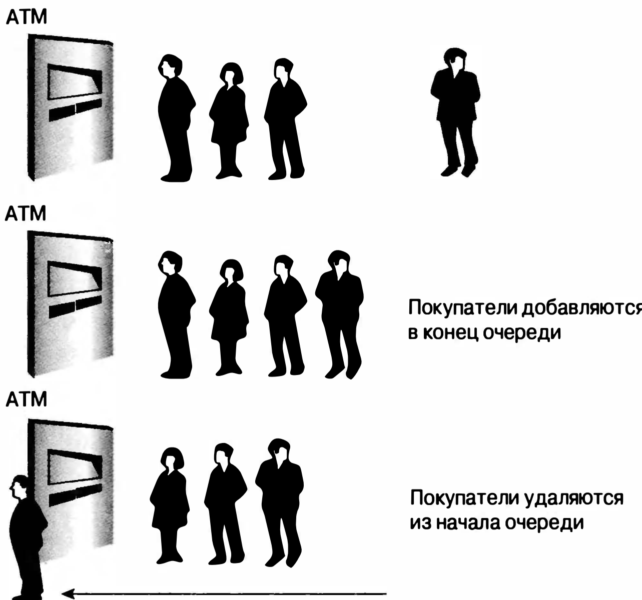


Рис. 12.7 Очередь

Интерфейс класса Queue

Атрибуты очереди, перечисленные в предыдущем разделе, предполагают следующий общедоступный интерфейс для класса Queue:

```
class Queue
{
   enum {Q_SIZE = 10};
private:
// приватное представление будет разрабатываться позднее
public:
   Queue(int qs = Q_SIZE); // создать очередь с пределом qs
   ~Queue();
```

```
    bool isempty() const;
    bool isfull() const;
    int queuecount() const;
    bool enqueue(const Item &item);   // добавить элемент в конец
    bool dequeue(Item &item);         // удалить элемент из начала
};
```

Конструктор создает пустую очередь. По умолчанию очередь может содержать до десяти элементов, но это ограничение может быть изменено с помощью аргумента при явной инициализации:

```
Queue line1;        // очередь с ограничением на 10 элементов
Queue line2(20);    // очередь с ограничением на 20 элементов
```

При использовании очереди вы можете применить `typedef` для определения `Item`. (В главе 14 рассматриваются вопросы применения шаблонов классов.)

Реализация класса Queue

После определения интерфейса вы можете приступить к его реализации. Во-первых, необходимо решить, как представлять данные в очереди. Одним подходом может быть применение операции `new` для динамического распределения массива с требуемым количеством элементов. Однако массивы не очень подходят для операций над очередями. Например, после удаления элемента в начале массива придется передвигать каждый оставшийся элемент на одну позицию ближе к началу. В противном случае нужно будет создавать более замысловатую конструкцию, такую как циклический массив. Использование связного списка в данной ситуации является наиболее разумным подходом, соответствующим всем требованиям очереди. *Связный список* состоит из последовательности узлов. Каждый *узел* содержит информацию, которую нужно удерживать в списке, вместе с указателем на следующий узел в списке. Для очереди в данном примере каждая часть данных является переменной типа `Item`, и для представления узла вы можете воспользоваться следующей структурой:

```
struct Node
{
    Item item;              // данные, хранящиеся в узле
    struct Node * next;     // указатель на следующий узел
};
```

На рис. 12.8 показан связный список.

Пример, показанный на рис. 12.8, называется *односвязным списком*, поскольку каждый узел имеет единственную ссылку, или указатель, на другой узел. Если вы знаете адрес первого узла, то с помощью указателей можно пройти по всем последующим узлам в списке. Как правило, указатель в последнем узле устанавливается в значение `NULL` (либо `0`), что служит признаком того, что больше узлов нет. Для того чтобы отслеживать связный список, необходимо знать адрес первого узла. Можно использовать данные-члены класса `Queue` для указания на начало списка. В принципе, это вся информация, которая вам нужна, так как любой узел можно найти, пройдя по цепочке узлов, начиная с первого. Однако, поскольку новый элемент всегда добавляется в конец очереди, удобно иметь также данные-члены, указывающие на последний узел (рис. 12.9). Кроме того, вы можете применять данные-члены для отслеживания максимального количества элементов, допустимых в очереди, и текущего числа элементов.

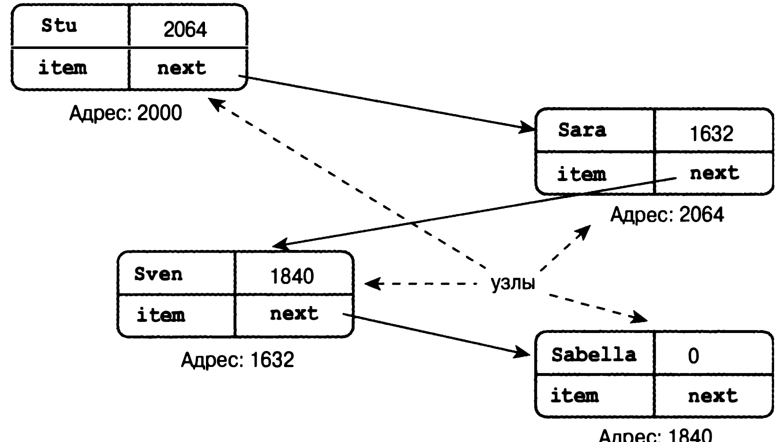


Рис. 12.8. Связный список

Таким образом, приватная часть объявления класса может выглядеть следующим образом:

```
class Queue
{
private:
//определения области действия класса
//Узел — это определение вложенной структуры, локальное для данного класса
   struct Node { Item item; struct Node * next;};
   enum {Q_SIZE = 10};
// приватные члены класса
   Node * front;    // указатель на начало Queue
   Node * rear;     // указатель на конец Queue
   int items;       // текущее количество элементов в Queue
   const int qsize; // максимальное количество элементов в Queue
   ...
public:
//...
};
```

В данном объявлении задействовано новое свойство C++: возможность вкладывать структуру или объявление класса внутри класса. Размещая объявление Node внутри класса Queue, вы определяете его область действия. Другими словами, Node — это тип, который вы можете применять для объявления членов класса и в качестве имени типа в методах класса, но использование данного типа ограничено классом. Таким образом, вам не нужно беспокоиться о том, что Node может противоречить какому-либо глобальному объявлению, либо Node, объявленному внутри другого класса. Не все компиляторы на данный момент поддерживают вложенные структуры и классы. Если ваш входит в их число, то структуру Node потребуется определить глобально, указывая в качестве области ее действия весь файл.

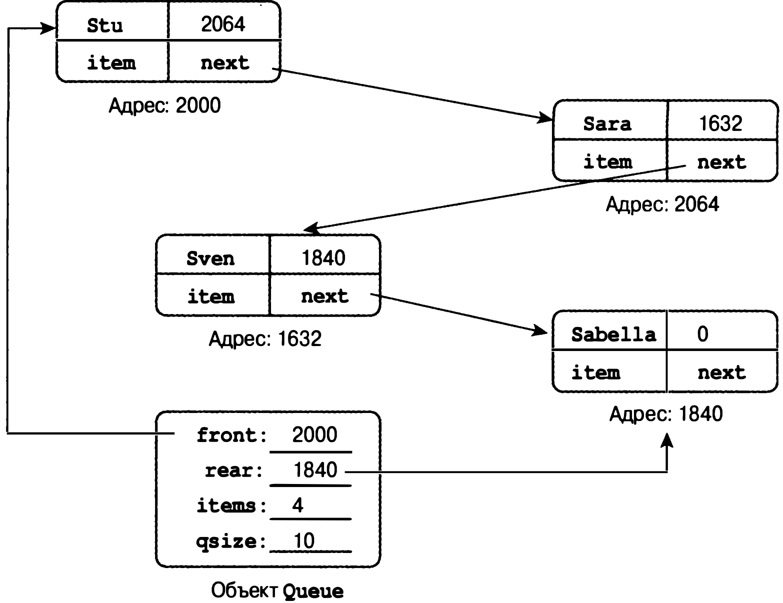


Рис. 12.9. Объект Queue

Вложенные структуры и классы

Структура, класс или перечисление, объявленное внутри объявления класса, называется *вложенным* в класс. Областью его действия является класс. Подобное объявление не создает объект данных. Точнее, оно устанавливает тип, который можно использовать внутри класса. Если объявление размещено в приватном разделе класса, то объявленный тип можно применять только внутри класса. Если объявление размещено в общедоступном разделе, то объявленный тип можно также использовать вне класса через операцию разрешения контекста. Например, если тип Node был объявлен в общедоступном разделе класса Queue, то вы можете объявить переменные типа Queue::Node вне класса Queue.

После того, как вы уладили все проблемы с представлением данных, следующим шагом является кодирование методов класса.

Методы класса

Конструктор класса должен предоставлять значения для членов класса. Поскольку очередь в данном примере начинается с пустого состояния, вы должны установить для начального и конечного указателя значение NULL (или 0) и для переменной items значение 0. Также аргументу конструктора qs необходимо присвоить максимальный размер очереди qsize. Ниже приводится реализация, которая не работает:

```
Queue::Queue(int qs)
{
   front = rear = NULL;
   items = 0;
   qsize = qs; // не допускается!
}
```

Проблема заключается в том, что qsize является переменной типа const, поэтому ее можно *инициализировать*, но ей нельзя *присвоить* некоторое значение. В принципе, вызов конструктора создает объект до того, как выполняется код внутри скобок. Таким образом, вызов конструктора Queue(int qs) вынуждает программу сначала выделить пространство для четырех переменных члена. Затем программа входит в скобки и использует обычное присваивание для помещения значений в выделенное пространство. Следовательно, если вы хотите инициализировать данные-члены const, то это необходимо делать во время создания объекта до того, как выполнение программы достигнет тела конструктора. В C++ именно для этого предусмотрен специальный синтаксис. Он называется *списком инициализаторов членов*. Этот список состоит из инициализаторов, разделенных запятыми, с двоеточием впереди. Он располагается после закрывающей скобки списка аргументов и перед открывающей скобкой тела функции. Если данные-члены имеют имя mdata и им нужно присвоить первоначальное значение val, то инициализатор имеет форму mdata(val). Используя данное представление, можно записать конструктор Queue следующим образом:

```
Queue::Queue(int qs) : qsize(qs)  // присвоить qs в качестве
                                  // первоначального значения qsize
{
   front = rear = NULL;
   items = 0;
}
```

В общем случае первоначальное значение может включать в себя константы и аргументы из списка аргументов конструктора. Инициализировать константы можно различными способами, например, конструктор Queue может также выглядеть так:

```
Queue::Queue(int qs) : qsize(qs), front(NULL), rear(NULL), items(0)
{
}
```

Такой синтаксис списка инициализаторов может применяться только в конструкторах. Как вы уже видели, этот синтаксис необходимо использовать для членов класса const. Вы также должны использовать его для членов класса, которые объявлены как ссылки:

```
class Agency {...};
class Agent
{
private:
   Agency & belong; // для инициализации необходимо использовать
                    // список инициализаторов
   ...
};
Agent::Agent(Agency & a) : belong(a) {...}
```

Это связано с тем, что ссылки, как и данные const, могут быть инициализированы только во время создания. Для простых членов данных вроде front и items нет особого различия, использовать для них список инициализаторов членов или присваивание в теле функции. Кроме того, в главе 14 будет показано, что более эффективно использовать список инициализаторов членов для тех членов, которые сами являются объектами класса.

Синтаксис списка инициализаторов членов

Если, например, `Classy` — это класс, а `mem1`, `mem2` и `mem3` — данные-члены этого класса, то конструктор класса может использовать следующий синтаксис для инициализации данных-членов:

```
Classy::Classy(int n, int m) :mem1(n), mem2(0), mem3(n*m + 2)
{
//...
}
```

При этом для `mem1` устанавливается значение `n`, для `mem2` — значение `0`, а для `mem3` — значение `n*m + 2`. В принципе, указанные инициализации происходят во время создания объекта и до того, как выполняется код в скобках. Обратите внимание на следующие моменты:

- Данная форма может применяться только с конструкторами.
- Данную форму необходимо использовать для инициализации нестатических членов данных типа `const`.
- Данную форму необходимо использовать для инициализации ссылочных данных-членов.

Данные-члены инициализируются в том порядке, в котором они появляются в объявлении класса, а не в том, в котором перечислены инициализаторы.

Внимание!

Вы не можете применять синтаксис списка инициализаторов членов в методах класса, отличных от конструкторов.

Кстати, форма с круглыми скобками, применяемая в списке инициализаторов членов, может также использоваться при обычной инициализации. Другими словами, если вам нравится, то вы можете заменить код

```
int games = 162;
double talk = 2.71828;
```

таким кодом:

```
int games(162);
double talk(2.71828);
```

При этом инициализация встроенных типов выглядит как инициализация объектов класса.

Код для функций `isempty()`, `isfull()` и `queuecount()` является несложным. Если величина `items` равна `0`, то очередь пустая. Если `items` равна `qsize`, то очередь полная. Возврат значения `items` отвечает на вопрос, сколько элементов содержится в очереди. Вы увидите этот код ниже в данной главе, в листинге 12.11.

Добавление элемента в конец очереди выполняется несколько сложнее. Ниже показан один из подходов:

```
bool Queue::enqueue(const Item & item)
{
   if (isfull())
      return false;
   Node * add = new Node; // создать узел
   if (add == NULL)
      return false;       // выйти, если ничего не доступно
   add->item = item;      // установить указатели узлов
   add->next = NULL;
   items++;
```

```
    if (front == NULL)          // если очередь пустая,
        front = add;            // поместить элемент в начало
    else
        rear->next = add;       // иначе поместить в конец
    rear = add;                 // присвоить указатель конца новому узлу
    return true;
}
```

Вкратце, метод состоит из следующих фаз, как показано на рис. 12.10.

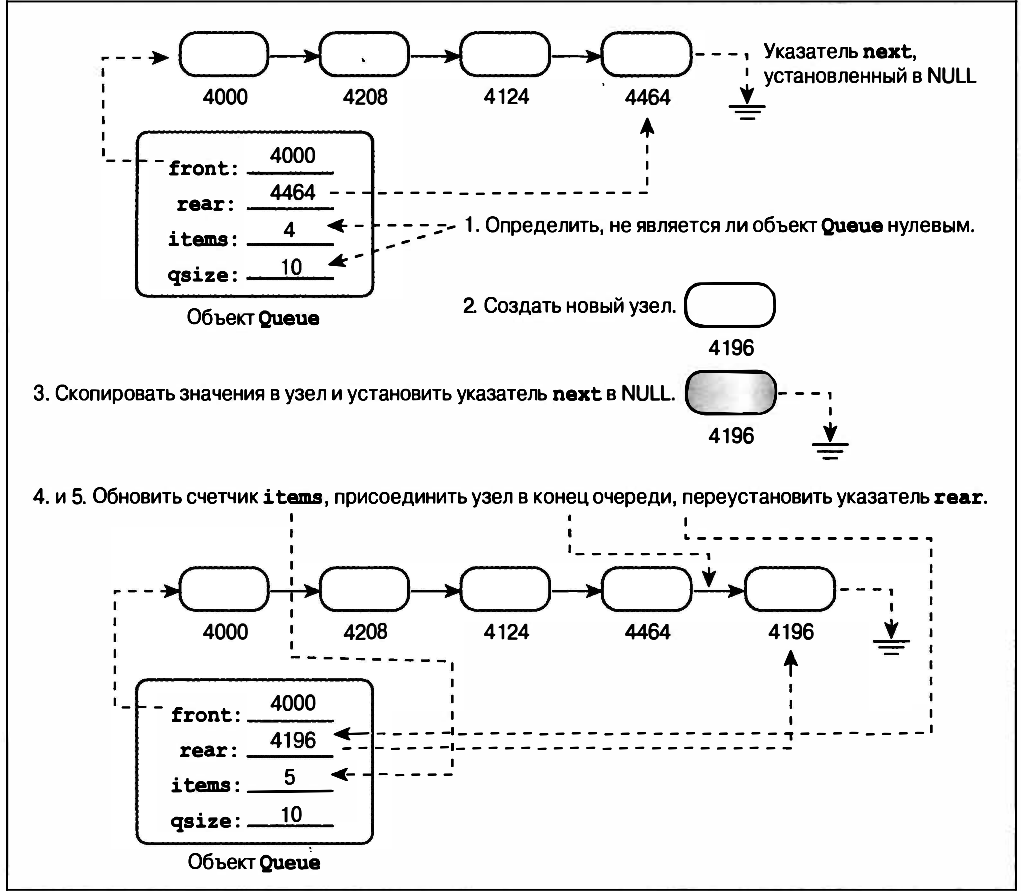


Рис. 12.10. Добавление элемента в конец очереди

1. Завершить программу, если очередь уже полная. (В данной реализации максимальный размер выбирается пользователем через конструктор.)
2. Создать новый узел и завершить работу, если для этого нет возможности (например, если в запросе на больший объем памяти отказано).
3. Поместить соответствующие значения в узел. В этом случае код копирует значение `Item` в часть данных узла и устанавливает указатель на `NULL`. Это подготовка к тому, что узел окажется последним в очереди.

4. Увеличить счетчик элементов (items) на единицу.
5. Присоединить узел в конец очереди. Этот процесс состоит из двух частей. Во-первых, привязывание узла к другим узлам в списке. Это осуществляется путем установки указателя next предыдущего конечного узла на новый конечный узел. Во-вторых, установка члена-указателя rear объекта Queue на новый узел, чтобы можно было получить доступ непосредственно к последнему узлу. Если очередь пустая, то также необходимо установить указатель front на новый узел. (Если в очереди всего один узел, то он является и начальным, и конечным.)

Удаление элемента из начала очереди также предусматривает несколько шагов. Вот один способ:

```
bool Queue::dequeue(Item & item)
{
if (front == NULL)
   return false;
item = front->item;    // установить элемент на первое место в очереди
items--;
Node * temp = front;   // сохранить расположение первого элемента
front = front->next;//переустановить начальный указатель на следующий элемент
delete temp;           // удалить предыдущий первый элемент
if (items == 0)
   rear = NULL;
return true;
}
```

Вкратце, метод состоит из следующих фаз (рис. 12.11):

1. Завершить программу, если очередь уже пустая.
2. Предоставить первый элемент в очереди для вызывающей функции. Это совершается путем копирования блока данных текущего узла front в ссылочную переменную, переданную в метод.
3. Уменьшить счетчик элементов (items) на единицу.
4. Сохранить расположение начального узла для последующего удаления.
5. Удалить узел из очереди. Это производится путем установки члена-указателя front объекта Queue на следующий узел, адрес которого предоставляется front->next.
6. Для сохранения памяти удалить предыдущий начальный узел.
7. Если список теперь пуст, то установить rear на NULL. (Начальный указатель при этом уже будет NULL после установки front->next.)

Шаг 4 является обязательным, поскольку шаг 5 очищает память, в которой находиkся предыдущий начальный узел.

Другие методы класса?

Вам необходимы еще методы? Конструктор класса не использует операцию new, поэтому, на первый взгляд, может показаться, что вам не следует беспокоиться о специальных требованиях классов, которые используют new в конструкторах. Конечно, это первое впечатление ошибочно, поскольку добавление объектов в очередь активизирует new для создания новых узлов.

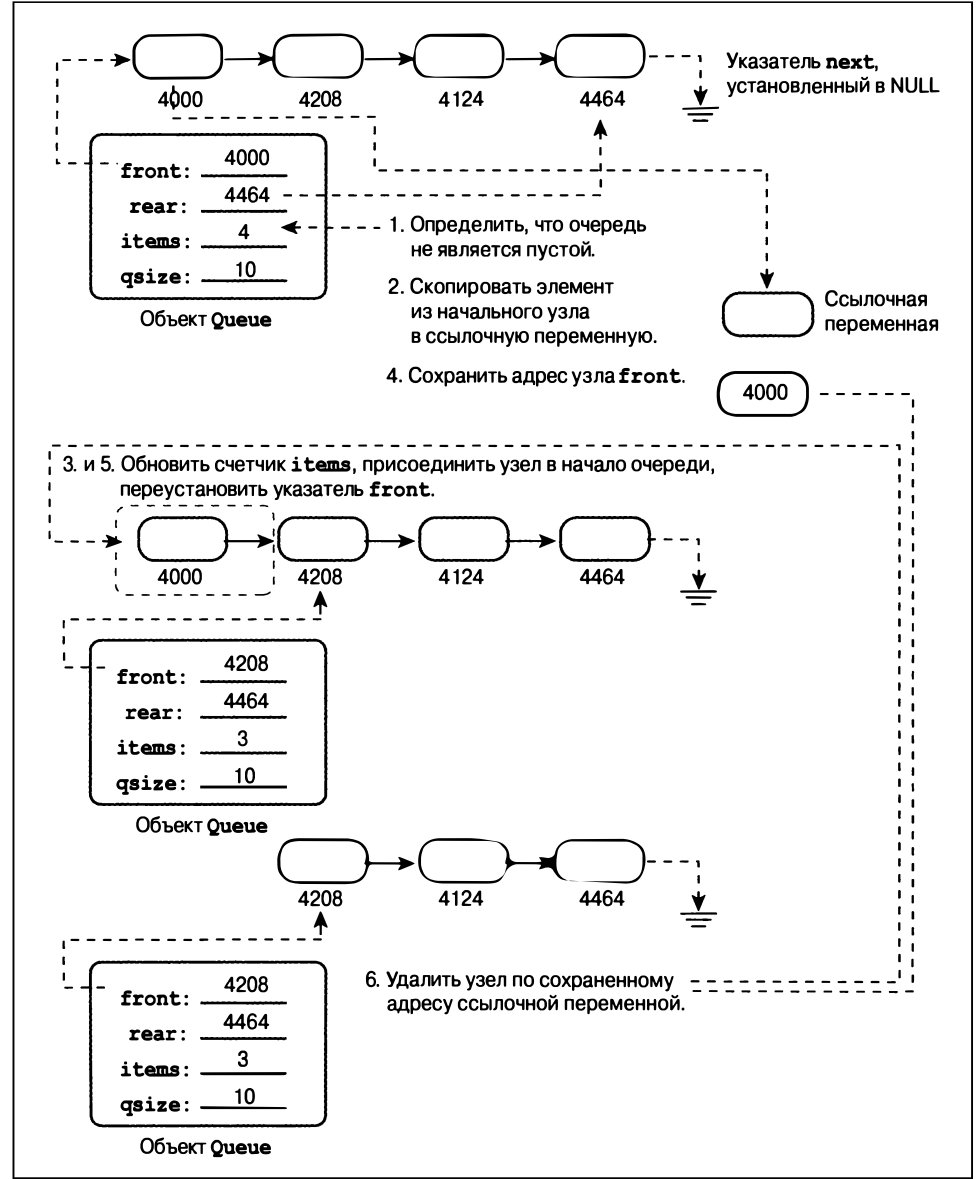


Рис. 12.11. Удаление элемента из начала очереди

Метод `dequeue()`, разумеется, наводит после этого порядок, удаляя узлы, но нет никакой гарантии, что очередь будет пустой, когда работа с ней завершается. Следовательно, класс требует явного деструктора — такого, который удалит все оставшиеся узлы.

Ниже приводится реализация, которая стартует с начала списка и удаляет каждый узел по очереди:

```
Queue::~Queue()
{
   Node * temp;
   while (front != NULL)  // до тех пор, пока очередь еще не пуста
   {
      temp = front;       // сохранить адрес начального элемента
      front = front->next;//переустановить указатель на следующий элемент
      delete temp;        // удалить предыдущий начальный элемент
   }
}
```

Интересно. Вы видели ранее, что классы, которые используют new, обычно требуют явные конструкторы копирования и операции присваивания, которые осуществляют глубокое копирование. Этот ли случай имеет место здесь? Первый вопрос, на который нужно ответить: делает ли почленное копирование по умолчанию то, что необходимо? Ответ отрицательный. Почленное копирование объекта Queue порождает новый объект, который указывает на начало и конец того же самого связного списка, что был первоначально. Таким образом, добавление элемента в копию объекта Queue изменяет общий связный список. Это очень плохо. Что может быть хуже, если конечный указатель обновляется только в копии, существенно искажая список с точки зрения исходного объекта. Очевидно, что клонирование или копирование очередей требует обеспечения конструктора копирования и операции присваивания, которые производят глубокое копирование.

Конечно, при этом возникает вопрос, для чего может понадобиться копирование очереди? Возможно, вы захотите сохранить снимки очереди во время различных стадий моделирования. Либо вам понадобится предоставить одинаковые входные данные для двух различных стратегий. Фактически, полезными могут оказаться операции разделения очереди, как это иногда происходит в супермаркетах при открытии дополнительной кассы. Аналогично может понадобиться объединение двух очередей в одну или усечение очереди.

При данном моделировании вам не потребуется делать ничего подобного. Можете ли вы просто проигнорировать данные соображения и применить те методы, которые уже у вас есть? Конечно, можете. Однако, когда-нибудь в будущем вам, возможно, снова придется использовать очереди, но уже с копированием. А вы можете забыть, что вы не создали код, соответствующий копированию. В таком случае, программы будут компилироваться и выполняться, но они будут выдавать сбивающие с толку результаты и создавать аварийные ситуации. Поэтому будет лучше предусмотреть сразу конструктор копирования и операцию присваивания, даже если они сейчас не нужны.

К счастью, существует хитрый способ избежать дополнительной работы и в то же время защититься от будущих аварийных ситуаций в работе программы. Идея заключается в определении требуемых методов в качестве фиктивных приватных методов:

```
class Queue
{
private:
   Queue(const Queue & q) : qsize(0) { } // примитивное определение
   Queue & operator=(const Queue & q) { return *this;}
//...
};
```

Это дает два эффекта. Во-первых, при этом перезаписываются определения методов по умолчанию, которые в противном случае генерируются автоматически.

Во-вторых, поскольку эти методы являются приватными, то они не могут применяться внешним миром. Другими словами, если `nip` и `tuck` являются объектами `Queue`, то компилятор не разрешит следующие шаги:

```
Queue snick(nip);   // не допускается
tuck = nip;         // не допускается
```

Следовательно, вместо того, чтобы сталкиваться с загадочными неполадками во время работы программы в будущем, вы получите легко отслеживаемую ошибку компилятора, констатирующую, что данные методы не доступны. Этот прием также полезен при определении класса, элементы которого нельзя копировать.

Есть ли еще какие-то результаты, на которые нужно обратить внимание? Да. Вспомните, что конструктор копирования активизируется, когда объекты передаются (или возвращаются) по значению. Однако это не составит проблему, если вы последуете рекомендуемой практике передачи объектов в качестве ссылок. Также конструктор копирования применяется для создания других временных объектов. Но в определении `Queue` отсутствуют операции, которые приводят к временным объектам, такие как перегрузка операции сложения.

Класс Customer

На этом этапе необходимо спроектировать класс покупателя — `Customer`. В общем случае, пользователи ATM имеют много свойств, такие как имя, номер счета и баланс счета. Однако, свойствами, необходимыми для моделирования, являются только два: когда покупатель присоединяется к очереди и время, необходимое для выполнения пользовательской транзакции. Когда модель создает нового покупателя, программа должна создать новый объект клиента, сохранив в нем время появления покупателя и сгенерированное случайным образом время транзакции. Когда клиент достигает начала очереди, программа должна отметить время и вычесть из него время присоединения к очереди. При этом получается время ожидания покупателя. Ниже приведен пример определения и реализации класса `Customer`:

```
class Customer
{
private:
   long arrive;      // время появления покупателя
   int processtime;  // время обслуживания покупателя
public:
   Customer() { arrive = processtime = 0; }
   void set(long when);
   long when() const { return arrive; }
   int ptime() const { return processtime; }
};
void Customer::set(long when)
{
   processtime = std::rand() % 3 + 1;
   arrive = when;
}
```

Конструктор по умолчанию создает нулевого покупателя. Функция-член `set()` устанавливает в качестве аргумента время прибытия и случайным образом выбирает значение от 1 до 3 для времени обслуживания.

В листинге 12.10 собраны вместе объявления классов Queue и Customer, а в листинге 12.11 предоставляются методы.

Листинг 12.10. queue.h

```
// queue.h -- интерфейс для очереди
#ifndef QUEUE_H_
#define QUEUE_H_
// Данная очередь будет содержать элементы Customer
class Customer
{
private:
   long arrive;        // время появления покупателя
   int processtime;    // время обслуживания покупателя
public:
   Customer() { arrive = processtime = 0; }
   void set(long when);
   long when() const { return arrive; }
   int ptime() const { return processtime; }
};

typedef Customer Item;

class Queue
{
private:
// определения области действия класса
   //Узел — это определение вложенной структуры, локальное для данного класса
   struct Node { Item item; struct Node * next;};
   enum {Q_SIZE = 10};
// приватные члены класса
   Node * front;     // указатель на начало Queue
   Node * rear;      // указатель на конец Queue
   int items;        // текущее количество элементов в Queue
   const int qsize; // максимальное количество элементов в Queue
   // вытесняющие объявления для предотвращения общедоступного копирования
   Queue(const Queue & q) : qsize(0) { }
   Queue & operator=(const Queue & q) { return *this;}
public:
   Queue(int qs = Q_SIZE);           // создать очередь с пределом qs
   ~Queue();
   bool isempty() const;
   bool isfull() const;
   int queuecount() const;
   bool enqueue(const Item &item);   // добавить элемент в конец
   bool dequeue(Item &item);         // удалить элемент из начала
};
#endif
```

Листинг 12.11. queue.cpp

```
// queue.cpp -- методы классов Queue и Customer
#include "queue.h"
#include <cstdlib> // (или stdlib.h) для rand()
```

```
// методы Queue
Queue::Queue(int qs) : qsize(qs)
{
   front = rear = NULL;
   items = 0;
}

Queue::~Queue()
{
   Node * temp;
   while (front != NULL)  // до тех пор, пока очередь не пуста
   {
      temp = front;       // сохранить адрес начального элемента
      front = front->next;// переустановить указатель на следующий элемент
      delete temp;        // удалить предыдущий начальный элемент
   }
}

bool Queue::isempty() const
{
   return items == 0;
}

bool Queue::isfull() const
{
   return items == qsize;
}

int Queue::queuecount() const
{
   return items;
}

// Добавить элемент в очередь
bool Queue::enqueue(const Item & item)
{
   if (isfull())
      return false;
   Node * add = new Node; // создать узел
   if (add == NULL)
      return false;       // выйти из программы, если не доступно
   add->item = item;      // установить указатели узлов
   add->next = NULL;
   items++;
   if (front == NULL)     // если очередь пустая,
      front = add;        // поместить элемент в начало
   else
      rear->next = add;   // иначе поместить в конец
   rear = add;            // установить конечный указатель на новый узел
   return true;
}
```

```
// Поместить начальный элемент в переменную элементов и удалить из очереди
bool Queue::dequeue(Item & item)
{
   if (front == NULL)
      return false;
   item = front->item;     // установить элемент на первое место в очереди
   items--;
   Node * temp = front;    // сохранить расположение первого элемента
   front = front->next;    // переустановить начальный указатель
                           // на следующий элемент
   delete temp;            // удалить предыдущий первый элемент
   if (items == 0)
      rear = NULL;
   return true;
}

// Метод Customer

// когда приходит время появления покупателя,
// фиксируется время прибытия, а время обслуживания
// выбирается случайным образом от 1 до 3
void Customer::set(long when)
{
   processtime = std::rand() % 3 + 1;
   arrive = when;
}
```

Замечание по совместимости

Вы можете иметь дело с компилятором, который не распознает `bool`. В таком случае можно использовать `int` вместо `bool`, `0` вместо `false`, и `1` вместо `true`. Возможно, также понадобится включать `stdlib.h` вместо более новой `cstdlib`.

Моделирование

Теперь у вас есть все средства, необходимые для моделирования ATM. Программа должна предусматривать ввод пользователем трех параметров: максимальный размер очереди, количество часов, которые моделируются программой, и среднее количество покупателей в час. В программе необходимо организовать цикл, в котором каждый полный период представляет собой одну минуту. Во время каждого минутного цикла программа должна выполнять следующие шаги:

1. Определить, появился ли новый покупатель. Если да, то добавить покупателя в очередь, если для него есть место; в противном случае, отвергнуть его.
2. Если не обслуживается ни одного клиента, взять первого человека из очереди. Определить, насколько долго он ожидает, и установить счетчик `wait_time` на необходимое ему время обслуживания.
3. Если в данный момент обслуживается клиент, уменьшить счетчик `wait_time` на одну минуту.
4. Проверить различные параметры: количество обслуженных покупателей, количество непринятых клиентов, совокупное время, проведенное в ожидании в очереди, общая длина очереди.

После завершения цикла моделирования программа должна выдать статистический отчет.

Интересной проблемой является способ, которым программа определяет, появился ли новый покупатель. Допустим, что в среднем за час появляется 10 покупателей. Это равносильно одному покупателю каждые шесть минут. Программа вычисляет и сохраняет эту величину в переменной `min_per_cust`. При этом очевидно, что появление следующего покупателя в точности через 6 минут нереально. То, что нужно на самом деле — это более случайный процесс, который моделирует появление одного покупателя за шесть минут. Программа использует эту функцию для выяснения, появился ли покупатель во время цикла:

```
bool newcustomer(double x)
{
   return (std::rand() * x / RAND_MAX < 1);
}
```

Рассмотрим, как работает данный код. Значение `RAND_MAX` определено в файле `cstdlib` (ранее `stdlib.h`) и представляет собой наибольшее значение, которое может возвращать функция `rand()` (самое маленькое — 0). Предположим, что x, среднее время между покупателями, равно 6. После этого значение `rand() * x / RAND_MAX` попадает где-то между 0 и 6. В частности, оно будет меньше 1 в среднем одну шестую часть всего времени. Другими словами, данная функция может выдать двух клиентов с промежутком в одну минуту между ними, а в другой раз двух клиентов с промежутком 20 минут между ними. Такое поведение несколько неуклюже, и им часто отличаются реальные процессы от хронологически точных поступлений клиентов по одному каждые 6 минут. Данный конкретный метод даст сбой, если среднее время между появлением покупателей оказывается меньше одной минуты, но моделирование не предназначено для управления подобным сценарием. Если вам необходимо иметь дело с таким случаем, то можно применить более подходящее временное разрешение, например, установить для каждого цикла 10 секунд.

Замечание по совместимости

Некоторые компиляторы не поддерживают определение `RAND_MAX`. Если вы столкнетесь с такой ситуацией, то можете самостоятельно определить значение для `RAND_MAX` с помощью `#define` или `const int`. Если не удается найти документированное корректное значение, можете попробовать наибольшее возможное значение `int`, выдаваемое `INT_MAX` в заголовочном файле `climits` или `limits.h`.

В листинге 12.12 представлены детали реализации моделирования. Запуск моделирования на длительный период времени предоставляет проникновение в долгосрочные статистические процессы, а на короткие промежутки — в краткие вариации.

Листинг 12.12. bank.cpp

```
// bank.cpp -- использование интерфейса Queue
// компилировать вместе с queue.cpp
#include <iostream>
#include <cstdlib> // для rand() и srand()
#include <ctime>   // для time()
#include "queue.h"
const int MIN_PER_HR = 60;
bool newcustomer(double x); // есть ли новый покупатель?
```

```
int main()
{
   using std::cin;
   using std::cout;
   using std::endl;
   using std::ios_base;
// установка параметров
   std::srand(std::time(0)); // случайная инициализация rand()

   cout << "Учебный пример: банкомат банка Bank of Heather\n";
   cout << "Введите максимальный размер очереди: ";
   int qs;
   cin >> qs;
   Queue line(qs);  // линия очереди останавливается при количестве людей qs

   cout << "Введите количество моделируемых часов: ";
   int hours;              // часы моделирования
   cin >> hours;
   // моделирование будет запускать один цикл в минуту
   long cyclelimit = MIN_PER_HR * hours; // количество циклов

   cout << "Введите среднее количество покупателей в час: ";
   double perhour;         // среднее количество появлений за час
   cin >> perhour;
   double min_per_cust;    // среднее время между появлениями
   min_per_cust = MIN_PER_HR / perhour;

   Item temp;            // данные нового покупателя
   long turnaways = 0; // не пропущен в очередь из-за того, что она заполнена
   long customers = 0; // присоединен к очереди
   long served = 0;    // обслужен во время моделирования
   long sum_line = 0;  // общая длина очереди
   int wait_time = 0;  // время до того, как освободится банкомат
   long line_wait = 0; // общее время в очереди

// запуск моделирования
   for (int cycle = 0; cycle < cyclelimit; cycle++)
   {
      if (newcustomer(min_per_cust)) // есть новенький
      {
         if (line.isfull())
            turnaways++;
         else
         {
            customers++;
            temp.set(cycle);       // цикл = время прибытия
            line.enqueue(temp);    // добавить новичка в очередь
         }
      }
      if (wait_time <= 0 && !line.isempty())
      {
         line.dequeue (temp);      // обслужить следующего покупателя
         wait_time = temp.ptime(); // для минут wait_time
```

```
        line_wait += cycle - temp.when();
        served++;
      }
      if (wait_time > 0)
         wait_time--;
      sum_line += line.queuecount();
   }

// результаты для отчета
   if (customers > 0)
   {
      cout << "   принято покупателей: " << customers << endl;
      cout << " обслужено покупателей: " << served << endl;
      cout << "отправлено покупателей: " << turnaways << endl;
      cout << "средний размер очереди: ";
      cout.precision(2);
      cout.setf(ios_base::fixed, ios_base::floatfield);
      cout.setf(ios_base::showpoint);
      cout << (double) sum_line / cyclelimit << endl;
      cout << "среднее время ожидания: "
           << (double) line_wait / served << " minutes\n";
   }
   else
      cout << "Покупателей нет!\n";
   cout << "Готово!\n";

   return 0;
}

// x = среднее время в минутах между покупателями
// возвращается значение true, если в эту минуту появляется покупатель
bool newcustomer(double x)
{
   return (std::rand() * x / RAND_MAX < 1);
}
```

Замечание по совместимости

Вы можете иметь дело с компилятором, не воспринимающим тип `bool`. В таком случае можно использовать `int` вместо `bool`, `0` вместо `false` и `1` вместо `true`. Возможно, также понадобится включить `stdlib.h` и `time.h` вместо более новых `cstdlib` и `ctime`. Кроме того, может потребоваться самостоятельно определить `RAND_MAX`.

Ниже показано несколько примеров выполнения программы, представленной в листингах 12.10, 12.11, 12.12, для длительного периода времени:

```
Учебный пример: банкомат банка Bank of Heather
Введите максимальный размер очереди: 10
Введите количество моделируемых часов: 100
Введите среднее количество покупателей в час: 15
   принято покупателей: 1485
 обслужено покупателей: 1485
отправлено покупателей: 0
средний размер очереди: 0.15
среднее время ожидания: 0.63 мин
```

```
Учебный пример: банкомат банка Bank of Heather
Введите максимальный размер очереди: 10
Введите количество моделируемых часов: 100
Введите среднее количество покупателей в час: 30
   принято покупателей: 2896
 обслужено покупателей: 2888
отправлено покупателей: 101
средний размер очереди: 4.64
среднее время ожидания: 9.63 мин

Учебный пример: банкомат банка Bank of Heather
Введите максимальный размер очереди: 20
Введите количество моделируемых часов: 100
Введите среднее количество покупателей в час: 30
   принято покупателей: 2943
 обслужено покупателей: 2943
отправлено покупателей: 93
средний размер очереди: 13.06
среднее время ожидания: 26.63 мин
```

Обратите внимание на то, что переход от 15 к 30 покупателям за один час, не удваивает среднее время ожидания, а оно увеличивается приблизительно в 15 раз. Допущение более длинной очереди лишь ухудшает ситуацию. При этом моделирование не предусматривает того факта, что многие покупатели, разозленные долгим ожиданием, могут запросто покинуть очередь.

Ниже приводится еще несколько примеров запуска программы из листинга 12.12. они иллюстрируют возможные кратковременные варианты, даже если среднее количество покупателей за час остается постоянным:

```
Учебный пример: банкомат банка Bank of Heather
Введите максимальный размер очереди: 10
Введите количество моделируемых часов: 4
Введите среднее количество покупателей в час: 30
   принято покупателей: 114
 обслужено покупателей: 110
отправлено покупателей: 0
средний размер очереди: 2.15
среднее время ожидания: 4.52 мин

Учебный пример: банкомат банка Bank of Heather
Введите максимальный размер очереди: 10
Введите количество моделируемых часов: 4
Введите среднее количество покупателей в час: 30
   принято покупателей: 121
 обслужено покупателей: 116
отправлено покупателей: 5
средний размер очереди: 5.28
среднее время ожидания: 10.72 мин

Учебный пример: банкомат банка Bank of Heather
Введите максимальный размер очереди: 10
Введите количество моделируемых часов: 4
Введите среднее количество покупателей в час: 30
```

```
   принято покупателей: 112
  обслужено покупателей: 109
 отправлено покупателей: 0
 средний размер очереди: 2.41
 среднее время ожидания: 5.16 мин
```

Резюме

В данной главе рассматривались многие важные аспекты определения и использования классов. Некоторые из этих аспектов являются хитроумными, и даже трудными, концепциями. Если какие-то из них покажутся вам непонятными или необычайно сложными — не огорчайтесь. Это характерно почти для всех новичков в С++. Часто единственный путь, который приводит к действительно ценным концепциям, таким как конструкторы копирования, проходит через практические неприятности из-за их игнорирования. Таким образом, кое-что из материала данной главы может показаться неясным до тех пор, пока ваш собственный опыт не обогатит ваше понимание.

Вы можете использовать операцию new в конструкторе класса для распределения памяти под данные и последующего присваивания адреса памяти члену класса. Это позволяет классу, например, управлять строками различных размеров без жесткого кодирования заранее установленного размера. Использование операции new в конструкторах класса также вызывает возможные проблемы во время прекращения существования объекта. Если объект имеет члены-указатели, указывающие на память, распределенную операцией new, то освобождение памяти, которая использовалась для хранения объекта, не освобождает автоматически память, на которую указывают члены-указатели объекта. Следовательно, если вы применяете new в конструкторе класса для распределения памяти, то вы должны использовать операцию delete в деструкторе класса для очистки этой памяти. Уничтожение объекта автоматически запускает удаление указываемой памяти.

Объекты, которые имеют члены, указывающие на память, распределенную операцией new, также имеют проблемы при инициализации одного объекта другим либо при простом присваивании одного объекта другому. По умолчанию С++ использует почленную инициализацию и присваивание, которые означают, что инициализированный или присвоенный объект имеют дело с точными копиями исходных членов объекта. Если исходный член указывает на блок данных, то скопированный член указывает на тот же самый блок. Если программа случайно удаляет два объекта, деструктор класса пытается удалить один и тот же блок дважды, что является ошибкой. Выходом служит определение специального конструктора копирования, который переопределяет инициализацию, и перегрузка операции присваивания. В каждом случае новое определение должно создавать копии всех данных, на которые имеются указатели, и заставлять новый объект указывать на копии. При этом и старый, и новый объекты указывают на отдельные, но одинаковые данные, не перекрывающие друг друга. Те же рассуждения применяются и к операции присваивания. В каждом случае целью является создание глубокой копии — другими словами необходимо копировать фактические данные, а не только указатели на них.

Если объект имеет автоматическую или внешнюю память, то деструктор для этого объекта вызывается автоматически, когда объект прекращает свое существование. Если вы распределяете память для объекта с помощью операции new и присваиваете его адрес указателю, то деструктор для данного объекта вызывается автоматически,

когда вы применяете операцию delete к указателю. Однако если вы распределяете память для объектов класса через операцию new с адресацией вместо стандартной new, то вы принимаете дополнительную ответственность за явный вызов деструктора для данного объекта путем передачи методу деструктора указателя на объект. C++ позволяет помещать определения структур, классов и перечислений внутрь класса. Подобные вложенные типы имеют областью видимости в пределах класса, что означает, что они являются локальными для класса и не противоречат структурам, классам, перечислениям с таким же именем, определенным где-либо еще.

В C++ предусмотрен специальный синтаксис для конструкторов класса, которые могут применяться для инициализации данных-членов. Упомянутый синтаксис состоит из двоеточия, за которым следует список инициализаторов, разделенных запятыми. Он размещается после закрывающей круглой скобки аргументов конструктора и до открывающей фигурной скобки тела функции. Каждый инициализатор состоит из имени инициализируемого члена, за которым следуют круглые скобки, содержащие начальное значение. Концептуально, такие инициализации выполняются во время создания объекта и до операторов в теле функции. Синтаксис выглядит следующим образом:

```
queue(int qs) : qsize(qs), items(0), front(NULL), rear(NULL) { }
```

Представленная форма является обязательной, если данные-члены являются нестатическим членом const или ссылкой.

Как вы уже заметили, классы требуют гораздо больше осторожности и внимания к деталям, нежели простые структуры C-стиля. В свою очередь, они делают гораздо больше для вас.

Вопросы для самоконтроля

1. Предположим, что класс String имеет следующие приватные члены:

```
class String
{
private:
    char * str; // указывает на строку, распределенную операцией new
    int len;    // хранит длину строки
//...
};
```

а. Что неправильно в данном конструкторе по умолчанию?

```
String::String() {}
```

б. Что неправильно в данном конструкторе?

```
String::String(const char * s)
{
    str = s;
    len = strlen(s);
}
```

в. Что неправильно в данном конструкторе?

```
String::String(const char * s)
{
    strcpy(str, s);
    len = strlen(s);
}
```

2. Назовите три проблемы, которые могут возникнуть, если вы определите класс, где член-указатель инициализируется с помощью операции new. Укажите, как их можно решить.
3. Какие методы класса компилятор генерирует автоматически, если вы не предусмотрели их явно? Опишите, как ведут себя эти неявно сгенерированные функции.
4. Найдите и исправьте ошибки в следующем объявлении класса:

```
class nifty
{
// данные
    char personality[];
    int talents;
// методы
    nifty();
    nifty(char * s);
    ostream & operator<<(ostream & os, nifty & n);
}

nifty:nifty()
{
    personality = NULL;
    talents = 0;
}

nifty:nifty(char * s)
{
    personality = new char [strlen(s)];
    personality = s;
    talents = 0;
}
ostream & nifty:operator<<(ostream & os, nifty & n)
{
    os << n;
}
```

5. Имеется следующее объявление класса:

```
class Golfer
{
private:
    char * fullname;//указывает на строку, содержащую имя игрока в гольф
    int games;     // хранит количество сыгранных игр в гольф
    int * scores;  // указывает на первый элемент массива счетов гольфа
public:
    Golfer();
    Golfer(const char * name, int g= 0);
    // создает пустой динамический массив из g элементов, если g > 0
    Golfer(const Golfer & g);
    ~Golfer();
};
```

а. Какие методы класса будут вызываться следующими операторами?

```
Golfer nancy;                       // #1
Golfer lulu("Little Lulu");         // №2
Golfer roy("Roy Hobbs", 12);        // №3
Golfer * par = new Golfer;          // №4
Golfer next = lulu;                 // №5
Golfer hazzard = "Weed Thwacker";   // №6
*par = nancy;                       // №7
nancy = "Nancy Putter";             // №8
```

б. Ясно, что классу требуется больше методов для того, чтобы сделать его действительно полезным. Какой дополнительный метод требуется для защиты от разрушения данных?

Упражнения по программированию

1. Имеется следующее объявление класса:

```
class Cow {
    char name[20];
    char * hobby;
    double weight;
public:
    Cow();
    Cow(const char * nm, const char * ho, double wt);
    Cow(const Cow c&);
    ~Cow();
    Cow & operator=(const Cow & c);
    void ShowCow() const; // отобразить все данные cow
};
```

Напишите реализацию для данного класса и короткую программу, использующую все функции-члены.

2. Усовершенствуйте объявление класса `String` (другими словами, обновите `string1.h` на `string2.h`) следующим образом:

а. Перегрузите операцию + для получения возможности добавлять две строки в одну.

б. Предусмотрите функцию члена `stringlow()`, которая преобразует все алфавитные символы в строке в нижний регистр. (Не забудьте о семействе `cctype` символьных функций.)

в. Предусмотрите функцию-член `stringup()`, которая преобразует все символы алфавита в строке в верхний регистр.

г. Создайте функцию-член, которая принимает аргумент `char` и возвращает количество раз, которое символ появляется в строке.

Протестируйте результаты проделанной работы в следующей программе:

```
// pe12_2.cpp
#include <iostream>
using namespace std;
#include "string2.h"
int main()
```

```
{
   String s1(" and I am a C++ student.");
   String s2 = "Please enter your name: ";
   String s3;
   cout << s2;                // перегруженная операция <<
   cin >> s3;                 // перегруженная операция <<
   s2 = "My name is " + s3;   // перегруженные операции =, +
   cout << s2 << ".\n";
   s2 = s2 + s1;
   s2.stringup();             // преобразует строку в верхний регистр
   cout << "The string\n" << s2 << "\ncontains " << s2.has('A')
        << " 'A' characters in it.\n";
   s1 = "red";     // String(const char *),
                   // затем String & operator=(const String&)
   String rgb[3] = { String(s1), String("green"), String("blue")};
   cout << "Enter the name of a primary color for mixing light: ";
   String ans;
   bool success = false;
   while (cin >> ans)
   {
      ans.stringlow();        // преобразует строку в нижний регистр
      for (int i = 0; i < 3; i++)
      {
         if (ans == rgb[i])   // перегруженная операция ==
         {
            cout << "That's right!\n";
            success = true;
            break;
         }
      }
      if (success)
         break;
      else
         cout << "Try again!\n";
   }
   cout << "Bye\n";
   return 0;
}
```

Выходные данные должны выглядеть приблизительно так:

```
Please enter your name: Fretta Farbo
My name is Fretta Farbo.
The string
MY NAME IS FRETTA FARBO AND I AM A C++ STUDENT.
contains 6 'A' characters in it.
Enter the name of a primary color for mixing light: yellow
Try again!
BLUE
That's right!
Bye
```

3. **Перепишите класс `Stock`, как описывается в листингах 10.7 и 10.8 главы 10, таким образом, чтобы он использовал динамически распределенную память непосредственно вместо применения объектов класса `string` для хранения на-**

званий пакетов. Также замените функцию-член `show()` перегруженным определением `operator<<()`. Протестируйте новый код с помощью программы из листинга 10.9.

4. Имеется следующий вариант класса `Stack`, определенный в листинге 10.10:

```
// stack.h -- объявление класса для ADT стека
typedef unsigned long Item;
class Stack
{
private:
    enum {MAX = 10};  // константы, характерные для класса
    Item * pitems;    // хранит элементы стека
    int size;         // количество элементов в стеке
    int top;          // индекс для верхнего элемента стека
public:
    Stack(int n = 10);// создает стек с n элементами
    Stack(const Stack & st);
    ~Stack();
    bool isempty() const;
    bool isfull() const;
    // push() возвращает значение false, если стек уже полный,
    // и true в противном случае
    bool push(const Item & item);   // добавить элемент в стек
    // pop() возвращает значение false, если стек уже пустой,
    // и true в противном случае
    bool pop(Item & item);          // извлечь элемент из стека
    Stack & operator=(const Stack & st);
};
```

Как подсказывают приватные члены, данный класс использует динамически распределенный массив для хранения элементов стека. Перепишите методы для соответствия новому представлению и напишите программу, которая демонстрирует все методы, включая конструктор копирования и операцию присваивания.

5. Банк Bank of Heather провел исследование, которое показало, что клиенты ATM не ожидают в очереди более одной минуты. Используя модель из листинга 12.10, найдите количество покупателей за час, которое приводит к среднему времени ожидания, равному одной минуте. (Используйте по меньшей мере 100-часовый испытательный срок.)

6. Банк Bank of Heather интересуется, что произойдет, если добавится второй ATM. Измените моделирование в данной главе таким образом, чтобы оно поддерживало две очереди. Допустите, что покупатель присоединяется к первой очереди, если в ней меньше людей, и ко второй — в противном случае. Найдите количество покупателей за час, которое приводит к среднему времени ожидания, равному одной минуте. (На заметку: это нелинейная задача, в которой удвоение числа ATM не удваивает количество покупателей, которые могут быть обслужены за час с максимальным ожиданием в одну минуту.)

ГЛАВА 13

Наследование классов

В этой главе:

- Наследование как отношение *is-a*
- Общедоступное порождение одного класса из другого
- Защищенный доступ
- Списки инициализаторов членов конструктора
- Восходящее и нисходящее преобразование
- Виртуальные функции-члены
- Раннее (статическое) связывание и позднее (динамическое) связывание
- Базовые абстрактные классы
- Чистые виртуальные функции
- Когда и как использовать общедоступное наследование

Одной из главных целей объектно-ориентированного программирования является обеспечение многократно используемого кода. Когда вы разрабатываете новый проект, особенно если проект крупный, то предпочтительнее иметь возможность повторно использовать уже проверенный код, чем заново изобретать его. Применение старого кода экономит время, а также, поскольку код уже был использован и проверен, это может помочь избежать внесения ошибок в программу. К тому же, чем меньше вы будете заняты мелкими деталями, тем лучше сосредоточитесь на общей стратегии программы.

Традиционные библиотеки функций C предусматривают возможность многократного использования стандартных, предварительно откомпилированных функций, таких как `strlen()` и `rand()`, которые вы можете применять в своих программах. Многие поставщики предоставляют специализированные библиотеки C, которые расширяют стандартную библиотеку C. Например, вы можете приобрести библиотеки функций управления базами данных и функций управления изображением на экране. Однако библиотеки функций имеют ограничение: если поставщик не предоставляет исходный код для своих библиотек (чаще всего так и происходит), то вы не сможете расширить или изменить функции для соответствия своим особым потребностям. Вместо этого вы должны формировать свою программу таким образом, чтобы соответствовать разработкам библиотеки. Даже если поставщик передает исходный код, при добавлении собственных переделок вы подвергаетесь риску непреднамеренного изменения работы части функции либо изменения отношений между функциями библиотеки.

Классы C++ обеспечивают более высокий уровень многократного использования. Многие поставщики сейчас предлагают библиотеки, которые состоят из объявлений и реализаций классов. Поскольку класс объединяет представление данных

с методами, образуется более интегрированный пакет, чем библиотека функций. Единственный класс, к примеру, может предусматривать все средства для управления диалоговым окном. Часто библиотеки классов доступны в виде исходных кодов, а это означает, что вы можете модифицировать их для соответствия своим потребностям. Однако в C++ для расширения и изменения классов имеется более удобный метод, нежели модификация кода. Данный метод, называемый *наследованием классов*, позволяет порождать новые классы от старых, называемых *базовыми классами*. Производный класс наследует все свойства, включая методы, старого класса. Унаследовать состояние обычно легче, чем заработать его с нуля. Точно также породить класс с помощью наследования гораздо проще, чем сконструировать новый. Вот что можно делать, благодаря наследованию:

- Добавлять функциональные возможности в существующий класс. Например, для данного базового класса массива можно добавить арифметические операции.
- Добавлять данные, которые представляет класс. Например, взяв за основу базовый класс строки, можно породить класс, в котором добавлен член данных, представляющий цвет, и который будет использоваться при отображении строки.
- Модифицировать поведение методов класса. Например, взяв за основу класс `Passenger`, который представляет услуги, предоставляемые пассажиру авиалинии, можно породить класс `FirstClassPassenger`, предусматривающий более высокий уровень обслуживания.

Конечно, вы можете достигнуть тех же целей, скопировав исходный код класса и модифицировав его, но механизм наследования позволяет продолжать разработку путем всего лишь добавления новых свойств. Для порождения класса вам даже не нужно иметь доступ к исходному коду. Таким образом, если вы приобрели библиотеку классов, которая предоставляет только заголовочные файлы и скомпилированный код для методов класса, вы все равно можете порождать новые классы, основанные на классах библиотеки. И наоборот, вы можете распространять свои собственные классы среди других пользователей, сохраняя части реализации в секрете, но предоставляя своим клиентам возможность добавления свойств к этим классам.

Наследование — это превосходная концепция, и его основная реализация достаточно простая. Однако управление наследованием так, чтобы оно работало должным образом во всех ситуациях, требует некоторых уточнений. В данной главе рассматриваются как простейшие, так и более сложные аспекты наследования.

Начало работы с простым базовым классом

Когда один класс наследуется от другого, то исходный класс называется *базовым классом*, а унаследованный — *производным классом*. Таким образом, для того чтобы проиллюстрировать наследование, давайте начнем с базового класса. Клуб Webtown Social Club решил отслеживать своих членов, играющих в настольный теннис. Как ведущий программист клуба, вы спроектировали простой класс `TableTennisPlayer`, определенный в листингах 13.1 и 13.2.

Листинг 13.1. tabtenn0.h

```
// tabtenn0.h -- базовый класс для клуба по настольному теннису
#ifndef TABTENN0_H_
#define TABTENN0_H_
// простой базовый класс
class TableTennisPlayer
{
private:
   enum {LIM = 20};
   char firstname[LIM];
   char lastname[LIM];
   bool hasTable;
public:
   TableTennisPlayer (const char * fn = "none",
                      const char * ln = "none", bool ht = false);
   void Name() const;
   bool HasTable() const { return hasTable; };
   void ResetTable(bool v) { hasTable = v; };
};
#endif
```

Листинг 13.2. tabtenn0.cpp

```
//tabtenn0.cpp -- методы простого базового класса
#include "tabtenn0.h"
#include <iostream>
#include <cstring>

TableTennisPlayer::TableTennisPlayer (const char * fn, const char * ln, bool ht)
{
   std::strncpy(firstname, fn, LIM - 1);
   firstname[LIM - 1] = '\0';
   std::strncpy(lastname, ln, LIM - 1);
   lastname[LIM - 1] = '\0';
   hasTable = ht;
}

void TableTennisPlayer::Name() const
{
   std::cout << lastname << ", " << firstname;
}
```

Класс TableTennisPlayer всего лишь отслеживает имена игроков, а также наличие у них столов. В листинге 13.3 показан этот скромный класс в действии.

Листинг 13.3. usett0.cpp

```
// usett0.cpp -- использование базового класса
#include <iostream>
#include "tabtenn0.h"
int main ( void )
{
   using std::cout;
```

```
    TableTennisPlayer player1("Чак", "Близзард", true);
    TableTennisPlayer player2("Тара", "Бумди", false);
    player1.Name();
    if (player1.HasTable())
       cout << ": имеет стол.\n";
    else
       cout << ": не имеет стола.\n";
    player2.Name();
    if (player2.HasTable())
       cout << ": имеет стол";
    else
       cout << ": не имеет стола.\n";
    return 0;
}
```

Ниже показан вывод программы, представленный в листингах 13.1, 13.2 и 13.3:

```
Близзард, Чак: имеет стол.
Бумди, Тара: не имеет стола.
```

Порождение класса

Некоторые члены клуба Webtown Social Club участвовали в местных турнирах по настольному теннису, и для них требуется класс, который включает в себя рейтинговые баллы, заработанные ими в играх. Вместо того чтобы начинать с пустого места, вы можете породить новый класс от `TableTennisPlayer`. Первый шаг заключается в создании объявления класса `RatedPlayer`, из которого видно, что он порожден от `TableTennisPlayer`:

```
// RatedPlayer порожден от базового класса TableTennisPlayer
class RatedPlayer : public TableTennisPlayer
{
...
};
```

Здесь двоеточие указывает на то, что класс `RatedPlayer` основан на классе `TableTennisPlayer`. Данный отдельный заголовок означает, что `TableTennisPlayer` является общедоступным базовым классом, этот процесс называется *общедоступным порождением*. Объект производного класса заключает в себе объект базового класса. Во время общедоступного порождения общедоступные члены базового класса становятся общедоступными членами производного класса. Приватные порции базового класса становятся частью производного класса, однако доступ к ним может быть получен только через общедоступные и защищенные методы базового класса. (Защищенные члены будут рассматриваться чуть позже.)

Что при этом происходит? Если вы объявите объект `RatedPlayer`, то он будет обладать следующими особыми свойствами:

- Объект производного типа хранит члены данных базового типа. (Производный класс наследует реализацию базового класса.)
- Объект производного типа может использовать методы базового типа. (Производный класс наследует интерфейс базового класса.)

Таким образом, объект RatedPlayer может хранить имя и фамилию каждого игрока, а также сведения о том, имеет ли игрок стол. Также объект RatedPlayer может применять методы Name(), HasTable() и ResetTable() из класса TableTennisPlayer (рис. 13.1).

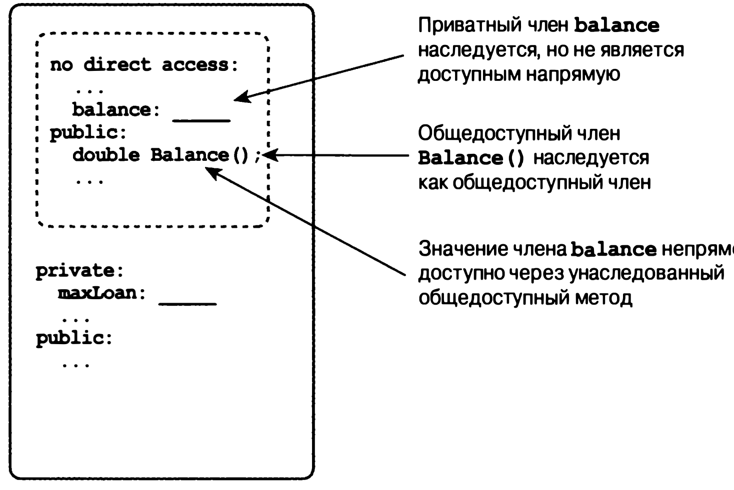


Рис. 13.1. Объекты базового и производного классов

Что нужно добавить к данным унаследованным свойствам?

- Производный класс нуждается в своих собственных конструкторах.
- Производный класс может добавлять дополнительные члены данных и методы, если это необходимо.

В этом конкретном случае классу требуется еще одни данные-члены ratings для хранения собственно рейтинга. В нем также необходимы методы для извлечения и возврата рейтинга. Итак, объявление класса может выглядеть следующим образом:

```
// простой производный класс
class RatedPlayer : public TableTennisPlayer
{
private:
   unsigned int rating; // добавить член данных
```

```
public:
   RatedPlayer (unsigned int r = 0, const char * fn = "none",
                const char * ln = "none", bool ht = false);
   RatedPlayer(unsigned int r, const TableTennisPlayer & tp);
   unsigned int Rating() { return rating; }          // добавить метод
   void ResetRating (unsigned int r) {rating = r;}   // добавить метод
};
```

Конструкторы должны обеспечивать данные для новых членов, если они есть, а также для унаследованных членов. Первый конструктор RatedPlayer использует отдельный формальный параметр для каждого члена, а второй конструктор — параметр TableTennisPlayer, связывающий три элемента (firstname, lastname и hasTable) в единое целое.

Конструкторы: анализ доступа

Производный класс не имеет прямого доступа к приватным членам базового класса; он вынужден обращаться к методам базового класса. Например, конструкторы RatedPlayer не могут непосредственно устанавливать значения унаследованных членов (firstname, lastname и hasTable). Вместо этого они должны использовать общедоступные методы базового класса, чтобы получить доступ к приватным членам базового класса. В частности, конструкторы производного класса должны использовать конструкторы базового класса.

Когда программа создает объект производного класса, сначала конструируется объект базового класса. В принципе, это означает, что объект базового класса должен быть создан до того, как программа войдет в тело конструктора производного класса. Для осуществления этого в C++ применяется синтаксис списка инициализаторов членов. Ниже для примера приводится код для первого конструктора RatedPlayer:

```
RatedPlayer::RatedPlayer(unsigned int r, const char * fn,
   const char * ln, bool ht) : TableTennisPlayer(fn, ln, ht)
{
   rating = r;
}
```

Выражение

```
: TableTennisPlayer(fn, ln, ht)
```

является списком инициализаторов членов. Это исполняемый код, он вызывает конструктор TableTennisPlayer. Предположим, например, что программа содержит следующее объявление:

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
```

Конструктор RatedPlayer присваивает формальным параметрам fn, ln и ht фактические аргументы "Mallory", "Duck" и true. Затем он передает данные параметры как фактические аргументы конструктору TableTennisPlayer. Этот конструктор, в свою очередь, создает вложенный объект TableTennisPlayer и сохраняет в нем данные "Mallory", "Duck" и true. Далее программа входит в тело конструктора RatedPlayer, завершает создание объекта RatedPlayer и присваивает значение параметра r (то есть 1140) члену rating (рис. 13.2).

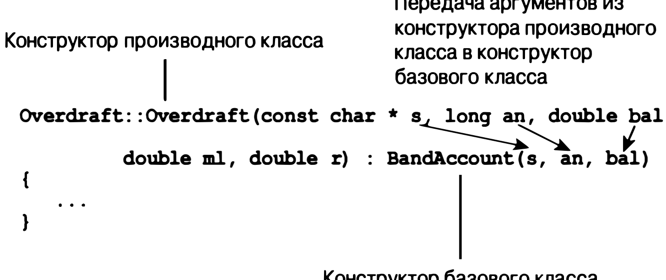


Рис. 13.2. Передача аргументов с помощью конструктора базового класса

Что произойдет, если вы пропустите список инициализаторов членов?

```
RatedPlayer::RatedPlayer(unsigned int r, const char * fn,
   const char * ln, bool ht) // что будет без списка инициализаторов?
{
   rating = r;
}
```

Прежде всего, должен быть создан объект базового класса, поэтому если вы пропустите вызов конструктора базового класса, то программа воспользуется стандартным конструктором базового класса. Следовательно, предыдущий код аналогичен следующему:

```
RatedPlayer::RatedPlayer(unsigned int r, const char * fn,
   const char * ln, bool ht) // : TableTennisPlayer()
{
   rating = r;
}
```

Если вы не хотите, чтобы использовался стандартный конструктор, вы должны предусмотреть явный вызов соответствующего конструктора базового класса. Давайте рассмотрим код для второго конструктора:

```
RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
   : TableTennisPlayer(tp)
{
   rating = r;
}
```

Информация `TableTennisPlayer` передается далее конструктору `TableTennisPlayer`:

```
TableTennisPlayer(tp)
```

Поскольку `tp` является переменной типа `const TableTennisPlayer &`, данный вызов активизирует конструктор копирования базового класса. В базовом классе не определяется конструктор копирования, однако следует вспомнить из главы 12, что компилятор автоматически генерирует конструктор копирования, если он необходим, но ранее не был определен. Неявный конструктор, который производит почленное копирование, полностью подходит в подобной ситуации, поскольку класс не использует динамическое распределение памяти.

Если потребуется, вы можете также использовать синтаксис списка инициализаторов для членов производного класса. В этом случае в списке используется имя члена вместо имени класса. Таким образом, второй конструктор можно также записать следующим образом:

```
// альтернативный вариант
RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
   : TableTennisPlayer(tp), rating(r)
{
}
```

Рассмотрим основные моменты относительно конструкторов для производного класса:

- Сначала создается объект базового класса.
- Конструктор производного класса должен передавать информацию базового класса конструктору базового класса через список инициализаторов членов.
- Конструктор производного класса должен инициализировать данные-члены, которые были добавлены к производному классу.

В настоящем примере не предусмотрены явные деструкторы, поэтому используются неявные деструкторы. Уничтожение объекта происходит в порядке, обратном тому, в котором он создавался. То есть сначала выполняется тело деструктора производного класса, а затем автоматически вызывается деструктор базового класса.

На память!

Во время создания объекта производного класса программа сначала вызывает конструктор базового класса, а потом конструктор производного класса. Конструктор базового класса отвечает за инициализацию унаследованных данных-членов. Конструктор производного класса отвечает за инициализацию всех добавляемых данных-членов. Конструктор производного класса всегда вызывает конструктор базового класса. Вы можете применить синтаксис списка инициализаторов для указания того, *какой* конструктор базового класса использовать. В противном случае вызывается стандартный конструктор базового класса.

Когда объект производного класса прекращает свое существование, программа сначала вызывает деструктор производного класса, а затем деструктор базового класса.

Списки инициализаторов членов

Конструктор для производного класса может использовать механизм списка инициализаторов для передачи значений конструктору базового класса. Вот пример:

```
derived::derived(type1 x, type2 y) : base(x,y) // список инициализаторов
{
...
}
```

Здесь `derived` — это производный класс, `base` — базовый класс, а `x` и `y` — переменные, которые используются конструктором базового класса. Если, к примеру, конструктор производного класса получает аргументы `10` и `12`, то данный механизм затем передает `10` и `12` конструктору базового класса, который определен как принимающий аргументы указанных типов. За исключением случая виртуальных базовых классов (см. главу 14), класс может передавать значения только своему непосредственному базовому классу. Однако данный класс может использовать тот же механизм для передачи информации своему непосредственному базовому классу и так далее. Если вы не предусмотрели конструктор базового класса в списке инициализаторов членов, то программа использует стандартный конструктор базового класса. Список инициализаторов членов может использоваться *только* с конструкторами.

Использование производного класса

Для того чтобы использовать производный класс, программе необходимо иметь доступ к объявлениям базового класса. В листинге 13.4 оба объявления классов располагаются в одном и том же заголовочном файле. Вы можете предоставить каждому классу свой собственный заголовочный файл, однако, в связи с тем, что классы зависят друг от друга, гораздо удобнее хранить объявления классов вместе.

Листинг 13.4. tabtenn1.h

```
// tabtenn1.h -- простое наследование
#ifndef TABTENN1_H_
#define TABTENN1_H_
// простой базовый класс
class TableTennisPlayer
{
private:
   enum {LIM = 20};
   char firstname[LIM];
   char lastname[LIM];
   bool hasTable;
public:
   TableTennisPlayer (const char * fn = "none",
                      const char * ln = "none", bool ht = false);
   void Name() const;
   bool HasTable() const { return hasTable; } ;
   void ResetTable(bool v) { hasTable = v; };
};

// простой производный класс
class RatedPlayer : public TableTennisPlayer
{
private:
   unsigned int rating;
public:
   RatedPlayer (unsigned int r = 0, const char * fn = "none",
                const char * ln = "none", bool ht = false);
   RatedPlayer(unsigned int r, const TableTennisPlayer & tp);
   unsigned int Rating() { return rating; }
   void ResetRating (unsigned int r) {rating = r;}
};

#endif
```

В листинге 13.5 представлены определения методов для обоих классов. Опять-таки, вы можете использовать отдельные файлы, однако проще хранить определения вместе.

Листинг 13.5. tabtenn1.cpp

```
// tabtenn1.cpp -- методы базового и производного классов
#include "tabtenn1.h"
#include <iostream>
```

```
#include <cstring>
// методы TableTennisPlayer
TableTennisPlayer::TableTennisPlayer (const char * fn, const char * ln, bool ht)
{
   std::strncpy(firstname, fn, LIM - 1);
   firstname[LIM - 1] = '\0';
   std::strncpy(lastname, ln, LIM - 1);
   lastname[LIM - 1] = '\0';
   hasTable = ht;
}

void TableTennisPlayer::Name() const
{
   std::cout << lastname << ", " << firstname;
}

// методы RatedPlayer
RatedPlayer::RatedPlayer(unsigned int r, const char * fn,
   const char * ln, bool ht) : TableTennisPlayer(fn, ln, ht)
{
   rating = r;
}

RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
   : TableTennisPlayer(tp), rating(r)
{
}
```

Код в листинге 13.6 создает объекты как класса `TableTennisPlayer`, так и класса `RatedPlayer`. Обратите внимание на то, что объекты обоих классов могут использовать методы `Name()` и `HasTable()` класса `TableTennisPlayer`.

Листинг 13.6. `usett1.cpp`

```
// usett1.cpp -- использование базового и производного классов
#include <iostream>
#include "tabtenn1.h"
int main ( void )
{
   using std::cout;
   using std::endl;
   TableTennisPlayer player1("Тара", "Бумди", false);
   RatedPlayer rplayer1(1140, "Мэллори", "Дюк", true);
   rplayer1.Name(); // производный объект использует базовый метод
   if (rplayer1.HasTable())
      cout << ": имеет стол.\n";
   else
      cout << ": не имеет стола.\n";
   player1.Name(); // базовый объект использует базовый метод
   if (player1.HasTable())
      cout << ": имеет стол";
   else
      cout << ": не имеет стола.\n";
```

```
    cout << "Имя: ";
    rplayer1.Name();
    cout << "; Рейтинг: " << rplayer1.Rating() << endl;
    RatedPlayer rplayer2(1212, player1);
    cout << "Имя: ";
    rplayer2.Name();
    cout << "; Рейтинг: " << rplayer2.Rating() << endl;
    return 0;
}
```

Ниже показан вывод программы, представленной в листингах 13.4, 13.5 и 13.6:

```
Дюк, Мэллори: имеет стол.
Бумди, Тара: не имеет стола.
Имя: Duck, Mallory; Рейтинг: 1140
Имя: Boomdea, Tara; Рейтинг: 1212
```

Специальные отношения между производным и базовым классами

Производный класс имеет некоторые специальные отношения с базовым классом. Одно, как вы уже видели только что, заключается в том, что объект производного класса может использовать методы базового класса, при условии, что методы не являются приватными:

```
RatedPlayer rplayer1(1140, "Мэллори", "Дюк", true);
rplayer1.Name(); // производный объект использует методы базового класса
```

Еще два важных момента заключаются в том, что указатель базового класса может указывать на объект производного класса без явного приведения типа, а также что ссылка базового класса может ссылаться на объект производного класса без явного приведения типа:

```
RatedPlayer rplayer1(1140, "Мэллори", "Дюк", true);
TableTennisPlayer & rt = rplayer;
TableTennisPlayer * pt = &rplayer;
rt.Name();  // вызвать Name() с ссылкой
pt->Name(); // вызвать Name() с указателем
```

Однако указатель или ссылка базового класса могут активизировать методы только базового класса, поэтому вы не сможете использовать `rt` или `pt` для того, чтобы обратиться, например, к методу `ResetRanking()`.

Обычно C++ требует, чтобы типы ссылок и указателя соответствовали присваиваемым типам, но это правило ослаблено для наследования. При этом оно ослаблено только в одном направлении. Вы не можете присваивать объекты базового класса и адреса для ссылок и указателей производного класса:

```
TableTennisPlayer player("Бетси", "Блуп", true);
RatedPlayer & rr = player; // НЕ РАЗРЕШАЕТСЯ
RatedPlayer * pr = player; // НЕ РАЗРЕШАЕТСЯ
```

Оба набора правил имеют смысл. Например, рассмотрим возможные последствия того, что ссылка базового класса будет указывать на производный объект. В таком

случае вы можете использовать ссылку базового класса для вызова методов базового класса в отношении объекта производного класса. Поскольку производный класс наследует методы и данные-члены базового класса, это не вызывает проблем. А теперь давайте представим, что могло бы произойти, если бы вы могли присвоить объект базового класса как ссылку на производный класс. Ссылка на производный класс должна активизировать методы производного класса для базового объекта, а это может вызвать проблемы. Например, использование метода RatedPlayer::Rating() для объекта TableTennisPlayer не имеет смысла, поскольку объект TableTennisPlayer не имеет члена rating.

Тот факт, что ссылки и указатели базового класса могут ссылаться на объекты производного класса, имеет несколько интересных следствий. Одно заключается в том, что функции, определенные со ссылкой или указателем базового класса в качестве аргументов, могут использоваться с объектами как базового, так и производного класса. Для примера рассмотрим следующую функцию:

```
void Show(const TableTennisPlayer & rt)
{
    cout << "Имя: ";
    rt.Name();
    cout << "\nСтол: ";
    if (rt.HasTable())
        cout << "да\n";
    else
        cout << "нет\n";
}
```

Формальный параметр rt является ссылкой на базовый класс, следовательно, он может указывать на объект базового или производного класса. Соответственно, вы можете использовать Show() и с аргументом TableTennis, и с аргументом RatedPlayer:

```
TableTennisPlayer player1("Тара", "Бумди", false);
RatedPlayer rplayer1(1140, "Мэллори", "Дюк", true);
Show(player1);  // работает с аргументом TableTennisPlayer
Show(rplayer1); // работает с аргументом RatedPlayer
```

Аналогичная зависимость сохраняется для функции, у которой в качестве формального параметра задан указатель на базовый класс. Она может использоваться как с адресом объекта базового класса, так и с адресом объекта производного класса в качестве фактического аргумента:

```
void Wohs(const TableTennisPlayer * pt); // функция с параметром-
указателем
...
TableTennisPlayer player1("Тара", "Бумди", false);
RatedPlayer rplayer1(1140, "Мэллори", "Дюк", true);
Wohs(&player1);  // работает с аргументом TableTennisPlayer *
Wohs(&rplayer1); // работает с аргументом RatedPlayer *
```

Свойство совместимости ссылок также позволяет указывать объект базового класса в качестве первоначального значения объекта производного класса, хотя и несколько косвенно. Предположим, что имеется следующий код:

```
RatedPlayer olaf1(1840, "Олаф", "Лоф", true);
TableTennisPlayer olaf2(olaf1);
```

Точным соответствием для инициализации olaf2 будет конструктор со следующим прототипом:

```
TableTennisPlayer(const RatedPlayer &); // не существует
```

Определения класса не включают в себя данный конструктор, однако существует неявный конструктор копирования:

```
// неявный конструктор копирования
TableTennisPlayer(const TableTennisPlayer &);
```

Формальный параметр является ссылкой на базовый тип, значит, он может ссылаться и на производный тип. Таким образом, при попытке присвоить olaf2 в качестве первоначального значения olaf1 используется данный конструктор, который копирует члены firstname, lastname и hasTable. Другими словами, он инициализирует olaf2 для объекта TableTennisPlayer, вложенного в olaf1 объекта RatedPlayer.

Аналогично вы можете присвоить объект производного класса объекту базового класса:

```
RatedPlayer olaf1(1840, "Олаф", "Лоф", true);
TableTennisPlayer winner;
winner = olaf1; // присвоить производный объект базовому объекту
```

В этом случае программа использует неявную перегруженную операцию присваивания:

```
TableTennisPlayer & operator=(const TableTennisPlayer &) const;
```

Ссылка базового класса указывает на объект производного класса, при этом в winner копируется только базовая часть olaf1.

Наследование: отношение *is-a*

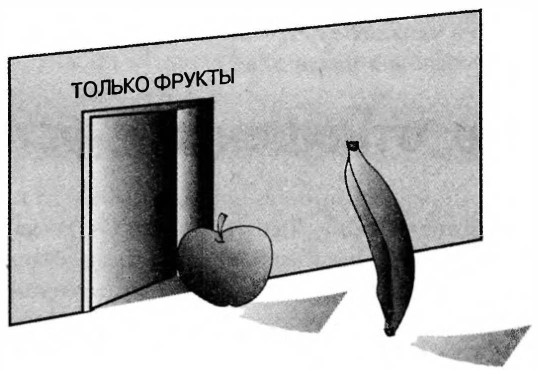
Специальное отношение между производным и базовым классами основано на внутренней модели наследования C++. Фактически, в C++ имеется три варианта наследования: общедоступное, защищенное и приватное. Общедоступное наследование является наиболее общей формой, оно моделирует отношение *is-a* ("является"). Это условное сокращение для обозначения того, что объект производного класса должен также быть объектом базового класса. У вас появляется возможность делать с объектом производного класса все, что вы делаете с объектом базового класса. Предположим, например, что у вас есть класс Fruit. Он хранит, скажем, вес и содержание калорий во фруктах. Поскольку банан является одним из видов фруктов, вы можете породить класс Banana от класса Fruit. Новый класс унаследует все члены данных исходного класса, то есть объект Banana будет иметь члены, представляющие вес и содержание калорий в банане. В новый класс Banana также можно добавлять члены, которые относятся отдельно к бананам и не относятся к фруктам вообще, например, Banana Institute Peel Index (Индекс кожуры института бананов). Поскольку в производном классе могут быть добавлены свойства, то, возможно, более точным

было бы название отношения как *is-a-kind-of* ("является разновидностью"), однако общепринятым является термин *is-a*.

Для того чтобы лучше понять отношение *is-a*, давайте рассмотрим несколько примеров, которые не соответствуют данной модели. Общедоступное наследование не моделирует отношение *has-a* ("содержит"). Обед, к примеру, может содержать фрукт, однако, сам обед, в общем-то, не является фруктом. Следовательно, не стоит порождать класс `Lunch` от класса `Fruit`, пытаясь поместить фрукт в обед. Правильным способом для ввода фрукта в обед является отношение *has-a*: обед содержит фрукт. Как вы увидите в главе 14, это легче всего моделировать путем включения объекта `Fruit` в качестве члена данных класса `Lunch` (рис. 13.3).

Общедоступное наследование не моделирует отношение *is-like-a* ("подобный") — то есть оно не делает сравнений. Очень часто говорят, что адвокаты похожи на акул. Но это не означает, что адвокат на самом деле акула. Известно, что акулы могут жить только под водой. Следовательно, вы не должны порождать класс `Lawyer` от класса `Shark`. Наследование может добавлять свойства в базовый класс, но оно не удаляет свойств из базового класса. В некоторых случаях общие характеристики могут обрабатываться при помощи конструирования класса, управляющего данными характеристиками, и последующего применения этого класса либо в отношении *is-a*, либо в отношении *has-a* для определения зависимых классов.

Общедоступное наследование не моделирует отношение *is-implemented-as-a* ("реализован как"). Например, вы можете реализовать стек с применением массива. Однако будет неправильно порождать класс `Stack` от класса `Array`. Стек не является массивом. Например, индексация массива не является свойством стека. Стек также можно реализовать другим способом, например, с помощью связного списка. Правильнее будет скрыть реализацию массива за счет предоставления приватного объекта `Array`.



Банан является фруктом, но обед содержит банан

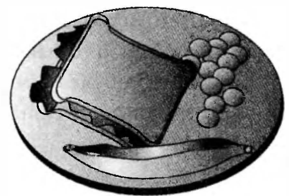


Рис. 13.3. Отношения is-a и has-a

Общедоступное наследование не моделирует отношение *uses-a* (“используется”). Например, компьютер может использовать лазерный принтер, однако не имеет смысла порождать класс `Printer` от класса `Computer`, и наоборот. При этом можно разрабатывать дружественные функции или классы для управления взаимодействием между объектами `Printer` и `Computer`.

В C++ ничто не мешает использовать общедоступное наследование для моделирования отношений *has-a*, *is-implemented-as-a* или *uses-a*. Однако это, как правило, приводит к проблемам с программированием. Поэтому давайте будем придерживаться отношений *is-a*.

Полиморфное общедоступное наследование

Пример наследования `RatedPlayer` является достаточно простым. Объекты производного класса используют методы базового класса без изменений. Однако возможны ситуации, когда понадобится, чтобы метод вел себя для производного класса не так, как для базового. Другими словами, способ поведения отдельного метода может отличаться в зависимости от объекта, который его вызывает. Это более сложное поведение называется *полиморфным*, поскольку у вас появляется несколько моделей поведения для метода в зависимости от контекста. Существует два основных механизма для реализации полиморфного общедоступного наследования:

- Переопределение методов базового класса в производном классе.
- Использование виртуальных функций.

Рассмотрим еще один пример. Вы получили неплохой опыт во время работы в клубе Webtown Social Club, и теперь готовы стать главным программистом банка Pontoon National Bank. Первое задание, которое вам дает банк, связано с разработкой двух классов. Один класс представляет чековый счет Brass Account, а второй — чековый счет Brass Plus, в который добавлено свойство защиты от превышения кредита. Другими словами, если клиент выписывает чек на сумму, большую (но не намного большую), чем его баланс, то банк покрывает этот чек, предоставляя ссуду клиенту для дополнительного платежа и начисляет пеню. Вы должны описать два счета в терминах хранящихся на них данных и допустимых операций.

Для начала, вот информация для счета Brass Account:

- Имя клиента.
- Номер счета.
- Текущий баланс.

А вот операции, которые должны быть представлены:

- Создание счета.
- Внесение денег на счет.
- Снятие денег со счета.
- Отображение состояния счета.

Для счета Brass Plus предусматриваются все свойства Brass Account, а также следующие дополнительные информационные элементы:

- Верхний предел защиты от превышения кредита.
- Процентная ставка, начисляемая на ссуду превышения кредита.
- Величина превышения кредита, которую клиент должен банку на данный момент.

Никаких дополнительных операций не требуется, однако две операции необходимо реализовать по-другому:

- Операция снятия денег для счета Brass Plus должна объединяться с защитой от превышения кредита.
- Операция отображения должна показывать всю дополнительную информацию, которая требуется для счета Brass Plus.

Предположим, что вы назвали один класс `Brass`, а второй — `BrassPlus`. Должны ли вы порождать `BrassPlus` из `Brass` с применением общедоступного наследования? Для того чтобы ответить на этот вопрос, сначала ответьте на другой: соответствует ли класс критерию отношения *is-a*? Конечно. Все, что подходит объекту `Brass`, будет подходить и объекту `BrassPlus`. Оба хранят имя клиента, номер счета и баланс. На оба счета вы можете вкладывать и снимать деньги, а также отображать информацию о текущем балансе. Обратите внимание, что отношение *is-a* в общем случае не является симметричным. Фрукт не обязательно будет бананом, аналогично объект `Brass` не будет обладать всеми свойствами объекта `BrassPlus`.

Разработка классов `Brass` и `BrassPlus`

Информация класса счета Brass Account довольно-таки прямолинейна, однако банк не предоставил достаточную информацию о том, как работает система превышения кредита. В ответ на ваш запрос о дополнительных деталях готовый помочь банк Pontoon National Bank уполномочен заявить следующее:

- Счет Brass Plus ограничивает сумму денег, которую банк может одолжить вам для покрытия суммы превышения кредита. Значение по умолчанию составляет $500, но для некоторых клиентов может быть установлен другой предел.
- Банк может изменять предел превышения кредита клиента.
- Счет Brass Plus предусматривает начисление процентов на ссуду. Значение по умолчанию составляет 10%, но для некоторых клиентов может быть установлена другая ставка.
- Банк может изменять процентную ставку клиента.
- Счет отслеживает, какую сумму клиент должен банку (превышение кредита плюс проценты). Клиент не может погасить эту сумму через обычный вклад или через перевод денег с другого счета. Он должен заплатить наличными специальному банковскому служащему, который, если понадобится, будет разыскивать клиента. Как только долг погашен, на счету устанавливается нулевое значение задолженности.

Последнее свойство нетипично для банковского способа ведения дел, однако, по счастливой случайности оно упрощает решение программных проблем.

Приведенный перечень наталкивает на мысль, что для нового класса нужны конструкторы, которые предоставляют информацию о состоянии счета и включают в себя предел долга со стандартным значением $500 и процентную ставку со стандартным значением 10%. Должны также существовать методы для обновления предела долга, процентной ставки и текущего долга. Это все, что требуется добавить в класс `Brass`. И это будет сделано в объявлении класса `BrassPlus`.

Информация о данных двух классах предполагает объявления классов, похожих на приведенные в листинге 13.7.

Листинг 13.7. brass.h

```
// brass.h -- классы банковских счетов
#ifndef BRASS_H_
#define BRASS_H_
// Класс счета Brass Account
class Brass
{
private:
   enum {MAX = 35};
   char fullName[MAX];
   long acctNum;
   double balance;
public:
   Brass(const char *s = "Nullbody", long an = -1,
            double bal = 0.0);
   void Deposit(double amt);
   virtual void Withdraw(double amt);
   double Balance() const;
   virtual void ViewAcct() const;
   virtual ~Brass() {}
};
// Класс счета Brass Plus
class BrassPlus : public Brass
{
private:
   double maxLoan;
   double rate;
   double owesBank;
public:
   BrassPlus(const char *s = "Nullbody", long an = -1,
            double bal = 0.0, double ml = 500,
            double r = 0.10);
   BrassPlus(const Brass & ba, double ml = 500, double r = 0.1);
   virtual void ViewAcct()const;
   virtual void Withdraw(double amt);
   void ResetMax(double m) { maxLoan = m; }
   void ResetRate(double r) { rate = r; };
   void ResetOwes() { owesBank = 0; }
};
#endif
```

В листинге 13.7 следует обратить внимание на ряд моментов:

- Класс BrassPlus добавляет в класс Brass три новых приватных данных-членов и три новых общедоступных метода.
- Оба класса объявляют методы ViewAcct() и Withdraw(); однако они будут вести себя для объекта BrassPlus иначе, нежели для объекта Brass.
- Класс Brass использует новое ключевое слово virtual при объявлении ViewAcct() и Withdraw(). Эти методы теперь называются виртуальными функциями.
- Класс Brass также объявляет виртуальный деструктор, несмотря на то, что он ничего не делает.

Первый пункт в данном списке не говорит ни о чем новом. Класс RatedPlayer делал нечто подобное, когда добавлял новый член данных и два новых метода в класс TableTennisPlayer.

Вторым важным моментом в списке является способ, которым в объявлениях устанавливается, что методы ведут себя по-другому для производного класса. Два прототипа ViewAcct() указывают, что должны существовать два отдельных определения метода. Уточненным именем для версии базового класса служит Brass::ViewAcct(), а для производного класса — BrassPlus::ViewAcct(). Программа будет использовать тип объекта для того, чтобы определить, какую версию применять:

```
Brass dom("Dominic Banker", 11224, 4183.45);
BrassPlus dot("Dorothy Banker", 12118, 2592.00);
dom.ViewAcct(); // использовать Brass::ViewAcct()
dot.ViewAcct(); // использовать BrassPlus::ViewAcct()
```

По аналогии употребляются две версии Withdraw(): одна для использования объектами Brass и одна — для объектов BrassPlus. Методы, которые ведут себя одинаково для обоих классов, такие как Deposit() и Balance(), объявляются только в базовом классе.

Третий вопрос (употребление virtual) является более сложным, чем первые два. Он определяет, какой метод используется, если метод вызывается через ссылку или указатель вместо объекта. Если вы не применяете ключевое слово virtual, то программа выбирает метод, основываясь на типе ссылки или указателя. Если вы используете ключевое слово virtual, то программа выбирает метод, основываясь на типе объекта, на который указывает ссылка или указатель. Вот как ведет себя программа, если ViewAcct() не является виртуальной:

```
// поведение при невиртуальной функции ViewAcct()
// метод выбирается в соответствии с типом ссылки
Brass dom("Dominic Banker", 11224, 4183.45);
BrassPlus dot("Dorothy Banker", 12118, 2592.00);
Brass & b1_ref = dom;
Brass & b2_ref = dot;
b1_ref.ViewAcct(); // использовать Brass::ViewAcct()
b2_ref.ViewAcct(); // использовать Brass::ViewAcct()
```

Ссылочные переменные относятся к типу Brass, поэтому выбирается Brass::ViewAccount(). Использование указателей на Brass вместо ссылки дает аналогичное поведение.

Для сравнения продемонстрируем поведение при виртуальной функции ViewAcct():

```
// поведение при виртуальной функции ViewAcct()
// метод выбирается в соответствии с типом объекта
Brass dom("Dominic Banker", 11224, 4183.45);
BrassPlus dot("Dorothy Banker", 12118, 2592.00);
Brass & b1_ref = dom;
Brass & b2_ref = dot;
b1_ref.ViewAcct(); // использовать Brass::ViewAcct()
b2_ref.ViewAcct(); // использовать BrassPlus::ViewAcct()
```

В этом случае обе ссылки относятся к типу Brass, но b2_ref ссылается на объект BrassPlus, поэтому для него применяется BrassPlus::ViewAcct(). Использование указателей на Brass вместо ссылки обеспечивает аналогичное поведение.

Оказывается, как вы увидите чуть позже, что такое поведение виртуальных функций весьма удобно. Поэтому общей рекомендацией будет объявление в базовом классе в качестве виртуальных тех методов, которые могут быть переопределены в производном классе. Если метод объявлен как виртуальный в базовом классе, то он автоматически является виртуальным в производном классе. Однако рекомендуется в объявлениях базового класса также информировать, какие функции являются виртуальными, используя ключевое слово virtual.

Четвертый момент заключается в том, что базовый класс объявляет виртуальный деструктор. Это необходимо для того, чтобы обеспечить корректную последовательность деструкторов, вызываемых при уничтожении производного объекта. Этот вопрос мы обсудим более подробно далее в данной главе.

На память!

Если вы планируете переопределять какой-либо метод базового класса в производном классе, то обычно этот метод объявляется как виртуальный в базовом классе. Это вынуждает программу выбирать версию метода, основываясь на типе объекта вместо типа ссылки и указателя. Также принято объявлять виртуальный деструктор в базовом классе.

Реализации классов

Следующим шагом является подготовка реализации классов. Часть этого уже была сделана с помощью встроенных определений функций в заголовочном файле. В листинге 13.8 предлагаются оставшиеся определения методов. Обратите внимание, что ключевое слово virtual присутствует только в прототипах методов в объявлении класса, а не в определениях методов в листинге 13.8.

Листинг 13.8. brass.cpp

```
// brass.cpp -- методы классов банковских счетов
#include <iostream>
#include <cstring>
#include "brass.h"
using std::cout;
using std::ios_base;
using std::endl;
```

```
// методы  Brass
Brass::Brass(const char *s, long an, double bal)
{
   std::strncpy(fullName, s, MAX - 1);
   fullName[MAX - 1] = '\0';
   acctNum = an;
   balance = bal;
}
void Brass::Deposit(double amt)
{
   if (amt < 0)
      cout << "Отрицательный вклад не допускается; "
           << "попытка вклада отменена.\n";
   else
      balance += amt;
}
void Brass::Withdraw(double amt)
{
   if (amt < 0)
      cout << "Снимаемая сумма должна быть положительной; "
           << "попытка снятия отменена.\n";
   else if (amt <= balance)
   balance -= amt;
   else
      cout << "Снимаемая сумма $" << amt
           << " превышает ваш баланс.\n"
           << "Попытка снятия отменена.\n";
}
double Brass::Balance() const
{
   return balance;
}
void Brass::ViewAcct() const
{
   // установить формат ###.##
   ios_base::fmtflags initialState =
      cout.setf(ios_base::fixed, ios_base::floatfield);
   cout.setf(ios_base::showpoint);
   cout.precision(2);
   cout << "Клиент: " << fullName << endl;
   cout << "Номер счета: " << acctNum << endl;
   cout << "Баланс: $" << balance << endl;
   cout.setf(initialState); // восстановить исходный формат
}

// методы BrassPlus
BrassPlus::BrassPlus(const char *s, long an, double bal,
        double ml, double r) : Brass(s, an, bal)
{
   maxLoan = ml;
   owesBank = 0.0;
   rate = r;
}
```

```
BrassPlus::BrassPlus(const Brass & ba, double ml, double r)
       : Brass(ba) // использует неявный конструктор копирования
{
   maxLoan = ml;
   owesBank = 0.0;
   rate = r;
}
// переопределить работу ViewAcct()
void BrassPlus::ViewAcct() const
{
   // установить формат ###.##
   ios_base::fmtflags initialState =
      cout.setf(ios_base::fixed, ios_base::floatfield);
   cout.setf(ios_base::showpoint);
   cout.precision(2);
   Brass::ViewAcct(); // отобразить базовый раздел
   cout << "Максимальный заем: $" << maxLoan << endl;
   cout << "Долг банку: $" << owesBank << endl;
   cout << "Процент на заем: " << 100 * rate << "%\n";
   cout.setf(initialState);
}
// переопределить работу Withdraw()
void BrassPlus::Withdraw(double amt)
{
   // установить формат ###.##
   ios_base::fmtflags initialState =
      cout.setf(ios_base::fixed, ios_base::floatfield);
   cout.setf(ios_base::showpoint);
   cout.precision(2);
   double bal = Balance();
   if (amt <= bal)
      Brass::Withdraw(amt);
   else if ( amt <= bal + maxLoan - owesBank)
   {
      double advance = amt - bal;
      owesBank += advance * (1.0 + rate);
      cout << "Банковский аванс: $" << advance << endl;
      cout << "Налог: $" << advance * rate << endl;
      Deposit(advance);
      Brass::Withdraw(amt);
   }
   else
      cout << "Предел кредита превышен. Транзакция отменена.\n";
   cout.setf(initialState);
}
```

Прежде чем рассмотреть некоторые детали листинга 13.8 вроде управления форматированием в некоторых методах давайте исследуем те аспекты, которые относятся непосредственно к наследованию. Вспомните о том, что производный класс не имеет прямого доступа к приватным данным базового класса. Он вынужден использовать общедоступные методы базового класса для получения доступа к этим данным. Средства доступа зависят от метода. Конструкторы используют одну методику, дру-

гие функции-члены — другие методики. Техническим приемом, который применяют конструкторы производного класса для инициализации приватных данных базового класса, является синтаксис списка инициализаторов членов. Этот прием употребляют конструкторы класса RatedPlayer, а также конструкторы BrassPlus:

```
BrassPlus::BrassPlus(const char *s, long an, double bal,
        double ml, double r) : Brass(s, an, bal)
{
   maxLoan = ml;
   owesBank = 0.0;
   rate = r;
}
BrassPlus::BrassPlus(const Brass & ba, double ml, double r)
        : Brass(ba) // использует неявный конструктор копирования
{
   maxLoan = ml;
   owesBank = 0.0;
   rate = r;
}
```

Каждый из этих конструкторов использует синтаксис списка инициализаторов членов для передачи информации базового класса конструктору базового класса, а затем применяет тело конструктора для инициализации новых элементов данных, добавляемых классом BrassPlus.

Не-конструкторы не могут использовать синтаксис списка инициализаторов членов. Однако метод производного класса может вызвать общедоступный метод базового класса. Например, игнорируя аспект форматирования, ядро версии BrassPlus функции ViewAcct() может быть таким:

```
// переопределить работу ViewAcct()
void BrassPlus::ViewAcct() const
{
...
   Brass::ViewAcct(); // отобразить базовый раздел
   cout << "Максимальный заем: $" << maxLoan << endl;
   cout << "Долг банку: $" << owesBank << endl;
   cout << "Процент на заем: " << 100 * rate << "%\n";
...
}
```

Другими словами, BrassPlus::ViewAcct() отображает добавленные члены данных BrassPlus и вызывает метод базового класса Brass::ViewAcct() для отображения членов данных базового класса. Использование операции разрешения контекста в методе производного класса для вызова метода базового класса является стандартным приемом.

Очень важно то, что код использует операцию разрешения контекста. Предположим, что вы вместо предыдущего написали следующий код:

```
// переопределить работу ViewAcct()
void BrassPlus::ViewAcct() const
{
...
   ViewAcct(); // ой! рекурсивный вызов
...
}
```

Если код не использует операцию разрешения контекста, то компилятор предполагает, что ViewAcct() есть BrassPlus::ViewAcct(), а последний метод создает рекурсивную функцию, которая не имеет завершения — это не очень хорошо, согласитесь?

Далее рассмотрим метод BrassPlus::Withdraw(). Если клиент снимает сумму, превышающую баланс, то метод должен организовать ссуду. Он может применить Brass::Withdraw() для доступа к члену баланса, но Brass::Withdraw() выдает сообщение об ошибке, если снимаемая сумма превышает баланс. В данной реализации можно избежать этого сообщения, если воспользоваться методом Deposit() для открытия ссуды, а затем, когда достаточные средства уже доступны, вызвать Brass::Withdraw():

```
// переопределить работу Withdraw()
void BrassPlus::Withdraw(double amt)
{
...
   double bal = Balance();
   if (amt <= bal)
      Brass::Withdraw(amt);
   else if ( amt <= bal + maxLoan - owesBank)
   {
      double advance = amt - bal;
      owesBank += advance * (1.0 + rate);
      cout << "Банковский аванс: $" << advance << endl;
      cout << "Налог: $" << advance * rate << endl;
      Deposit(advance);
      Brass::Withdraw(amt);
   }
   else
      cout << "Предел кредита превышен. Транзакция отменена.\n";
...
}
```

Обратите внимание, что метод использует функцию базового класса Balance() для определения исходного баланса. Код не должен применять операцию разрешения контекста для Balance(), поскольку этот метод не переопределялся в производном классе.

Методы ViewAcct() применяют команды форматирования для установки следующего режима вывода для величин с плавающей запятой: с фиксированной запятой и с двумя знаками после нее. Когда эти режимы установлены, вывод остается в предыдущем режиме, потому все, что делают эти методы — это сбрасывают режим форматирования в состояние, которое имело место до вызова этих методов. Таки образом, данные методы фиксируют исходное состояние форматирования с помощью следующего кода:

```
ios_base::fmtflags initialState =
   cout.setf(ios_base::fixed, ios_base::floatfield);
```

Метод setf() возвращает значение, представляющее состояние формата до вызова функции. Новые реализации C++ определяют тип ios_base::fmtflags в качестве типа для этого значения, а данный оператор сохраняет состояние в переменной initialState этого типа. (В более старых версиях C++ вместо этого типа приме-

няется unsigned int.) Когда функция ViewAcct() завершает работу, она передает initialState в setf() в качестве аргумента.

```
cout.setf(initialState);
```

Эта строка возвращает исходные установки форматирования.

Использование классов Brass и BrassPlus

В листинге 13.9 представлен код, тестирующий классы Brass и BrassPlus.

Листинг 13.9. usebrass1.cpp

```
// usebrass1.cpp -- тестирование классов банковских счетов
// компилировать вместе с brass.cpp
#include <iostream>
#include "brass.h"
int main()
{
   using std::cout;
   using std::endl;

   Brass Piggy("Porcelot Pigg", 381299, 4000.00);
   BrassPlus Hoggy("Horatio Hogg", 382288, 3000.00);
   Piggy.ViewAcct();
   cout << endl;
   Hoggy.ViewAcct();
   cout << endl;
   cout << "Вклад $1000 на счет Horatio Hogg:\n";
   Hoggy.Deposit(1000.00);
   cout << "Новый баланс счета Horatio Hogg: $" << Hoggy.Balance() << endl;
   cout << "Снятие $4200 со счета Porcelot Pigg:\n";
   Piggy.Withdraw(4200.00);
   cout << "Баланс счета Porcelot Pigg: $" << Piggy.Balance() << endl;
   cout << "Снятие $4200 со счета Horatio Hogg:\n";
   Hoggy.Withdraw(4200.00);
   Hoggy.ViewAcct();
   return 0;
}
```

В приведенных ниже выходных данных программы из листинга 13.9 обратите внимание на то, что к Horatio Hogg применяется защита от превышения кредита, а к Porcelot Pigg — нет:

```
Клиент: Porcelot Pigg
Номер счета: 381299
Баланс: $4000.00

Клиент: Horatio Hogg
Номер счета: 382288
Баланс: $3000.00
Максимальный заем: $500.00
Долг банку: $0.00
Процент на заем: 10.00%
```

```
Вклад $1000 на счет Horatio Hogg:
Новый баланс счета Horatio Hogg: $4000.00
Снятие $4200 со счета Porcelot Pigg:
Снимаемая сумма $4200.00 превышает ваш баланс.
Попытка снятия отменена.
Баланс счета Porcelot Pigg: $4000.00
Снятие $4200 со счета Horatio Hogg:
Банковский аванс: $200.00
Налог: $20.00
Клиент: Horatio Hogg
Номер счета: 382288
Баланс: $0.00
Максимальный заем: $500.00
Долг банку: $220.00
Процент на заем: 10.00%
```

Демонстрация поведения виртуальных методов

В листинге 13.9 методы активизируются объектами, а не указателями или ссылками, поэтому программа не использует возможности виртуальных методов. Давайте рассмотрим пример, в котором виртуальные методы начинают действовать. Предположим, что вам требуется управлять смесью счетов Brass и BrassPlus. Было бы удобно иметь единственный массив, хранящий набор объектов Brass и BrassPlus, однако это невозможно. Каждый элемент массива должен относиться к одному и тому же типу, а Brass и BrassPlus — два отдельных типа. При этом вы можете создать массив указателей на Brass. В этом случае все элементы будут одного типа, но благодаря модели общедоступного наследования указатель на Brass может ссылаться либо на объект Brass, либо на объект BrassPlus. Таким образом, фактически у вас имеется способ представления совокупности данных более чем одного типа в единственном массиве. Это и есть полиморфизм, и в листинге 13.10 показан простой пример.

Листинг 13.10. usebrass2.cpp

```
// usebrass2.cpp -- пример полиморфизма
// компилировать вместе с brass.cpp
#include <iostream>
#include "brass.h"
const int CLIENTS = 4;
const int LEN = 40;
int main()
{
   using std::cin;
   using std::cout;
   using std::endl;
   Brass * p_clients[CLIENTS];

   int i;
   for (i = 0; i < CLIENTS; i++)
   {
     char temp[LEN];
     long tempnum;
     double tempbal;
```

```
    char kind;
    cout << "Введите имя клиента: ";
    cin.getline(temp, LEN);
    cout << "Введите номер счета клиента: ";
    cin >> tempnum;
    cout << "Введите начальный баланс: $";
    cin >> tempbal;
    cout << "Введите 1 для счета Brass Account или "
         << "2 для счета Brass Plus: ";
    while (cin >> kind && (kind != '1' && kind != '2'))
        cout <<"Введите 1 или 2: ";
    if (kind == '1')
     p_clients[i] = new Brass(temp, tempnum, tempbal);
    else
    {
       double tmax, trate;
       cout << "Введите предел превышения кредита: $";
       cin >> tmax;
       cout << "Введите ставку процента "
           << "как десятичную дробь: ";
       cin >> trate;
       p_clients[i] = new BrassPlus(temp, tempnum, tempbal,
                           tmax, trate);
    }
    while (cin.get() != '\n')
        continue;
  }
  cout << endl;
  for (i = 0; i < CLIENTS; i++)
  {
     p_clients[i]->ViewAcct();
     cout << endl;
  }

  for (i = 0; i < CLIENTS; i++)
  {
     delete p_clients[i]; // освободить память
  }
  cout << "Готово.\n";

  return 0;
}
```

Программа в листинге 13.10 позволяет ввести тип добавляемого счета. Затем используется операция new для создания и инициализации объекта соответствующего типа.

Ниже показан пример выполнения программы из листинга 13.10:

```
Введите имя клиента: Harry Fishsong
Введите номер счета клиента: 112233
Введите начальный баланс: $1500
Введите 1 для счета Brass Account или 2 для счета Brass Plus: 1
Введите имя клиента: Dinah Otternoe
```

```
Введите номер счета клиента: 121213
Введите начальный баланс: $1800
Введите 1 для счета Brass Account или 2 для счета Brass Plus: 2
Введите предел превышения кредита: $350
Введите ставку процента как десятичную дробь: 0.12
Введите имя клиента: Brenda Birdherd
Введите номер счета клиента: 212118
Введите начальный баланс: $5200
Введите 1 для счета Brass Account или 2 для счета Brass Plus: 2
Введите предел превышения кредита: $800
Введите ставку процента как десятичную дробь: 0.10
Введите имя клиента: Tim Turtletop
Введите номер счета клиента: 233255
Введите начальный баланс: $688
Введите 1 для счета Brass Account или 2 для счета Brass Plus: 1

Клиент: Harry Fishsong
Номер счета: 112233
Баланс: $1500.00

Клиент: Dinah Otternoe
Номер счета: 121213
Баланс: $1800.00
Максимальный заем: $350.00
Долг банку: $0.00
Процент на заем: 12.00%

Клиент: Brenda Birdherd
Номер счета: 212118
Баланс: $5200.00
Максимальный заем: $800.00
Долг банку: $0.00
Процент на заем: 10.00%

Клиент: Tim Turtletop
Номер счета: 233255
Баланс: $688.00

Готово.
```

Полиморфизм обеспечивается с помощью следующего кода:

```
for (i = 0; i < CLIENTS; i++)
{
   p_clients[i]->ViewAcct();
   cout << endl;
}
```

Если элемент массива указывает на объект `Brass`, вызывается `Brass::ViewAcct()`, а если на объект `BrassPlus` — то `BrassPlus::ViewAcct()`. Если функция `Brass::ViewAcct()` была объявлена как виртуальная, то во всех случаях будет вызываться `Brass:ViewAcct()`.

Необходимость в виртуальных деструкторах

Код в листинге 13.10, в котором используется операция `delete` для освобождения объектов, память под которые выделена операцией `new`, показывает, почему базовый класс должен иметь виртуальный деструктор, даже если деструкторов вообще не нужно. Если деструкторы не виртуальные, то тогда вызывается только деструктор, соответствующий типу указателя. Для листинга 13.10 это означает, что всегда будет вызываться только деструктор `Brass`, даже если указатель ссылается на объект `BrassPlus`. При наличии же виртуальных деструкторов, если указатель ссылается на объект `BrassPlus`, вызывается деструктор `BrassPlus`. Когда деструктор `BrassPlus` завершает свою работу, он автоматически вызывает конструктор базового класса. Таким образом, применение виртуальных деструкторов гарантирует, что деструкторы вызываются в корректной последовательности. В листинге 13.10 такое правильное поведение не является важным, поскольку деструкторы ничего не делают. Однако, если, например, `BrassPlus` имел бы функционирующий деструктор, то он обязательно должен бы быть виртуальным.

Статическое и динамическое связывание

Какой блок рабочей программы выполняется, когда программа вызывает функцию? За этот вопрос несет ответственность компилятор. Интерпретация вызова функции в исходном коде в виде запуска определенной части кода во время выполнения называется *связыванием* имени функции. В С эта задача является простой, так как каждое имя соответствует отдельной функции. В C++ эта задача гораздо сложнее из-за перегрузки функций. Компилятор должен просматривать аргументы функции, а также ее имя для того, чтобы вычислить, какую функцию использовать. Несмотря на все это, такой тип связывания является задачей, которую компилятор С или C++ может выполнить во время компиляции. Связывание, происходящее во время компиляции, называется *статическим связыванием* (или *ранним связыванием*). Однако виртуальные функции существенно усложняют работу. Как показано в листинге 13.10, решение о том, какую функцию использовать, не может быть принято во время компиляции, поскольку компилятор не знает, с объектом какого типа собирается работать пользователь. Следовательно, компилятор должен генерировать код, который позволяет выбирать соответствующий виртуальный метод во время работы программы. Такой процесс называется *динамическим связыванием* (или *поздним связыванием*).

Теперь, когда вы уже видели виртуальные методы в работе, давайте рассмотрим этот процесс более глубоко. Начнем с того, как C++ поддерживает совместимость типов указателя и ссылки.

Совместимость типов указателя и ссылки

Динамическое связывание в C++ взаимодействует с методами, вызываемыми по указателям и ссылкам, и управляется, в частности, процессом наследования. Один способ, с помощью которого общедоступное наследование моделирует отношение *is-a*, заключается в управлении указателями и ссылками на объекты. Как правило, C++ не позволяет присваивать адрес одного типа указателю другого типа. Также не разрешается ссылке одного типа ссылаться на другой тип:

```
double x = 2.5;
int * pi = &x; //неправильное присваивание, несоответствие типов указателей
long & rl = x; //неправильное присваивание, несоответствие типов указателей
```

Однако, как вы уже видели, ссылка или указатель на базовый класс может ссылаться на объект производного класса без использования явного приведения типа. Например, допустимы следующие инициализации:

```
BrassPlus dilly ("Annie Dill", 493222, 2000);
Brass * pb = &dilly; // нормально
Brass & rb = dilly;  // нормально
```

Преобразование ссылки или указателя производного класса в ссылку или указатель базового класса называется *восходящим*. Оно всегда разрешено для общедоступного наследования без необходимости явного приведения типа. Это правило является частью выражения отношения *is-a*. Объект `BrassPlus` является объектом `Brass` в том смысле, что он наследует все члены данных и функции объекта `Brass`. Поэтому все, что вы можете делать с объектом `Brass`, вы можете также делать и с объектом `BrassPlus`. Таким образом, функция, разработанная для управления ссылкой `Brass`, может без опасения вызвать проблемы выполнять все те же действия и для объекта `BrassPlus`. Та же самая идея имеет место, если вы передаете указатель на объект в качестве аргумента функции. Восходящее преобразование является транзитивным. Другими словами, если вы будете порождать класс `BrassPlusPlus` от `BrassPlus`, то указатель или ссылка `Brass` смогут обращаться к объекту `Brass`, `BrassPlus` или `BrassPlusPlus`.

Обратный процесс, то есть преобразование указателя или ссылки базового класса в указатель или ссылку производного класса, называется *нисходящим*, и оно недопустимо без явного приведения типа. Причиной этого ограничения является то, что отношение *is-a* не является симметричным. Производный класс может добавлять новые данные-члены, и методы класса, которые используются с этими данными-членами, не могут применяться для базового класса. Например, предположим, что вы порождаете класс `Singer` от класса `Employee`, добавляя при этом член данных, представляющий вокальный диапазон певца, и метод `range()`, который сообщает величину вокального диапазона. Применять метод `range()` к объекту `Employee` бессмысленно. Но если бы было допустимо неявное нисходящее преобразование, то вы могли бы случайно установить адрес объекта `Employee` для указателя на `Singer` и применить указатель для вызова метода `range()` (рис. 13.4).

Благодаря неявному восходящему преобразованию становится возможным то, что указатель или ссылка базового класса могут обращаться либо к объекту базового класса, либо к объекту производного класса. Это приводит к необходимости динамического связывания. Виртуальные методы C++ отвечают этой необходимости.

Виртуальные методы и динамическое связывание

Давайте вернемся к процессу вызова метода через ссылку или указатель. Рассмотрим следующий код:

```
BrassPlus ophelia;  // объект производного класса
Brass * bp;         // указатель базового класса
bp = &ophelia;      // указатель Brass на объект BrassPlus
bp->ViewAcct();     // какая версия будет вызвана?
```

```
class Employee
{
private:
    char name[40];
    ...
public:
    void show_name();
    ...
};
class Singer : public Employee
{
    ...
public:
    void range();
    ...
};
...
Employee veep;
Singer trala;
...
Employee * pe = &trala;
Singer * ps = (Singer *) &veep;
...
pe->show_name();
ps->range();
```

Восходящее преобразование — допустимо неявное приведение типа

Нисходящее преобразование — требуется явное приведение типа

Восходящее преобразование приводит к безопасной операции, поскольку **Singer** является **Employee** (каждый **Singer** наследует **name**)

Нисходящее преобразование может привести к опасной операции, так как **Employee** не является **Singer** (**Employee** не нуждается в методе **range()**)

Рис. 13.4. Восходящее и нисходящее преобразование

Как обсуждалось ранее, если `ViewAcct()` не объявлена как виртуальная в базовом классе, то `bp->ViewAcct()` руководствуется типом указателя (`Brass *`) и вызывает `Brass::ViewAcct()`. Тип указателя известен во время компиляции, поэтому компилятор может связать `ViewAcct()` с `Brass::ViewAcct()` еще на этапе компиляции. Короче говоря, компилятор использует статическое связывание для невиртуальных методов.

Однако если `ViewAcct()` объявлена в базовом классе как виртуальная, то `bp->ViewAcct()` руководствуется типом объекта (`BrassPlus`) и вызывает `BrassPlus::ViewAcct()`. В этом примере видно, что тип объекта `BrassPlus`, но в общем случае (как в листинге 13.10) тип объекта может быть определен только во время выполнения. Поэтому компилятор генерирует код, который связывает `ViewAcct()` с `Brass::ViewAcct()` или `BrassPlus::ViewAcct()` в зависимости от типа объекта во время выполнения программы. Короче говоря, компилятор использует динамическое связывание для виртуальных методов.

В большинстве случаев динамическое связывание играет хорошую роль, поскольку оно позволяет программе выбирать метод, предназначенный для конкретного типа. Зная этот факт, вы можете поинтересоваться следующим:

- Когда применяются два типа связывания?
- Если динамическое связывание такое удобное, то почему оно не используется по умолчанию?
- Как работает динамическое связывание?

Давайте изучим ответы на эти вопросы.

Зачем существуют два типа связывания, и почему статическое связывание используется по умолчанию

Если динамическое связывание позволяет переопределять методы класса, а статическое — только частично, то зачем вообще нужно иметь статическое связывание? На то имеются две причины: эффективность и концептуальная модель.

Во-первых, давайте коснемся вопроса эффективности. Для того чтобы программа могла принимать решения во время выполнения, она должна обладать некоторым способом для отслеживания того, к какому типу объекта обращается указатель или ссылка базового класса. Это влечет за собой дополнительные накладные расходы на обработку. (Вы увидите один метод динамического связывания позже.) Если, например, вы разрабатываете класс, который не будет использоваться в качестве базового для наследования, то вам не нужно динамическое связывание. Также оно вам не понадобится, если вы используете производный класс (вроде `RatedPlayer`), который не переопределяет методы. В таких случаях имеет смысл применять статическое связывание, тем самым немного увеличивая эффективность. Тот факт, что статическое связывание более рационально, является причиной того, что оно выбирается в C++ по умолчанию. Страуструп указывает, что один из руководящих принципов C++ заключается в том, что вы не должны платить (в смысле использования памяти или времени обработки) за те свойства, которые вы не используете. Поэтому к виртуальным функциям стоит обращаться только тогда, когда дизайн программы требует этого.

Во-вторых, рассмотрим концептуальную модель. Когда вы разрабатываете класс, появляются методы, которые нежелательно переопределять в производных классах. Например, функция `Brass::Balance()`, которая возвращает баланс счета — одна из подобных функций. Объявив эту функцию невиртуальной, вы сделаете две вещи. Во-первых, вы сделаете ее более эффективной. Во-вторых, вы даете понять, что эта функция не должна переопределяться. Рекомендуется оставлять виртуальными только те методы, которые предположительно будут переопределяться.

Совет

Если метод базового класса будет переопределяться в производном классе, то его необходимо сделать виртуальным. Если метод не будет переопределяться, то он должен быть невиртуальным.

Конечно, во время разработки класса не всегда точно известно, в какую категорию попадает метод. Подобно многим процессам реальной жизни конструирование классов не является линейным процессом.

Как работают виртуальные функции

Язык C++ определяет, как должны вести себя виртуальные функции, но реализация этого механизма оставляется разработчику компилятора. Вам не нужно знать метод реализации для того, чтобы использовать виртуальные функции, однако изучение принципа работы поможет лучше понять идею. Поэтому давайте рассмотрим его.

Обычно компиляторы управляют виртуальными функциями, добавляя скрытый член к каждому объекту. Этот член хранит указатель на массив адресов функций. Такой массив, как правило, называется *таблицей виртуальных функций*. Таблица виртуальных функций хранит адреса виртуальных функций, объявленных для объектов данного класса. Например, объект базового класса содержит указатель на таблицу адресов всех виртуальных функций для этого класса. Объект производного класса со-

держит указатель на отдельную таблицу адресов. Если производный класс дает новое определение виртуальной функции, то в таблице виртуальных функций сохраняется адрес новой функции. Если же производный класс не переопределяет виртуальную функцию, таблица виртуальных функций хранит адрес исходной версии функции. Если производный класс определяет новую функцию и объявляет ее виртуальной, ее адрес добавляется в таблицу виртуальных функций (рис. 13.5). Обратите внимание на то, что независимо от количества виртуальных функций для класса в объект добавляется только один член-адрес. Различным будет только размер самой таблицы.

При вызове виртуальной функции программа просматривает адрес таблицы виртуальных функций, хранящийся в объекте, и переходит к соответствующей таблице адресов функций. Если применяется первая виртуальная функция, определенная в объявлении класса, программа берет первый адрес в массиве и выполняет функцию с этим адресом. Если вызывается третья виртуальная функция в объявлении класса, программа запускает функцию, адрес которой хранится в третьем элементе массива.

```
class Scientist{
{
 ...
 char name[40];
public:
 virtual void show_name();
 virtual void show_all();
 ...
};
class Physicist : public Scientist
{
 ...
 char field[40];
public;
 void show_all(); // переопределена
 virtual void show_field(); // новая
 ...
};
```

Адрес Scientist::show_name() Адрес Scientist::show_all()

4064	6400

Таблица виртуальных функций Scientist

Таблица адресов 2008

4064	6820	7280

Таблица виртуальных функций Physicist

2096

Адрес Scientist::show_name() (функция не переопределена)

Адрес Physicist::show_all() (функция переопределена)

Адрес Physicist:show_field() (новая функция)

...	Sophie Fant	2008
...	Имя	vptr

Объект Scientist со скрытым указателем vptr, который ссылается на таблицу виртуальных функций Scientist

...	Adam Crusher	2096	nuclear structure
...	Имя	vptr	Поле

Объект Physicist со скрытым указателем vptr, который ссылается на таблицу виртуальных функций Physicist

```
Physicist adam("Adam Crusher", "nuclear structure");
Scientist * psc = &adam;
psc->show_all();
```

1. Найти значение psc->vptr, которое равно 2096.
2. Перейти к таблице по адресу 2096.
3. Найти адрес второй функции в таблице, который равен 6820.
4. Перейти по этому адресу (6820) и выполнить найденную там функцию.

Рис. 13.5. Механизм работы виртуальных функций

Кратко можно сказать, что использование виртуальных функций влечет за собой следующие умеренные затраты памяти и скорости:

- Каждый объект имеет свой размер, который увеличивается на значение, необходимое для хранения адреса.
- Для каждого класса компилятор создает таблицу (массив) адресов виртуальных функций.
- При каждом вызове функции выполняется дополнительный шаг для нахождения адреса в таблице.

Помните, что хотя невиртуальные функции более эффективны, чем виртуальные, они не предусматривают динамического связывания.

Что следует знать о виртуальных методах

Мы уже обсудили главные вопросы, связанные с виртуальными методами:

- Если в базовом классе начать объявление метода класса с ключевого слова `virtual`, то функция становится виртуальной для базового класса и для всех классов, производных от данного, включая вложенные производные классы.
- Если виртуальный метод вызывается через ссылку или указатель на объект, то программа использует метод, определенный для типа объекта, а не для типа указателя или ссылки. Этот процесс называется динамическим, или поздним, связыванием. Подобное поведение очень важно, поскольку указатель или ссылка базового класса всегда имеют право обращаться к объекту производного типа.
- Если вы определяете класс, который будет использоваться в качестве базового для наследования, то вы должны объявить виртуальными те методы класса, которые могут быть переопределены в производных классах.

Существует также еще несколько моментов, которые необходимо знать о виртуальных методах. Некоторые из них уже были бегло раскрыты. Давайте рассмотрим их подробнее.

Конструкторы

Конструкторы не могут быть виртуальными. Создание объекта производного класса активизирует конструктор производного, а не базового класса. Конструктор производного класса затем использует конструктор базового класса, однако, эта последовательность явно вытекает из механизма наследования. Таким образом, производный класс не наследует конструкторы базового класса, поэтому нет смысла делать их виртуальными.

Деструкторы

Деструкторы должны быть виртуальными, за исключением тех классов, которые не будут использоваться в качестве базовых. Например, предположим, что `Employee` — это базовый класс, а `Singer` — производный класс, добавляющий член `char *`, который указывает на память, выделенную операцией `new`. Затем, когда объект `Singer` завершает свою работу, необходимо вызвать деструктор `~Singer()` для того, чтобы освободить эту память.

Теперь посмотрите на следующий код:

```
Employee * pe = new Singer;  // допустимо, так как Employee является
                             // базовым классом для Singer
...
delete pe;                   // ~Employee() или ~Singer()?
```

Если применяется стандартное статическое связывание, то операция delete активизирует деструктор ~Employee(). При этом освобождается память, на которую ссылаются компоненты Employee объекта Singer, но не память, на которую указывают новые члены класса. При этом если деструкторы виртуальные, то тот же самый код активизирует деструктор ~Singer(), освобождающий память, на которую указывает компонент Singer, а затем вызывает деструктор ~Employee() для освобождения памяти, на которую указывает компонент Employee.

Обратите внимание, что даже если базовый класс не требует явного деструктора, вам не стоит надеяться на стандартный конструктор. Вместо этого вы должны предусмотреть виртуальный деструктор, даже если он ничего не будет делать:

```
virtual ~BaseClass() { }
```

Кстати, не будет ошибкой ввод виртуального деструктора для класса даже в том случае, если вы не планируете сделать его базовым. Это нужно из соображений эффективности.

Совет

Как правило, вы должны обеспечить в базовом классе виртуальный деструктор, даже если класс в нем не нуждается.

Друзья

Друзья не могут быть виртуальными функциями, поскольку они не являются членами класса, а виртуальными функциями могут быть только члены. Если это вызывает проблему при разработке, то вы можете обойти ее за счет использования виртуальных методов внутри дружественных функций.

Отсутствие переопределения

Если в производном классе не происходит переопределение функции (виртуальной или нет), то класс будет использовать версию функции базового класса. Если класс является частью длинной цепочки порождений, то будет использоваться самая последняя версия функции. Исключение составляет случай, когда базовая версия скрыта, как описано ниже.

Переопределение скрывает методы

Предположим, что вы создали нечто наподобие:

```
class Dwelling
{
public:
   virtual void showperks(int a) const;
...
};
```

```
class Hovel : public Dwelling
{
{
public:
   virtual void showperks() const;
...
};
```

Это вызывает проблему. Вы можете получить предупреждение компилятора вроде следующего:

```
Warning: Hovel::showperks(void) hides Dwelling::showperks(int)
Warning: Hovel::showperks(void) скрывает Dwelling::showperks(int)
```

Возможно, вы и не получите предупреждение. В любом случае, код дает следующие результаты:

```
Hovel trump;
trump.showperks();  // верно
trump.showperks(5); // неверно
```

Новое определение создает функцию showperks(), которая не принимает аргументов. Вместо того чтобы получить результат в виде двух перегруженных версий функции, это переопределение *скрывает* версию базового класса, в которой был аргумент int. Другими словами, переопределение унаследованных методов не является разновидностью перегрузки. Если вы переопределяете функцию в производном классе, то при этом происходит не просто перегрузка объявления базового класса с той же самой сигнатурой функции. Вместо этого скрываются *все* методы базового класса с тем же именем вне зависимости от сигнатур аргументов.

Этот факт приводит к нескольким практическим правилам. Во-первых, если вы переопределяете унаследованный метод, необходимо убедиться в точном совпадении с исходным прототипом. Одно сравнительно новое исключение из этого правила состоит в том, что возвращаемый тип (указатель или ссылка на базовый класс) может быть заменен указателем или ссылкой на производный класс. Это свойство называется *изменчивостью возвращаемого типа*, поскольку возвращаемый тип допускается изменять параллельно с типом класса:

```
class Dwelling
{
public:
// базовый метод
   virtual Dwelling & build(int n);
   ...
};
class Hovel : public Dwelling
{
public:
// производный метод с измененным возвращаемым типом
   virtual Hovel & build(int n); // та же самая сигнатура функции
   ...
};
```

Обратите внимание, что данное исключение относится только к возвращаемым значениям, но не к аргументам.

Во-вторых, если объявление класса перегружается, вам необходимо переопределить все версии базового класса в производном классе:

```
class Dwelling
{
public:
// три перегруженных функции showperks()
   virtual void showperks(int a) const;
   virtual void showperks(double x) const;
   virtual void showperks() const;
   ...
};
class Hovel : public Dwelling
{
public:
// три переопределенных функции showperks()
   virtual void showperks(int a) const;
   virtual void showperks(double x) const;
   virtual void showperks() const;
   ...
};
```

Если вы переопределяете только одну версию, то две остальных становятся скрытыми и не могут использоваться объектами производного класса. Обратите внимание, что если не нужны никакие изменения, то переопределение может просто вызывать версию базового класса.

Управление доступом: protected

До настоящего времени в примерах классов использовались ключевые слова `public` и `private` для управления доступом к членам класса. Имеется еще одна категория доступа, обозначаемая ключевым словом `protected` (защищенный). Ключевое слово `protected` подобно `private` в том смысле, что доступ к членам класса в разделе `protected` можно получить извне только за счет использования общедоступных членов класса. Различие между `private` и `protected` начинает действовать только внутри классов, производных от базового класса. Члены производного класса имеют прямой доступ к защищенным членам базового класса, но не имеют прямого доступа к приватным членам базового класса. Таким образом, члены из категории `protected` ведут себя как приватные члены для внешнего мира и как общедоступные члены для производных классов.

Например, предположим, что в классе `Brass` член `balance` объявлен как защищенный:

```
class Brass
{
protected:
   double balance;
...
};
```

В этом случае класс BrassPlus имеет прямой доступ к balance без применения методов Brass. Например, ядро функции BrassPlus::Withdraw() может быть реализовано так:

```
void BrassPlus::Withdraw(double amt)
{
if (amt < 0)
   cout << "Снимаемая сумма должна быть положительной; "
        << "попытка снятия отменена.\n";
else if (amt <= balance) // прямой доступ к balance
   balance -= amt;
else if ( amt <= balance + maxLoan - owesBank)
{
   double advance = amt - balance;
   owesBank += advance * (1.0 + rate);
   cout << "Банковский аванс: $" << advance << endl;
   cout << "Налог: $" << advance * rate << endl;
   Deposit(advance);
   balance -= amt;
}
else
   cout << "Предел кредита превышен. Транзакция отменена.\n";
}
```

Защищенные члены данных могут упростить код, однако в коде уже имеется один проектный изъян. Например, продолжая рассматривать пример BrassPlus, если бы член balance был защищенным, то код можно было бы записать так:

```
void BrassPlus::Reset(double amt)
{
   balance = amt;
}
```

Класс Brass был разработан таким образом, что интерфейс функций Deposit() и Withdraw() предусматривает только один способ для изменения balance. Однако метод Reset(), по сути, делает balance общедоступной переменной для объектов BrassPlus, игнорируя защитные меры, которые есть в функции Withdraw().

Внимание!

Вы должны отдавать предпочтение защищенному доступу перед приватным для данных-членов класса, при этом используя методы базового класса для предоставления производным классам доступа к данным базового класса.

Однако защищенный доступ может оказаться достаточно полезным для методов, предоставляя производным классам доступ к внутренним функциям, которые не являются общедоступными.

Пример из практики: одноэлементный проектный шаблон

Часто можно найти единую общую модель, которая решает множество проблем. Это верно для человеческих взаимоотношений, например, существует рецепт вроде "сделать глубокий вздох и досчитать до десяти, прежде чем ответить". В программировании при изучении проблем разработки программного обеспечения также появляются общие шаблоны. Проектный шаблон — это программный эквивалент определенного стиля изготовления, в котором один и тот же принцип можно применить к различным наборам ингредиентов. Вы можете применять проектный шаблон

для создания элегантного и последовательного решения для повторяющихся проблемных областей. Например, вы можете воспользоваться одноэлементным шаблоном, если необходимо, чтобы в вызывающую программу возвращался в точности один экземпляр класса. Ниже показано, как может быть объявлен такой класс:

```
class TheOnlyInstance
{
public:
    static TheOnlyInstance* GetTheOnlyInstance();
    // другие методы
protected:
    TheOnlyInstance() {}
private:
    // приватные данные
};
```

Объявив конструктор `TheOnlyInstance` как `protected` и не предусмотрев ни одного общедоступного конструктора, вы можете гарантировать, что не будет создано ни одного локального экземпляра:

```
int main()
{
    TheOnlyInstance noCanDo; // не допускается
```

Общедоступный статический метод `GetTheOnlyInstance` обслуживает класс в качестве единственной точки доступа в течение всего времени существования класса. При вызове он возвращает экземпляр класса `TheOnlyInstance`:

```
TheOnlyInstance* TheOnlyInstance::GetTheOnlyInstance()
{
    static TheOnlyInstance objTheOnlyInstance;
    return &objTheOnlyInstance;
}
```

Метод `GetTheOnlyInstance` просто создает статический экземпляр класса `TheOnlyInstance` при первом вызове. Статический объект, созданный таким способом, остается действующим до тех пор, пока программа не завершается, после чего он автоматически уничтожается. Для извлечения указателя на единственный экземпляр класса программа просто вызывает статический метод `GetTheOnlyInstance`, который возвращает адрес одноэлементного объекта:

```
TheOnlyInstance* pTheOnlyInstance = TheOnlyInstance::GetTheOnlyInstance();
```

Поскольку статическая переменная остается в памяти между двумя вызовами функции, то последующие вызовы `GetTheOnlyInstance` возвращают адрес того же самого статического объекта.

Абстрактные базовые классы

До сих пор мы имели дело с простым наследованием и более сложным полиморфным наследованием. Следующим шагом к увеличению сложности является абстрактный базовый класс (АБК). Давайте рассмотрим некоторые программные ситуации, которые предоставляют основу для концепции АБК.

Иногда использование отношения *is-a* является не настолько простым, как может показаться. Допустим, например, что вы разрабатываете графическую программу, в которой предполагается отображение среди прочих объектов окружностей и эллипсов. Окружность — это отдельный случай эллипса; это эллипс, у которого больший диаметр равен меньшему. Таким образом, все окружности являются эллипсами, и очень заманчивой идеей становится порождение класса `Circle` от класса `Ellipse`. Однако когда вы вникните в детали, то можете столкнуться с проблемами.

Для того чтобы их увидеть, сначала нужно обсудить, что будет включаться в класс Ellipse. Члены данных могут включать в себя координаты центра, большую полуось (половина большего диаметра), малую полуось (половина меньшего диаметра), координатный угол, который равен углу между горизонтальной координатной осью и большой полуосью. Также в класс могут входить методы для перемещения эллипса, для вычисления площади эллипса, для вращения эллипса, для масштабирования большой и малой полуосей:

```
class Ellipse
{
private:
   double x;     // координата x центра эллипса
   double y;     // координата y центра эллипса
   double a;     // большая полуось
   double b;     // малая полуось
   double angle; // координатный угол в градусах
   ...
public:
   ...
   void Move(int nx, ny) { x = nx; y = ny; }
   virtual double Area() const { return 3.14159 * a * b; }
   virtual void Rotate(double nang) { angle += nang; }
   virtual void Scale(double sa, double sb) { a *= sa; b *= sb; }
   ...
};
```

Теперь предположим, что вы порождаете класс Circle от класса Ellipse:

```
class Circle : public Ellipse
{
   ...
};
```

Хотя окружность и является эллипсом, данное порождение очень неудобное. Например, для окружности требуется только одно измерение (ее радиус), чтобы описать ее размер и форму, вместо большой полуоси (a) и малой полуоси (b). Конструкторы Circle могут присвоить одно и то же значение членам a и b, однако при этом вы получите избыточное представление одной и той же информации. Параметр angle и метод Rotate() фактически не имеют смысла для окружности, а метод Scale() может превратить окружность в не окружность путем разного масштабирования двух осей. Вы можете попытаться решить эти проблемы с помощью различных ухищрений, таких как размещение метода Rotate() в приватном разделе класса Circle (при этом Rotate() станет недоступным для окружности). Однако в целом гораздо легче определить класс Circle без применения наследования:

```
class Circle     // без наследования
{
private:
   double x;     // координата x центра окружности
   double y;     // координата y центра окружности
   double r;     // радиус
   ...
```

```
public:
   ...
   void Move(int nx, ny) { x = nx; y = ny; }
   double Area() const { return 3.14159 * r * r; }
   void Scale(double sr) { r *= sr; }
   ...
};
```

Теперь в классе присутствуют только те переменные, которые ему необходимы. Хотя это решение тоже оказывается слабым. Классы `Circle` и `Ellipse` имеют много общего, однако при их отдельном определении этот факт игнорируется.

Существует другое решение. Вы можете извлечь из классов `Ellipse` и `Circle` то, что для них является общим, и поместить эти свойства в АБК. Затем можно породить от АБК оба класса `Circle` и `Ellipse`. Далее, например, можно использовать массив указателей базового класса для управления набором объектов `Ellipse` и `Circle` — другими словами, можно применить принцип полиморфизма. В данном случае для двух классов общими являются координаты центра фигуры, метод `Move()`, который работает одинаково для двух классов, а также метод `Area()`,работающий по-разному. На самом деле метод `Area()` даже невозможно реализовать для АБК, так как в нем нет необходимых членов данных. В C++ имеется способ для представления нереализованной функции — *чистая виртуальная функция*. Чистая виртуальная функция в конце своего объявления содержит конструкцию `= 0`, как показано для метода `Area()`:

```
class BaseEllipse   // абстрактный базовый класс
{
private:
   double x;        // координата x центра
   double y;        // координата y центра
   ...
public:
   BaseEllipse(double x0 = 0, double y0 = 0) : x(x0),y(y0) {}
   virtual ~BaseEllipse() {}
   void Move(int nx, ny) { x = nx; y = ny; }
   virtual double Area() const = 0; // чистая виртуальная функция
   ...
}
```

Если объявление класса содержит чистую виртуальную функцию, то вы не сможете создать объект этого класса. Идея заключается в том, что классы с чистыми виртуальными функциями существуют только для использования в качестве базовых классов. Для того чтобы класс был настоящим АБК, он должен иметь, по крайней мере, одну чистую виртуальную функцию. Обычная виртуальная функция превращается в чистую благодаря конструкции `= 0` в прототипе. В случае метода `Area()` функция не имеет определения, но в C++ даже для чистой виртуальной функции допускается иметь определение. Например, все базовые методы подобны `Move()` в том смысле, что они могут быть определены для базового класса, однако вам все равно нужно сделать класс абстрактным. Тогда можно сделать виртуальным прототип:

```
void Move(int nx, ny) = 0;
```

Базовый класс при этом становится абстрактным. Но после этого вы все равно можете предусмотреть определение в файле реализации:

```
void BaseEllipse::Move(int nx, ny) { x = nx; y = ny; }
```

Короче говоря, конструкция = 0 в прототипе указывает на то, что класс является АБК, и в нем необязательно определять функцию.

Теперь можно породить классы Ellipse и Circle от класса BaseEllipse, добавляя члены, необходимые для завершения каждого класса. Один момент, на который следует обратить внимание, состоит в том, что класс Circle всегда представляет окружности, в то время как класс Ellipse представляет эллипсы, которые также могут быть окружностями. При этом окружность класса Ellipse можно масштабировать в не окружность, а окружность класса Circle должна оставаться окружностью.

Программа, использующая эти классы, будет иметь возможность создавать объекты Ellipse и Circle, но не BaseEllipse. Поскольку объекты Circle и Ellipse имеют один и тот же базовый класс, то набором таких объектов можно управлять с помощью массива указателей на BaseEllipse. Такие классы, как Circle и Ellipse, иногда называются *конкретными* классами для указания того, что вы можете создавать объекты данных типов.

Кратко можно сказать, что АБК описывает интерфейс, который использует, по крайней мере, одну чистую виртуальную функцию. Классы, порождаемые от АБК, применяют обычные виртуальные функции для реализации интерфейса со свойствами конкретного производного класса.

Использование концепции АБК

Вы, возможно, хотите увидеть полный пример АБК, поэтому давайте применим этот принцип к отображению счетов Brass и BrassPlus, начав с АБК AcctABC. Этот класс должен содержать все методы и данные-члены, которые являются общими для классов Brass и BrassPlus. Методы, работа которых отличается для классов BrassPlus и Brass, необходимо объявлять как виртуальные функции. По крайней мере, одна виртуальная функция должна быть чистой для того, чтобы сделать класс AcctABC абстрактным.

В листинге 13.11 представлен заголовочный файл, который объявляет класс AcctABC (АБК), а также классы Brass и BrassPlus (конкретные классы). Для облегчения доступа производного класса к данным базового класса в AcctABC предусмотрено несколько защищенных методов. Вспомните, что защищенные методы — это методы, которые может вызывать производный класс, однако не являющиеся частью общедоступного интерфейса для объектов производного класса. Класс AcctABC также предусматривает защищенный метод для управления форматированием, которое ранее выполнялось в нескольких методах. В классе AcctABC содержатся две чистые виртуальные функции, поэтому он, очевидно, является абстрактным классом.

Листинг 13.11. acctabc.h

```
// acctabc.h -- классы банковских счетов
#ifndef ACCTABC_H_
#define ACCTABC_H_
// Абстрактный базовый класс
class AcctABC
{
private:
   enum {MAX = 35};
   char fullName[MAX];
   long acctNum;
   double balance;
```

```
protected:
   const char * FullName() const {return fullName;}
   long AcctNum() const {return acctNum;}
   std::ios_base::fmtflags SetFormat() const;
public:
   AcctABC(const char *s = "Nullbody", long an = -1,
               double bal = 0.0);
   void Deposit(double amt) ;
   virtual void Withdraw(double amt) = 0;  // чистая виртуальная функция
   double Balance() const {return balance;};
   virtual void ViewAcct() const = 0;      // чистая виртуальная функция
   virtual ~AcctABC() {}
};
// Класс банковского счета Brass
class Brass :public AcctABC
{
public:
   Brass(const char *s = "Nullbody", long an = -1,
            double bal = 0.0) : AcctABC(s, an, bal) { }
   virtual void Withdraw(double amt);
   virtual void ViewAcct() const;
   virtual ~Brass() {}
};
// Класс банковского счета Brass Plus
class BrassPlus : public AcctABC
{
private:
   double maxLoan;
   double rate;
   double owesBank;
public:
   BrassPlus(const char *s = "Nullbody", long an = -1,
               double bal = 0.0, double ml = 500,
               double r = 0.10);
   BrassPlus(const Brass & ba, double ml = 500, double r = 0.1);
   virtual void ViewAcct()const;
   virtual void Withdraw(double amt);
   void ResetMax(double m) { maxLoan = m; }
   void ResetRate(double r) { rate = r; };
   void ResetOwes() { owesBank = 0; }
};
#endif
```

Следующим шагом является реализация методов, которые еще не имеют встроенных определений. Это осуществляется в листинге 13.12.

Листинг 13.12. acctabc.cpp

```
// acctabc.cpp -- методы класса банковских счетов
#include <iostream>
#include <cstring>
using std::cout;
using std::ios_base;
using std::endl;
```

```
#include "acctabc.h"

// Абстрактный базовый класс
AcctABC::AcctABC(const char *s, long an, double bal)
{
   std::strncpy(fullName, s, MAX - 1);
   fullName[MAX - 1] = '\0';
   acctNum = an;
   balance = bal;
}

void AcctABC::Deposit(double amt)
{
   if (amt < 0)
      cout << "Отрицательный вклад не допускается; "
           << "попытка вклада отменена.\n";
   else
      balance += amt;
}
void AcctABC::Withdraw(double amt)
{
   balance -= amt;
}

// защищенный метод
ios_base::fmtflags AcctABC::SetFormat() const
{
   // установка формата ###.##
    ios_base::fmtflags initialState =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    cout.setf(ios_base::showpoint);
    cout.precision(2);
    return initialState;
}

// методы класса Brass
void Brass::Withdraw(double amt)
{
   if (amt < 0)
      cout << "Снимаемая сумма должна быть положительной; "
           << "попытка снятия отменена.\n";
   else if (amt <= Balance())
      AcctABC::Withdraw(amt);
   else
      cout << "Снимаемая сумма $" << amt
           << " превышает ваш баланс.\n"
           << "Попытка снятия отменена.\n";
}
void Brass::ViewAcct() const
{

   ios_base::fmtflags initialState = SetFormat();
   cout << "Клиент Brass: " << FullName() << endl;
```

```
   cout << "Номер счета: " << AcctNum() << endl;
   cout << "Баланс: $" << Balance() << endl;
   cout.setf(initialState);
}

// методы класса BrassPlus
BrassPlus::BrassPlus(const char *s, long an, double bal,
        double ml, double r) : AcctABC(s, an, bal)
{
   maxLoan = ml;
   owesBank = 0.0;
   rate = r;
}
BrassPlus::BrassPlus(const Brass & ba, double ml, double r)
        : AcctABC(ba) // использует неявный конструктор копирования
{
   maxLoan = ml;
   owesBank = 0.0;
   rate = r;
}

void BrassPlus::ViewAcct() const
{
   ios_base::fmtflags initialState = SetFormat();
   cout << "Клиент BrassPlus: " << FullName() << endl;
   cout << "Номер счета: " << AcctNum() << endl;
   cout << "Баланс: $" << Balance() << endl;
   cout << "Максимальный заем: $" << maxLoan << endl;
   cout << "Долг банку: $" << owesBank << endl;
   cout << "Процент на заем: " << 100 * rate << "%\n";
   cout.setf(initialState);
}

void BrassPlus::Withdraw(double amt)
{
   ios_base::fmtflags initialState = SetFormat();

   double bal = Balance();
   if (amt <= bal)
      AcctABC::Withdraw(amt);
   else if ( amt <= bal + maxLoan - owesBank)
   {
     double advance = amt - bal;
     owesBank += advance * (1.0 + rate);
     cout << "Банковский аванс: $" << advance << endl;
     cout << "Налог: $" << advance * rate << endl;
     Deposit(advance);
     AcctABC::Withdraw(amt);
   }
   else
      cout << "Предел кредита превышен. Транзакция отменена.\n";
   cout.setf(initialState);
}
```

Защищенные методы `FullName()` и `AcctNum()` предоставляют доступ только для чтения к членам данных `fullName` и `acctNum`, а также делают возможной индивидуальную настройку функции `ViewAcct()` для каждого производного класса.

Новую реализацию счетов `Brass` и `BrassPlus` можно использовать таким же образом, как и старую, поскольку методы класса имеют те же имена и интерфейсы, что и ранее. Например, для того, чтобы преобразовать листинг 13.10 для применения новой реализации, необходимо предпринять следующие шаги (и преобразовать файл `usebrass2.cpp` в `usebrass3.cpp`):

- Связать `usebrass2.cpp` с `acctabc.cpp` вместо `brass.cpp`.
- Включить `acctabc.h` вместо `brass.h`.
- Заменить

  ```
  Brass * p_clients[CLIENTS];
  ```

  на

  ```
  AcctABC * p_clients[CLIENTS];
  ```

Философия АБК

Методология АБК является гораздо более систематическим, упорядоченным способом подхода к наследованию, чем более специализированный, ситуационный принцип, который использовался в примере с `RatedPlayer`. Прежде чем разрабатывать АБК, сначала нужно сконструировать модель программной проблемы, в которой определить, какие необходимы классы и как они зависят друг от друга. Один из взглядов на этот вопрос заключается в том, что если вы проектируете иерархию наследования классов, то только конкретные классы никогда не будут работать в качестве базовых классов. Такой подход приводит к производству более ясных конструкций при меньшей сложности.

Пример из практики: требование выполнения правил интерфейса с помощью АБК

Одно из мнений об АБК состоит в том, что они требуют обязательного выполнения правил интерфейса. АБК требует, чтобы его чистые виртуальные функции перегружались во всех конкретных производных классах — другими словами, он заставляет производный класс подчиняться правилам интерфейса, установленным в АБК. Эта модель является общей в компонентных программных парадигмах, в которых применение АБК позволяет разработчику компонентов создавать "контракт на интерфейс". В таком контракте гарантируется, что все компоненты, порожденные от АБК, поддерживают, по меньшей мере, общие функциональные возможности, установленные АБК.

Наследование и динамическое распределение памяти

Как наследование взаимодействует с динамическим распределением памяти (использованием операций `new` и `delete`)? Например, если базовый класс применяет динамическое распределение памяти и перегружает операцию присваивания и конструктор копирования, то как это отражается на реализации производного класса? Ответ зависит от природы производного класса. Если производный класс непосред-

ственно не использует динамическое распределение памяти, то вам не нужно предпринимать каких-либо специальных шагов. Если же использует, то нужно будет освоить несколько новых приемов. Давайте рассмотрим эти два случая подробнее.

Случай 1: производный класс не использует операцию new

Предположим, что вы начинаете работу со следующим базовым классом, который использует динамическое распределение памяти:

```
// Базовый класс, использующий динамическое распределение памяти
class baseDMA
{
private:
   char * label;
   int rating;

public:
   baseDMA(const char * l = "null", int r = 0);
   baseDMA(const baseDMA & rs);
   virtual ~baseDMA();
   baseDMA & operator=(const baseDMA & rs);
...
};
```

Объявление содержит специальные методы, которые требуются, когда конструкторы используют new: деструктор, конструктор копирования, перегруженную операцию присваивания.

Теперь предположим, что вы порождаете класс lackDMA от baseDMA, при этом lackDMA не применяет new либо имеет другие нестандартные конструкторские свойства, которые требуют особого обращения:

```
// производный класс, не использующий ДРП
class lacksDMA :public baseDMA
{
private:
   char color[40];
public:
...
};
```

Должны ли вы сейчас определять явный деструктор, конструктор копирования и операцию присваивания для класса lackDMA? Ответ отрицательный.

Во-первых, давайте проанализируем потребность в деструкторе. Если вы не определили ни одного деструктора, то компилятор генерирует деструктор по умолчанию, который ничего не делает. На самом деле деструктор по умолчанию для производного класса всегда что-то делает — он вызывает деструктор базового класса после выполнения собственного кода. Поскольку члены lackDMA, как мы предполагаем, не требуют никаких особых действий, то деструктор по умолчанию вполне подходит.

Далее давайте рассмотрим конструктор копирования. Как было показано в главе 12, конструктор копирования по умолчанию осуществляет почленное копирование, что не подходит при динамическом распределении памяти. Однако почленное копи-

рование подходит для нового члена lacksDMA. Оно сохраняет сущность унаследованного объекта baseDMA. При этом нужно знать, что почленное копирование использует форму копирования, которая определена для указанного типа данных. Итак, копирование long в long производится с помощью обычного присваивания. Однако копирование члена класса или унаследованного компонента класса осуществляется с использованием конструктора копирования для данного класса. Таким образом, конструктор копирования по умолчанию для класса lacksDMA использует явный конструктор копирования baseDMA для того, чтобы скопировать блок baseDMA объекта lacksDMA. Итак, конструктор копирования по умолчанию подходит для нового члена lacksDMA, а также для унаследованного объекта baseDMA.

По сути, та же самая ситуация имеет место для присваивания. Операция присваивания по умолчанию для класса автоматически применяет операцию присваивания базового класса для базового компонента. Таким образом, она также полностью устраивает.

Данные свойства унаследованных объектов также сохраняются для членов класса, которые сами являются объектами. Например, в главе 10 было описано, как можно реализовать класс Stock путем использования для представления названия компании объекта string вместо массива char. Стандартный класс string, подобно нашему примеру String, использует динамическое распределение памяти. Теперь вы уже знаете, почему это не вызывает проблем. Конструктор копирования по умолчанию Stock будет использовать конструктор копирования string для копирования члена company объекта. Операция присваивания по умолчанию Stock будет использовать операцию присваивания string для присваивания члена company объекту. Деструктор Stock (по умолчанию или любой другой) будет автоматически вызывать деструктор string.

Случай 2: производный класс не использует операцию new

Предположим, что производный класс использует операцию new:

```
// производный класс, использующий ДРП
class hasDMA :public baseDMA
{
private:
   char * style; // использовать new в конструкторах
public:
...
};
```

В этом случае, конечно, вам необходимо определить явный деструктор, конструктор копирования и операцию присваивания для производного класса. Давайте рассмотрим упомянутые методы по очереди.

Деструктор производного класса автоматически вызывает деструктор базового класса, поэтому он сам отвечает только за завершение работы после того, что сделали конструкторы производного класса. Другими словами, деструктор hasDMA должен освободить память, управляемую указателем style, после чего деструктор baseDMA освободит память, управляемую указателем label:

```
baseDMA::~baseDMA()    // заботится о baseDMA
{
   delete [] label;
}

hasDMA::~hasDMA()      // заботится о hasDMA
{
   delete [] style;
}
```

Далее проанализируем конструкторы копирования. Конструктор копирования baseDMA следует обычной модели:

```
baseDMA::baseDMA(const baseDMA & rs)
{
   label = new char[std::strlen(rs.label) + 1];
   std::strcpy(label, rs.label);
   rating = rs.rating;
}
```

Конструктор копирования hasDMA имеет доступ только к данным hasDMA, поэтому он должен вызвать конструктор копирования baseDMA для управления частью данных baseDMA:

```
hasDMA::hasDMA(const hasDMA & hs)
      : baseDMA(hs)
{
   style = new char[std::strlen(hs.style) + 1];
   std::strcpy(style, hs.style);
}
```

Необходимо обратить внимание на то, что список инициализаторов члена передает ссылку hasDMA конструкторе baseDMA. Не существует ни одного конструктора baseDMA с параметром ссылки типа hasDMA, но они и не требуются. Это связано с тем, что конструктор копирования baseDMA имеет параметр ссылки baseDMA, а ссылка базового класса может обращаться к производному типу. Таким образом, конструктор копирования baseDMA использует порцию baseDMA аргумента hasDMA для создания порции baseDMA нового объекта.

Наконец, давайте рассмотрим операции присваивания. Операция присваивания baseDMA следует обычной модели:

```
baseDMA & baseDMA::operator=(const baseDMA & rs)
{
   if (this == &rs)
      return *this;
   delete [] label;
   label = new char[std::strlen(rs.label) + 1];
   std::strcpy(label, rs.label);
   rating = rs.rating;
   return *this;
}
```

Поскольку hasDMA также применяет динамическое распределение памяти, он также нуждается в явной операции присваивания. Будучи методом hasDMA, он имеет прямой доступ только к данным hasDMA. Несмотря на это, явная операция при-

сваивания для производного класса также должна позаботиться о присваивании для унаследованного объекта baseDMA базового класса. Вы можете осуществить это путем явного вызова операции присваивания базового класса, как показано ниже:

```
hasDMA & hasDMA::operator=(const hasDMA & hs)
{
   if (this == &hs)
      return *this;
   baseDMA::operator=(hs); // копировать базовый блок
   style = new char[std::strlen(hs.style) + 1];
   std::strcpy(style, hs.style);
   return *this;
}
```

Оператор

```
baseDMA::operator=(hs); // копировать базовый блок
```

может показаться лишним. Однако применение нотации в виде функции вместо нотации в виде операции позволяет использовать операцию разрешения контекста. В сущности, приведенный выше оператор означает следующее:

```
*this = hs; // использовать baseDMA::operator=()
```

Однако, разумеется, компилятор игнорирует комментарии, поэтому, если вы примените последний код, то компилятор фактически обратиться к hasDMA::operator=() и тем самым создаст рекурсивный вызов. Использование нотации в виде функции приводит к вызову правильной операции присваивания.

Подведем итоги. Если и базовый, и производный классы используют динамическое распределение памяти, то деструктор, конструктор копирования и операция присваивания производного класса должны применять свои аналоги из базового класса для управления базовой компонентой. Это общее требование удовлетворяется тремя различными способами. Для деструктора оно выполняется автоматически. Для конструктора — с помощью вызова конструктора копирования базового класса в списке инициализаторов члена либо автоматической активизации стандартного конструктора. Для операции присваивания — за счет использования операции разрешения контекста в вызове операции присваивания базового класса.

Пример наследования с динамическим распределением памяти и друзьями

Для иллюстрации представленных рассуждений о наследовании и динамическом распределении памяти давайте объединим классы baseDMA, lacksDMA и hasDMA, которые только что обсуждались, в один пример. В листинге 13.13 показан заголовочный файл для данных классов. Ко всему тому, что мы уже обсуждали, он добавляет дружественную функцию, которая иллюстрирует, как производные классы могут получать доступ к друзьям базового класса.

Листинг 13.13. dma.h

```
// dma.h -- наследование и динамическое распределение памяти
#ifndef DMA_H_
#define DMA_H_
#include <iostream>
```

```
// Базовый класс, использующий динамическое распределение памяти
class baseDMA
{
private:
   char * label;
   int rating;

public:
   baseDMA(const char * l = "null", int r = 0);
   baseDMA(const baseDMA & rs);
   virtual ~baseDMA();
   baseDMA & operator=(const baseDMA & rs);
   friend std::ostream & operator<<(std::ostream & os,
                           const baseDMA & rs);
};

// производный класс без динамического распределения памяти
// не нуждается в деструкторе
// использует неявный конструктор копирования
// использует неявную операцию присваивания
class lacksDMA :public baseDMA
{
private:
   enum { COL_LEN = 40};
   char color[COL_LEN];
public:
   lacksDMA(const char * c = "blank", const char * l = "null",
            int r = 0);
   lacksDMA(const char * c, const baseDMA & rs);
   friend std::ostream & operator<<(std::ostream & os,
                           const lacksDMA & rs);
};

// производный класс с динамическим распределением памяти
class hasDMA :public baseDMA
{
private:
   char * style;
public:
   hasDMA(const char * s = "none", const char * l = "null",
            int r = 0);
   hasDMA(const char * s, const baseDMA & rs);
   hasDMA(const hasDMA & hs);
   ~hasDMA();
   hasDMA & operator=(const hasDMA & rs);
   friend std::ostream & operator<<(std::ostream & os,
                           const hasDMA & rs);
};

#endif
```

В листинге 13.14 представлены определения методов для классов baseDMA, lacksDMA и hasDMA.

Листинг 13.14. dma.cpp

```
// dma.cpp -- методы классов с динамическим распределением памяти

#include "dma.h"
#include <cstring>

// методы baseDMA
baseDMA::baseDMA(const char * l, int r)
{
   label = new char[std::strlen(l) + 1];
   std::strcpy(label, l);
   rating = r;
}

baseDMA::baseDMA(const baseDMA & rs)
{
   label = new char[std::strlen(rs.label) + 1];
   std::strcpy(label, rs.label);
   rating = rs.rating;
}

baseDMA::~baseDMA()
{
   delete [] label;
}

baseDMA & baseDMA::operator=(const baseDMA & rs)
{
   if (this == &rs)
      return *this;
   delete [] label;
   label = new char[std::strlen(rs.label) + 1];
   std::strcpy(label, rs.label);
   rating = rs.rating;
   return *this;
}

std::ostream & operator<<(std::ostream & os, const baseDMA & rs)
{
   os << "Название: " << rs.label << std::endl;
   os << "Рейтинг: " << rs.rating << std::endl;
   return os;
}

// методы lacksDMA
lacksDMA::lacksDMA(const char * c, const char * l, int r)
         : baseDMA(l, r)
{
   std::strncpy(color, c, 39);
   color[39] = '\0';
}
```

```
lacksDMA::lacksDMA(const char * c, const baseDMA & rs)
        : baseDMA(rs)
{
   std::strncpy(color, c, COL_LEN - 1);
   color[COL_LEN - 1] = '\0';
}
std::ostream & operator<<(std::ostream & os, const lacksDMA & ls)
{
   os << (const baseDMA &) ls;
   os << "Цвет: " << ls.color << std::endl;
   return os;
}

// методы hasDMA
hasDMA::hasDMA(const char * s, const char * l, int r)
        : baseDMA(l, r)
{
   style = new char[std::strlen(s) + 1];
   std::strcpy(style, s);
}
hasDMA::hasDMA(const char * s, const baseDMA & rs)
        : baseDMA(rs)
{
   style = new char[std::strlen(s) + 1];
   std::strcpy(style, s);
}
hasDMA::hasDMA(const hasDMA & hs)
        : baseDMA(hs) // вызвать конструктор копирования базового класса
{
   style = new char[std::strlen(hs.style) + 1];
   std::strcpy(style, hs.style);
}
hasDMA::~hasDMA()
{
   delete [] style;
}
hasDMA & hasDMA::operator=(const hasDMA & hs)
{
   if (this == &hs)
      return *this;
   baseDMA::operator=(hs); // копировать базовый блок
   style = new char[std::strlen(hs.style) + 1];
   std::strcpy(style, hs.style);
   return *this;
}

std::ostream & operator<<(std::ostream & os, const hasDMA & hs)
{
   os << (const baseDMA &) hs;
   os << "Стиль: " << hs.style << std::endl;
   return os;
}
```

Новая особенность, на которую нужно обратить внимание в листингах 13.13 и 13.14, заключается в том, как производные классы могут применять друзей базового класса. Проанализируйте, например, следующую дружественную функцию класса hasDMA:

```
friend std::ostream & operator<<(std::ostream & os,
                                 const hasDMA & rs);
```

Из-за того, что данная функция является дружественной для класса hasDMA, она получает доступ к члену style. Однако здесь возникает проблема: эта функция не является дружественной для класса baseDMA, поэтому каким образом она может получить доступ к членам label и rating? Решением может быть применение функции operator<<(), которая является дружественной для класса baseDMA. Следующая проблема связана с тем, что поскольку друзья не являются методами, вы не сможете использовать операцию разрешения контекста для указания того, какую функцию применять. Эту проблему можно решить, если употребить приведение типа таким образом, чтобы сопоставление прототипов выбирало соответствующую функцию. Таким образом, код приводит тип параметра const hasDMA & к типу аргумента const baseDMA &:

```
std::ostream & operator<<(std::ostream & os, const hasDMA & hs)
{
// привести тип для соответствия operator<<(ostream & , const baseDMA &)
    os << (const baseDMA &) hs;
    os << "Стиль: " << hs.style << endl;
    return os;
}
```

Код в листинге 13.15 позволяет протестировать классы baseDMA, lacksDMA и hasDMA.

Листинг 13.15. usedma.cpp

```
// usedma.cpp -- наследование, друзья и динамическое распределение памяти
// компилировать вместе с dma.cpp
#include <iostream>
#include "dma.h"
int main()
{
    using std::cout;
    using std::endl;

    baseDMA shirt("Portabelly", 8);
    lacksDMA balloon("red", "Blimpo", 4);
    hasDMA map("Mercator", "Buffalo Keys", 5);
    cout << shirt << endl;
    cout << balloon << endl;
    cout << map << endl;
    lacksDMA balloon2(balloon);
    hasDMA map2;
    map2 = map;
    cout << balloon2 << endl;
    cout << map2 << endl;
    return 0;
}
```

Ниже показан вывод программы, представленной в листингах 13.13, 13.14 и 13.15:

```
Название: Portabelly
Рейтинг: 8

Название: Blimpo
Рейтинг: 4
Цвет: red

Название: Buffalo Keys
Рейтинг: 5
Стиль: Mercator

Название: Blimpo
Рейтинг: 4
Цвет: red

Название: Buffalo Keys
Рейтинг: 5
Стиль: Mercator
```

Обзор проекта класса

Язык C++ можно применять к очень широкому кругу программных проблем, и вы не сможете свести разработку класса к некоторым регламентированным стандартным процедурам. Однако существует несколько общих направлений, которые часто используются, и сейчас самое подходящее время тщательно рассмотреть их, выполнив обзор и уточнение предыдущих рассуждений.

Функции-члены, которые генерирует компилятор

Как обсуждалось в главе 12, компилятор автоматически генерирует определенные общедоступные функции-члены. Тот факт, что компилятор это делает, говорит о том, что данные функции-члены особенно важны. Давайте еще раз рассмотрим некоторые из них.

Конструкторы по умолчанию

Стандартный конструктор, как правило, либо не имеет аргументов вообще, либо все его аргументы имеют значения по умолчанию. Если вы не определяете ни одного конструктора, то компилятор генерирует для вас конструктор по умолчанию. Его наличие позволяет создавать объекты. Например, предположим, что `Star` — это класс. Для применения следующего кода потребуется конструктор по умолчанию:

```
Star rigel;          // создать объект без явной инициализации
Star pleiades[6];    // создать массив объектов
```

Еще один важный момент, который осуществляет автоматический конструктор по умолчанию — это вызов конструкторов по умолчанию для всех базовых классов и для всех членов, которые являются объектами другого класса.

Если вы создали конструктор производного класса без явной активизации конструктора базового класса в списке инициализаторов членов, то компилятор также использует конструктор по умолчанию базового класса для конструирования порции базового класса в новом объекте. Если же в базовом классе нет конструктора по умолчанию, то в этой ситуации появится сообщение об ошибке компиляции.

Если вы определяете некоторый конструктор, то компилятор не генерирует конструктор по умолчанию. В таком случае вам нужно предусмотреть конструктор по умолчанию, если он по каким-то причинам необходим.

Обратите внимание на то, что одним из мотивов для введения конструкторов является возможность гарантировать, что объекты всегда будут инициализированы должным образом. К тому же, если в классе есть члены-указатели, то они обязательно будут инициализированы. Таким образом, рекомендуется вводить явный конструктор по умолчанию, который инициализирует все члены данных класса подходящими значениями.

Конструкторы копирования

Конструктор копирования для класса — это конструктор, который принимает объект класса в качестве аргумента. Как правило, объявленный параметр является константной ссылкой на тип класса. Например, конструктор копирования для класса `Star` может иметь прототип вроде:

```
Star(const Star &);
```

Конструктор копирования класса используется в следующих ситуациях:

- Когда в качестве первоначального значения нового объекта присваивается объект того же самого класса.
- Когда объект передается в функцию по значению.
- Когда функция возвращает объект по значению.
- Когда компилятор генерирует временный объект.

Если в программе не употребляется конструктор копирования (явно или неявно), то компилятор предоставляет прототип, однако не определение функции. В противном случае программа определяет конструктор копирования, который осуществляет почленную инициализацию. Другими словами, каждому члену нового объекта в качестве первоначального присваивается значение соответствующего члена исходного объекта.

В некоторых случаях использование почленнной инициализации нежелательно. Например, указатели членов, инициализированные с помощью `new`, как правило, требуют глубокого копирования, как в примере класса `baseDMA`. Либо класс может содержать статическую переменную, которую нужно изменить. В подобных ситуациях необходимо определить свой собственный конструктор копирования.

Операции присваивания

Операция присваивания по умолчанию управляет присваиванием одного объекта другому объекту того же самого класса. Не путайте присваивание с инициализацией. Если оператор создает новый объект, то это инициализация, а если оператор изменяет значение существующего объекта, то это присваивание:

```
Star sirius;
Star alpha = sirius;    // инициализация (одна нотация)
Star dogstar;
dogstar = sirius;       // присваивание
```

Если вам нужно явно определить конструктор копирования, то по ряду причин потребуется также явно определить операцию присваивания. Прототип для операции присваивания класса Star выглядит следующим образом:

```
Star & Star::operator=(const Star &);
```

Обратите внимание, что функция операции присваивания возвращает ссылку на объект Star. Класс baseDMA демонстрирует типичный пример явной операции присваивания.

Компилятор не генерирует операции присваивания для присваивания одного типа другому. Предположим, что вам нужна возможность присвоить строку объекту Star. Одним из способов является явное определение такой операции:

```
Star & Star::operator=(const char *) {...}
```

Второй подход заключается в использовании функции преобразования (смотри раздел “Анализ преобразования” далее в главе) для преобразования строки в объект Star и последующее использование функции присваивания объекта Star объекту Star. Первый метод работает быстрее, однако требует большего объема кода. Применение функции преобразования может привести к ситуациям, которые компилятор может не распознать.

Анализ других методов класса

Существует несколько важных моментов, о которых нужно помнить во время определения класса. В следующих разделах описаны некоторые из них.

Анализ конструкторов

Конструкторы отличаются от других методов класса тем, что они создают новые объекты, тогда как остальные методы вызываются существующими объектами. Это одна из причин, по которым конструкторы не наследуются. Наследование подразумевает, что производный объект может использовать метод базового класса, а в случае конструкторов объект даже не существует до тех пор, пока конструктор не сделает свою работу.

Анализ деструкторов

Нельзя забывать об определении явного деструктора, который очищает всю память, выделенную операцией new в конструкторах класса, и занимается уничтожением всего тог, что создал объект класса. Если класс будет использоваться в качестве базового, то необходимо предусмотреть виртуальный деструктор, даже если классу вообще не требуется деструктор.

Анализ преобразования

Любой конструктор, который может быть вызван с ровно одним аргументом, определяет преобразование типа аргумента в тип класса. Для примера посмотрите на следующие прототипы конструкторов для класса Star:

```
Star(const char *);                    // преобразует char * в Star
Star(const Spectral &, int members = 1); // преобразует Spectral в Star
```

Конструкторы преобразования используются, скажем, когда преобразуемый тип передается функции, которая была определена как принимающая класс в качестве аргумента. Например, рассмотрим следующее:

```
Star north;
north = "polaris";
```

Второй оператор активизирует функцию `Star::operator=(const Star &)`, используя `Star::Star(const char *)` для создания объекта `Star`, который будет применен в качестве аргумента для функции операции присваивания. При этом предполагается, что вы не определяли операцию присваивания `(char *)` для `Star`.

Включение выражения `explicit` в прототип для одноаргументного конструктора блокирует неявные преобразования, однако допускает явные:

```
class Star
{
...
public:
   explicit Star(const char *);
...
};
Star north;
north = "polaris";        // не допускается
north = Star("polaris");  // допускается
```

Для преобразования объекта класса в какой-то другой тип определяется функция преобразования (см. главу 11). Функция преобразования — это метод класса без аргументов или с объявленным возвращаемым типом; имя этой функции совпадает с типом, в который нужно преобразовать. Несмотря на отсутствие объявленного возвращаемого типа, функция должна возвращать требуемое преобразованное значение. Вот несколько примеров:

```
Star::Star double() {...}         // преобразует star в double
Star::Star const char * () {...}  // преобразует char в const
```

Вы должны проявлять благоразумие в отношении таких функций, применяя их только в тех случаях, когда они действительно полезны. Также в некоторых разработках классов наличие функций преобразования увеличивает вероятность написания неоднозначного кода. Например, предположим, что вы определяете преобразование `double` для типа `vector` из главы 11 с помощью следующего кода:

```
vector ius(6.0, 0.0);
vector lux = ius + 20.2; // не определено
```

Компилятор может преобразовать `ius` в `double` и применить сложение `double`, либо преобразовать `20.2` в `vector` (используя один из конструкторов) и применить сложение `vector`. На самом деле он ничего не сделает и проинформирует вас о неоднозначной конструкции.

Сравнение передач объекта по значению и по ссылке

В общем случае, если вы создаете функцию с аргументом-объектом, вы должны передавать объект по ссылке, а не по значению. Одной из причин для этого является эффективность. Передача объекта по значению влечет за собой создание временной копии, что подразумевает вызов конструктора копирования и последующего вызова деструктора. Вызов этих функций занимает время, причем копирование большого объекта может длиться гораздо дольше, чем передача по ссылке. Если функция не модифицирует объект, то вы должны объявить аргумент как ссылку `const`.

Еще одним основанием для передачи объектов по ссылке является то, что в случае наследования с применением виртуальных функций, функция, определенная как принимающая ссылку базового класса в качестве аргумента, может также успешно использоваться с производными классами, как упоминалось ранее в данной главе. (Этот вопрос также рассматривается в разделе "Виртуальные методы" далее в главе.)

Сравнение возврата объекта и возврата ссылки

Некоторые методы класса возвращают объекты. Вы, возможно, уже заметили, что некоторые члены возвращают непосредственно объекты, в то время как остальные возвращают ссылки. Иногда требуется, чтобы метод возвращал объект, однако если это не обязательно, то вместо объекта лучше использовать ссылку. Давайте рассмотрим это более подробно.

Во-первых, единственное различие при кодировании между возвратом непосредственно объекта и возвратом ссылки заключается в прототипе функции и заголовке:

```
Star nova1(const Star &);    // возвращает объект Star
Star & nova2(const Star &);  // возвращает ссылку на Star
```

Второй причиной, по которой возврат ссылки предпочтительнее возврата объекта, состоит в том, что возврат объекта влечет за собой создание временной копии возвращаемого объекта. Вызывающая программа получает доступ к этой копии. Таким образом, возврат объекта влечет за собой потерю времени на вызов конструктора копирования для создания копии и на вызов деструктора для уничтожения копии. Возврат ссылки экономит время и память. Возврат непосредственно объекта подобен передаче объекта по значению: оба процесса генерируют временные копии. Аналогично возврат ссылки похож на передачу объекта по ссылке: и вызывающая, и вызываемая функция оперируют одним и тем же объектом.

Однако не всегда имеется возможность возвращать ссылку. Функция не может возвращать ссылку на временный объект, созданный функцией, поскольку ссылка становится недействительной, когда функция завершает свою работу и объект исчезает. В этом случае код должен возвращать объект для создания копии, которая будет доступна вызывающей программе.

Если функция возвращает временный объект, созданный в функции, то вы не должны использовать ссылку. Например, следующий метод применяет конструктор для создания нового объекта, и затем возвращает копию данного объекта:

```
Vector Vector::operator+(const Vector & b) const
{
   return Vector(x + b.x, y + b.y);
}
```

Если функция возвращает объект, который был передан в нее через ссылку или указатель, то нужно возвращать объект по ссылке. Например, следующий код возвращает по ссылке либо объект, который активизирует функцию, либо объект, переданный в качестве аргумента:

```
const Stock & Stock::topval(const Stock & s) const
{
   if (s.total_val > total_val)
      return s; // объект аргумента
   else
      return *this; // вызывающий объект
}
```

Использование const

Вы должны быть внимательны с возможностями применения `const`. Вы можете использовать его для гарантии того, что метод не изменит аргумент:

```
Star::Star(const char * s) {...}  // не изменяет строку,
                                  // на которую указывает s
```

Вы можете также применять `const` для гарантии того, что метод не будет модифицировать объект, который его вызывает:

```
void Star::show() const {...} // не изменяет вызывающий объект
```

Здесь `const` означает `const Star * this`, где `this` указывает на вызывающий объект.

Как правило, функция, которая возвращает ссылку, может располагаться в левой части оператора присваивания, который на самом деле означает, что вы можете присвоить значение указываемому объекту. При этом вы можете применять `const` для обеспечения того, что ссылка или указатель не будут использоваться для модификации данных в объекте:

```
const Stock & Stock::topval(const Stock & s) const
{
   if (s.total_val > total_val)
      return s; // объект аргумента
   else
      return *this; // вызывающий объект
}
```

Здесь метод возвращает ссылку либо на `this`, либо на `s`. Поскольку и `this`, и `s` объявлены как `const`, то функция не может изменять их. При этом подразумевается, что возвращаемая ссылка также должна быть объявлена как `const`.

Обратите внимание, что если функция объявляет аргумент как ссылку или указатель на `const`, то она не сможет передавать этот аргумент в другую функцию до тех пор, пока данная функция не будет также гарантировать, что она не изменяет аргумент.

Анализ общедоступного наследования

Как и следовало ожидать, добавление наследования в программу увеличивает количество соглашений. Давайте рассмотрим некоторые из них.

Отношения Is-a

Вы должны руководствоваться отношением *is-a*. Если ваш предложенный производный класс не является отдельной разновидностью базового класса, то не стоит применять общедоступное наследование. Например, не нужно порождать класс `Programmer` от класса `Brain`. Если вы хотите отразить представление о том, что программист имеет мозг, то нужно использовать объект класса `Brain` в качестве члена класса `Programmer`.

В некоторых случаях самым лучшим подходом будет создание абстрактного класса данных с чистыми виртуальными функциями и последующее порождение из него остальных классов.

Помните, что одно из выражений отношения *is-a* проявляется в том, что указатель базового класса может указывать на объект производного класса, а ссылка базового класса может обращаться к объекту производного класса без явного приведения типа. Также помните о том, что обратное неверно. Другими словами, указатель или ссылка производного класса не могут ссылаться на объект базового класса без явного приведения типа. В зависимости от объявлений класса подобное явное приведение типа (нисходящее) может иметь либо не иметь смысла. (Проанализируйте рис. 13.4.)

Что не наследуется?

Конструкторы не наследуются. То есть создание производного объекта требует вызова конструктора производного класса. Однако, как правило, конструкторы производного класса используют синтаксис списка инициализаторов членов для вызова конструкторов базового класса с целью построения блока базового класса в производном объекте. Если конструктор производного класса не вызывает явно конструктор базового класса с помощью указанного синтаксиса, то он использует стандартный конструктор базового класса. В цепочке наследования каждый класс может применять список инициализаторов члена для передачи информации обратно своему ближайшему базовому классу.

Деструкторы не наследуются. При этом во время уничтожения объекта программа сначала вызывает деструктор производного класса, а затем деструктор базового класса. Если в базовом классе используется деструктор по умолчанию, то компилятор генерирует деструктор по умолчанию производного класса. В общем случае, если класс применяется в качестве базового, то его деструктор должен быть виртуальным.

Операции присваивания

Операции присваивания не наследуются. Причина очень проста. Унаследованный метод имеет ту же самую сигнатуру функции в производном классе, что и в базовом. Однако сигнатура операции присваивания изменяется от класса к классу, поскольку она содержит формальный параметр, который совпадает с типом класса. Операции присваивания обладают некоторыми интересными свойствами, которые будут рассмотрены далее.

Если компилятор обнаруживает, что программа присваивает один объект другому объекту того же класса, то он автоматически снабжает этот класс операцией присваивания. Стандартная, или неявная, версия этой операции использует почленное копирование, присваивая каждому члену целевого объекта значение соответствующего члена исходного объекта. При этом если объект принадлежит производному классу,

то компилятор применяет операцию присваивания базового класса для управления присваиванием порции базового класса в объекте производного класса. Если для базового класса предусмотрена явная операция присваивания, то используется она. Аналогично, если класс содержит член, являющийся объектом другого класса, то для этого члена применяется операция присваивания собственного класса.

Как уже упоминалось несколько раз, если конструкторы применяют new для инициализации указателей, то необходимо предусмотреть явную операцию присваивания. Поскольку в C++ для базового блока производных объектов используется операция присваивания базового класса, вам нет необходимости переопределять операцию присваивания для производного класса *за исключением* тех случаев, когда добавляются члены данных, которые требуют особой осторожности. Например, в классе baseDMA присваивание определяется явно, но производный класс lacksDMA использует неявную операцию присваивания, сгенерированную для данного класса.

Предположим, что производный класс использует new, и вам необходимо предоставить явный операцию присваивания. Операция должна предусматриваться для каждого члена класса, а не только для новых членов. Класс hasDMA демонстрирует, как это можно сделать:

```
hasDMA & hasDMA::operator=(const hasDMA & hs)
{
   if (this == &hs)
      return *this;
   baseDMA::operator=(hs); // копировать базовый блок
   style = new char[std::strlen(hs.style) + 1];
   std::strcpy(style, hs.style);
   return *this;
}
```

А что можно сказать о присваивании объекта производного класса объекту базового класса? (На заметку: это не то же самое, что инициализация ссылки базового класса на объект производного класса.) Давайте рассмотрим пример:

```
Brass blips; // базовый класс
BrassPlus snips("Rafe Plosh", 91191,3993.19, 600.0, 0.12); // производный
                                                           // класс
blips = snips; // присвоить производный объект базовому объекту
```

Чья операция присваивания используется? Помните о том, что оператор присваивания транслируется в метод, который вызывается объектом, расположенным в левой части:

```
blips.operator=(snips);
```

Здесь объектом, расположенным слева, является Brass, поэтому он активизирует функцию Brass::operator=(const Brass &). Отношение *is-a* позволяет ссылке Brass указывать на объект производного класса, такой как snips. Операция присваивания имеет дело только с членами базового класса, поэтому член maxLoan и остальные члены класса BrassPlus объекта snips в присваивании игнорируются. Говоря кратко, вы можете присвоить производный объект базовому, но в этот процесс вовлекаются только члены базового класса.

Что можно сказать об обратном? Можно ли присвоить объект базового класса объекту производного класса? Давайте рассмотрим пример:

```
Brass gp("Griff Hexbait", 21234, 1200); // базовый класс
BrassPlus temp;                          // производный класс
temp = gp;                               // возможно?
```

Здесь оператор присваивания транслируется следующим образом:

```
temp.operator=(gp);
```

Слева располагается объект `BrassPlus`, он активизирует функцию `BrassPlus::operator=(const BrassPlus &)`. При этом ссылка производного класса не может автоматически указывать на объект базового класса, поэтому данный код *не будет* работать до тех пор, пока в нем не появится *также* конструктор преобразования:

```
BrassPlus(const Brass &);
```

Возможно, как и в случае для класса `BrassPlus`, что конструктор преобразования — это конструктор с базовыми аргументами и дополнительными аргументами, при условии, что дополнительные аргументы имеют значения по умолчанию:

```
BrassPlus(const Brass & ba, double ml = 500, double r = 0.1);
```

Если предусмотрен конструктор преобразования, то программа использует его для создания временного объекта `BrassPlus` из `gp`, который затем применяется в качестве аргумента операции присваивания. В качестве альтернативы вы можете определить операцию присваивания для присваивания базового класса производному классу:

```
BrassPlus & BrassPlus ::operator=(const Brass &) {...}
```

Здесь типы в точности соответствуют оператору присваивания, поэтому в преобразованиях типа нет необходимости. Говоря кратко, ответом на вопрос "Можно ли присваивать объект базового класса производному объекту?" является: "Возможно". Можно, если производный класс имеет конструктор, который определяет преобразование объекта базового класса в объект производного класса. Также можно, если производный класс определяет операцию присваивания базового объекта производному объекту. Если ни одно из двух данных условий не выполняется, то вы не сможете производить присваивание за исключением использования явного приведения типа.

Сравнение приватных и защищенных членов

Помните о том, что защищенные члены ведут себя как общедоступные члены для производного класса и как приватные члены для внешнего мира. Производный класс имеет прямой доступ к защищенным членам базового класса, однако доступ к приватным членам он может получить только через методы базового класса. Таким образом, обозначение членов базового класса приватными предполагает бо́льшую защиту, в то время как обозначение их защищенными упрощает кодирование и ускоряет доступ. Страуструп в одной из своих книг указывает, что лучше применять приватные члены данных, нежели защищенные, однако защищенные методы все равно очень полезны.

Виртуальные методы

Во время разработки базового класса вам необходимо решить, делать ли методы класса виртуальными. Если вам нужно переопределять метод в производном классе,

то в базовом классе его следует определять как виртуальный. При этом становится возможным позднее, или динамическое, связывание. Если метод не нужно переопределять, то не стоит делать его виртуальным. Это не помешает кому-либо переопределить метод, однако это продемонстрирует, что вы не хотите его переопределять.

Обратите внимание, что некорректный код может "перехитрить" динамическое связывание. Посмотрите, например, на две следующих функции:

```
void show(const Brass & rba)
{
   rba.ViewAcct();
   cout << endl;
}

void inadequate(Brass ba)
{
   ba.ViewAcct();
   cout << endl;
}
```

Первая функция передает объект по ссылке, а вторая — по значению.

Теперь предположим, что вы используете каждую из данных функций с аргументом производного класса:

```
BrassPlus buzz("Buzz Parsec", 00001111, 4300);
show(buzz);
inadequate(buzz);
```

В вызове функции `show()` аргумент `rba` является ссылкой на объект `buzz` класса `BrassPlus`, поэтому `rba.ViewAcct()` интерпретируется как версия `BrassPlus`, что и должно быть. Но в функции `inadequate()`, которая передает объект по значению, `ba` является объектом `Brass`, созданным конструктором `Brass(const Brass &)`. (Автоматическое восходящее преобразование позволяет аргументу конструктора ссылаться на объект `BrassPlus`.) Таким образом, в `inadequate()` `ba.ViewAcct()` является версией `Brass`, поэтому отображается только компонент `Brass` объекта `buzz`.

Деструкторы

Как отмечалось ранее, деструктор базового класса должен быть виртуальным. В результате, когда вы удаляете производный объект через указатель или ссылку базового класса на этот объект, программа использует деструктор производного класса, а затем деструктор базового класса.

Друзья

Поскольку дружественная функция фактически не является членом класса, то она не наследуется. Однако вам может понадобиться, чтобы друг производного класса использовал друга базового класса. Способ осуществления этого предполагает приведение типа ссылки или указателя производного класса к эквиваленту базового класса и последующее применение нового указателя или ссылки для обращения к другу базового класса:

```
ostream & operator<<(ostream & os, const hasDMA & hs)
{
// приведение типа для соответствия operator<<(ostream & , const baseDMA &)
   os << (const baseDMA &) hs;
   os << "Style: " << hs.style << endl;
   return os;
}
```

Вы можете также использовать для приведения типа операцию `dynamic_cast<>`, которая рассматривается в главе 15:

```
os << dynamic_cast<const baseDMA &> (hs);
```

По причинам, которые обсуждаются в главе 15, эта форма приведения типа является наиболее предпочтительной.

Наблюдения за использованием методов базового класса

Общедоступные производные объекты могут использовать методы базового класса многими способами:

- Производный объект автоматически использует унаследованные методы базового класса, если в производном классе методы не переопределялись.
- Деструктор производного класса автоматически вызывает конструктор базового класса.
- Конструктор производного класса автоматически вызывает стандартный конструктор базового класса, если в списке инициализаторов членов не указан другой конструктор.
- Конструктор производного класса явно вызывает конструктор базового класса, указанный в списке инициализаторов членов.
- Методы производного класса могут применять операцию разрешения контекста для вызова общедоступных и защищенных методов базового класса.
- Друзья производного класса могут приводить тип ссылки или указателя производного класса к ссылке или указателю базового класса и затем использовать данную ссылку или указатель для вызова друга базового класса.

Сводка функций классов

Функции классов C++ представлены многими вариациями. Некоторые могут наследоваться, некоторые нет. Некоторые функции могут быть как функциями-членами, так и дружественными, остальные могут быть только методами. В табл. 13.1 подводятся итоги по указанным свойствам. В ней конструкция op= обозначает операции присваивания в форме +=, *= и так далее. Обратите внимание на то, что свойства для операций op= не отличаются от свойств категории "Другие операции". Причиной отдельного перечисления op= является стремление подчеркнуть, что эти операции ведут себя не так, как операция =.

Таблица 13.1. Свойства функций-членов

Функция	Наследование	Член или друг	Генерируется по умолчанию	Может быть виртуальной	Может иметь возвращаемый тип
Конструктор	Нет	Член	Да	Нет	Нет
Деструктор	Нет	Член	Да	Да	Нет
`=`	Нет	Член	Да	Да	Да
`&`	Да	Оба	Да	Да	Да
Преобразование	Да	Член	Нет	Да	Нет
`()`	Да	Член	Нет	Да	Да
`[]`	Да	Член	Нет	Да	Да
`->`	Да	Член	Нет	Да	Да
`op=`	Да	Оба	Нет	Да	Да
`new`	Да	Статический член	Нет	Нет	`void *`
`delete`	Да	Статический член	Нет	Нет	`void`
Другие операции	Да	Оба	Нет	Да	Да
Другие члены	Да	Член	Нет	Да	Да
Друзья	Нет	Друг	Нет	Да	Да

Резюме

Наследование предоставляет возможность адаптировать программный код под ваши конкретные потребности за счет определения нового класса (производного) из существующего (базового). Общедоступное наследование моделирует отношение *is-a*, которое означает, что объект производного класса должен быть также объектом базового класса. Как часть модели *is-a*, производный класс наследует члены данных и большинство методов базового класса. Однако производный класс не наследует конструкторы, деструкторы и операции присваивания базового класса. Производный класс имеет прямой доступ к общедоступным и защищенным членам базового класса, а также доступ к приватным членам базового класса через общедоступные и защищенные методы базового класса. Затем вы можете добавлять в класс новые члены данных и методы, а также использовать производный класс в качестве базового для дальнейших разработок. Каждый производный класс требует собственных конструкторов. Когда программа создает объект производного класса, она сначала вызывает конструктор базового класса, а затем конструктор производного класса. Когда программа удаляет объект, она сначала вызывает деструктор производного класса, а затем деструктор базового класса.

Если предполагается использовать класс в качестве базового, вы можете отдать предпочтение защищенным членам взамен приватных, чтобы производные классы имели прямой доступ к данным членам. Однако использование приватных членов,

в общем случае, уменьшает количество программных ошибок. Если вы планируете переопределение в производном классе метода базового класса, его необходимо сделать виртуальной функцией, объявив его с ключевым словом virtual. Это позволяет управлять объектами, к которым обращаются указатели или ссылки, на основе типа объекта, а не на основе типа ссылки или указателя. В частности, деструктор для базового класса должен быть виртуальным.

Возможно, вы захотите определить абстрактный базовый класс (АБК), который определяет интерфейс без подробного рассмотрения вопросов реализации. Например, вы можете определить абстрактный класс Shape, а от него порождать отдельные классы фигур, такие как Circle и Square. АБК должен включать в себя, по крайней мере, один чистый виртуальный метод. Объявить чистую виртуальную функцию можно, поместив конструкцию = 0 после закрывающей скобки в объявлении:

```
virtual double area() const = 0;
```

Вы не должны определять чистые виртуальные методы, вы также не сможете создать объект класса, который содержит чистые виртуальные члены. Чистые виртуальные функции служат только для определения общего интерфейса, который будет использоваться производными классами.

Вопросы для самоконтроля

1. Что производный класс наследует от базового класса?
2. Что производный класс не наследует от базового класса?
3. Предположим, что возвращаемый тип для функции baseDMA::operator=() был определен как void вместо baseDMA &. Какой результат последует из этого? Что произошло бы, если бы возвращаемый тип был baseDMA вместо baseDMA &?
4. В каком порядке вызываются конструкторы и деструкторы класса во время создания и удаления объекта?
5. Если производный класс не добавляет членов данных в базовый класс, то требуются ли для производного класса конструкторы?
6. Предположим, что базовый и производный классы определяют метод с одним и тем же именем. Производный класс вызывает метод. Чей метод будет фактически вызван?
7. В каких случаях производный класс должен определять операцию присваивания?
8. Можно ли присвоить адрес объекта производного класса указателю на базовый класс? Можно ли присвоить адрес объекта базового класса указателю на производный класс?
9. Можно ли присвоить объект производного класса объекту базового класса? Можно ли присвоить объект базового класса объекту производного класса?
10. Предположим, что вы определяете функцию, которая принимает в качестве аргумента ссылку на объект базового класса. Почему данная функция может также использовать объект производного класса в качестве аргумента?
11. Предположим, что вы определяете функцию, которая принимает в качестве аргумента объект базового класса (другими словами, функция передает объект

базового класса по значению). Почему данная функция может также использовать объект производного класса в качестве аргумента?

12. Почему предпочтительнее передавать объекты по ссылке, чем по значению?

13. Предположим, что Corporation является базовым классом, PublicCorporation — производным классом. Также допустим, что каждый класс определяет метод head(), ph является указателем на тип Corporation, а также что ph присваивается адрес объекта PublicCorporation. Как интерпретируется ph->head(), если базовый класс определяет head() как:

    а. обычный невиртуальный метод,

    б. виртуальный метод.

14. Какие ошибки допущены (если они есть) в следующем коде?

```
class Kitchen
{
private:
   double kit_sq_ft;
public:
   Kitchen() {kit_sq_ft = 0.0; }
   virtual double area() const { return kit_sq_ft * kit_sq_ft; }
};
class House : public Kitchen
{
private:
   double all_sq_ft;
public:
   House() {all_sq_ft += kit_sq_ft;}
   double area(const char *s) const { cout << s; return all_sq_ft; }
};
```

Упражнения по программированию

1. Начните со следующего объявления класса:

```
// базовый класс
class Cd { // представляет компакт-диск
private:
   char performers[50];
   char label[20];
   int selections;  // количество сборников
   double playtime; // время воспроизведения в минутах
public:
   Cd(char * s1, char * s2, int n, double x);
   Cd(const Cd & d);
   Cd();
   ~Cd();
   void Report() const; // отображает все данные компакт-диска
   Cd & operator=(const Cd & d);
};
```

Породите класс Classic, добавляющий массив членов char, которые будут хранить строки, указывающие ведущее произведение на компакт-диске. Если не-

обходимо, чтобы все функции в базовом классе были виртуальными, модифицируйте объявление класса. Если объявленный метод не нужен, удалите его из объявления. Протестируйте результат с помощью следующей программы:

```
#include <iostream>
using namespace std;
#include "classic.h" // который будет содержать #include cd.h
void Bravo(const Cd & disk);
int main()
{
   Cd c1("Beatles", "Capitol", 14, 35.5);
   Classic c2 = Classic("Piano Sonata in B flat, Fantasia in C",
                 "Alfred Brendel", "Philips", 2, 57.17);
   Cd *pcd = &c1;

   cout << "Using object directly:\n";
   c1.Report(); // использовать метод Cd
   c2.Report(); // использовать метод Classic

   cout << "Using type cd * pointer to objects:\n";
   pcd->Report(); // использовать метод Cd для объекта cd
   pcd = &c2;
   pcd->Report(); // использовать метод Classic для объекта classic

   cout << "Calling a function with a Cd reference argument:\n";
   Bravo(c1);
   Bravo(c2);

   cout << "Testing assignment: ";
   Classic copy;
   copy = c2;
   copy.Report()

   return 0;
}

void Bravo(const Cd & disk)
{
   disk.Report();
}
```

2. Выполните упражнение 1, но воспользуйтесь динамическим распределением памяти вместо массивов фиксированного размера для различных строк, используемых двумя классами.
3. Пересмотрите иерархию классов `baseDMA-lacksDMA-hasDMA` таким образом, чтобы все три класса были порождены из АБК. Протестируйте результат с помощью программы, подобной той, которая приведена в листинге 13.10. Другими словами, она должна поддерживать массив указателей на АБК и позволять пользователю принимать во время работы программы решения о том, объекты какого типа создавать. Добавьте в определение класса виртуальные методы `View()` для управления отображением данных.

4. Благотворительный орден программистов (Benevolent Order of Programmers — BOP) поддерживает коллекцию бутылочного портвейна. Для ее описания администратор по портвейну BOP разработал класс `Port`, который описан ниже:

```
#include <iostream>
using namespace std;
class Port
{
private:
    char * brand;
    char style[20]; // то есть tawny, ruby, vintage
    int bottles;
public:
    Port(const char * br = "none", const char * st = "none", int b = 0);
    Port(const Port & p); // конструктор копирования
    virtual ~Port() { delete [] brand; }
    Port & operator=(const Port & p);
    Port & operator+=(int b); //добавляет b к bottles
    Port & operator-=(int b); //вычитает b из bottles, если это возможно
    int BottleCount() const { return bottles; }
    virtual void Show() const;
    friend ostream & operator<<(ostream & os, const Port & p);
};
```

Метод `Show()` представляет информацию в таком виде:

```
Brand: Gallo
Kind: tawny
Bottles: 20
```

Функция `operator<<()` представляет информацию в следующем формате (без символа новой строки в конце):

```
Gallo, tawny, 20
```

Администратор по портвейну завершил определения методов для класса `Port` и затем породил класс `VintagePort`, как показано ниже:

```
class VintagePort : public Port // необходимый стиль = "vintage"
{
private:
    char * nickname; // то есть "The Noble" или "Old Velvet" и так далее
    int year;        // год сбора
public:
    VintagePort();
    VintagePort(const char * br, int b, const char * nn, int y);
    VintagePort(const VintagePort & vp);
    ~VintagePort() { delete [] nickname; }
    VintagePort & operator=(const VintagePort & vp);
    void Show() const;
    friend ostream & operator<<(ostream & os, const VintagePort & vp);
};
```

Вам поручается завершить работу над `VintagePort`.

а. Первое задание заключается в том, что нужно заново создать определения методов `Port`, поскольку предыдущий администратор по портвейну был принесен в жертву.

б. Второе задание состоит в объяснении факта, почему одни методы переопределены, а остальные нет.

в. На третьем этапе нужно объяснить, почему `operator=()` и `operator<<()` не являются виртуальными.

г. Четвертое задание заключается в обеспечении определений для методов `VintagePort`.

ГЛАВА 14

Повторное использование кода в C++

В этой главе:

- Связь типа *has-a*
- Классы с членами-объектами (включение)
- Класс шаблона **valarray**
- Приватное и защищенное наследование
- Множественное наследование
- Виртуальные базовые классы
- Создание шаблонов классов
- Использование шаблонов классов
- Специализации шаблонов

Возможность повторного использования кода — основная цель языка C++. Общедоступное наследование является одним из механизмов достижения этой цели, хотя и не единственным. В этой главе исследуются другие механизмы. Одна из техник предусматривает использование членов класса, являющихся объектами другого класса. Это называется *включением* или *композицией* или *иерархическим представлением*. К другим техникам относится приватное или защищенное наследование. Включение, приватное и защищенное наследование обычно используются для создания отношений *has-a* – отношений, при которых вновь создаваемый класс содержит в себе объект другого, уже существующего класса. Например, класс `HomeTheater` может содержать объект `DvdPlayer`. Множественное наследование позволяет создавать классы, которые наследуются от двух или более базовых классов и обладают их совокупной функциональностью.

В главе 10 были рассмотрены шаблоны функций. В этой главе описываются шаблоны классов, являющиеся еще одной возможностью повторного использования кода. Шаблон класса позволяет определить общие свойства класса. Затем можно использовать этот шаблон для создания специфических классов, предназначенных для конкретных целей. Например, можно создать общий шаблон стека и потом использовать его для создания класса, являющегося стеком для значений типа `int`, и другого класса, являющегося стеком для значений типа `double`. Возможно даже создание класса, представляющего стек стеков.

Классы с членами-объектами

Рассмотрим классы, включающие в качестве членов объекты других классов. Некоторые классы вроде string или стандартных шаблонов классов C++, рассмотренные в главе 16, предоставляют хорошую основу для создания более развитых классов. Рассмотрим пример.

Кто такой студент? Тот, кто поступил в учебное заведение? Тот, кто занимается исследовательской работой? Беглец от трудностей реального мира? Тот, у кого есть имя и набор оценок? Ясно, что последнее определение — совершенно неадекватная характеристика студента, однако она упрощает его компьютерное представление. Создадим класс Student, основанный на этом определении.

Сведение представления студента к имени и набору оценок наводит на мысль использовать класс, содержащий два члена — один для представления имени и другой для набора оценок. Для имени можно использовать символьный массив, но он налагает ограничения на длину имени. Или же можно использовать указатель на char и динамическое распределение памяти. Но как показано в главе 12, это требует написания большого количества строк кода. Лучше воспользоваться объектом существующего класса, для которого кто-то уже проделал всю необходимую работу. Например, можно использовать объект класса String (см. главу 12) или стандартный класс C++ string. Проще выбрать класс string, так как библиотека C++ уже содержит весь необходимый код. (Для использования класса string потребуется включить в проект файл string1.cpp.)

Создание набора оценок можно выполнить аналогично. Использование массива фиксированной длины приведет к ограничению размера. Применение динамического распределения памяти вызовет увеличение размера исходного кода. Можно разработать собственный класс, используя динамическое распределение памяти для создания массива. Можно воспользоваться подходящим классом из стандартной библиотеки C++.

В третьем случае (собственный класс) трудность заключается в том, что необходимо разработать такой класс. Это не сложно, поскольку массив double имеет много общего с массивом char. Поэтому при создании класса массива из double можно использовать класс String. Собственно, это и делалось в предыдущих изданиях этой книги.

Еще проще, если бы библиотека содержала подходящий класс. И такой класс существует — это класс valarray.

Класс valarray: краткий обзор

Класс valarray поддерживается заголовочным файлом valaray. Исходя из имени этого класса, можно предположить, что он предназначен для работы с числовыми величинами (или с классами с аналогичными свойствами). Он поддерживает операции суммирования содержимого массива и поиск максимального и минимального значения в массиве. Поскольку класс valarray может работать с данными разного типа, он определен как шаблон класса. Далее в этой главе будет показано, как создавать шаблон класса. А пока просто достаточно знать, как его использовать.

При объявлении объекта, являющегося шаблоном, необходимо точно определить его тип. Так, за идентификатором valarray должны следовать угловые скобки, содержащие требуемый тип:

```
valarray<int> q_values;      // массив значений int
valarray<double> weights;    // массив значений double
```

Это новый синтаксис, который нужно запомнить. Он очень прост.

При использовании объектов valarray необходимо иметь представление о конструкторах и других методах класса. Вот несколько примеров, использующих конструкторы:

```
double gpa[5] = {3.1, 3.5, 3.8, 2.9, 3.3};
valarray<double> v1;         // массив элементов double, размерность 0
valarray<int> v2(8);         // массив из 8 элементов int
valarray<int> v3(10,8);      // массив из 8 элементов int,
                             // каждому из которых присвоено значение 10
valarray<double> v4(gpa, 4);// массив из 4 элементов, первые 4 элемента
                             // которого установлены в gpa
```

Из примера видно, что можно создавать пустой массив нулевой размерности, пустой массив заданной размерности, массив, в котором всем элементам присвоены одинаковые значения, и массив, инициализированный значениями перечислимого типа.

Ниже описаны некоторые методы класса Valarray:

- operator[]() обеспечивает доступ отдельным элементам.
- size() возвращает количество элементов.
- sum() возвращает сумму значений элементов.
- max() возвращает максимальный элемент.
- min() возвращает минимальный элемент.

Существует еще много методов этого класса, причем часть из них описана в главе 16. Для работы с рассматриваемым примером достаточно методов, представленных в этом разделе.

Проект класса Student

При проектировании класса Student планируется использовать объект string для представления имени и объект valarray<double> для представления набора оценок. Как это можно сделать? Возникает желание породить класс Student от этих двух классов. Это было бы примером множественного общедоступного наследования, но в данном случае это неприемлемо. Дело в том, что отношение класса Student с этими классами не соответствует модели *is-a*. Студент — это не имя. Студент — это не массив оценок. На самом деле здесь имеется отношение типа *has-a*. У студента есть имя и у студента есть набор оценок. Обычно для моделирования отношений *has-a* в C++ используется композиция или включение. В этом случае созданный класс содержит члены, являющиеся объектами других классов. Например, можно начать определение класса Student следующим образом:

```
class Student
{
private:
   string name; // используем объект string для имени
   valarray<double> scores;//используем объект valarray<double> для оценок
   ...
};
```

Обычно при создании класса данные-члены делают приватными. Это значит, что функции-члены класса `Student` могут использовать общедоступные интерфейсы `sting` и `valarray<double>` для доступа к объектам `name` и `scores`. Доступ извне к этим объектам невозможен. Единственная возможность получения доступа к `string` и `valarray<double>` извне заключается в использовании общедоступных интерфейсов класса `Student` (рис. 14.1). Объясняется это тем, что класс `Student`, включая реализацию своих членов-объектов, не наследует их интерфейсы. Например, объект `Student` для представления имени использует реализацию `string` вместо `char * name` или `char name[26]`. Однако объект `Student` не может использовать функцию `string operator+=()` для добавления символов.

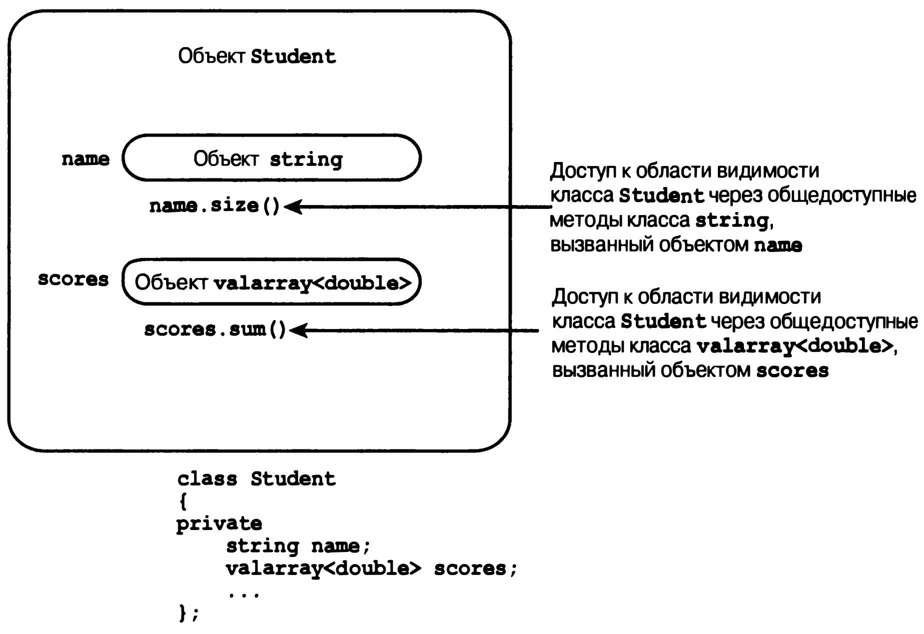


Рис. 14.1. Объекты внутри объектов: включение

Интерфейсы и реализации

При общедоступном наследовании класс наследует интерфейс и, возможно, реализацию. (Чистые виртуальные функции базового класса могут иметь интерфейс без его реализации.) Владение интерфейсом — признак отношения *is-a*. С другой стороны, при композиции класс получает реализацию без интерфейса. Не наследование интерфейса — признак отношения *has-a*.

То, что объект класса автоматически не наследует интерфейс включаемого объекта, полезно для отношения *has-a*. Например, string перегружает операцию + для конкатенации двух строк, но нет смысла выполнять конкатенацию двух объектов Student. Поэтому в данном случае не имеет смысла использовать общедоступное наследование. С другой стороны, часть интерфейса наследуемого класса может быть полезна для нового класса. Например, можно использовать метод operator<() из интерфейса объекта string для сортировки объектов Student по имени. Это можно сделать, определив функцию-член Student::operator<(), которая внутри себя использует функцию string::operator<(). Рассмотрим это подробнее.

Пример класса Student

Давайте, разработаем определение класса Student. Оно должно содержать конструкторы и, как минимум, несколько функций, реализующих интерфейс для класса Student. Это можно видеть в листинге 14.1. В нем определены конструкторы и несколько дружественных функций для ввода и вывода.

Листинг 14.1. studentc.h

```
// studentc.h -- определение класса Student с использованием включения
#ifndef STUDENTC_H_
#define STUDENTC_H_
#include <iostream>
#include <string>
#include <valarray>
class Student
{
private:
   typedef std::valarray<double> ArrayDb;
   std::string name;    // включаемый объект
   ArrayDb scores;      // включаемый объект
   // приватный метод для вывода scores
   std::ostream & arr_out(std::ostream & os) const;
public:
   Student() : name("Null Student"), scores() {}
   Student(const std::string & s)
      : name(s), scores() {}
   explicit Student(int n) : name("Nully"), scores(n) {}
   Student(const std::string & s, int n)
      : name(s), scores(n) {}
   Student(const std::string & s, const ArrayDb & a)
      : name(s), scores(a) {}
   Student(const char * str, const double * pd, int n)
      : name(str), scores(pd, n) {}
   ~Student() {}
   double Average() const;
   const std::string & Name() const;
   double & operator[](int i);
   double operator[](int i) const;
// друзья
   // ввод
```

```
   friend std::istream & operator>>(std::istream & is,
         Student & stu);  // 1 слово
   friend std::istream & getline(std::istream & is,
      Student & stu);     // 1 строка
   // вывод
   friend std::ostream & operator<<(std::ostream & os,
      const Student & stu);
};
#endif
```

Для простоты класс Student содержит следующее определение typedef:

```
typedef std::valarray<double> ArrayDb;
```

Это позволяет в остальном коде использовать более удобную нотацию ArrayDb вместо std::valarray<double>. В результате методы и друзья класса могут ссылаться на тип ArrayDb. Расположение этого typedef в защищенном разделе определения класса означает, что его можно использовать только внутри реализации класса Student, а не извне, внешними пользователями класса.

Обратите внимание на применение ключевого слова explicit:

```
explicit Student(int n) : name("Nully"), scores(n) {}
```

Вспомните, что конструктор, который можно вызвать с одним аргументом, работает как функция неявного преобразования типа аргумента в тип класса. Поскольку в этом примере первый аргумент представляет количество элементов в массиве, а не значение для массива, то выполняемое конструктором преобразование int в Student не имеет смысла. Использование ключевого слова explicit отключает неявное преобразование. Если опустить это ключевое слово, то станет возможным следующий код:

```
Student doh("Homer", 10);//сохраняет "Homer", создает массив из 10 элементов
doh = 5; // устанавливает имя в "Nully", очищает массив из 5 элементов
```

Предположим, невнимательный программист напечатает doh вместо doh[0]. Если в конструкторе опустить ключевое слово explicit, 5 будет преобразована во временный объект Student . Вызов конструктора Student(5) со значением параметра "Nully" присвоит значение члену name. Затем операция присваивания заменит оригинальный doh временным объектом. При использовании ключевого слова explicit компилятор выдаст ошибку во время выполнения операции присваивания.

Пример из практики: C++ и ограничения

Язык C++ содержит средства, позволяющие программисту накладывать определенные ограничения на программные конструкции: explicit — отключить неявное преобразование в конструкторах с одним аргументом, const — ограничить использование методов преобразования данных и так далее. Объяснение простое — ошибки времени компиляции лучше ошибок времени выполнения.

Инициализация включенных объектов

Обратите внимание, что для инициализации объектов-членов name и scores все конструкторы используют уже хорошо известный синтаксис списка инициализаторов членов. В некоторых случаях ранее в этой книге конструкторы используют его для инициализации членов, являющихся встроенными типами. Например:

```
Queue::Queue(int qs) : qsize(qs) {...} //инициализирует qsize значением qs
```

В этом коде в списке инициализаторов членов применяется имя члена (qsize). В предыдущих примерах конструкторы использовали список инициализаторов членов для инициализации порции базового класса производного объекта, например:

```
hasDMA::hasDMA(const hasDMA & hs) : baseDMA(hs) {...}
```

Для *унаследованных* объектов конструкторы используют имя класса, содержащееся в списке инициализаторов членов, чтобы вызвать конструктор конкретного базового класса. Для объектов-*членов* конструкторы используют имя *члена*. Посмотрите на последний конструктор в листинге 14.1:

```
Student(const char * str, const double * pd, int n)
   : name(str), scores(pd, n) {}
```

Поскольку этот конструктор инициализирует объекты-члены, а не унаследованные объекты, в списке инициализаторов он использует имя члена, а не имя класса. Каждый элемент в списке инициализаторов вызывает соответствующий конструктор. Так, name(str) вызывает конструктор string(const char *), scores(pd, n) — конструктор ArrayDb(const double *, int), который благодаря наличию typedef в действительности является конструктором valarray<double>(const double *, int).

Что произойдет, если не использовать список инициализаторов? В C++ все объекты-члены унаследованных компонентов должны быть созданы до того, как будут созданы все остальные объекты. Таким образом, если не использовать список инициализаторов, C++ использует конструктор по умолчанию, определенный для классов объектов-членов.

Порядок инициализации

При инициализации более одного объекта объекты инициализируются в том порядке, в котором они объявлены, а не в порядке, в котором они содержатся в списке инициализаторов. Предположим, что конструктор Student имеет следующий вид:

```
student(const char * str, const double * pd, int n)
     : scores(pd, n), name(str) {}
```

Член name будет инициализирован первым, поскольку он объявлен первым в определении класса. В данном случае точный порядок инициализации не важен. Однако он будет таковым, если в коде значение одного члена используется в выражении для инициализации другого члена.

Использование интерфейса для включенного объекта

Интерфейс для включенных объектов не является общедоступным (public), но его можно использовать внутри методов класса. Например, можно определить функцию, возвращающую среднее значение оценок студента:

```
double Student::Average() const
{
   if (scores.size() > 0)
      return scores.sum()/scores.size();
   else
      return 0;
}
```

Эта функция определяет метод, который может быть вызван объектом Student. Внутренне этот метод использует методы sum() и size(). Поскольку scores является объектом типа valarray, он может вызывать функции-члены класса valarray.

Аналогично можно определить дружественную функцию, которая будет использовать string-версию операции <<:

```
// использует string-версию операции <<
ostream & operator<<(ostream & os, const Student & stu)
{
   os << "Оценки для " << stu.name << ":\n";
   ...
}
```

Поскольку stu.name является объектом типа string, она вызывает функцию operator<<(ostream &, const string&), являющуюся частью реализации класса string. Обратите внимание, что функция operator<<(ostream & os, const Student & stu) должна быть дружественной классу Student, для того чтобы иметь доступ к члену name. (В качестве альтернативы функция может использовать общедоступный метод Name() вместо приватного члена name.)

Аналогично в функции можно применять valarray-реализацию операции << для вывода; к сожалению этого не сделано. Вот как в классе можно определить вспомогательный метод для решения этой задачи:

```
// приватный метод
ostream & Student::arr_out(ostream & os) const
{
   int i;
   int lim = scores.size();
   if (lim > 0)
   {
      for (i = 0; i < lim; i++)
      {
         os << scores[i] << " ";
         if (i % 5 == 4)
            os << endl;
      }
      if (i % 5 != 0)
         os << endl;
   }
```

```
      else
         os << " пустой массив ";
      return os;
   }
```

Использование такой вспомогательной функции собирает разбросанные фрагменты кода в одном месте и делает код дружественной функции более ясным:

```
   // использует string-версию операции operator<<()
   ostream & operator<<(ostream & os, const Student & stu)
   {
      os << "Оценки для " << stu.name << ":\n";
      stu.arr_out(os); // использует приватный метод для scores
      return os;
   }
```

Если есть желание, вспомогательную функцию можно также применить в качестве строительного блока для функций вывода пользовательского уровня.

В листинге 14.2 показаны методы класса `Student`. Он включает методы, которые позволяют использовать операцию `[]` для доступа к отдельным оценкам в объекте `Student`.

Листинг 14.2. studentc.cpp

```
// studentc.cpp -- класс Student, использующий включение
#include "studentc.h"
using std::ostream;
using std::endl;
using std::istream;
using std::string;
// общедоступные методы
double Student::Average() const
{
   if (scores.size() > 0)
      return scores.sum()/scores.size();
   else
      return 0;
}
const string & Student::Name() const
{
   return name;
}
double & Student::operator[](int i)
{
   return scores[i]; // использует valarray<double>::operator[]()
}
double Student::operator[](int i) const
{
   return scores[i];
}
// приватный метод
ostream & Student::arr_out(ostream & os) const
{
   int i;
```

```
   int lim = scores.size();
   if (lim > 0)
   {
      for (i = 0; i < lim; i++)
      {
         os << scores[i] << " ";
         if (i % 5 == 4)
            os << endl;
      }
      if (i % 5 != 0)
         os << endl;
   }
   else
      os << " пустой массив ";
   return os;
}
// друзья
// использует string-версию operator>>()
istream & operator>>(istream & is, Student & stu)
{
   is >> stu.name;
   return is;
}
// использует string-версию getline(ostream &, const string &)
istream & getline(istream & is, Student & stu)
{
   getline(is, stu.name);
   return is;
}
// использует string-версию operator<<()
ostream & operator<<(ostream & os, const Student & stu)
{
   os << "Оценки для " << stu.name << ":\n";
   stu.arr_out(os); // использует приватный метод для scores
   return os;
}
```

За исключением приватного вспомогательного метода, листинг 14.2 практически не требует написания нового кода. Техника включения позволяет использовать преимущества кодов, которые уже написаны вами или кем-либо еще.

Использование нового класса Student

Давайте создадим небольшую программу для тестирования класса `Student`. Для простоты она должна использовать массив из трех объектов `Student`, каждый из которых содержит 5 списков оценок. Она должна использовать простой цикл ввода без проверки вводимых значений и не позволять вам быстро закончить ввод. Тестовая программа показана в листинге 14.3. Компилировать ее нужно вместе с `studentc.cpp`.

Листинг 14.3. use_stuc.cpp

```
// use_stuc.cpp -- использует составной класс
// компилировать вместе с studentc.cpp
#include <iostream>
#include "studentc.h"
using std::cin;
using std::cout;
using std::endl;
void set(Student & sa, int n);
const int pupils = 3;
const int quizzes = 5;
int main()
{
   Student ada[pupils] =
      {Student(quizzes), Student(quizzes), Student(quizzes)};
   int i;
   for (i = 0; i < pupils; ++i)
      set(ada[i], quizzes);
   cout << "\nСписок студентов:\n";
   for (i = 0; i < pupils; ++i)
      cout << ada[i].Name() << endl;
   cout << "\nРезультаты:";
   for (i = 0; i < pupils; ++i)
   {
      cout << endl << ada[i];
      cout << "средняя: " << ada[i].Average() << endl;
   }
   cout << "Готово.\n";
   return 0;
}
void set(Student & sa, int n)
{
   cout << "Пожалуйста, введите имя и фамилию студента: ";
   getline(cin, sa);
   cout << "Пожалуйста, введите " << n << " оценок по тестам:\n";
   for (int i = 0; i < n; i++)
      cin >> sa[i];
   while (cin.get() != '\n')
      continue;
}
```

Замечание по совместимости

Если система корректно не поддерживает дружественную функцию `getline()`, выполнить программу не удастся. Можно изменить программу и воспользоваться `operator>>()` из листинга 14.2 вместо функции `getline()`. Поскольку эта дружественная функция читает только одно слово, нужно изменить подсказку для ввода только фамилии.

Ниже приведен пример вывода программы, представленной в листингах 14.1, 14.2 и 14.3:

```
Пожалуйста, введите имя и фамилию студента: Gil Bayts
Пожалуйста, введите 5 оценок по тестам:
92 94 96 93 95
```

```
Пожалуйста, введите имя и фамилию студента: Pat Roone
Пожалуйста, введите 5 оценок по тестам:
83 89 72 78 95
Пожалуйста, введите имя и фамилию студента: Fleur O'Day
Пожалуйста, введите 5 оценок по тестам:
92 89 96 74 64
Список студентов:
Gil Bayts
Pat Roone
Fleur O'Day
Результаты:
Оценки для Gil Bayts:
92 94 96 93 95
средняя: 94
Оценки для Pat Roone:
83 89 72 78 95
средняя: 83.4
Оценки для Fleur O'Day:
92 89 96 74 64
средняя: 83
Готово.
```

Приватное наследование

В C++ имеется и другое средство реализации отношений *has-a* — приватное (`private`) наследование. При использовании *приватного наследования* общедоступные (`public`) и защищенные (`protected`) члены базового класса становятся частью общедоступного интерфейса производного объекта. Они могут быть использованы внутри функций-членов производного класса.

Давайте внимательнее посмотрим на интерфейс. При общедоступном наследовании общедоступные методы базового класса становятся общедоступными методами производного класса. Короче говоря, производный класс наследует интерфейс базового класса. Это соответствует отношению *is-a*. При приватном наследовании общедоступные методы базового класса становятся приватными методами производного класса. То есть производный класс не наследует интерфейс базового класса. Для включаемых объектов, как мы уже выяснили, отсутствие наследования приводит к возникновению отношения *has-a*.

При приватном наследовании класс наследует реализацию. Например, если базовым классом для класса `Student` является класс `string`, класс `Student` будет иметь компонент унаследованного класса `string`, которую можно использовать для хранения строк. Кроме того, методы класса `Student` могут использовать методы класса `string` для внутреннего доступа к компоненту `string`. Включение добавляет к классу именованный объект-член, тогда как приватное наследование добавляет к классу неименованный унаследованный объект. В этой книге для обозначения объектов, добавленных путем наследования или включения, используется термин *субобъект*. Приватное наследование обеспечивает те же свойства, что и включение — наследует реализацию и не наследует интерфейс. Поэтому его так же можно использовать для создания отношения *has-a*. По сути, можно создать класс `Student`, использующий приватное наследование и имеющий тот же общедоступный интерфейс, что и у вер-

сии класса Student с технологией включения. Таким образом, различия между двумя подходами влияют на реализацию, а не на интерфейс. Посмотрим, как можно переделать класс Student, используя приватное наследование.

Пример класса Student (новая версия)

Для создания приватного интерфейса при определении класса нужно использовать ключевое слово private вместо public. (В действительности, private задано по умолчанию, поэтому отсутствие квалификатора доступа приведет к приватному наследованию.) Класс Student должен наследовать два класса, поэтому в объявлении класса Student они должны быть указаны оба:

```
class Student : private std::string, private std::valarray<double>
{
public:
   ...
};
```

Наследование от нескольких базовых классов называется *множественным наследованием.* В общем, множественное наследование и, в частности, общедоступное множественное наследование, может привести к проблемам, которые устраняются с помощью дополнительных синтаксических правил. Мы поговорим об этих случаях позже. В данном частном случае будем считать, что этих проблем не существует.

Заметим, что новый класс не требует приватных данных, поскольку оба наследуемых базовых класса содержат все необходимые данные-члены. Версия этого примера с использованием включения содержит два явно именованных объекта в качестве членов. Версия же с приватным наследованием содержит два неименованных субобъекта в виде унаследованных членов. Это одно из главных отличий между двумя рассматриваемыми подходами.

Инициализация компонентов базового класса

Наличие неявно унаследованных компонентов вместо объектов-членов влияет на исходный текст этого примера, поскольку для описания объекта больше нельзя использовать name и score. Вместо этого придется вернуться к технологии, применяемой при общедоступном наследовании. Конструктор, используемый при включении, имеет вид:

```
Student(const char * str, const double * pd, int n)
  : name(str), scores(pd, n) {} //использует имена объектов для включения
```

В новой версии примера для наследуемых классов должен быть использован список инициализаторов членов, в котором для указания конструктора вместо имени члена применяется имя класса:

```
Student(const char * str, const double * pd, int n)
  : std::string(str), ArrayDb(pd, n) {}  // использует имена классов
                                         // для наследования
```

В предыдущем примере ArrayDb является typedef для std::valarray<double>. Нужно иметь в виду, что список инициализации членов использует определение std::string(str) вместо name(str). Это второе главное отличие между двумя рассматриваемыми подходами.

В листинге 14.4. показано новое определение класса. Единственным отличием является отсутствие явно указанных имен объектов и использование имен классов вместо имен членов в конструкторах.

Листинг 14.4. `studenti.h`

```
// studenti.h -- определение класса Student,
// использующего приватное наследование
#ifndef STUDENTC_H_
#define STUDENTC_H_
#include <iostream>
#include <valarray>
#include <string>
class Student : private std::string, private std::valarray<double>
{
private:
   typedef std::valarray<double> ArrayDb;
   // приватный метод для вывода scores
   std::ostream & arr_out(std::ostream & os) const;
public:
   Student() : std::string("Null Student"), ArrayDb() {}
   Student(const std::string & s)
      : std::string(s), ArrayDb() {}
   Student(int n) : std::string("Nully"), ArrayDb(n) {}
   Student(const std::string & s, int n)
      : std::string(s), ArrayDb(n) {}
   Student(const std::string & s, const ArrayDb & a)
      : std::string(s), ArrayDb(a) {}
   Student(const char * str, const double * pd, int n)
      : std::string(str), ArrayDb(pd, n) {}
   ~Student() {}
   double Average() const;
   double & operator[](int i);
   double operator[](int i) const;
   const std::string & Name() const;
// друзья
   // ввод
   friend std::istream & operator>>(std::istream & is,
      Student & stu); // 1 слово
   friend std::istream & getline(std::istream & is,
      Student & stu); // 1 строка
   // вывод
   friend std::ostream & operator<<(std::ostream & os,
      const Student & stu);
};
#endif
```

Доступ к методам базового класса

Приватное (`private`) наследование делает возможным использование методов базового класса только внутри производного класса. Тем не менее, иногда необходимо иметь доступ к методам базового класса извне. Например, класс `Student` дает воз-

можность применять функцию `Average()`. В технологии включения это делается вызовом методов `valarray size()` и `sum()` внутри общедоступной (`public`) функции `Student::average()`, как показано на рис. 14.2. Методы вызываются внутри объекта:

```
double Student::Average() const
{
   if (scores.size() > 0)
      return scores.sum()/scores.size();
   else
      return 0;
}
```

Объект **Student**

Объект **string**

string::size()

Доступ к области видимости класса **Student** через общедоступные методы класса **string**, при помощи операции разрешения контекста

Объект **valarray<double>**

valarray<double>::sum()

Доступ к области видимости класса **Student** через общедоступные методы класса **valarray<double>**, при помощи операции разрешения контекста

```
class Student:private string,
private valarray<double>
{
 ...
};
```

Рис. 14.2. Объекты внутри объектов: приватное наследование

Здесь для вызова базовых классов наследование позволяет использовать имя класса и операцию разрешения контекста:

```
double Student::Average() const
{
   if (ArrayDb::size() > 0)
      return ArrayDb::sum()/ArrayDb::size();
   else
      return 0;
}
```

Короче говоря, в технологии включения для вызова методов применяются имена объектов, тогда как при общедоступном наследовании используются имя класса и операция разрешения контекста.

Доступ к объектам базового класса

Операция разрешения контекста обеспечивает доступ к методам базового класса. А что, если нужен доступ к собственно объекту базового класса? Например, в технологии включения класс Student реализует метод Name(), получая значение от члена name объекта string. Но при приватном наследовании объект string не имеет имени. Как же код класса Student может получить доступ к внутреннему объекту string?

Ответ — используя приведение типов. Поскольку тип Student порожден от string, можно привести тип объекта Student к типу string. Вспомним, что указатель this ссылается на вызываемый объект. Тогда *this является вызываемым объектом, в данном случае — объектом Student. Для того чтобы избежать вызова конструкторов и создания новых объектов, необходимо использовать приведение типа для создания ссылки:

```
const string & Student::Name() const
{
   return (const string &) *this;
}
```

Этот код возвращает ссылку на унаследованный объект string, содержащийся в вызываемом объекте Student.

Доступ к друзьям базового класса

Техника явного указания имени функции с именем ее класса не работает для дружественных функций, поскольку дружественная функция не принадлежит этому классу. Тем не менее, для корректного вызова функций можно использовать явное приведение типа к базовому классу. В основном это та же техника, что и при доступе к объекту базового класса в методах класса. Но в случае с друзьями доступно имя объекта Student, поэтому в коде используется имя объекта вместо *this. Например, рассмотрим следующее определение дружественной функции:

```
ostream & operator<<(ostream & os, const Student & stu)
{
   os << "Оценки для " << (const string &) stu << ":\n";
   ...
}
```

Если plato — это объект типа Student, то оператор

```
cout << plato;
```

вызывает эту функцию, где stu ссылается на plato, а os ссылается на cout. Внутри кода приведение типа в операторе

```
os << "Оценки для " << (const string &) stu << ":\n";
```

явно преобразует stu в ссылку на тип объекта string и это соответствует функции operator<<(ostream &, const string &). Ссылка stu не преобразуется автоматически в ссылку sting. Фундаментальная причина этого в том, что при приватном наследовании ссылка или указатель на базовый класс не может быть назначен ссылке или указателю на производный класс без явного приведения типа.

Однако даже при общедоступном наследовании нужно использовать явное приведение типов. Одна из причин состоит в том, что без приведения типа код вроде

```
os << stu;
```

соответствующий прототипу дружественной функции, приводит к рекурсивному вызову.

Другая причина связана с тем, что поскольку класс использует множественное наследование, компилятор не может определить, какой базовый класс преобразовывать, так как оба базовых класса поддерживают функцию `operator<<()`. В листинге 14.5 показаны все методы класса `Student`, за исключением приведенных в определении класса.

Листинг 14.5. `studenti.cpp`

```
// studenti.cpp -- класс Student, использующий приватное наследование
#include "studenti.h"
using std::ostream;
using std::endl;
using std::istream;
using std::string;
// общедоступные методы
double Student::Average() const
{
   if (ArrayDb::size() > 0)
      return ArrayDb::sum()/ArrayDb::size();
   else
      return 0;
}
const string & Student::Name() const
{
   return (const string &) *this;
}
double & Student::operator[](int i)
{
   return ArrayDb::operator[](i); // use ArrayDb::operator[]()
}
double Student::operator[](int i) const
{
   return ArrayDb::operator[](i);
}
// приватный метод
ostream & Student::arr_out(ostream & os) const
{
   int i;
   int lim = ArrayDb::size();
   if (lim > 0)
   {
      for (i = 0; i < lim; i++)
      {
         os << ArrayDb::operator[](i) << " ";
         if (i % 5 == 4)
            os << endl;
      }
```

```
      if (i % 5 != 0)
         os << endl;
   }
   else
      os << " пустой массив ";
   return os;
}
// друзья
// использует string-версию operator>>()
istream & operator>>(istream & is, Student & stu)
{
   is >> (string &)stu;
   return is;
}
// использует string-версию getline(ostream &, const string &)
istream & getline(istream & is, Student & stu)
{
   getline(is, (string &)stu);
   return is;
}
// использует string-версию operator<<()
ostream & operator<<(ostream & os, const Student & stu)
{
   os << "Оценки для " << (const string &) stu << ":\n";
   stu.arr_out(os); // использует приватный метод для scores
   return os;
}
```

Поскольку в примере повторно используется код string и valarray, к приватному вспомогательному методу нужно дописать совсем немного кода.

Использование пересмотренного класса Student

Настало время протестировать новый класс. Обратите внимание, что обе версии класса Student имеют совершенно одинаковые общедоступные интерфейсы, поэтому можно протестировать обе версии с помощью одной программы. Единственное отличие состоит в том, что нужно включить файл studenti.h вместо studentc.h и компоновать программу с файлом studenti.cpp вместо studentc.cpp. Программа показана в листинге 14.6. Не забудьте скомпилировать ее вместе с файлом studenti.cpp.

Листинг 14.6. use_stui.cpp

```
// use_stui.cpp -- использование класса с приватным наследованием
// компилировать с studenti.cpp
#include <iostream>
#include "studenti.h"
using std::cin;
using std::cout;
using std::endl;
void set(Student & sa, int n);
const int pupils = 3;
const int quizzes = 5;
```

```
int main()
{
   Student ada[pupils] =
      {Student(quizzes), Student(quizzes), Student(quizzes)};
   int i;
   for (i = 0; i < pupils; i++)
      set(ada[i], quizzes);
   cout << "\nСписок студентов:\n";
   for (i = 0; i < pupils; ++i)
      cout << ada[i].Name() << endl;
   cout << "\nРезультаты:";
   for (i = 0; i < pupils; i++)
   {
      cout << endl << ada[i];
      cout << "средняя: " << ada[i].Average() << endl;
   }
   cout << "Готово.\n";
   return 0;
}
void set(Student & sa, int n)
{
   cout << "Пожалуйста, введите имя и фамилию студента: ";
   getline(cin, sa);
   cout << "Пожалуйста, введите " << n << " оценок по тестам:\n";
   for (int i = 0; i < n; i++)
      cin >> sa[i];
   while (cin.get() != '\n')
      continue;
}
```

Замечание по совместимости

Если система корректно не поддерживает дружественную функцию `getline()`, выполнить программу не удастся. Можно изменить программу и использовать `operator>>()` из листинга 14.5 вместо функции `getline()`. Поскольку эта дружественная функция читает только одно слово нужно изменить подсказку для ввода только фамилии.

Это пример вывода программы из листинга 14.6

```
Пожалуйста, введите имя и фамилию студента: Gil Bayts
Пожалуйста, введите 5 оценок по тестам:
92 94 96 93 95
Пожалуйста, введите имя и фамилию студента: Pat Roone
Пожалуйста, введите 5 оценок по тестам:
83 89 72 78 95
Пожалуйста, введите имя и фамилию студента: Fleur O'Day
Пожалуйста, введите 5 оценок по тестам:
92 89 96 74 64

Список студентов:
Gil Bayts
Pat Roone
Fleur O'Day
```

```
Результаты:
Оценки для Gil Bayts:
92 94 96 93 95
средняя: 94
Оценки для Pat Roone:
83 89 72 78 95
средняя: 83.4
Оценки для Fleur O'Day:
92 89 96 74 64
средняя: 83
Готово.
```

Результирующий вывод программы такой же, как и у версии, использующей технологию включения.

Включение или приватное наследование?

Из приведенного следует, что моделирование отношения *has-a* возможно с помощью технологий включения или приватного наследования. Какой путь избрать? Большинство программистов на C++ применяют технологию включения. Во-первых, она проще в использовании. Если посмотреть на определение класса, то будут видны явно именованные объекты, представляющие содержащиеся классы, и можно ссылаться на эти объекты по именам. При наследовании же отношение является более абстрактным. Во-вторых, наследование может приводить к трудностям, особенно, если класс наследуется от нескольких базовых классов. Возможно, придется столкнуться с проблемой, когда разные базовые классы имеют методы с одинаковыми именами, или когда разные базовые классы имеют общего предка. В конце концов, при использовании технологии включения меньше возможностей столкнуться с проблемами. Включение также позволяет иметь несколько субобъектов с одинаковыми именами. Если класс должен иметь три объекта `string`, можно определить три отдельных члена `string`, используя технику включения. При наследовании можно иметь только один объект. (Трудно вызвать объекты извне, если у них нет имени.)

Тем не менее, приватное наследование дает возможности, отсутствующие у технологии включения. Предположим, что класс имеет защищенных членов, которые могут быть данными или функциями. Такие члены доступны только для класса-наследника, но не для внешних пользователей класса. Если вы включите этот класс в другой класс с помощью композиции, то новый класс не будет классом наследником. Поэтому он не будет иметь доступа к защищенным членам. Если же использовать наследование, то новый класс будет наследником, а значит, будет иметь доступ к защищенным членам.

Другой ситуацией, в которой придется использовать приватное наследование, является переопределение виртуальных функций. Это привилегия порожденных, а не контейнерных классов. При использовании приватного наследования переопределенные функции могут применяться только внутри класса, но не извне.

Совет

В общем случае, для создания отношения *has-a* нужно использовать технику включения. Если класс должен иметь доступ к защищенным членам базовых классов или необходимо переопределить виртуальную функцию, необходимо применять приватное наследование.

Защищенное наследование

Защищенное (protected) наследование это разновидность приватного (private) наследования. В этом случае при объявлении базового класса указывается ключевое слово (protected):

```
class Student : protected std::string,
                protected std::valarray<double>
{...};
```

При защищенном наследовании общедоступные и защищенные члены базового класса становятся защищенными членами производного класса. Как и при приватном наследовании, интерфейсы базовых классов доступны для производного класса, но не доступны для внешних пользователей. Главное отличие между защищенным и приватным наследованием проявляется при наследовании класса, который сам является производным классом. При приватном наследовании производный класс не получает доступа к интерфейсам базовых классов. Это происходит потому, что общедоступные (public) методы базовых классов становятся приватными в классе-наследнике, и доступ к приватным членам и методам из следующего уровня наследования невозможен. При защищенном наследовании общедоступные методы базового класса становятся защищенными членами производного класса и доступны для следующего уровня наследования.

В табл. 14.1 приведены свойства общедоступного, приватного и защищенного наследования. Выражение *неявное восходящее преобразование* означает, что можно иметь указатель или ссылку базового класса, ссылающуюся на объект производного класса без использования явного преобразования типов.

Таблица 14.1. Разновидности наследования

Свойство	Общедоступное наследование	Защищенное наследование	Приватное наследование
Общедоступные члены становятся	Общедоступными членами производного класса	Защищенными членами производного класса	Приватными членами производного класса
Защищенные члены становятся	Защищенными членами производного класса	Защищенными членами производного класса	Приватными членами производного класса
Приватные члены становятся	Доступными только через интерфейс базового класса	Доступными только через интерфейс базового класса	Доступными только через интерфейс базового класса
Неявное восходящее преобразование	Да	Да, только внутри производного класса	Нет

Переопределение доступа с помощью using

При использовании защищенного или приватного наследования общедоступные методы базового класса становятся защищенными или приватными. Предположим, что нужно создать специальный метод базового класса, общедоступный в производном классе. Одна из возможностей — определить метод производного класса, использующий метод базового класса. Например, нужно, чтобы класс Student мог

использовать метод sum() из valarray. Можно определить метод sum() в определении класса, а затем определить метод следующим образом:

```
double Student::sum() const // общедоступный метод Student
{
   return std::valarray<double>::sum(); // использует приватно
                                         // унаследованный метод
}
```

Тогда из объектов Student можно вызывать метод Student::sum(), который применяет метод valarray<double>::sum() к встроенному объекту valarray. (Если определение ArrayDb находится в области видимости, можно использовать ArrayDb вместо std::valarray<double>.)

В качестве альтернативы вызову одной функции из другой можно использовать using для объявления специального метода базового класса, доступного из производного класса, даже если наследование является приватным. Допустим, нужно использовать методы min() и max() из valarray с классом Student. В этом случае в раздел public файла studenti.h можно добавить объявление using:

```
class Student : private std::string, private std::valarray<double>
{
...
public:
   using std::valarray<double>::min;
   using std::valarray<double>::max;
   ...
};
```

Объявление using делает методы valarray<double>::min() и valarray<double>::max() доступными, как будто они являются общедоступными методами класса Student:

```
cout << "высшая оцента: " << ada[i].max() << endl;
```

Обратите внимание, что объявление using использует только имя члена, а не имя родителя, функции или возвращаемого типа. Например, для того чтобы сделать метод operator[]() из valarray доступным в классе Student, поместите объявление using в раздел private определения класса Student:

```
using std::valarray<double>::operator[];
```

Это сделает доступными обе версии. Затем можно удалить существующие прототипы и определения для Student::operator[](). Подход с использованием объявления using работает только в случае наследования и не работает при технике включения.

Существует и более старый подход — переопределить методы базового класса в классе, порожденном с помощью приватного наследования. Достаточно расположить имя метода в разделе public класса-наследника. Это можно сделать следующим образом:

```
class Student : private std::string, private std::valarray<double>
{
public:
   std::valarray<double>::operator[]; //переопределена как общедоступная
   ...
};
```

Это выглядит как объявление using без ключевого слова using. Такой подход не приветствуется, поскольку он не соответствует концепции. Поэтому если ваш компилятор поддерживает объявление using, можно использовать его для создания методов базовых классов, доступных для классов-наследников.

Множественное наследование

При использовании множественного наследования создается производный класс, происходящий от нескольких базовых классов. Подобно одиночному наследованию, множественное наследование устанавливает отношение *is-a*. Например, если имеются классы Waiter и Singer, от них можно породить класс SingingWaiter:

```
class SingingWaiter : public Waiter, public Singer {...};
```

Обратите внимание, что нужно оба базовых класса должны сопровождаться ключевым словом public. Это необходимо, поскольку по умолчанию компилятор подразумевает приватное наследование:

```
class SingingWaiter : public Waiter, Singer {...}; // Singer является
                                                   // приватным базовым классом
```

Как обсуждалось ранее в этой главе, приватное и защищенное множественное наследование может выражать отношение *has-a;* примером может послужить реализация класса Student в файле studenti.h. Давайте сосредоточимся на общедоступном наследовании.

Использование множественного наследования может приводить к трудностям для программистов. Двумя главными проблемами являются: наследование разных методов с одинаковыми именами от разных базовых классов и наследование нескольких экземпляров класса от нескольких непосредственно связанных базовых классов. Решение этих проблем приводит к появлению новых правил и вариантов синтаксиса. Таким образом, реализация множественного наследования сложнее, чем одиночного наследования. По этой причине многие специалисты, принадлежащие к сообществу C++, возражают против использования множественного наследования, а некоторые хотели бы вообще изъять множественное наследование из языка. Другим наоборот — нравится множественное наследование, они считают его полезным для отдельных проектов. Тем не менее, они советуют использовать множественное наследование аккуратно и с осторожностью.

Исследуем отдельный пример и посмотрим, какие возникают проблемы и какие возможны решения. Для примера с множественным наследованием необходимо иметь несколько классов. В этом примере нужно определить абстрактный базовый класс Worker, и путем наследования от него определить классы Waiter и Singer. Потом, используя множественное наследование и наследуя от классов Waiter и Singer, можно определить класс SingingWaiter (рис. 14.3). В этом случае базовый класс Worker будет дважды наследоваться различными классами, что вызывает наибольшие трудности при использовании множественного наследования. Начнем с определений классов Worker, Waiter и Singer, которые представлены в листинге 14.7.

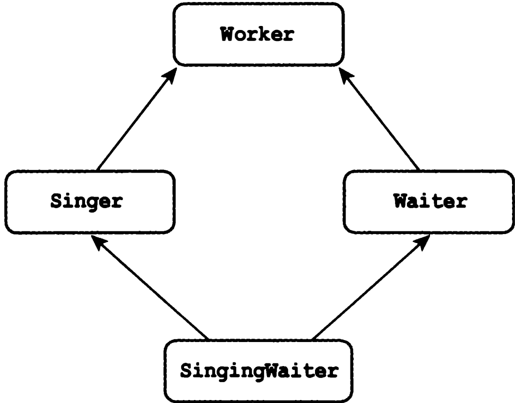


Рис. 14.3. Множественное наследование с совместно используемым предком

Листинг 14.7. worker0.h

```
// worker0.h -- классы сотрудников
#ifndef WORKER0_H_
#define WORKER0_H_
#include <string>
class Worker // абстрактные базовые классы
{
private:
   std::string fullname;
   long id;
public:
   Worker() : fullname("no one"), id(0L) {}
   Worker(const std::string & s, long n)
      : fullname(s), id(n) {}
   virtual ~Worker() = 0; // чистый виртуальный деструктор
   virtual void Set();
   virtual void Show() const;
};
class Waiter : public Worker
{
private:
   int panache;
public:
   Waiter() : Worker(), panache(0) {}
   Waiter(const std::string & s, long n, int p = 0)
      : Worker(s, n), panache(p) {}
   Waiter(const Worker & wk, int p = 0)
      : Worker(wk), panache(p) {}
   void Set();
   void Show() const;
};
class Singer : public Worker
{
protected:
```

```
   enum {other, alto, contralto, soprano,
      bass, baritone, tenor};
   enum {Vtypes = 7};
private:
   static char *pv[Vtypes]; // строковые эквиваленты разновидностей голоса
   int voice;
public:
   Singer() : Worker(), voice(other) {}
   Singer(const std::string & s, long n, int v = other)
      : Worker(s, n), voice(v) {}
   Singer(const Worker & wk, int v = other)
      : Worker(wk), voice(v) {}
   void Set();
   void Show() const;
};
#endif
```

Определение классов в листинге 14.7 включает несколько внутренних констант, представляющих разновидности голоса. Перечислением вводятся символьные константы типов голоса `alto`, `contralto` и так далее. Статический массив `pv` содержит указатели на строковые эквиваленты C-стиля. В файле реализации, приведенном в листинге 14.8, осуществлятся инициализация массива и определение методов.

Листинг 14.8. worker0.cpp

```
// worker0.cpp -- методы классов сотрудников
#include "worker0.h"
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
// методы Worker
// должен реализовать виртуальный деструктор, даже если он чистый
Worker::~Worker() {}
void Worker::Set()
{
   cout << "Введите имя и фамилию сотрудника: ";
   getline(cin, fullname);
   cout << "Введите идентификатор сотрудника: ";
   cin >> id;
   while (cin.get() != '\n')
      continue;
}
void Worker::Show() const
{
   cout << "Имя: " << fullname << "\n";
   cout << "Идентификатор: " << id << "\n";
}
// методы Waiter
void Waiter::Set()
{
   Worker::Set();
   cout << "Введите индекс щегольства официанта: ";
```

```
   cin >> panache;
   while (cin.get() != '\n')
      continue;
}
void Waiter::Show() const
{
   cout << "Категория: официант\n";
   Worker::Show();
   cout << "Индекс щегольства: " << panache << "\n";
}
// методы Singer
char * Singer::pv[] = {"other", "alto", "contralto",
"soprano", "bass", "baritone", "tenor"};
void Singer::Set()
{
   Worker::Set();
   cout << "Введите вокальный диапазон певца:\n";
   int i;
   for (i = 0; i < Vtypes; i++)
   {
      cout << i << ": " << pv[i] << " ";
      if ( i % 4 == 3)
         cout << endl;
   }
   if (i % 4 != 0)
      cout << endl;
   cin >> voice;
   while (cin.get() != '\n')
      continue;
}
void Singer::Show() const
{
   cout << "Категория: певец\n";
   Worker::Show();
   cout << "Вокальный диапазон: " << pv[voice] << endl;
}
```

Код в листинге 14.9 предназначен для простого тестирования классов с использованием полиморфного массива указателей.

Листинг 14.9. `worktest.cpp`

```
// worktest.cpp -- тестирование иерархии классов сотрудников
#include <iostream>
#include "worker0.h"
const int LIM = 4;
int main()
{
   Waiter bob("Bob Apple", 314L, 5);
   Singer bev("Beverly Hills", 522L, 3);
   Waiter w_temp;
   Singer s_temp;
   Worker * pw[LIM] = {&bob, &bev, &w_temp, &s_temp};
```

```
    int i;
    for (i = 2; i < LIM; i++)
        pw[i]->Set();
    for (i = 0; i < LIM; i++)
    {
        pw[i]->Show();
        std::cout << std::endl;
    }
    return 0;
}
```

Замечание по совместимости

Если система корректно не поддерживает дружественную функцию `getline()`, выполнить программу не удастся. Можно изменить программу и использовать `operator>>()` из листинга 14.8 вместо функции `getline()`. Поскольку эта дружественная функция читает только одно слово, потребуется изменить подсказку для ввода только фамилии.

Ниже показан вывод программы, приведенной в листингах 14.7, 14.8 и 14.9:

```
Введите имя и фамилию сотрудника: Waldo Dropmaster
Введите идентификатор сотрудника: 442
Введите индекс щегольства официанта: 3
Введите имя и фамилию сотрудника: Sylvie Sirenne
Введите идентификатор сотрудника: 555
Введите вокальный диапазон певца:
0: other 1: alto 2: contralto 3: soprano
4: bass 5: baritone 6: tenor
3
Категория: waiter
Имя: Bob Apple
Идентификатор: 314
Индекс щегольства: 5
Категория: singer
Имя: Beverly Hills
Идентификатор: 522
Вокальный диапазон: soprano
Категория: waiter
Имя: Waldo Dropmaster
Идентификатор: 442
Индекс щегольства: 3
Категория: singer
Имя: Sylvie Sirenne
Идентификатор: 555
Вокальный диапазон: soprano
```

Казалось бы, реализация работает: указатели на `Waiter` вызывают `Waiter::Show()`, указатели на `Singer` — `Singer::Show()` и `Singer::Set()`. Однако трудности возникают, если нужно добавить класс `SingingWaiter`, порожденный от двух классов `Singer` и `Waiter`. В частности, возникают два вопроса:

- Сколько всего сотрудников?
- Какой использовать метод?

Сколько всего сотрудников?

Начнем с общедоступного наследования SingingWaiter от Singer и Waiter:

```
class SingingWaiter: public Singer, public Waiter {...};
```

Поскольку Singer и Waiter наследуют компонент Worker, то SingingWaiter будет иметь два компонента Worker (рис. 14.4).

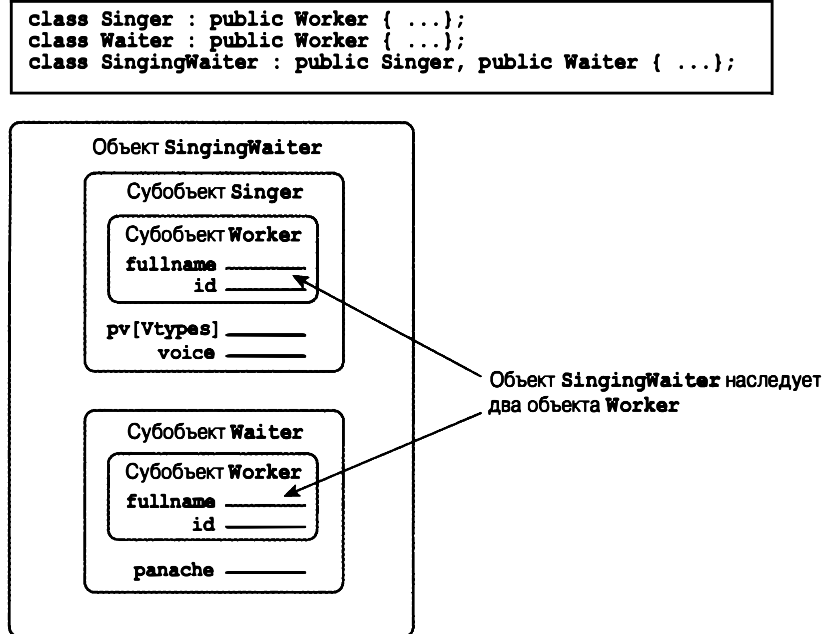


Рис. 14.4. Наследование двух объектов базового класса

Можно ожидать, что это приведет к трудностям. Например, назначение адреса объекта производного класса указателю на объект базового класса становится неоднозначным:

```
SingingWaiter ed;
Worker * pw = &ed; // неоднозначность
```

Обычно такое присваивание назначает указателю базового класса адрес объекта базового класса внутри производного объекта. Но ed содержит два объекта Worker, то есть нужно выбирать из двух адресов. Можно выбрать объект, используя приведение типа:

```
Worker * pw1 = (Waiter *) &ed; // Worker в Waiter
Worker * pw2 = (Singer *) &ed; // Worker в Singer
```

Это, безусловно, усложняет технику использования массива указателей на базовый класс для ссылок на множество объектов (полиморфизм).

Наличие двух копий объектов Worker приводит также и к другим сложностям. Тем не менее, главный вопрос состоит в том, зачем вообще нужны две копии объек-

та Worker? Поющий официант, как и любой другой сотрудник, должен иметь только одно имя и один идентификатор. Вместе с введением множественного наследования в C++ появились виртуальные базовые классы, делающие упомянутое наследование возможным.

Виртуальные базовые классы

Виртуальные базовые классы позволяют объекту, порожденному от нескольких базовых классов, имеющих, в свою очередь, один совместно используемый базовый класс, наследовать только один объект от этого базового класса. Например, можно сделать Worker виртуальным базовым классом для Singer и Waiter, указывая ключевое слово virtual в определении класса (virtual и public можно использовать в любом порядке):

```
class Singer : virtual public Worker {...};
class Waiter : public virtual Worker {...};
```

Затем необходимо определить SingingWaiter, как это делалось раньше:

```
class SingingWaiter: public Singer, public Waiter {...};
```

Теперь объект SingingWaiter будет иметь один объект Worker. Singer и Waiter имеют один общий базовый объект Worker вместо двух его копий.

В сущности, производные объекты Singer и Waiter совместно используют один общий базовый объект Worker вместо того, чтобы создавать две его копии (рис. 14.5). Поскольку SingingWaiter теперь содержит один субобъект Worker, снова можно использовать полиморфизм.

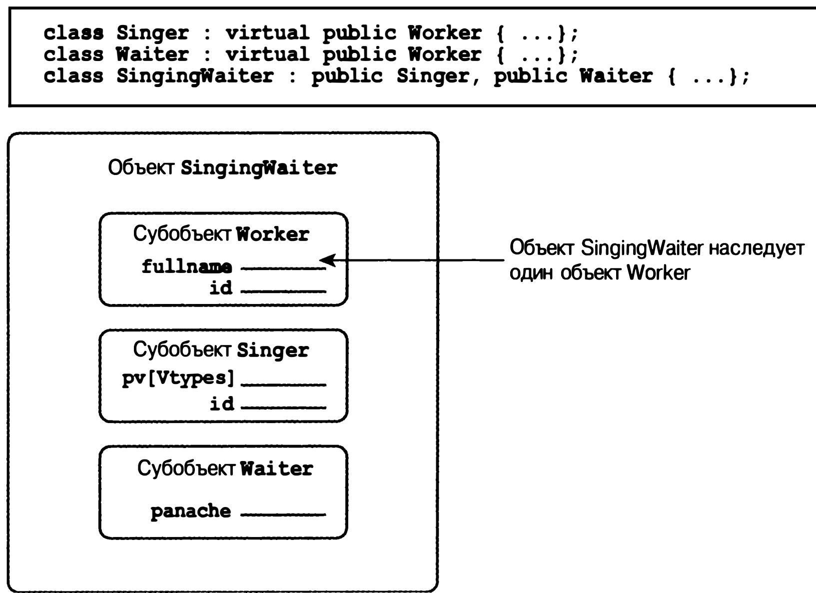


Рис. 14.5. Наследование с виртуальными базовыми классами

Рассмотрим несколько вопросов, которые могут возникнуть:

- Почему используется термин виртуальный?
- Почему нельзя обойтись без определения базовых классов виртуальными вместо того, чтобы сделать виртуальное поведение нормой для множественного наследования?
- Существуют ли здесь какие-то ловушки?

Во-первых, зачем же все-таки применяется термин "виртуальный"? После всего сказанного совсем не очевидна ясная связь между концепциями виртуальных функций и виртуальных базовых классов. Оказывается, существует сильное давление со стороны сообщества C++, препятствующее введению новых ключевых слов. Будет неудобно, например, если новое ключевое слово совпадет с именем важной функции или переменной в главной программе. Таким образом, C++ просто переопределяет ключевое слово `virtual` для новой возможности — своего рода небольшая "перегрузка" ключевого слова.

Далее, почему нельзя обойтись без определения базовых классов виртуальными вместо того, чтобы сделать виртуальное поведение нормой для множественного наследования? Во-первых, бывают случаи, когда нужно иметь несколько копий базового класса. Во-вторых, если сделать базовые классы виртуальными, программа будет выполнять дополнительные вычисления. То есть, за эту возможность придется платить, даже если вы в ней не нуждаетесь. В-третьих, существуют трудности, которые будут рассмотрены чуть ниже.

Наконец, есть ли еще препятствия? Да, есть. Использование виртуальных базовых классов требует изменения правил C++, и придется кодировать некоторые вещи по-другому. Кроме того, использование виртуальных базовых классов потребует изменения существующего кода. Например, добавление класса `SingingWaiter` в иерархию класса `Worker` требует возврата назад и добавления ключевого слова `virtual` в определения классов `Singer` и `Waiter`.

Новые правила для конструкторов

Наличие виртуальных базовых классов требует нового подхода к конструкторам класса. При использовании невиртуальных базовых классов в списке инициализации могут присутствовать только конструкторы непосредственных базовых классов. Однако эти конструкторы, в свою очередь, могут передавать информацию своим базовым классам. Например, допустима следующая организация конструкторов:

```
class A
{
   int a;
public:
   A(int n = 0) { a = n; }
   ...
};
class B: public A
{
   int b;
public:
   B(int m = 0, int n = 0) : A(n) { b = m; }
...
};
```

```
class C : public B
{
   int c;
public:
   C(int q = 0, int m = 0, int n = 0) : B(m, n) { c = q; }
   ...
};
```

Конструктор класса C может вызывать только конструкторы класса B, а конструктор B может вызывать только конструкторы класса A. В примере конструктор C использует значение q и передает значения m и n обратно конструктору B. Конструктор B использует значение m и передает значение n конструктору A. Если же Worker будет виртуальным базовым классом, автоматическая передача информации работать не будет. Рассмотрим следующий конструктор для случая множественного наследования:

```
SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
      : Waiter(wk,p), Singer(wk,v) {} // некорректно
```

Проблема заключается в том, что в данном случае wk автоматически передается в объект Worker двумя разными путями (Waiter и Singer). Для устранения потенциального конфликта C++ отключит автоматическую передачу информации через промежуточный класс в базовый класс, являющийся виртуальным. Таким образом, предыдущий конструктор инициализирует члены panache и voice, однако аргумент wk не будет передан в субобъект Waiter. Тем не менее, компилятор должен создать объект базового компонента перед созданием производных компонентов. В данном случае он будет использовать конструктор Worker по умолчанию.

Если для создания виртуального базового класса нужно использовать конструктор, отличный от конструктора по умолчанию, необходимо вызывать его явно. Это делается так:

```
SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
      : Worker(wk), Waiter(wk,p), Singer(wk,v) {}
```

Здесь явно вызывается конструктор Worker(const Worker &). Отметим, что такое использование допустимо и часто необходимо для виртуальных базовых классов, но не допустимо для невиртуальных.

Внимание!

Если у класса есть непрямой виртуальный базовый класс, конструктор этого класса должен явно вызывать конструктор виртуального базового класса, за исключением случаев, когда достаточно конструктора по умолчанию.

Какой использовать метод?

Вдобавок к изменениям правил для конструкторов, множественное наследование часто требует уточнений в исходном коде. Рассмотрим случай расширения метода Show() для класса SingingWaiter. Поскольку у объекта SingingWaiter нет членов данных, можно подумать, что достаточно использовать унаследованные методы. Однако это создает первую проблему. Не будем создавать новую версию Show() и попытаемся применить объект SingingWaiter для вызова унаследованного метода Show():

```
SingingWaiter newhire("Elise Hawks", 2005, 6, soprano);
newhire.Show(); // неоднозначность
```

При одиночном наследовании сбой переопределения функции `Show()` приведет к использованию самого последнего наследственного определения этой функции. В данном же случае у каждого прямого предка имеется метод `Show()`, который делает этот вызов неоднозначным.

Внимание!

Множественное наследование может приводить к неоднозначным вызовам функций. Например, класс `BadDude` может наследовать два совершенно разных метода `Draw()` от классов `Gunslinger` и `PokerPlayer`.

Для ясности можно использовать операцию разрешения контекста:

```
SingingWaiter newhire("Elise Hawks", 2005, 6, soprano);
newhire.Singer::Show(); // использует версию Singer
```

Однако лучше переопределить `Show()` для `SingingWaiter` и указать, какой `Show()` использовать. Например, если нужно, чтобы `SingingWaiter` использовал версию из `Singer`, поступите следующим образом:

```
void SingingWaiter::Show()
{
   Singer::Show();
}
```

В этом методе производный метод вызывает базовый метод. Это хорошо работает для одиночного наследования. Предположим, что класс `HeadWaiter` унаследован от класса `Waiter`. Можно использовать следующую последовательность определений, где в каждый производный класс добавлена информация базового класса:

```
void Worker::Show() const
{
   cout << "Имя: " << fullname << "\n";
   cout << "Идентификатор: " << id << "\n";
}
void Waiter::Show() const
{
   Worker::Show();
   cout << "Индекс щегольства: " << panache << "\n";
}
void HeadWaiter::Show() const
{
   Waiter::Show();
   cout << "Индекс присутствия: " << presence << "\n";
}
```

Тем не менее, для `SingingWaiter` этот подход не работает. Метод

```
void SingingWaiter::Show()
{
   Singer::Show();
}
```

даст сбой, поскольку в нем игнорируется компонент `Waiter`. Это можно исправить, вызвав версию из `Waiter`:

```
void SingingWaiter::Show()
{
   Singer::Show();
   Waiter::Show();
}
```

В этом фрагменте имя и идентификатор сотрудника будут отображены дважды, поскольку `Singer::Show()` и `Waiter::Show()` оба вызывают `Worker::Show()`.

Как это устранить? Один из способов — воспользоваться модульным подходом вместо инкрементного. Так, нужно определить метод, отображающий только компоненты `Worker`, затем метод, отображающий только компоненты `Waiter` (вместо компонентов `Waiter` плюс `Worker`), и, наконец, метод, отображающий компоненты `Singer`. Далее необходимо собрать эти компоненты вместе в методе `SingingWaiter::Show()`. Например, можно поступить следующим образом:

```
void Worker::Data() const
{
   cout << "Имя: " << fullname << "\n";
   cout << "Идентификатор: " << id << "\n";
}
void Waiter::Data() const
{
   cout << "Индекс щегольства: " << panache << "\n";
}
void Singer::Data() const
{
   cout << "Вокальный диапазон: " << pv[voice] << "\n";
}
void SingingWaiter::Data() const
{
   Singer::Data();
   Waiter::Data();
}
void SingingWaiter::Show() const
{
   cout << "Категория: поющий официант\n";
   Worker::Data();
   Data();
}
```

Аналогично, другие методы `Show()` можно построить из соответствующих компонентов `Data()`. При таком подходе объекты по-прежнему используют метод `Show()` общедоступно. С другой стороны, методы `Data()` должны быть внутренними для класса; они должны быть вспомогательными методами, обеспечивающими общедоступный интерфейс. Если сделать методы `Data()` приватными, то вызов `Worker::Data()` из кода `Waiter` будет невозможным. Это разновидность ситуации, в которой полезны защищенные классы. Если методы `Data()` являются защищенными, их можно использовать внутри всех классов иерархии, в то время как извне они будут недоступными.

Другой подход — все компоненты данных сделать не приватными, а защищенными. Использование защищенных методов вместо защищенных данных позволяет жестче контролировать доступ к данным.

Метод Set(), запрашивающий данные для установки значений объекта, представляет сходную проблему. Например, SingingWaiter::Set() запрашивает информацию для Worker один раз, а не два. Аналогично работает и Show(). Можно определить защищенный метод Get(), который запрашивает информацию только для одного класса, а затем объединить методы Set(), использующие методы Get() в качестве строительных блоков.

Короче говоря, введение множественного наследования с совместно используемым предком требует введения виртуальных базовых классов, изменения правил для списка инициализации конструкторов, и, возможно, перекодирования классов, если они написаны с учетом множественного наследования. В листинге 14.10 показаны измененные определения классов, а в листинге 14.11 — их реализации.

Листинг 14.10. workermi.h

```
// workermi.h -- классы сотрудников с множественным наследованием
#ifndef WORKERMI_H_
#define WORKERMI_H_
#include <string>
class Worker // абстрактный базовый класс
{
private:
   std::string fullname;
   long id;
protected:
   virtual void Data() const;
   virtual void Get();
public:
   Worker() : fullname("no one"), id(0L) {}
   Worker(const std::string & s, long n)
      : fullname(s), id(n) {}
   virtual ~Worker() = 0; // чистая виртуальная функция
   virtual void Set() = 0;
   virtual void Show() const = 0;
};
class Waiter : virtual public Worker
{
private:
   int panache;
protected:
   void Data() const;
   void Get();
public:
   Waiter() : Worker(), panache(0) {}
   Waiter(const std::string & s, long n, int p = 0)
      : Worker(s, n), panache(p) {}
   Waiter(const Worker & wk, int p = 0)
      : Worker(wk), panache(p) {}
   void Set();
   void Show() const;
};
class Singer : virtual public Worker
{
```

```
protected:
   enum {other, alto, contralto, soprano,
   bass, baritone, tenor};
   enum {Vtypes = 7};
   void Data() const;
   void Get();
private:
   static char *pv[Vtypes]; // строка, содержащая разновидности голосов
   int voice;
public:
   Singer() : Worker(), voice(other) {}
   Singer(const std::string & s, long n, int v = other)
      : Worker(s, n), voice(v) {}
   Singer(const Worker & wk, int v = other)
      : Worker(wk), voice(v) {}
   void Set();
   void Show() const;
};
// множественное наследование
class SingingWaiter : public Singer, public Waiter
{
protected:
   void Data() const;
   void Get();
public:
   SingingWaiter() {}
   SingingWaiter(const std::string & s, long n, int p = 0,
   int v = other)
      : Worker(s,n), Waiter(s, n, p), Singer(s, n, v) {}
   SingingWaiter(const Worker & wk, int p = 0, int v = other)
      : Worker(wk), Waiter(wk,p), Singer(wk,v) {}
   SingingWaiter(const Waiter & wt, int v = other)
      : Worker(wt),Waiter(wt), Singer(wt,v) {}
   SingingWaiter(const Singer & wt, int p = 0)
      : Worker(wt),Waiter(wt,p), Singer(wt) {}
   void Set();
   void Show() const;
};
#endif
```

Листинг 14.11. workermi.cpp

```
// workermi.cpp -- методы классов сотрудников с множественным наследованием
#include "workermi.h"
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
// методы Worker
Worker::~Worker() { }
// защищенные методы
void Worker::Data() const
```

```
{
   cout << "Имя: " << fullname << endl;
   cout << "Идентификатор: " << id << endl;
}
void Worker::Get()
{
   getline(cin, fullname);
   cout << "Введите идентификатор сотрудника: ";
   cin >> id;
   while (cin.get() != '\n')
      continue;
}
// методы Waiter
void Waiter::Set()
{
   cout << "Введите имя и фамилию официанта: ";
   Worker::Get();
   Get();
}
void Waiter::Show() const
{
   cout << "Категория: официант\n";
   Worker::Data();
   Data();
}
// защищенные методы
void Waiter::Data() const
{
   cout << "Индекс щегольства: " << panache << endl;
}
void Waiter::Get()
{
   cout << "Введите индекс щегольства официанта: ";
   cin >> panache;
   while (cin.get() != '\n')
      continue;
}
// методы Singer
char * Singer::pv[Singer::Vtypes] = {"other", "alto", "contralto",
      "soprano", "bass", "baritone", "tenor"};
void Singer::Set()
{
   cout << "Введите имя и фамилию певца: ";
   Worker::Get();
   Get();
}
void Singer::Show() const
{
   cout << "Категория: певец\n";
   Worker::Data();
   Data();
}
```

```
// защищенные методы
void Singer::Data() const
{
   cout << "Вокальный диапазон: " << pv[voice] << endl;
}
void Singer::Get()
{
   cout << "Введите число, соответствующее вокальному диапазону певца:\n";
   int i;
   for (i = 0; i < Vtypes; i++)
   {
      cout << i << ": " << pv[i] << " ";
      if ( i % 4 == 3)
         cout << endl;
   }
   if (i % 4 != 0)
      cout << '\n';
   cin >> voice;
   while (cin.get() != '\n')
      continue;
}
// методы SingingWaiter
void SingingWaiter::Data() const
{
   Singer::Data();
   Waiter::Data();
}
void SingingWaiter::Get()
{
   Waiter::Get();
   Singer::Get();
}
void SingingWaiter::Set()
{
   cout << "Введите имя и фамилию поющего официанта: ";
   Worker::Get();
   Get();
}
void SingingWaiter::Show() const
{
   cout << "Категория: поющий официант\n";
   Worker::Data();
   Data();
}
```

Конечно, ради любопытства нужно протестировать эти классы. В листинге 14.12 приведен код, пригодный для этого. Обратите внимание, что в программе при назначении адресов различных типов классов указателям базовых классов используется полиморфизм. Применяется также функция `strchr()` из библиотеки обработки строк С-стиля:

```
while (strchr("wstq", choice) == NULL)
```

Эта функция возвращает адрес первого вхождения символа choice в строку "wstq"; функция возвращает указатель NULL, если символ не найден. Использовать этот тест проще, чем писать оператор if для сравнения с choice каждого символа.

Не забудьте скомпилировать листинг 14.12 с файлом workermi.cpp.

Листинг 14.12. workmi.cpp

```
// workmi.cpp -- множественное наследование
// компилировать с workermi.cpp
#include <iostream>
#include <cstring>
#include "workermi.h"
const int SIZE = 5;
int main()
{
   using std::cin;
   using std::cout;
   using std::endl;
   using std::strchr;
   Worker * lolas[SIZE];
   int ct;
   for (ct = 0; ct < SIZE; ct++)
   {
      char choice;
      cout << "Введите категорию сотрудника:\n"
         << "w: официант s: певец "
         << "t: поющий официант q: завершение\n";
      cin >> choice;
      while (strchr("wstq", choice) == NULL)
      {
         cout << "Пожалуйста, введите w, s, t или q: ";
         cin >> choice;
      }
      if (choice == 'q')
         break;
      switch(choice)
      {
         case 'w': lolas[ct] = new Waiter;
            break;
         case 's': lolas[ct] = new Singer;
            break;
         case 't': lolas[ct] = new SingingWaiter;
            break;
      }
      cin.get();
      lolas[ct]->Set();
   }
   cout << "\nВаш персонал:\n";
   int i;
   for (i = 0; i < ct; i++)
   {
      cout << endl;
      lolas[i]->Show();
   }
```

```
   for (i = 0; i < ct; i++)
      delete lolas[i];
   cout << "Всего наилучшего.\n";
   return 0;
}
```

Замечание по совместимости

Если система корректно не поддерживает дружественную функцию `getline()`, выполнить программу не удастся. Можно изменить программу и воспользоваться `operator>>()` из листинга 14.11 вместо функции `getline()`. Поскольку эта дружественная функция читает только одно слово, потребуется изменить подсказку для ввода только фамилии.

Ниже показан пример выполнения программы, содержащейся в листингах 14.10, 14.11 и 14.12:

```
Введите категорию сотрудника:
w: официант s: певец t: поющий официант q: завершение
w
Введите имя и фамилию официанта: Wally Slipshod
Введите идентификатор сотрудника: 1040
Введите индекс щегольства официанта: 4
Введите категорию сотрудника:
w: waiter s: singer t: singing waiter q: quit
s
Введите имя и фамилию певца: Sinclair Parma
Введите идентификатор сотрудника: 1044
Введите число, соответствующее вокальному диапазону певца:
0: other 1: alto 2: contralto 3: soprano
4: bass 5: baritone 6: tenor
5
Введите категорию сотрудника:
w: официант s: певец t: поющий официант q: завершение
t
Введите имя и фамилию поющего официанта: Natasha Gargalova
Введите идентификатор сотрудника: 1021
Введите индекс щегольства официанта: 6
Введите число, соответствующее вокальному диапазону певца:
0: other 1: alto 2: contralto 3: soprano
4: bass 5: baritone 6: tenor
3
Введите категорию сотрудника:
w: официант s: певец t: поющий официант q: завершение
q
Ваш персонал:
Категория: официант
Имя: Wally Slipshod
Идентификатор: 1040
Индекс щегольства: 4
Категория: певец
Имя: Sinclair Parma
Идентификатор: 1044
Вокальный диапазон: baritone
Категория: поющий официант
```

```
Имя: Natasha Gargalova
Идентификатор: 1021
Вокальный диапазон: soprano
Индекс щегольства: 6
Всего наилучшего.
```

Давайте рассмотрим еще насколько вопросов, касающихся множественного наследования.

Смешанный виртуальный и невиртуальный базис

Снова рассмотрим случай с производным классом, который наследует базовый класс несколькими путями. Если базовый класс виртуальный, то производный класс содержит один субобъект базового класса. Если базовый класс невиртуальный, производный класс содержит несколько субобъектов. А что, если их смешать? Предположим, что класс B является виртуальным базовым классом для классов C и D, и невиртуальным базовым классом для классов X и Y. Представим так же, что класс M унаследован от классов C, D, X и Y. В этом случае класс M содержит один субобъект класса B для всех виртуально унаследованных родителей (классов C и D) и отдельный субобъект класса B для каждого невиртуального родителя (классов X и Y). Можно сказать, что он будет содержать три субобъекта класса B. При наследовании одного базового класса несколькими виртуальными и несколькими невиртуальными путями производный класс будет иметь один объект базового класса для представления всех виртуальных путей и отдельные объекты базового класса для каждого невиртуального пути.

Виртуальные базовые классы и доминирование

Использование виртуальных базовых классов меняет механизм, которым C++ разрешает неоднозначности. При использовании невиртуальных базовых классов правила просты. Если класс наследует несколько членов (данных или методов) с одинаковыми именами от разных классов, использование этого имени без квалификатора класса приведет к неоднозначности. Однако если виртуальные базовые классы связаны, то такое использование имени без квалификатора класса может и не приводить к неоднозначности. Если одно имя *доминирует* над всеми остальными, его можно использовать без квалификатора класса.

Каким образом одно имя члена может доминировать над другим? Имя в производном классе доминирует над такими же именами в родительских классах независимо от того, прямые они родители или нет. Рассмотрим следующие определения:

```
class B
{
public:
   short q();
   ...
};
class C : virtual public B
{
public:
   long q();
   int omb()
   ...
};
```

```
class D : public C
{
   ...
};
class E : virtual public B
{
private:
   int omb();
   ...
};
class F: public D, public E
{
   ...
};
```

Здесь определение q() в классе C доминирует над определением в классе B, поскольку C порожден от B. Таким образом, для вызова C::q() методы класса F могут использовать q(). С другой стороны, определение omb() не может доминировать над другими, поскольку ни C ни E не являются базовыми классами для других. Таким образом, попытка класса F использовать вызов omb() без квалификатора класса приведет к неоднозначности. Правила виртуальной неоднозначности не принимают во внимание правила доступа. Даже несмотря на то что E::omb() является приватным и поэтому недоступным непосредственно для класса F, вызов omb() будет неоднозначным. Аналогично, если даже C::q() будет приватным, он будет доминировать над D::q(). В этом случае возможен вызов B::q() в классе B, но неуточненный q() будет ссылаться на недоступный C::q().

Краткий обзор множественного наследования

Во-первых, давайте рассмотрим множественное наследование без виртуальных базовых классов. Эта форма множественного наследования не вводит новых правил. Тем не менее, если класс наследует два члена с одинаковыми именами, но от разных классов, в производном классе для разделения двух членов нужно использовать квалификатор класса. Так, методы класса BadDude, унаследованные от Gunslinger и PokerPlayer, должны использовать Gunslinger::draw() и PokerPlayer::draw() для разделения методов Drow(), унаследованных от двух разных классов. В противном случае компилятор выдаст сообщение о неоднозначном использовании.

Если производный класс наследуется от невиртуального базового класса несколькими путями, то он наследует по одному объекту базового класса для каждого экземпляра базового класса. В некоторых случаях — это то, что нужно, но чаще несколько экземпляров базового класса приводят к проблемам.

Далее, рассмотрим множественное наследование с виртуальными базовыми классами. Класс становится виртуальным базовым классом, когда производный класс использует ключевое слово virtual при описании наследования:

```
class marketing : public virtual reality { ... };
```

Главное отличие и причина применения виртуальных базовых классов заключается в том, что класс, порождаемый от одного или более экземпляров виртуального базового класса, наследует только один объект базового класса. Реализация этот свойства приводит к следующим требованиям:

- Производный класс с непрямым виртуальным базовым классом должен иметь конструкторы, непосредственно вызывающие конструкторы непрямых базовых классов, что недопустимо для непрямых виртуальных базовых классов.
- Неоднозначность имен должна разрешаться с помощью правил доминирования.

Как видно, множественное наследование может приводить к сложностям в программировании. Эти сложности возникают, когда производный класс наследуется несколькими путями от одного и того же базового класса. Во избежание этого нужно, при необходимости, уточнять наследуемые имена именем класса.

Шаблоны класса

Наследование (общедоступное, приватное и защищенное) и включение не являются решением, требующим обязательного повторного использования кода. Рассмотрим, например, класс `Stack` (см. главу 10) и класс `Queue` (см. главу 12). Это примеры *контейнерных классов*, содержащих другие объекты или типы данных. Класс `Stack` из главы 10, например, содержит значения `unsigned long`. Легко можно создать класс для хранения значений `double` или объектов `sting`. Код будет аналогичным, за исключением типов хранимых объектов. Тем не менее, прежде чем определять новые классы, было бы хорошо определить стек в общей (не зависящей от типа) форме и определить специфические типы как параметры класса. Тогда можно будет использовать один и тот же общий код для создания стеков разных типов величин. В главе 10 в примере `Stack` используется `typedef` в качестве первого шага в достижении этой цели. Но этот подход имеет множество недостатков. Во-первых, придется редактировать заголовочный файл каждый раз, как только вы измените тип. Во-вторых, можно использовать эту технику для создания только одного вида стека на программу. Это происходит потому, что `typedef` не может определять два разных типа одновременно, и поэтому невозможно использовать метод для создания одновременно, скажем, стека целочисленных значений и стека строк.

В C++ шаблоны класса предлагают более подходящий способ создания общих определений класса. (C++ в оригинале не поддерживает шаблоны, которые с момента своего появления постоянно развиваются. Так что вполне возможно, что ваш компилятор поддерживает не все рассматриваемые здесь возможности.) Шаблоны вводят параметризованные типы — это значит, что можно передавать имя типа в качестве аргумента при создании класса или функции. Передавая имя типа `int` в шаблон `Queue`, например, можно заставить компилятор создать класс `Queue` для хранения в очереди целых значений.

Библиотека C++ содержит несколько шаблонов классов. Ранее в этой главе рассматривался шаблон класса `valarray`. Стандартная библиотека шаблонов (Standard Template Library — STL) C++, рассматриваемая в главе 16, предлагает мощные и гибкие реализации шаблонов для нескольких контейнерных классов. В этой главе рассматриваются наиболее простые сущности.

Определение шаблона класса

Воспользуемся классом `Stack` из главы 10 в качестве модели для построения шаблона. Ниже приведено исходное определение класса:

```
typedef unsigned long Item;
class Stack
{
private:
   enum {MAX = 10};               // специфическая для класса константа
   Item items[MAX];               // содержит элементы стека
   int top;                       // индекс вершины стека
public:
   Stack();
   bool isempty() const;
   bool isfull() const;
   // push() возвращает false, если стек полон, иначе - true
   bool push(const Item & item);  // добавляет элемент в стек
   // pop() возвращает false, если стек пуст, иначе - true
   bool pop(Item & item);         // выталкивает элемент с вершины стека
};
```

При построении шаблона определение класса Stack меняется на определение шаблона, а функции-члены — на функции-члены шаблона. Как и для шаблонных функций, необходимо предварить шаблонный класс следующим кодом:

```
template <class Type>
```

Ключевое слово template говорит компилятору, что нужно определить шаблон. Часть кода в угловых скобках аналогична списку аргументов в функции. Можно считать, что ключевое слово class служит именем типа для переменной, которая получает тип как значение, а *Type* является именем этой переменной. Использование class не означает, что *Type* должен быть классом; это означает только, что *Type* служит в качестве общего спецификатора типа, который будет заменен реальным типом при использовании шаблона. Последние реализации С++ позволяют применять вместо class более точное ключевое слово typename:

```
template <typename Type> // новый вариант
```

Можно указать общее имя типа вместо *Type*; правила наименования здесь те же, что и для любого другого идентификатора. Обычно используют T и Type; в данном случае следует применять Type. При вызове шаблона *Type* заменяется реальным типом int или string. Внутри определения шаблона для определения типа, который будет храниться в стеке, можно использовать общее имя типа. В случае с классом Stack нужно использовать *Type* везде, где в старом определении формально применялся typedef-идентификатор Item. Например:

```
Item items[MAX]; // содержит элементы стека
```

будет выглядеть как

```
Type items[MAX]; // содержит элементы стека
```

Аналогично можно заменить методы исходного класса функциями членами шаблона. Каждая функция должна предваряться аналогичным объявлением шаблона:

```
template <class Type>
```

И снова необходимо заменить typedef идентификатор Item обобщенным именем типа Type. Также нужно заменить квалификатор класса с Stack:: на Stack<Type>::. Например:

```
bool Stack::push(const Item & item)
{
...
}
```

преобразуется в:

```
template <class Type> // или шаблон <typename Type>
bool Stack<Type>::push(const Type & item)
{
...
}
```

Если определять метод внутри определения класса, можно опустить преамбулу шаблона и квалификатор класса.

В листинге 14.13 показаны комбинированный класс и шаблоны функций-членов. Очень важно понимать, что эти шаблоны не являются определениями классов и функций-членов. Они являются инструкциями для компилятора C++ о том, как сгенерировать определения класса и функций-членов. Частная актуализация шаблона, например, класса стека для управления строковыми объектами, называется *реализацией* или *специализацией*. Если ваш компилятор не поддерживает новое ключевое слово export, располагать шаблонные функции в отдельном файле реализации нельзя. Поскольку шаблоны не являются функциями, их нельзя компилировать отдельно. Шаблоны необходимо использовать совместно с вызовами частной реализации. Проще всего это сделать, расположив всю шаблонную информацию в заголовочном файле и включив заголовочный файл в файл, использующий шаблоны.

Листинг 14.13. stacktp.h

```
// stacktp.h -- шаблон стека
#ifndef STACKTP_H_
#define STACKTP_H_
template <class Type>
class Stack
{
private:
   enum {MAX = 10};                // специфическая для класса константа
   Type items[MAX];                // содержит элементы стека
   int top;                        // индекс вершины стека
public:
   Stack();
   bool isempty();
   bool isfull();
   bool push(const Type & item);   // добавляет элемент в стек
   bool pop(Type & item);          // выталкивает элемент с вершины стека
};
template <class Type>
Stack<Type>::Stack()
{
```

```
   top = 0;
}
template <class Type>
bool Stack<Type>::isempty()
{
   return top == 0;
}
template <class Type>
bool Stack<Type>::isfull()
{
   return top == MAX;
}
template <class Type>
bool Stack<Type>::push(const Type & item)
{
   if (top < MAX)
   {
      items[top++] = item;
      return true;
   }
   else
      return false;
}
template <class Type>
bool Stack<Type>::pop(Type & item)
{
   if (top > 0)
   {
      item = items[--top];
      return true;
   }
   else
      return false;
}
#endif
```

Если ваш компилятор поддерживает новое ключевое слово export, можно расположить определения шаблонных методов в отдельном файле, предваряя каждое определение шаблона следующим образом:

```
export:
// stacktp.h -- шаблон стека
#ifndef STACKTP_H_
#define STACKTP_H_
export template <class Type> // предваряется словом export
class Stack
{
...
};
```

Затем можно следовать соглашениям, которые используются для обычных классов:

1. Расположите определение шаблонного класса (с ключевым словом export) в заголовочном файле и используйте директиву #include, для того чтобы сделать это определение доступным программе.
2. Расположите определения методов шаблонного класса в файле с исходным кодом, включите заголовочный файл в этот файл и используйте файл проекта либо его эквивалент, для того чтобы сделать определения доступными программе.

Использование шаблонного класса

Включение шаблона в программу не генерирует шаблонный класс. Необходимо запросить реализацию. Для того чтобы это сделать, потребуется объявить объект с типом шаблонного класса и заменить имя общего типа конкретным типом. Например, ниже показано создание двух стеков, одного для хранения значений int и другого — для объектов string.

```
Stack<int> kernels;       // создает стек для значений int
Stack<string> colonels;   // создает стек для объектов string
```

Встретив эти два определения, компилятор с помощью шаблона Stack<Type> сгенерирует два различных определения класса и два различных набора методов класса. Объявление класса Stack<int> везде заменит Type на int, а объявление класса Stack<string> везде заменит Type на string. Конечно, используемый алгоритм должен быть совместим по типам. Класс Stack, например, допускает присваивание одних элементов другим. Это присваивание справедливо только для базовых типов, структур и классов (пока вы не сделаете операцию присваивания приватной), но не для массивов. Идентификаторы общих типов, такие как Type в примере, называются *параметрами типа*. Подразумевается, что они работают почти как переменные, но вместо присваивания числового значения параметру типа ему присваивается тип. Так, в объявлении kernels параметр типа Type имеет значение int.

Заметим, что необходимо определять требуемый тип явно. В этом заключается отличие от обычных шаблонных функций. В них компилятор может использовать типы аргументов функции для определения вида функции, которую нужно сгенерировать:

```
Template <class T>
void simple(T t) { cout << t << '\n';}
...
simple(2);        // генерирует void simple(int)
simple("two")     // генерирует void simple(char *)
```

В листинге 14.14 приведена программа тестирования стека (листинг 10.12), модифицированная для использования строкового представления идентификаторов заказов вместо их представления как unsigned long.

Листинг 14.14. stacktem.cpp

```
// stacktem.cpp -- тестирование шаблона класса стека
#include <iostream>
#include <string>
#include <cctype>
#include "stacktp.h"
using std::cin;
```

```
using std::cout;
int main()
{
   Stack<std::string> st; // создание пустого стека
   char ch;
   std::string po;
   cout << "Пожалуйста, введите A для добавления заказа,\n"
        << "P - для обработки заказа, Q - для выхода.\n";
   while (cin >> ch && std::toupper(ch) != 'Q')
   {
      while (cin.get() != '\n')
         continue;
      if (!std::isalpha(ch))
      {
         cout << '\a';
         continue;
      }
      switch(ch)
      {
         case 'A':
         case 'a':  cout << "Введите номер заказа для добавления: ";
                    cin >> po;
                    if (st.isfull())
                       cout << "Стек уже полон\n";
                    else
                       st.push(po);
                    break;
         case 'P':
         case 'p':  if (st.isempty())
                       cout << "Стек уже пуст\n";
                    else {
                       st.pop(po);
                       cout << "Заказ #" << po << " вытолкнут\n";
                       break;
                    }
      }
      cout << "Пожалуйста, введите A для добавления заказа,\n"
           << "P - для обработки заказа, Q - для выхода.\n";
   }
   cout << "Всего наилучшего!\n";
   return 0;
}
```

Замечание по совместимости

Если версия C++ не поддерживает заголовочный файл `cctype`, нужно использовать старый заголовочный файл `ctype.h`.

Ниже показан вывод программы из листинга 14.14:

```
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода.
A
Введите номер заказа для добавления: red911porsche
```

```
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода.
A
Введите номер заказа для добавления: greenS8audi
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода.
A
Введите номер заказа для добавления: silver747boeing
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода.
P
Заказ #silver747boeing вытолкнут
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода.
P
Заказ #greenS8audi вытолкнут
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода.
P
Заказ #red911porsche вытолкнут
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода.
P
stack already empty
Пожалуйста, введите A для добавления заказа,
P - для обработки заказа, Q - для выхода.
Q
Всего наилучшего!
```

Более пристальный взгляд на шаблонные классы

Можно использовать встроенный тип или объект класса в качестве типа для шаблона класса Stack<Type>. А как насчет указателей? Например, можно ли использовать в листинге 14.14 указатель на char вместо объекта sting? Такие указатели в C++ являются встроенным средством для работы со строками. Ответ заключается в том, что конечно, можно создать стек указателей, но он будет плохо работать без существенной переделки программы. Компилятор создаст такой класс, однако задача программиста — следить за корректным его использованием. Посмотрим, почему такой стек указателей плохо работает в листинге 14.14, а затем рассмотрим пример удачного использования стека указателей.

Некорректное использование стека указателей

Рассмотрим вкратце три простых, но некорректных попытки адаптации листинга 14.14 к использованию стека указателей. Нужно извлечь урок из этих примеров с тем, чтобы в дальнейшем при создании шаблонов не действовать вслепую. Все три примера начинаются с совершенно верного вызова шаблона Stack<Type>:

```
Stack<char *> st; // создает стек указателей на символы
```

В версии 1 оператор

```
string po;
```

меняется на

```
char * po;
```

Идея состоит в том, чтобы для клавиатурного ввода использовать указатель на char вместо объекта string. Этот подход приведет к аварийному завершению программы, поскольку одно только создание указателя не выделяет пространства в памяти под хранение входных строк. (Программа будет откомпилирована, но завершится с ошибкой, после того как cin попытается сохранить ввод в неопределенном участке памяти.)

В версии 2 оператор

```
string po;
```

меняется на

```
char po[40];
```

В данном случае выделяется пространство в памяти для хранения входных строк. Более того, po имеет тип char * и поэтому его можно расположить в стеке. Однако массив фундаментально не согласуется с допущениями, сделанными для метода pop():

```
template <class Type>
bool Stack<Type>::pop(Type & item)
{
   if (top > 0)
   {
      item = items[--top];
      return true;
   }
   else
      return false;
}
```

Во-первых, ссылочная переменная item ссылается на l-значение определенного типа, но не на имя массива. Во-вторых, предполагается, что можно присваивать значения переменной item. Даже если бы переменная item могла ссылаться на массив, невозможно присвоить значение имени массива. Так что этот подход тоже не годится.

В версии 3 оператор

```
string po;
```

меняется на

```
char * po = new char[40];
```

Здесь выделяется пространство для хранения входных строк. Более того, po является переменной и поэтому совместима с кодом для pop(). Однако и здесь мы сталкиваемся с фундаментальной проблемой: имеется только одна переменная po и она всегда указывает на одно и тоже место в памяти. Правда, содержимое памяти меняется при каждом чтении новой строки, но каждая операция заталкивания помещает в стек в точности один и тот же адрес. Таким образом, при выталкивании данных из стека мы всегда получим один и тот же адрес, и он всегда будет указывать на последнюю строку, сохраненную в памяти. Такой стек не сохраняет отдельно каждую новую строку по мере прихода последних, а значит, он попросту бесполезен.

Корректное использование стека указателей

Один из способов применения стека указателей предусматривает создание в вызывающей программе массива указателей, в котором каждый указатель ссылается на отдельную строку. Расположение таких указателей в стеке имеет смысл, так как эти указатели ссылаются на разные строки. Следует отметить, что создание массива указателей — обязанность вызывающей программы, а не стека. Задача стека — управлять указателями, а не создавать их.

Предположим, что нужно смоделировать следующую ситуацию. Некто доставил Плодсону (Plodson) тележку с папками. Если входной ящик Плодсона пуст, он берет верхнюю папку из тележки и кладет во входной ящик. Если входной ящик заполнен, Плодсон берет верхнюю папку из ящика, обрабатывает ее и кладет в выходной ящик. Если входной ящик заполнен частично, Плодсон может обработать верхнюю папку из входного ящика или может взять верхнюю папку из тележки и положить во входной ящик. Он подбрасывает монету, дабы решить, как поступить в этом случае. Попытаемся исследовать влияние его действий на исходный порядок папок.

Описанную ситуацию можно смоделировать с помощью массива указателей на строки, представляющие папки в тележке. Каждая строка содержит имя человека, информация о котором содержится папке. Для представления входного ящика можно использовать стек. Другой массив указателей можно применить для представления выходного ящика. Добавление папки во входной ящик представляется заталкиванием указателя из входного массива в стек. Обработка папки представляется выталкиванием элемента из стека и добавлением его в выходной ящик.

Учитывая важность исследования всех аспектов проблемы, будет полезно попробовать разные размеры стека. В листинге 14.15 класс `Stack<Type>` слегка переопределяется с тем, чтобы конструктор `Stack` принимал размер стека в качестве аргумента. Это приводит к внутреннему использованию динамического массива, поэтому классу теперь требуется деструктор, конструктор и операция присваивания. Такое определение сокращает код примера, поскольку некоторые методы являются встроенными.

Листинг 14.15. stcktp1.h

```
// stcktp1.h -- модифицированный шаблон Stack
#ifndef STCKTP1_H_
#define STCKTP1_H_
template <class Type>
class Stack
{
private:
   enum {SIZE = 10};    // размер по умолчанию
   int stacksize;
   Type * items;        // содержит элементы стека
   int top;             // индекс вершины стека
public:
   explicit Stack(int ss = SIZE);
   Stack(const Stack & st);
   ~Stack() { delete [] items; }
   bool isempty() { return top == 0; }
   bool isfull() { return top == stacksize; }
   bool push(const Type & item);   // добавляет элемент в стек
   bool pop(Type & item);          // выталкивает элемент с вершины стека
```

```
    Stack & operator=(const Stack & st);
};
template <class Type>
Stack<Type>::Stack(int ss) : stacksize(ss), top(0)
{
    items = new Type [stacksize];
}
template <class Type>
Stack<Type>::Stack(const Stack & st)
{
    stacksize = st.stacksize;
    top = st.top;
    items = new Type [stacksize];
    for (int i = 0; i < top; i++)
        items[i] = st.items[i];
}
template <class Type>
bool Stack<Type>::push(const Type & item)
{
    if (top < stacksize)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
}
template <class Type>
bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}
template <class Type>
Stack<Type> & Stack<Type>::operator=(const Stack<Type> & st)
{
    if (this == &st)
        return *this;
    delete [] items;
    stacksize = st.stacksize;
    top = st.top;
    items = new Type [stacksize];
    for (int i = 0; i < top; i++)
        items[i] = st.items[i];
    return *this;
}
#endif
```

Замечание по совместимости

Некоторые реализации C++ могут не распознавать ключевое слово `explicit`.

Обратите внимание, что прототип объявляет тип возвращаемого значения функции операции присваивания как ссылку на `Stack`, а существующее определение шаблонной функции задает тип как `Stack<Type>`. Первое определение типа является сокращением последнего и может использоваться только внутри области видимости класса. То есть можно использовать `Stack` внутри определения шаблонов и шаблонных функций. Вне класса, как и при определении возвращаемых типов и использовании операции разрешения контекста, необходимо применять полную форму `Stack<Type>`.

Программа в листинге 14.16 для моделирования действий Плодсона использует новое определение шаблона стека. Как и в предыдущих примерах, для генерации случайных чисел в ней используются функции `rand()`, `srand()` и `time()`. Случайно сгенерированные 0 и 1 моделируют подбрасывание монеты.

Листинг 14.16. stkoptr1.cpp

```
// stkoptr1.cpp -- тестирование стека указателей
#include <iostream>
#include <cstdlib> // для rand(), srand()
#include <ctime> // для time()
#include "stcktp1.h"
const int Num = 10;
int main()
{
   std::srand(std::time(0)); // рандомизация rand()
   std::cout << "Пожалуйста, введите размер стека: ";
   int stacksize;
   std::cin >> stacksize;
   // создает пустой стек размером stacksize
   Stack<const char *> st(stacksize);
   // входной ящик
   const char * in[Num] = {
      " 1: Hank Gilgamesh", " 2: Kiki Ishtar",
      " 3: Betty Rocker", " 4: Ian Flagranti",
      " 5: Wolfgang Kibble", " 6: Portia Koop",
      " 7: Joy Almondo", " 8: Xaverie Paprika",
      " 9: Juan Moore", "10: Misha Mache"
   };
   // выходной ящик
   const char * out[Num];
   int processed = 0;
   int nextin = 0;
   while (processed < Num)
   {
      if (st.isempty())
         st.push(in[nextin++]);
      else if (st.isfull())
         st.pop(out[processed++]);
      else if (std::rand() % 2 && nextin < Num) // шансы "50 на 50"
         st.push(in[nextin++]);
      else
```

```
      st.pop(out[processed++]);
   }
   for (int i = 0; i < Num; i++)
      std::cout << out[i] << std::endl;
   std::cout << "Всего наилучшего!\n";
   return 0;
}
```

Замечание по совместимости

Некоторые реализации C++ требуют использования `stdlib.h` вместо `cstdlib` и `time.h` вместо `ctime`.

Ниже показаны два примера выполнения программы из листинга 14.16 (благодаря фактору случайности конечный порядок расположения папок может несколько изменяться, даже если размер стека остается постоянным):

```
Пожалуйста, введите размер стека: 5
2: Kiki Ishtar
1: Hank Gilgamesh
3: Betty Rocker
5: Wolfgang Kibble
4: Ian Flagranti
7: Joy Almondo
9: Juan Moore
8: Xaverie Paprika
6: Portia Koop
10: Misha Mache
Всего наилучшего!
Пожалуйста, введите размер стека: 5
3: Betty Rocker
5: Wolfgang Kibble
6: Portia Koop
4: Ian Flagranti
8: Xaverie Paprika
9: Juan Moore
10: Misha Mache
7: Joy Almondo
2: Kiki Ishtar
1: Hank Gilgamesh
Всего наилучшего!
```

Замечания по программе

Строки в программе, представленной листинге 14.16, никогда не перемещаются. При заталкивании строки в стек просто создается новый указатель на уже существующую строку. То есть создается указатель, значение которого является адресом существующей строки. При выталкивании строки из стека этот адрес копируется в выходной массив.

В программе в качестве типа используется `const char *`, поскольку массив указателей инициализирован набором строковых констант.

Как воздействует деструктор на строки? Никак. Для создания массива, содержащего указатели, конструктор использует операцию `new`. Деструктор класса уничтожает этот массив, а не строки, на которые ссылаются элементы массива.

Пример шаблона массива и нетипизированные аргументы

Шаблоны часто используются для контейнерных классов, поскольку идея параметров типа хорошо согласуется с необходимостью создания общего плана хранения для разных типов. В самом деле, желание повторного использования кода для контейнерных классов было главным мотивом введения шаблонов. Рассмотрим другой пример и исследуем несколько новых аспектов в разработке и использовании шаблонов. В частности, рассмотрим нетипизированные аргументы, или аргументы-выражения, и применение массива для управления наследованием.

Начнем с простого шаблона массива, который позволяет определять размер массива. Одна из техник, использованных в последней версии шаблона `Stack`, предусматривает организацию динамического массива внутри класса и аргумента в конструкторе для задания количества элементов. Другой подход состоит в применении аргумента в шаблоне для задания размера обычного массива. В листинге 14.17 показано, как это можно сделать.

Листинг 14.17. `arraytp.h`

```
//arraytp.h -- шаблон массива
#ifndef ARRAYTP_H_
#define ARRAYTP_H_
#include <iostream>
#include <cstdlib>
template <class T, int n>
class ArrayTP
{
private:
   T ar[n];
public:
   ArrayTP() {};
   explicit ArrayTP(const T & v);
   virtual T & operator[](int i);
   virtual T operator[](int i) const;
};
template <class T, int n>
ArrayTP<T,n>::ArrayTP(const T & v)
{
   for (int i = 0; i < n; i++)
      ar[i] = v;
}
template <class T, int n>
T & ArrayTP<T,n>::operator[](int i)
{
   if (i < 0 || i >= n)
   {
      std::cerr << "Ошибка в пределах массива: " << i
                << " выходит за пределы диапазона\n";
      std::exit(EXIT_FAILURE);
   }
   return ar[i];
}
```

```
template <class T, int n>
T ArrayTP<T,n>::operator[](int i) const
{
   if (i < 0 || i >= n)
   {
      std::cerr << "Ошибка в пределах массива: " << i
         << " выходит за пределы диапазона\n";
      std::exit(EXIT_FAILURE);
   }
   return ar[i];
}
#endif
```

Обратите внимание на следующий заголовок в листинге 14.17:

```
template <class T, int n>
```

Ключевое слово `class` (или `typename`, что в данном контексте эквивалентно) объявляет `T` как параметр типа, или аргумент типа. `int` объявляет `n` с типом `int`. Этот вид параметра, определяющий конкретный тип вместо общего имени типа, называется *нетипизированным параметром*, или *параметром-выражением*. Предположим, что имеется следующее определение:

```
ArrayTP<double, 12> eggweights;
```

Оно заставит компилятор определить класс `ArrayTP<double,12>` и создать объект `eggweights` этого класса. При определении класса компилятор заменит `T` на `double` и `n` на `12`.

Аргументы-выражения имеют некоторые ограничения. Аргумент-выражение может быть целого типа, перечислимого типа, ссылкой или указателем. Так, `double m` — неверное объявление, а `double & rm` и `double * pm` — допустимы. Также код шаблона не может изменять значение аргумента или затрагивать его адрес. Например, в шаблоне `ArrayTP` выражения `n++` или `&n` не допускаются. При инициализации шаблона значение, используемое для аргумента-выражения, должно быть константным.

Подход с заданием размера массива обладает одним преимуществом перед вариантом с конструктором, применяемым в `Stack`. Вариант с конструктором использует кучу в памяти, управляемую `new` и `delete`, тогда как подход с аргументом-выражением использует стек памяти, поддерживаемый для автоматических переменных. Это работает быстрее, особенно если имеется много небольших массивов.

Главный недостаток подхода с аргументами-выражениями состоит в том, что для каждого размера массива создается собственный шаблон. Так, определения:

```
ArrayTP<double, 12> eggweights;
ArrayTP(double, 13> donuts;
```

создают два раздельных определения класса. Однако определения:

```
Stack<int> eggs(12);
Stack<int> dunkers(13);
```

создают только одно определение класса, и информация о размере передается конструктору этого класса.

Другое отличие заключается в том, что подход с конструктором более гибок, потому что размер массива хранится как член класса, а не жестко закодирован в определении

класса. Поэтому можно, например, определить присваивание массива одного размера массиву другого размера или создать класс с массивами переменной размерности.

Универсальность шаблонов

В шаблонных классах можно применять те же техники программирования, что и в обычных классах. Шаблонные классы могут служить в качестве базовых классов, и они же могут быть компонентными классами. Например, можно создать шаблон стека, используя шаблон массива. Или можно взять шаблон массива и применить его для создания массива, элементы которого являются стеками, основанными на шаблоне стека. Рассмотрим следующий код:

```
template <class T>
class Array
{
private:
   T entry;
   ...
};
template <class Type>
class GrowArray : public Array<Type> {...}; // наследование
template <class Tp>
class Stack
{
   Array<Tp> ar;          // использует Array<> в качестве компонента
   ...
};
...
Array < Stack<int> > asi; // массив стеков значений int
```

В последнем операторе, во избежание конфликта с операцией >>, необходимо разделить два символа > как минимум одним пробелом.

Рекурсивное использование шаблонов

Другим примером универсальности шаблонов является возможность рекурсивного использования шаблонов. Например, приведенное ранее определение шаблона массива можно использовать следующим образом:

```
ArrayTP< ArrayTP<int,5>, 10> twodee;
```

Здесь создается массив `twodee`, состоящий из 10 элементов, каждый из которых, в свою очередь, является массивом из пяти целых (`int`). Эквивалентный обычный массив определяется следующим образом:

```
int twodee[10][5];
```

Заметим, что в синтаксисе шаблона размерности массива приведены в порядке, отличном от обычного двумерного массива. Эта идея задействована в программе, показанной в листинге 14.18. Там же для создания одномерного массива, содержащего суммы и средние значения для каждого из десяти наборов по пять чисел, применяется шаблон `ArrayTP`. Вызов метода `cout.width(2)` приводит к отображению следующего элемента массива в виде двух символов, до тех пор, пока не потребуется большая длина поля для отображения всего числа.

Листинг 14.18. twod.cpp

```
// twod.cpp -- создание двумерного массива
#include <iostream>
#include "arraytp.h"
int main(void)
{
   using std::cout;
   using std::endl;
   ArrayTP<int, 10> sums;
   ArrayTP<double, 10> aves;
   ArrayTP< ArrayTP<int,5>, 10> twodee;
   int i, j;
   for (i = 0; i < 10; i++)
   {
      sums[i] = 0;
      for (j = 0; j < 5; j++)
      {
         twodee[i][j] = (i + 1) * (j + 1);
         sums[i] += twodee[i][j];
      }
      aves[i] = (double) sums[i] / 10;
   }
   for (i = 0; i < 10; i++)
   {
      for (j = 0; j < 5; j++)
      {
         cout.width(2);
         cout << twodee[i][j] << ' ';
      }
      cout << ": сумма = ";
      cout.width(3);
      cout << sums[i] << ", среднее = " << aves[i] << endl;
   }
   cout << "Готово.\n";
   return 0;
}
```

В выводе программы из листинга 14.18 для каждого из 10 элементов twodee, в свою очередь, являющихся 5-элементными массивами, предусмотрена одна строка:

```
 1 2 3 4 5 : сумма = 15, среднее = 1.5
 2 4 6 8 10 : сумма = 30, среднее = 3
 3 6 9 12 15 : сумма = 45, среднее = 4.5
 4 8 12 16 20 : сумма = 60, среднее = 6
 5 10 15 20 25 : сумма = 75, среднее = 7.5
 6 12 18 24 30 : сумма = 90, среднее = 9
 7 14 21 28 35 : сумма = 105, среднее = 10.5
 8 16 24 32 40 : сумма = 120, среднее = 12
 9 18 27 36 45 : сумма = 135, среднее = 13.5
10 20 30 40 50 : сумма = 150, среднее = 15
Готово.
```

Использование нескольких параметров типа

Допускается создание шаблонов с несколькими параметрами типа. Предположим, что требуется класс, содержащий два вида значений. Для этой цели можно создать шаблонный класс Pair. (Между прочим, STL содержит подобный шаблон, именуемый pair.) Короткий пример показан в листинге 14.19. В нем методы first() const и second() const печатают хранимые значения, а методы first() и second(), благодаря возврату ссылок на данные-члены класса Pair, позволяют присвоить хранимым величинам новые значения.

Листинг 14.19. pairs.cpp

```
// pairs.cpp -- определение и использование шаблона Pair
include <iostream>
#include <string>
template <class T1, class T2>
class Pair
{
private:
   T1 a;
   T2 b;
public:
   T1 & first();
   T2 & second();
   T1 first() const { return a; }
   T2 second() const { return b; }
   Pair(const T1 & aval, const T2 & bval) : a(aval), b(bval) { }
   Pair() {}
};
template<class T1, class T2>
T1 & Pair<T1,T2>::first()
{
   return a;
}
template<class T1, class T2>
T2 & Pair<T1,T2>::second()
{
   return b;
}
int main()
{
   using std::cout;
   using std::endl;
   using std::string;
   Pair<string, int> ratings[4] =
   {
      Pair<string, int>("The Purple Duke", 5),
      Pair<string, int>("Jake's Frisco Al Fresco", 4),
      Pair<string, int>("Mont Souffle", 5),
      Pair<string, int>("Gertie's Eats", 3)
   };
   int joints = sizeof(ratings) / sizeof (Pair<string, int>);
   cout << "Рейтинг:\t Закусочная\n";
```

```
    for (int i = 0; i < joints; i++)
        cout << ratings[i].second() << ":\t "
             << ratings[i].first() << endl;
    cout << "Ой! Пересмотренный рейтинг:\n";
    ratings[3].first() = "Gertie's Fab Eats";
    ratings[3].second() = 6;
    cout << ratings[3].second() << ":\t "
         << ratings[3].first() << endl;
    return 0;
}
```

Обратите внимание, что в листинге 14.19 в функции `main()` для вызова конструкторов и в качестве аргумента для `sizeof` необходимо использовать `Pair<string,int>`. Это потому, что `Pair<string,int>`, а не `Pair` является именем класса. Например, `Pair<char *,double>` представляет собой имя уже совершенно другого класса.

Ниже показан вывод программы из листинга 14.19:

```
Рейтинг:   Закусочная
5:         The Purple Duke
4:         Jake's Frisco Al Fresco
5:         Mont Souffle
3:         Gertie's Eats
Ой! Пересмотренный рейтинг:
6:         Gertie's Fab Eats
```

Тип параметров шаблона по умолчанию

Другим новым свойством шаблонных классов является возможность определения типов параметров по умолчанию:

```
template <class T1, class T2 = int> class Topo {...};
```

Это заставляет компилятор использовать `int` в качестве типа `T2` если значение `T2` опущено:

```
Topo<double, double> m1;  // T1 является double, T2 является double
Topo<double> m2;          // T1 является double, T2 является int
```

Это свойство часто используется в STL (рассматривается в главе 16) для типов по умолчанию существующих классов. Можно задать значения по умолчанию для типов параметров шаблонных классов. Для шаблонных функций этого делать нельзя. Тем не менее, можно задать значения по умолчанию нетипизированных параметров как для шаблонных классов, так и для шаблонных функций.

Специализация шаблона

В шаблонах классов, как и в шаблонах функций, существуют неявная реализация, явная реализация и явная специализация, в общем известные под названием *специализации*. В то время как шаблоны описывают классы в терминах общих типов, специализация является определением класса, использующим специальные типы.

Неявная реализация

В примерах шаблонов, приводимых в этой главе, используется *неявная реализация*. При этом определяется один или более объектов требуемого типа, и компилятор генерирует специализированное определение класса, используя правила задаваемые общим шаблоном:

```
ArrayTP<int, 100> stuff;        // неявная реализация
```

Компилятор не создает неявную реализацию класса до тех пор, пока не потребуется объект:

```
ArrayTP<double, 30> * pt;       // указатель, пока объекты не нужны
pt = new ArrayTP<double, 30>;   // теперь требуется объект
```

Второй оператор заставляет компилятор генерировать определение класса и объект в соответствии с этим определением.

Явная реализация

Компилятор генерирует *явную реализацию* класса, если класс определен и с помощью ключевого слова `template`, и указанием желаемого типа или типов. Объявление класса должно находиться в том же пространстве имен, что и определение шаблона. Например:

```
template class ArrayTP<string, 100>;//генерирует класс ArrayTP<string, 100>
```

объявляет `ArrayTP<string, 100>` как класс. В этом случае компилятор генерирует определение класса, включая определение методов, даже если при этом не создаются объекты класса. Как и в случае неявной реализации, общий шаблон служит образцом для создания специализации.

Явная специализация

Явная специализация – это определение специального типа или типов, используемых вместо общего шаблона. Иногда необходимо изменить шаблон с тем, чтобы он вел себя по другому при присваивании значения определенному типу; в этом случае можно создать явную специализацию. Предположим, что определен шаблон класса, представляющий отсортированный массив, элементы которого сортируются при добавлении к нему:

```
template <class T>
class SortedArray
{
...// детали не показаны
};
```

Предположим также, что шаблон использует операцию > для сравнения значений. Это хорошо работает для чисел. Это также работает, если `T` является типом класса при условии, что определен метод `T::operator>()`. Однако это не работает, если `T` является строкой, представляемой типом `char *`. Собственно, шаблон будет работать, но строки будут отсортированы не по алфавиту, а по адресам. Все что необходимо — это определение класса, которое использует `strcmp()` вместо >. В этом случае можно применить явную специализацию шаблона. Это имеет вид шабло-

на, определенного для одного определенного типа вместо общего типа. При необходимости выбора между специализированным шаблоном и общим шаблоном, удовлетворяющим запросу специализации, компилятор использует специализированную версию.

Определение специализированного шаблона класса имеет вид:

```
template <> class Classname<имя-специализированного-типа> { ... };
```

Устаревшие компиляторы могут распознавать только ранние формы, не содержащие угловых скобок <>:

```
class Classname<имя-специализированного-типа> { ... };
```

Для создания шаблона `SortedArray`, специализированного для типа `char *`, с использованием новой нотации нужно применять следующий код:

```
template <> class SortedArray<char *>
{
...// детали не указаны
};
```

Здесь для сравнения значений массива код реализации должен использовать `strcmp()` вместо >. Теперь запросы шаблона `SortedArray`, содержащего `char *`, будут применять специализированные определения вместо более общих определений шаблона:

```
SortedArray<int> scores;    //использовать общее определение
SortedArray<char *> dates;  //использовать специализированное определение
```

Частичная специализация

В C++ разрешена *частичная специализация*, которая частично ограничивает общность шаблона. Например, используя частичную специализацию, можно задать конкретный тип для одного из типизированных параметров:

```
// общий шаблон
template <class T1, class T2> class Pair {...};
// специализация, в которой T2 установлен в int
template <class T1> class Pair<T1, int> {...};
```

Угловые скобки <>, следующие за ключевым словом `template`, объявляют типизированные параметры, которые пока еще не специализированы. Следующее определение специализирует `T2` в `int` и оставляет `T1` открытым. Отметим, что спецификация всех типов приводит к пустым угловым скобкам и завершенной явной специализации:

```
// специализация T1 и T2 установлена в int
template <> class Pair<int, int> {...};
```

Если у компилятора есть выбор, он использует наиболее специализированный шаблон:

```
Pair<double, double> p1; // использовать общий шаблон Pair
Pair<double, int> p2;    // использовать частичный шаблон Pair<T1, int>
Pair<int, int> p3;       // использовать явную специализацию Pair<int, int>
```

Можно частично специализировать существующий шаблон за счет введения специальной версии указателей:

```
template<class T>   // общая версия
class Feeb { ... };
template<class T*>  // частичная специализация
class Feeb { ... }; // измененный код
```

Если использовать не ссылочный тип, компилятор выберет общую версию; если использовать указатель, компилятор отдаст предпочтение ссылочной специализации:

```
Feeb<char> fb1;     // использует общий шаблон Feeb, T - это char
Feeb<char *> fb2;   // использует Feeb T* специализацию, T — это char
```

Без частичной специализации во втором определении будет применен общий шаблон, интерпретирующий `T` как тип `char *`. При использовании частичной специализации будет выбран специализированный шаблон, интерпретирующий `T` как тип `char`.

Частичная специализация позволяет преодолеть многие ограничения. Например, могут существовать следующие операторы:

```
// общий шаблон
template <class T1, class T2, class T3> class Trio{...};
// специализация T3 в T2
template <class T1, class T2> class Trio<T1, T2, T2> {...};
// специализация T3 и T2 в T1*
template <class T1> class Trio<T1, T1*, T1*> {...};
```

Встретив такое определение, компилятор выберет следующие варианты:

```
Trio<int, short, char *> t1;    // использует общий шаблон
Trio<int, short> t2;            // использует Trio<T1, T2, T2>
Trio<char, char *, char *> t3;  // использует Trio<T1, T1*, T1*>
```

Члены-шаблоны

Еще одно расширение C++ в части поддержки шаблонов состоит в том, что шаблоны могут быть членами структуры, класса или шаблонного класса. Эти свойства необходимы STL для полного определения своей структуры. В листинге 14.20 показан короткий пример шаблонного класса со встроенными в качестве членов шаблонным классом и шаблонной функцией.

Листинг 14.20. `tempmemb.cpp`

```
// tempmemb.cpp -- шаблонные члены
#include <iostream>
using std::cout;
using std::endl;

template <typename T>
class beta
{
private:
   template <typename V> // встроенный шаблонный класс-член
```

```
    class hold
    {
    private:
       V val;
    public:
       hold(V v = 0) : val(v) {}
       void show() const { cout << val << endl; }
       V Value() const { return val; }
    };
    hold<T> q;              // шаблонный объект
    hold<int> n;            // шаблонный объект
public:
    beta( T t, int i) : q(t), n(i) {}
    template<typename U>   // шаблонный метод
    U blab(U u, T t) { return (n.Value() + q.Value()) * u / t; }
    void Show() const { q.show(); n.show();}
};
int main()
{
    beta<double> guy(3.5, 3);
    guy.Show();
    cout << guy.blab(10, 2.3) << endl;
    cout << "Готово.\n";
    return 0;
}
```

Шаблон hold определен в приватном разделе, поэтому он имеет область видимости в пределах класса beta. Класс beta использует шаблон hold для определения двух членов данных:

```
hold<T> q;     // шаблонный объект
hold<int> n;   // шаблонный объект
```

n является объектом hold, основанном на типе int, а член q — объектом hold, основанном на типе T (параметр beta шаблона). В функции main() определение

```
beta<double> guy(3.5, 3);
```

присваивает T тип double и q тип hold<double>.

Метод blab() имеет один тип (U), определенный неявно, путем задания значения аргумента при вызове метода, и другой тип (T), определенный типом реализации объекта. В данном примере определение для guy назначает T тип double. Первый аргумент в вызове метода в

```
cout << guy.blab(10, 2.5) << endl;
```

назначает U тип int, соответствующий значению 10. Таким образом, хотя автоматическое преобразование типов, вызываемое смешанными типами, приводит к вычислению blab() как double, возвращаемая величина, имеющая тип U, будет целым (int), что и подтверждает вывод программы:

```
3.5
3
28
Готово.
```

Если заменить 10 на 10.0 в вызове guy.blab(), U будет иметь тип double, и выходной результат будет типа double:

```
3.5
3
28.2609
Готово.
```

Как говорилось выше, тип второго параметра установлен в double в определении объекта guy. В отличие от первого параметра, тип второго параметра не устанавливается вызовом функции. Например, оператор

```
cout << guy.blab(10, 3) << endl;
```

определит blah() как blah(int, double), и значение 3 будет преобразовано в тип double по обычным правилам прототипирования функций.

Можно объявить класс hold и метод blah в шаблоне beta и определить их вне шаблона beta. Тем не менее, поскольку реализация шаблонов остается новым процессом, возможно, будет трудно заставить компилятор воспринять такую форму. Некоторые компиляторы могут вообще не воспринимать шаблонные члены. Другие же, как показано в листинге 14.20, могут воспринимать такую форму, но не допускают определения вне класса. Если же вы имеете подходящий компилятор, ниже показано, как определить методы шаблона за пределами шаблона beta:

```
template <typename T>
class beta
{
private:
   template <typename V> // declaration
   class hold;
   hold<T> q;
   hold<int> n;
public:
   beta( T t, int i) : q(t), n(i) {}
   template<typename U> // declaration
   U blab(U u, T t);
   void Show() const { q.show(); n.show();}
};
// определение члена
template <typename T>
template<typename V>
class beta<T>::hold
{
private:
   V val;
public:
   hold(V v = 0) : val(v) {}
   void show() const { std::cout << val << std::endl; }
   V Value() const { return val; }
};
// определение члена
template <typename T>
template <typename U>
```

```
U beta<T>::blab(U u, T t)
{
   return (n.Value() + q.Value()) * u / t;
}
```

В определении T, V и U являются параметрами шаблона. Поскольку шаблоны встроенные, необходимо использовать синтаксис

```
template <typename T>
template <typename V>
```

вместо синтаксиса

```
template<typename T, typename V>
```

hold и blab в определении должны быть заданы как члены класса beta<T> с использованием операции разрешения контекста.

Шаблоны как параметры

Мы уже видели, что шаблоны могут иметь параметры типа, такие как typename T, и нетипизированные параметры вроде int n. Шаблоны также могут иметь параметры, которые сами являются шаблонами. Это еще одно расширение, реализованное в STL.

В листинге 14.21 показан пример, в котором шаблонный параметр представлен как template <typename T>class Thing. Здесь template <typename T> class — это тип, а Thing — параметр. Что это означает? Предположим, имеется объявление

```
Crab<King> legs;
```

Для того чтобы это работало, аргумент шаблона King должен быть шаблонным классом, определение которого должно соответствовать параметру шаблона Thing:

```
template <typename T>
class King {...};
```

Определение Crab объявляет два объекта:

```
Thing<int> s1;
Thing<double> s2;
```

Предыдущее объявление для legs приводило к замене King<int> на Thing<int> и King<double> на Thing<double>. Однако, в листинге 14.21 приведено следующее определение

```
Crab<Stack> nebula;
```

Поэтому Thing<int> реализовано как Stack<int>, и Thing<double> – как Stack<double>. Короче говоря, параметр шаблона Thing меняется на любой шаблонный тип, используемый в качестве шаблонного аргумента в объявлении объекта Crab.

В определении класса Crab сделано три допущения о шаблонном классе, представляемом параметром Thing. Класс должен иметь метод push(), класс должен иметь метод pop(), и эти методы должны иметь определенный интерфейс. Класс Crab может использовать любой шаблонный класс, который соответствует типу Ting и обязательно имеет методы push() и pop(). В этой главе рассматривается один такой

класс — шаблон Stack, определенный в stacktp.h. Этот класс и используется в примере.

Листинг 14.21. tempparm.cpp

```
// tempparm.cpp -- шаблоны как параметры
#include <iostream>
#include "stacktp.h"
template <template <typename T> class Thing>
class Crab
{
private:
   Thing<int> s1;
   Thing<double> s2;
public:
   Crab() {};
   // предположим, что класс thing имеет члены push() и pop()
   bool push(int a, double x) { return s1.push(a) && s2.push(x); }
   bool pop(int & a, double & x){ return s1.pop(a) && s2.pop(x); }
};
int main()
{
   using std::cout;
   using std::cin;
   using std::endl;
   Crab<Stack> nebula;
   // Stack должен соответствовать шаблону <typename T> класса thing
   int ni;
   double nb;
   cout << "Введите пары пары int и double, такие как 4 3.5 (0 0 для завершения):\n";
   while (cin>> ni >> nb && ni > 0 && nb > 0)
   {
      if (!nebula.push(ni, nb))
         break;
   }
   while (nebula.pop(ni, nb))
      cout << ni << ", " << nb << endl;
   cout << "Готово.\n";
   return 0;
}
```

Ниже приведен пример вывода программы из листинга 12.21:

```
Введите пары пары int и double, такие как 4 3.5 (0 0 для завершения):
50 22.48
25 33.87
60 19.12
0 0
60, 19.12
25, 33.87
50, 22.48
Done.
```

Можно смешивать шаблонные параметры с обычными параметрами. Например, определение класса Crab может начинаться так:

```
template <template <typename T> class Thing, typename U, typename V>
class Crab
{
private:
   Thing<U> s1;
   Thing<V> s2;
   ...
```

Сейчас типы, сохраняемые в членах s1 и s2, являются общими типами, а не жестко закодированными. Для этого в программе потребуется изменить определение nebula следующим образом:

```
Crab<Stack, int, double> nebula; // T=Stack, U=int, V=double
```

Шаблонный параметр T является шаблонным типом, а параметры типа U и V — нешаблонными типами.

Шаблонные классы и друзья

Определения шаблонных классов могут также содержать друзей. Друзей шаблонов можно классифицировать на три категории:

- Нешаблонные друзья.
- Ограниченные шаблонные друзья; то есть тип друга определяется типом класса при его реализации.
- Неограниченные шаблонные друзья; то есть все специализации друга являются друзьями для всех специализаций класса.
- Рассмотрим пример для каждого случая.

Нешаблонные дружественные функции для шаблонных классов

Объявим в качестве друга обычную функцию в шаблонном классе:

```
template <class T>
class HasFriend
{
friend void counts(); // дружественная для всех реализаций HasFriend
...
};
```

В этом определении функция counts() становится дружественной для всех возможных реализаций шаблона. Например, она будет дружественной для класса HasFriend<int> и для класса HasFriend<string>.

Функция counts() не вызывается из объекта (она друг, а не член) и она не имеет параметров объекта. Как же она получает доступ к объекту HasFriend? Существует несколько возможностей. Она может иметь доступ к глобальному объекту; она может иметь доступ к локальному объекту через глобальный указатель; она может создать собственные объекты; она может иметь доступ к статическим членам-данным, существующим отдельно от объекта. Предположим, требуется создать аргумент типа

шаблонного класса для дружественной функции. Можно ли иметь, например, следующее определение друга?

```
friend void report(HasFriend &);          // возможно ли?
```

Ответ — нельзя. Дело в том, что объект HasFriend не существует. Существует только частная специализация HasFriend<short>. Для создания аргумента типа шаблонного класса необходимо указать специализацию. Например, следующим образом:

```
template <class T>
class HasFriend
{
   friend void report(HasFriend<T> &);   // связать друга шаблона
   ...
};
```

Для того чтобы понять, что здесь происходит, представьте специализацию, выполняемую при объявлении объекта определенного типа:

```
HasFriend<int> hf;
```

Компилятор заменит шаблонный параметр T на int, придав определению друга следующую форму:

```
class HasFriend<int>
{
   friend void report(HasFriend<int> &); // связать друга шаблона
   ...
};
```

Таким образом, report() с параметром HasFriend<int> станет другом для класса HasFriend<int>. Аналогично, report() с параметром HasFriend<double> станет перегруженной версией report(), являющейся другом для класса HasFriend<double>.

Отметим, что report() не является шаблонной функцией. У нее только есть параметр, являющийся шаблоном. Это означает, что для создания друга необходимо использовать явную специализацию:

```
void report(HasFriend<short> &) { ... }; // явная специализация для short
void report(HasFriend<int> &) { ... };   // явная специализация для int
```

В листинге 14.22 иллюстрируется этот момент. У шаблона HasFriend имеется статический член ct. Это означает, что в любой частной специализации класса присутствует свой собственный статический член. Метод counts(), являющийся другом для всех специализаций HasFriend, передает значение ct двум частным специализациям: HasFriend<int> и HasFriend<double>. В программе создаются также две функции report(), каждая из которых является другом для одной частной специализации HasFriend.

Листинг 14.22. frnd2tmp.cpp

```
// frnd2tmp.cpp -- шаблонный класс без нешаблонных друзей
#include <iostream>
using std::cout;
using std::endl;
```

```
template <typename T>
class HasFriend
{
private:
   T item;
   static int ct;
public:
   HasFriend(const T & i) : item(i) {ct++;}
   ~HasFriend() {ct--; }
   friend void counts();
   friend void reports(HasFriend<T> &); // шаблонный параметр
};

// каждая специализация имеет свои собственные
// статические данные-члены
template <typename T>
int HasFriend<T>::ct = 0;
// нешаблонный друг для всех классов HasFriend<T>
void counts()
{
   cout << "счетчик int: " << HasFriend<int>::ct << "; ";
   cout << "счетчик double: " << HasFriend<double>::ct << endl;
}
// нешаблонный друг для класса HasFriend<int>
void reports(HasFriend<int> & hf)
{
   cout <<"HasFriend<int>: " << hf.item << endl;
}
// нешаблонный друг для класса HasFriend<double>
void reports(HasFriend<double> & hf)
{
   cout <<"HasFriend<double>: " << hf.item << endl;
}

int main()
{
   cout << "Нет объявленных объектов: ";
   counts();
   HasFriend<int> hfi1(10);
   cout << "После объявления hfi1: ";
   counts();
   HasFriend<int> hfi2(20);
   cout << "После объявления hfi2: ";
   counts();
   HasFriend<double> hfdb(10.5);
   cout << "После объявления hfdb: ";
   counts();
   reports(hfi1);
   reports(hfi2);
   reports(hfdb);
   return 0;
}
```

Ниже показан вывод программы из листинга 14.22:

```
Нет объявленных объектов: счетчик int: 0; счетчик double: 0
После объявления hfi1: счетчик int: 1; счетчик double: 0
После объявления hfi2: счетчик int: 2; счетчик double: 0
После объявления hfdb: счетчик int: 2; счетчик double: 1
HasFriend<int>: 10
HasFriend<int>: 20
HasFriend<double>: 10.5
```

Замечание по совместимости

Некоторые компиляторы C++ не могут работать с друзьями шаблонов.

Связанные шаблонные функции-друзья для шаблонных классов

Можно изменить рассмотренный пример, сделав дружественные функции шаблонами. В частности, можно сделать так, что каждая специализация класса получит соответствующую специализацию друга. Эта техника чуть более сложная, чем для нешаблонных друзей, и состоит из трех шагов.

На первом шаге перед объявлением класса необходимо объявить каждую шаблонную функцию:

```
template <typename T> void counts();
template <typename T> void report(T &);
```

Далее, нужно снова внутри функции объявить шаблоны в качестве друзей. Код, приведенный ниже, определяет специализацию, основанную на типе параметра шаблонного класса:

```
template <typename TT>
class HasFriendT
{
   ...
   friend void counts<TT>();
   friend void report<>(HasFriendT<TT> &);
};
```

Угловые скобки <> в объявлении класса определяют его как специализацию шаблона. В случае `report()` скобки <> могут быть незначащими слева? поскольку аргумент шаблонного типа

```
HasFriendT<TT>
```

может быть получен из аргумента функции. Тем не менее, в качестве альтернативы можно использовать

```
report<HasFriendT<TT> >(HasFriendT<TT> &)
```

Функция `counts()` не имеет параметров, поэтому для определения ее специализации нужно применять синтаксис шаблонного аргумента (`<TT>`). Отметим также, что `TT` является типом параметра для класа `HasFriendT`.

Проще всего понять эти определения, представив, чем они станут при объявлении объекта методом частной специализации. Допустим, что определен следующий объект:

```
HasFriendT<int> squack;
```

Компилятор подставит int вместо TT и сгенерирует показанное ниже класса:

```
class HasFriendT<int>
{
   ...
   friend void counts<int>();
   friend void report<>(HasFriendT<int> &);
};
```

Одна специализация основана на TT, которое преобразуется в int, а другая — на HasFriendT<TT>, преобразуемое в HasFriendT<int>. Таким образом, специализации шаблона counts<int>() и report<HasFriendT<int> >() объявлены как друзья класса HasFriendT<int>.

Третье требование, которому должна удовлетворять программа — необходимость определения шаблонов друзей. Все три рассматриваемых аспекта отражены в листинге 14.23. Обратите внимание, что в листинге 14.22 присутствует одна функция counts(), являющаяся другом для всех классов HasFriend. В листинге 14.23 имеются две функции counts(), каждая из которых является другом каждому реализуемому типу класса. Поскольку вызовы функции counts() не имеют параметров, из которых компилятор мог бы извлечь требуемую специализацию, в этих вызовах используются формы count<int>() и count<double>(). Для вызовов reports() компилятор при определении специализации может применять тип параметра. С таким же успехом можно использовать форму <>:

```
report<HasFriendT<int> >(hfi2); //то же, что и report(hfi2);
```

Листинг 14.23. tmp2tmp.cpp

```
// tmp2tmp.cpp -- шаблонные друзья для шаблонного класса
#include <iostream>
using std::cout;
using std::endl;

// шаблонные прототипы
template <typename T> void counts();
template <typename T> void report(T &);

// шаблонный класс
template <typename TT>
class HasFriendT
{
private:
   TT item;
   static int ct;
public:
   HasFriendT(const TT & i) : item(i) {ct++;}
   ~HasFriendT() { ct--; }
   friend void counts<TT>();
   friend void report<>(HasFriendT<TT> &);
};

template <typename T>
int HasFriendT<T>::ct = 0;
```

```
// определение шаблонных дружественных функций
template <typename T>
void counts()
{
   cout << "размер шаблона: " << sizeof(HasFriendT<T>) << "; ";
   cout << "counts() из шаблона: " << HasFriendT<T>::ct << endl;
}
template <typename T>
void report(T & hf)
{
   cout << hf.item << endl;
}
int main()
{
   counts<int>();
   HasFriendT<int> hfi1(10);
   HasFriendT<int> hfi2(20);
   HasFriendT<double> hfdb(10.5);
   report(hfi1); // генерирует report(HasFriendT<int> &)
   report(hfi2); // генерирует report(HasFriendT<int> &)
   report(hfdb); // генерирует report(HasFriendT<double> &)
   cout << "вывод counts<int>():\n";
   counts<int>();
   cout << "вывод counts<double>():\n";
   counts<double>();
   return 0;
}
```

Ниже показан вывод этой программы:

```
template size: 4; template counts(): 0
10
20
10.5
вывод counts<int>():
размер шаблона: 4; counts() из шаблона: 2
вывод counts<double>():
размер шаблона: 8; counts()из шаблона: 1
```

Как видно, `counts<double>` выводит размер шаблона, отличный от выводимого `counts<int>`. То есть, каждый тип `T` имеет собственную дружественную функцию `count()`.

Несвязанные шаблонные функции-друзья для шаблонных классов

В предыдущем разделе связанные шаблонные дружественные функции являются специализациями для шаблона, определенного вне класса. Специализация `int` класса получает специализацию функции `int` и так далее. Объявив шаблон внутри класса, можно создать несвязанные дружественные функции, для которых каждая специализация функции будет дружественной для каждой специализации класса. У несвязанных друзей параметры типа шаблонов друзей отличаются от параметров типа шаблонных классов:

```
template <typename T>
class ManyFriend
{
   ...
   template <typename C, typename D> friend void show2(C &, D &);
};
```

В листинге 14.24 приведен пример применения несвязанных друзей. В этом примере вызов функции `show2(hfi1, hfi2)` соответствует следующей специализации:

```
void show2<ManyFriend<int> &, ManyFriend<int> &>
        (ManyFriend<int> & c, ManyFriend<int> & d);
```

Поскольку этот вызов дружественен всем специализациям `ManyFriend`, эта функция имеет доступ к членам `item` всех специализаций. Однако она использует доступ только к объектам `ManyFriend<int>`.

Аналогично `show2(hfd, hfi2)` соответствует специализации:

```
void show2<ManyFriend<double> &, ManyFriend<int> &>
        (ManyFriend<double> & c, ManyFriend<int> & d);
```

Вызов также дружественен всем специализациям `ManyFriend` и использует доступ к члену `item` объекта `ManyFriend<int>` и к члену `item` объекта `ManyFriend<double>`.

Листинг 14.24. manyfrnd.cpp

```
// manyfrnd.cpp -- несвязанный шаблонный друг для шаблонного класса
#include <iostream>
using std::cout;
using std::endl;
template <typename T>
class ManyFriend
{
private:
   T item;
public:
   ManyFriend(const T & i) : item(i) {}
   template <typename C, typename D> friend void show2(C &, D &);
};
template <typename C, typename D> void show2(C & c, D & d)
{
   cout << c.item << ", " << d.item << endl;
}
int main()
{
   ManyFriend<int> hfi1(10);
   ManyFriend<int> hfi2(20);
   ManyFriend<double> hfdb(10.5);
   cout << "hfi1, hfi2: ";
   show2(hfi1, hfi2);
   cout << "hfdb, hfi2: ";
   show2(hfdb, hfi2);
   return 0;
}
```

Вот как выглядит вывод программы из листинга 14.24:

```
hfi1, hfi2: 10, 20
hfdb, hfi2: 10.5, 20
```

Замечание по совместимости

Некоторые компиляторы C++ не могут работать с друзьями шаблонов.

Резюме

В C++ существует несколько способов повторного использования кода. Общедоступное наследование, рассмотренное в главе 13, приводит к созданию отношения *is-a* и позволяет производным классам повторно использовать код базовых классов. Приватное и защищенное наследование также позволяет повторно использовать код базовых классов, но в этом случае имеет место отношение *has-a*. При приватном наследовании общедоступные и защищенные члены базового класса становятся приватными членами производного класса. При защищенном наследовании общедоступные и защищенные члены базового класса становятся защищенными членами производного класса. Таким образом, в обоих случаях общедоступный интерфейс базового класса становится внутренним интерфейсом для производного класса. Иногда это называют наследованием реализации, а не интерфейса, поскольку производный объект не может явно использовать интерфейс базового класса. Поэтому нельзя считать производный объект разновидностью базового объекта. По той же причине указатель или ссылку на базовый объект нельзя применять для ссылки на объект производного класса без явного преобразования типов.

Другая возможность повторного использования кода класса предусматривает разработку класса, члены которого сами являются объектами. Этот подход называется *включением*, *иерархическим представлением* или *композицией*, и также приводит к созданию отношения *has-a*. Включение проще, нежели приватное и защищенное наследование, и поэтому предпочтительнее использовать именно его. И все же приватное и защищенное наследование предоставляет дополнительные возможности. Например, наследование позволяет производному классу получить доступ к защищенным членам базового класса. Оно также позволяет производному классу переопределить виртуальные функции, унаследованные от базового класса. Поскольку включение не является разновидностью наследования, то оно таких возможностей не обеспечивает. С другой стороны, если нужно создать несколько объектов одного класса, предпочтительней применять включение. Например, класс `State` может иметь массив объектов `Country`.

Множественное наследование при создании класса позволяет использовать код нескольких классов. Приватное и защищенное множественное наследование приводит к созданию отношений *has-a*, а общедоступное множественное наследование — к созданию отношений *is-a*. Применение множественного наследования приводит к возникновению трудностей, связанных с неоднозначностью имен и неоднозначным наследованием базового класса. Для разрешения неоднозначности имен нужно использовать квалификатор класса, а для преодоления неоднозначности наследования — виртуальные базовые классы. Использование виртуальных базовых классов вводит новые правила для списка инициализации конструктов и для разрешения неоднозначности.

Шаблоны класса позволяют создать общую структуру класса, в которой тип, обычно тип члена, представляется параметром типа. Обычно шаблон выглядит следующим образом:

```
template <class T>
class Ic
{
T v;
...
public:
Ic(const T & val) : v(val) { }
...
};
```

Здесь T является параметром типа, и работает как заполнитель для реального типа, который будет задан позднее. (Этот параметр может иметь любое допустимое в С++ имя, обычно T или Type.) В этом контексте можно применять typename вместо class:

```
template <typename T>     // то же, что и шаблон <class T>
class Rev {...} ;
```

Определение класса (реализация) генерируется при объявлении объекта класса и спецификации конкретного типа. Например, объявление

```
class Ic<short> sic;      // неявная реализация
```

заставляет компилятор сгенерировать определение класса, в котором каждое вхождение параметра типа T в определении класса будет заменено типом short. В этом случае именем класса будет Ic<short>, а не Ic. Объявление Ic<short> называется *специализацией* шаблона. В данном случае имеет место неявная реализация.

Явная реализация возникает при объявлении специализации класса с использованием ключевого слова template:

```
template class IC<int>;   // явная реализация
```

В этом случае компилятор использует общий шаблон для генерации int-специализации Ic<int>, даже если не создается объект класса.

Можно создать явную специализацию, являющуюся специализированным определением класса, которая подменяет определение шаблона. Достаточно определить класс, начав с template<>, затем использовать имя шаблонного класса, после которого идут угловые скобки, содержащие тип требуемой специализации. Например, можно создать специализированный класс Ic для указателей на char:

```
template <> class Ic<char *>.
{
   char * str;
   ...
public:
   Ic(const char * s) : str(s) { }
   ...
};
```

Объявление в виде

```
class Ic<char *> chic;
```

вместо применения общего шаблона будет использовать специализированное определение для chic.

Шаблон класса может определять несколько общих типов и может иметь нетипизированные параметры:

```
template <class T, class TT, int n>
class Pals {...};
```

Объявление

```
Pals<double, string, 6> mix;
```

генерирует неявную реализацию, используя double для T, string для TT и 6 для n.

Шаблон класса может также иметь параметры, являющиеся шаблонами:

```
template < template <typename T> class CL, typename U, int z>
class Trophy {...};
```

Здесь z предназначено для значения int, U — для имени типа, а CL — для шаблона класса, определенного с использованием template <typename T>.

Шаблоны класса могут быть частично специализированы:

```
template <class T> Pals<T, T, 10> {...};
template <class T, class TT> Pals<T, TT, 100> {...};
template <class T, int n> Pals <T, T*, n> {...};
```

В первом примере создается специализация, в которой оба типа одинаковы и n имеет значение 6. Во втором примере создается специализация для n, равного 100. В третьем примере создается специализация, в которой второй тип является указателем на первый тип.

Шаблонные классы могут быть членами других классов, структур и шаблонов.

Главная цель создания всех рассмотренных методов — предоставить программисту возможность повторного использования протестированного кода, не занимаясь его копированием вручную. Это упрощает программирование и делает программы надежнее.

Вопросы для самоконтроля

1. Для каждого из набора классов укажите, какое наследование, общедоступное или приватное, лучше подходит для столбца Б.

А	Б
class Bear	class PolarBear
class Kitchen	class Home
class Person	class Programmer
class Person	class HorseJockey
class Person, class Automobile	class Drive

2. Предположим, что имеется следующее определение:

```
class Frabjous {
private:
   char fab[20];
public:
   Frabjous(const char * s = "C++") : fab(s) { }
   virtual void tell() { cout << fab; }
};
class Gloam {
private:
   int glip;
   Frabjous fb;
public:
   Gloam(int g = 0, const char * s = "C++");
   Gloam(int g, const Frabjous & f);
   void tell();
};
```

   Кроме того, известно, что функция `tell()` класса `Gloam` должна отображать значения `glip` и `fb`. Создайте определения для трех методов класса `Gloam`.

3. Предположим, что имеются следующие определения:

```
class Frabjous {
private:
   char fab[20];
public:
   Frabjous(const char * s = "C++") : fab(s) { }
   virtual void tell() { cout << fab; }
};
class Gloam : private Frabjous{
private:
   int glip;
public:
   Gloam(int g = 0, const char * s = "C++");
   Gloam(int g, const Frabjous & f);
   void tell();
};
```

   Известно, что версия функции `tell()` класса `Gloam` должна отображать значения `glip` и `fb`. Создайте определения для трех методов класса `Gloam`.

4. Предположим, что есть следующее определение, основанное на шаблоне `Stack` из листинга 14.13 и на классе `Worker` из листинга 14.10:

```
Stack<Worker *> sw;
```

   Напишите определение класса, который будет сгенерирован. Предоставьте только определение класса, а не его невстроенных методов.

5. Воспользуйтесь определениями шаблонов, рассмотренных в этой главе, для определения следующих сущностей:

   - Массива объектов `string`.
   - Стека массивов значений типа `double`.
   - Массива стеков указателей на объекты `Worker`.

   Сколько определений шаблонов классов представлено в листинге 14.18?

6. Объясните разницу между виртуальными и невиртуальными базовыми классами.

Упражнения по программированию

1. Класс Wine имеет член string (см. главу 4), содержащий наименование вина, и объекты Pair и valarray<int> (рассмотрены в этой главе). Первый член объекта Pair содержит год сбора винограда, а второй член — количество бутылок, разлитых в соответствующем году. Например, первый объект valarray объекта Pair содержит годы 1988, 1992 и 1996, а второй объект valarray — количества бутылок: 24, 48 и 144. Неплохо было бы, если бы объект Wine содержал член int для хранения возраста вина в годах. Для упрощения кода будут полезны несколько typedef-объявлений:

   ```
   typedef std::valarray<int> ArrayInt;
   typedef Pair<ArrayInt, ArrayInt> PairArray;
   ```

   Таким образом, тип PairArray представляет тип Pair<std::valarray<int>, std::valarray<int>>. Оеалзуйте класс Wine, используя технику включения. Класс должен иметь конструктор по умолчанию и, как минимум, следующие конструкторы:

   ```
   // инициализировать label значением l, количество лет - y,
   // год сбора винограда - yr[], бутылки - bot[]
   Wine(const char * l, int y, const int yr[], const int bot[]);
   // инициализировать label значением l, количество лет - y,
   // создать объекты массива размерностью y
   Wine(const char * l, int y);
   ```

   Класс Wine должен иметь метод GetBottles(), который для объекта Wine заданного возраста предлагает пользователю ввести соответствующее значение года сбора винограда и количество бутылок. Метод Label() будет возвращать ссылку на наименование вина. Метод sum() будет возвращать общее количество бутылок во втором объекте valarray<int> объекта Pair.

   Программа должна предлагать пользователю ввести наименование вина, количество элементов в массиве, а также год и количество бутылок для каждого элемента массива. Программа будет использовать эти данные для создания объекта Wine и отображения информации, хранимой в объекте. Для справки ниже приведен пример тестовой программы:

   ```
   // pe14-1.cpp -- классWine с использованием техники включения
   #include <iostream>
   #include "winec.h"
   int main ( void )
   {
       using std::cin;
       using std::cout;
       using std::endl;
       cout << "Введите наименование вина: ";
       char lab[50];
       cin.getline(lab, 50);
       cout << "Введите количество сборов урожая: ";
       int yrs;
       cin >> yrs;
       Wine holding(lab, yrs);//сохранение наименования, количества сборов,
                              // создание массива из yrs элементов
   ```

```
    holding.GetBottles(); // запрос на ввод года и количества бутылок
    holding.Show();       // вывод содержимого объекта
    const int YRS = 3;
    int y[YRS] = {1993, 1995, 1998};
    int b[YRS] = { 48, 60, 72};
    // создание нового объекта, инициализация с использованием данных
    // из массивов y и b
    Wine more("Gushing Grape Red",YRS, y, b);
    more.Show();
    cout << "Общее количество бутылок " << more.Label() // использует
                                                        // метод Label()
         << ": " << more.sum() << endl; // использует метод sum()
    cout << "Всего наилучшего!\n";
    return 0;
}
```

Ниже показан пример вывода тестовой программы:

```
Введите наименование вина: Gully Wash
Введите количество сборов урожая: 4
Введите данные для Gully Wash по 4 годам:
Введите год: 1988
Введите количество бутылок для этого года: 42
Введите год: 1994
Введите количество бутылок для этого года: 58
Введите год: 1998
Введите количество бутылок для этого года: 122
Введите год: 2001
Введите количество бутылок для этого года: 144
Вино: Gully Wash
    Год     Бутылок
    1988    42
    1994    58
    1998    122
    2001    144
Вино: Gushing Grape Red
    Год     Бутылок
    1993    48
    1995    60
    1998    72
Общее количество бутылок Gushing Grape Red: 180
Всего наилучшего!
```

2. Следующее упражнение такое же, как и упражнение 1, за исключением того, что вместо включения нужно использовать приватное наследование. И снова несколько объявлений `typedef` должны упростить жизнь. Можно также рассматривать значение операторов следующим образом:

```
PairArray::operator=(PairArray(ArrayInt(),ArrayInt()));
cout << (const string &)(*this);
```

Класс будет работать с тестовой программой, приведенной в упражнении 1.

3. Определите шаблон `QueueTp`. Протестируйте его, создав очередь указателей на `Worker` (как показано в листинге 14.10), и используйте его в программе, подобной листингу 14.12.

4. Класс `Person` содержит имя и фамилию человека. Вдобавок к своим конструкторам он имеет метод `Show()`, отображающий эти реквизиты. Класс `Gunslinger` (стрелок) виртуально унаследован от класса `Person`. Он имеет член `Draw()`, который возвращает значение типа `double`, являющееся временем реакции стрелка. Класс также содержит член `int`, содержащий количество насечек на ружье. И, наконец, класс имеет функцию `Show()`, которая отображает всю эту информацию.

   Класс `PokerPlayer` виртуально унаследован от класса `Person`. Он имеет метод `Draw()`, который возвращает случайное число в диапазоне от 1 до 52, являющееся значением карты. (Можно создать класс `Card` с членами, определяющими лицо и рубашку карты, и использовать значение, возвращаемое классом `Card`, для метода отрисовки `Draw()`). Класс `PokerPlayer` использует функцию `Show()` класса `Person`. Класс `BadDude` общедоступно наследуется от классов `Gunslinger` и `PokerPlayer`. Он имеет член `Gdraw()`, возвращающий время отрисовки класса и член `Cdraw()`, возвращающий изображение следующей карты. У него есть соответствующая функция `Show()`. Определите все упомянутые классы и методы вместе с другими необходимыми методами (такими как методы для установки значений объекта) и протестируйте их с помощью простой программы, аналогичной показанной в листинге 14.12.

5. Ниже приведено несколько объявлений класса:

```
// emp.h -- заголовочный файл для класса abstr_emp и его дочерних
классов
#include <iostream>
#include <string>
class abstr_emp
{
private:
    std::string fname; // имя abstr_emp
    std::string lname; // фамилия abstr_emp
    std::string job;
public:
    abstr_emp();
    abstr_emp(const std::string & fn, const std::string & ln,
    const std::string & j);
    virtual void ShowAll() const; // отображает все данные
    virtual void SetAll(); // запрос на ввод пользовательских значений
    friend std::ostream & operator<<(std::ostream & os, const abstr_emp & e);
    // отображает только имя и фамилию
    virtual ~abstr_emp() = 0; // виртуальный базовый класс
};
class employee : public abstr_emp
{
public:
    employee();
    employee(const std::string & fn, const std::string & ln,
    const std::string & j);
    virtual void ShowAll() const;
    virtual void SetAll();
};
```

```
class manager: virtual public abstr_emp
{
private:
   int inchargeof; // количество управляемых abstr_emps
protected:
   int InChargeOf() const { return inchargeof; }    // вывод
   int & InChargeOf(){ return inchargeof; }         // ввод
public:
   manager();
   manager(const std::string & fn, const std::string & ln,
   const std::string & j, int ico = 0);
   manager(const abstr_emp & e, int ico);
   manager(const manager & m);
   virtual void ShowAll() const;
   virtual void SetAll();
};
class fink: virtual public abstr_emp
{
private:
   std::string reportsto; // кому выводить отчеты
protected:
   const std::string ReportsTo() const { return reportsto; }
   std::string & ReportsTo(){ return reportsto; }
public:
   fink();
   fink(const std::string & fn, const std::string & ln,
      const std::string & j, const std::string & rpo);
   fink(const abstr_emp & e, const std::string & rpo);
   fink(const fink & e);
   virtual void ShowAll() const;
   virtual void SetAll();
};
class highfink: public manager, public fink
{
public:
   highfink();
   highfink(const std::string & fn, const std::string & ln,
      const std::string & j, const std::string & rpo, int ico);
   highfink(const abstr_emp & e, const std::string & rpo, int ico);
   highfink(const fink & f, int ico);
   highfink(const manager & m, const std::string & rpo);
   highfink(const highfink & h);
   virtual void ShowAll() const;
   virtual void SetAll();
};
```

Отметим, что в иерархии классов используется множественное наследование с виртуальными базовыми классами. Поэтому не забывайте о специальных правилах для списка инициализации конструкторов. Отметим также наличие нескольких приватных методов. Это упрощает код некоторых методов `highfink`. (Например, если `highfink::ShowAll()` просто вызывает `fink::ShowAll()` и `manager::ShowAll()`, то это дважды приводит к вызову `abstr_emp::ShowAll()`.) Реализуйте эти методы и протестируйте классы. Ниже показана минимальная тестовая программа:

```
// pe14-5.cpp
// useempl.cpp -- использует классы abstr_emp
#include <iostream>
using namespace std;
#include "emp.h"
int main(void)
{
    employee em("Trip", "Harris", "Thumper");
    cout << em << endl;
    em.ShowAll();

    manager ma("Amorphia", "Spindragon", "Nuancer", 5);
    cout << ma << endl;
    ma.ShowAll();
    fink fi("Matt", "Oggs", "Oiler", "Juno Barr");
    cout << fi << endl;
    fi.ShowAll();
    highfink hf(ma, "Curly Kew"); // наем?
    hf.ShowAll();

    cout << "Нажмите любую клавишу для следующей фазы:\n";
    cin.get();
    highfink hf2;
    hf2.SetAll();

    cout << "Использование указателя abstr_emp *:\n";
    abstr_emp * tri[4] = {&em, &fi, &hf, &hf2};
    for (int i = 0; i < 4; i++)
        tri[i]->ShowAll();
    return 0;
}
```

Почему не определена операция присваивания?

Почему `ShowAll()` **и** `SetAll()` **виртуальные?**

Почему `abstr_emp` **является виртуальным базовым классом?**

Почему `highfink` **не имеет раздела данных?**

Почему необходима только одна версия операции `operator<<()`**?**

Что произойдет, если код в конце программы заменить следующим:

```
abstr_emp tri[4] = {em, fi, hf, hf2};
for (int i = 0; i < 4; i++)
    tri[i].ShowAll();
```

ГЛАВА 15

Дружественность, исключения и другие понятия

В этой главе:

- Дружественные классы
- Методы дружественных классов
- Вложенные классы
- Генерация исключений и блоки `try` и `catch`
- Классы исключений
- Динамическая идентификация типов (RTTI)
- `dynamic_cast` и `typeid`
- `static_cast`, `const_cast` и `reinterpret_cast`

В этой главе связываются воедино несколько разрозненных понятий и рассматриваются последние расширения языка C++. К разрозненным понятиям, в данном случае, относятся дружественные классы, дружественные функции-члены и вложенные классы, являющиеся классами, определенными внутри других классов. К последним расширениям, рассматриваемым здесь, относятся исключения, динамическая идентификация типов (RTTI) и улучшенный контроль приведения типов. Управление исключениями в C++ предлагает механизм работы с нестандартными ситуациями, которые обычно приводят к останову программы. RTTI представляет собой механизм определения объектных типов. Новые операции приведения типов повышают надежность таких приведений. Последние три возможности сравнительно новы для C++ и могут не поддерживаться старыми компиляторами.

Друзья

В некоторых примерах этой книги использовались дружественные функции как часть расширенного интерфейса класса. Такие функции — не единственный вид друзей, которых может иметь класс. Класс также может быть другом. В этом случае любой метод дружественного класса может иметь доступ к приватным и защищенным методам исходного класса. Можно, конечно, ограничиться определением только отдельных функций-членов в качестве дружественных функций другого класса. Класс сам определяет, какие функции, функции-члены или классы являются для него друзьями; дружественность не может быть предписана извне. Таким образом, несмотря

на то, что друзья предоставляют доступ к приватной части класса, они не нарушают концепции объектно-ориентированного программирования. Наоборот, они придают большую гибкость общедоступному интерфейсу.

Дружественные классы

Когда возникает необходимость сделать один класс дружественным другому? Рассмотрим пример. Предположим, что нужно написать программу, имитирующую телевизор и дистанционное управление им. Создадим класс `Tv`, представляющий телевизор, и класс `Remote`, представляющий пульт дистанционного управления. Понятно, что должно существовать какое-то отношение между этими классами, но какое? Пульт дистанционного управления не является телевизором и наоборот, поэтому отношение *is-a*, возникающее при общедоступном наследовании, не подходит. Ни один, ни другой класс не являются компонентом другого, поэтому отношение *has-a*, возникающее при включении, приватном и защищенном наследовании, тоже не подходит. В действительности пульт дистанционного управления позволяет изменять состояние телевизора, и это наводит на мысль сделать класс `Remote` другом для класса `Tv`.

Давайте определим класс `Tv`. Можно представить телевизор как набор членов, определяющих состояние телевизора. Ниже приведен список возможных состояний телевизора:

- Включен/выключен.
- Настройка каналов.
- Настройка уровня громкости.
- Режим настройки кабеля и антенны.
- ТВ-тюнер или аудио-видео-вход.

Режим настройки отражает тот факт, что в Соединенных Штатах частотное расстояние между каналами для 14 канала и выше различно для кабельного и эфирного UHF (Ultra High Frequency — сверхвысокие частоты) телевидения. Выбор входа отражает возможность телевизора работать в режиме кабельного или эфирного телевидения или в качестве VCR (Video Card Recorder — видеомагнитофон). Некоторые устройства предлагают еще более разнообразный выбор, например наличие входов VCR/DVD, но для нашего примера достаточно приведенного списка.

У телевизора есть свойства, которые нельзя представить переменными, описывающими его состояние. Например, телевизоры отличаются по количеству каналов, которые они могут принимать. Можно создать член, описывающий это свойство.

Дальше нужно создать класс с методами, позволяющими изменять настройки телевизора. У многих современных телевизоров органы управления скрыты за лицевыми панелями, однако всегда есть возможность изменять настройки без пульта дистанционного управления. При этом можно последовательно переключаться на соседние каналы, но нельзя выбрать канал произвольно. Аналогично, для изменения громкости существуют кнопки — одна для увеличения, другая для уменьшения громкости.

Пульт дистанционного управления дублирует органы управления телевизора. Многие из его методов могут быть построены с использованием методов класса `Tv`. Вдобавок, пульт дистанционного управления позволяет выбирать каналы вещания произвольно. То есть можно сразу перейти со 2-го на 20-й канал, не перебирая все каналы подряд. Многие пульты дистанционного управления могут работать в двух режимах — управления телевизором и управления VCR. Подобные рассуждения наводят

на мысль создать определения, приведенные в листинге 15.1. Определения содержат несколько констант, заданных перечислением. Следующий оператор делает Remote дружественным классом:

```
friend class Remote;
```

Дружественное объявление может появляться в общедоступном, приватном или защищенном разделе; место не имеет значения. Поскольку класс Remote ссылается на класс Tv, компилятор должен знать о классе Tv до того, как начнет оперировать классом Remote. Самый простой путь — определить класс Tv первым. В качестве альтернативы можно использовать объявление forward; мы рассмотрим эту возможность позже.

Замечание по совместимости

Если ваш компилятор не поддерживает тип bool, используйте int-значения 0 и 1 вместо true и false.

Листинг 15.1. tv.h

```
// tv.h -- классы Tv и Remote
#ifndef TV_H_
#define TV_H_

class Tv
{
public:
   friend class Remote; // Remote имеет доступ к приватной части Tv
   enum {Off, On};
   enum {MinVal,MaxVal = 20};
   enum {Antenna, Cable};
   enum {TV, VCR};
   Tv(int s = Off, int mc = 100) : state(s), volume(5),
      maxchannel(mc), channel(2), mode(Cable), input(TV) {}
   void onoff() {state = (state == On)? Off : On;}
   bool ison() const {return state == On;}
   bool volup();
   bool voldown();
   void chanup();
   void chandown();
   void set_mode() {mode = (mode == Antenna)? Cable : Antenna;}
   void set_input() {input = (input == TV)? VCR : TV;}
   void settings() const; // вывод всех настроек
private:
   int state;       // On или Off
   int volume;      // предполагается, что будет оцифрован
   int maxchannel;  // максимальное количество каналов
   int channel;     // текущий канал
   int mode;        // эфирное или кабельное телевидение
   int input;       // TV или VCR
};
class Remote
{
private:
   int mode; // управление TV или VCR
public:
   Remote(int m = Tv::TV) : mode(m) {}
```

```
    bool volup(Tv & t) { return t.volup();}
    bool voldown(Tv & t) { return t.voldown();}
    void onoff(Tv & t) { t.onoff(); }
    void chanup(Tv & t) {t.chanup();}
    void chandown(Tv & t) {t.chandown();}
    void set_chan(Tv & t, int c) {t.channel = c;}
    void set_mode(Tv & t) {t.set_mode();}
    void set_input(Tv & t) {t.set_input();}
};
#endif
```

Большинство методов в листинге 15.1 определены как встроенные. Отметим, что каждый метод класса Remote, отличный от конструктора, принимает ссылку на объект Tv в качестве параметра. Это отражает то, что пульт дистанционного управления воздействует на конкретный телевизор. В листинге 15.2 приведены остальные определения. Функции управления громкостью изменяют уровень звука на единицу до тех пор, пока он не достигнет максимального или минимального значения. Функции выбора канала используют циклический возврат — за минимальным значением канала, равным 1, сразу следует максимальный канал, равный maxchannel.

Многие методы для переключения между двумя настройками используют условную операцию.

```
void onoff() {state = (state == On)? Off : On;}
```

При условии, что значениями настройки являются 0 и 1, это можно записать более компактно с помощью поразрядной операции исключающего ИЛИ и операции присваивания (^=), как показано в приложении Д:

```
void onoff() {state ^= 1;}
```

Фактически можно хранить до восьми двоичных состояний в переменной unsigned char и индивидуально переключать их. Но уже это другая история.

Листинг 15.2. tv.cpp

```
// tv.cpp -- методы класса Tv (методы Remote являются встроенными)
#include <iostream>
#include "tv.h"
bool Tv::volup()
{
    if (volume < MaxVal)
    {
        volume++;
        return true;
    }
    else
        return false;
}
bool Tv::voldown()
{
    if (volume > MinVal)
    {
        volume--;
        return true;
    }
```

```
   else
      return false;
}
void Tv::chanup()
{
   if (channel < maxchannel)
      channel++;
   else
      channel = 1;
}
void Tv::chandown()
{
   if (channel > 1)
      channel--;
   else
      channel = maxchannel;
}
void Tv::settings() const
{
   using std::cout;
   using std::endl;
   cout << "Телевизор " << (state == Off? "выключен" : "включен") << endl;
   if (state == On)
   {
      cout << "Настройка громкости = " << volume << endl;
      cout << "Настройка канала = " << channel << endl;
      cout << "Режим = "
           << (mode == Antenna? "антенна" : "кабель") << endl;
      cout << "Вход = "
           << (input == TV? "телевизор" : "видеомагнитофон") << endl;
   }
}
```

В листинге 15.3 показана короткая программа для тестирования предыдущего кода. Один и тот же пульт используется для управления двумя разными телевизорами.

Листинг 15.3. use_tv.cpp

```
//use_tv.cpp -- использование классов Tv и Remote
#include <iostream>
#include "tv.h"
int main()
{
   using std::cout;
   Tv s27;
   cout << "Начальные настройки для телевизора с диагональю 27\":\n";
   s27.settings();
   s27.onoff();
   s27.chanup();
   cout << "\nУточненные настройки для телевизора с диагональю 27\":\n";
   s27.settings();
   Remote grey;
   grey.set_chan(s27, 10);
   grey.volup(s27);
   grey.volup(s27);
```

```
   cout << "\nНастройки для телевизора с диагональю 27\" после использования
пульта:\n";
   s27.settings();
   Tv s32(Tv::On);
   s32.set_mode();
   grey.set_chan(s32,28);
   cout << "\nНастройки для телевизора с диагональю 32\":\n";
   s32.settings();
   return 0;
}
```

Ниже приведен вывод программ в листингах 15.1, 15.2 и 15.3:

```
Начальные настройки для телевизора с диагональю 27":
Телевизор выключен
Уточненные настройки для телевизора с диагональю 27":
Телевизор включен
Настройка громкости = 5
Настройка канала = 3
Режим = кабель
Вход = телевизор
Настройки для телевизора с диагональю 27" после использования пульта:
Телевизор включен
Настройка громкости = 7
Настройка канала = 10
Режим = кабель
Вход = телевизор
Настройки для телевизора с диагональю 32":
Телевизор включен
Настройка громкости = 5
Настройка канала = 28
Режим = антенна
Вход = телевизор
```

Главное в этом упражнении то, что дружба между классами является естественным понятием, подчеркивающим наличие некоторых отношений между ними. Без определенных форм дружественных связей придется сделать приватные части класса `Tv` общедоступными или создать один громоздкий класс, представляющий и телевизор, и пульт дистанционного управления. С помощью такого решения невозможно отразить тот факт, что один пульт дистанционного управления можно использовать с различными телевизорами.

Дружественные функции-члены

Просматривая последний пример, можно заметить, что большинство методов класса `Remote` реализовано с использованием общедоступного интерфейса класса `Tv`. Это значит, что эти методы, в действительности, не требуют дружественного статуса. В самом деле, единственным методом класса `Remote`, который получает прямой доступ к приватному члену класса `Tv`, является `Remote::set_chan()`, поэтому он должен быть другом. Можно сделать дружественными только отдельные члены класса, а не весь класс целиком, однако это менее удобно. Нужно быть внимательным к порядку, в котором создаются различные определения и объявления. Посмотрим, почему это происходит.

Сделать `Remote::set_chan()` другом для класса `Tv` можно, объявив его в качестве друга в объявлении класса `Tv`:

```
class Tv
{
   friend void Remote::set_chan(Tv & t, int c);
   ...
};
```

Однако компилятор для обработки этого оператора должен уже видеть определение класса `Remote`. С другой стороны, он не должен знать, что `Remote` является классом и что `set_chan()` – метод этого класса. Это наводит на мысль расположить определение `Remote` над определением `Tv`. Но методы класса `Remote` ссылаются на объекты `Tv`, а это значит, что определение `Tv` должно следовать раньше, чем определение `Remote`. Избежать циклических ссылок можно, используя *опережающее объявление*. Для этого вставим оператор

```
class Tv;          // опережающее объявление
```

перед определением `Remote`. Это приводит к следующему порядку:

```
class Tv;          // опережающее объявление
class Remote { ... };
class Tv { ... };
```

Можно ли использовать другой порядок?

```
class Remote;      // опережающее объявление
class Tv { ... };
class Remote { ... };
```

Ответ — нельзя. Причина заключается в том, что, как уже говорилось выше, когда компилятор видит, что метод класса `Remote` объявлен как дружественный в объявлении класса `Tv`, он должен видеть объявление всего класса `Remote` и метода `set_chan()` в частности.

Остаются и другие сложности. В листинге 15.1 объявление `Remote` содержит следующий встроенный код:

```
void onoff(Tv & t) { t.onoff(); }
```

Поскольку здесь вызывается метод `Tv`, компилятору нужно видеть объявление класса `Tv`, для того чтобы знать, какие у него есть методы. Но как мы видели, за этим объявлением обязательно следует объявление `Remote`. Решением в данном случае может быть раздельное объявление методов `Remote` до определения класса `Tv` и их непосредственное определение после класса `Tv`. Это приводит к следующему порядку:

```
class Tv; // опережающее объявление
class Remote { ... }; // методы, использующие Tv, в виде только прототипов
class Tv { ... };
// здесь расположить определение методов Remote
```

Прототипы `Remote` выглядят следующим образом:

```
void onoff(Tv & t);
```

Все что компилятору нужно знать, для того чтобы обработать этот прототип — это что Tv является классом, и опережающее объявление предоставляет ему эту информацию. Через время компилятор достигнет непосредственного определения методов. К этому моменту он уже прочитает объявление класса Tv, и будет иметь дополнительную информацию для компиляции этих методов.

Используя в определениях методов ключевое слово inline, можно сделать методы встроенными. В листинге 15.4 показан измененный заголовочный файл.

Листинг 15.4. tvfm.h

```
// tvfm.h -- классы Tv и Remote используют членов-друзей
#ifndef TVFM_H_
#define TVFM_H_
class Tv; // опережающее объявление
class Remote
{
public:
   enum State{Off, On};
   enum {MinVal,MaxVal = 20};
   enum {Antenna, Cable};
   enum {TV, VCR};
private:
   int mode;
public:
   Remote(int m = TV) : mode(m) {}
   bool volup(Tv & t); // только прототип
   bool voldown(Tv & t);
   void onoff(Tv & t) ;
   void chanup(Tv & t) ;
   void chandown(Tv & t) ;
   void set_mode(Tv & t) ;
   void set_input(Tv & t);
   void set_chan(Tv & t, int c);
};
class Tv
{
public:
   friend void Remote::set_chan(Tv & t, int c);
   enum State{Off, On};
   enum {MinVal,MaxVal = 20};
   enum {Antenna, Cable};
   enum {TV, VCR};
   Tv(int s = Off, int mc = 100) : state(s), volume(5),
      maxchannel(mc), channel(2), mode(Cable), input(TV) {}
   void onoff() {state = (state == On)? Off : On;}
   bool ison() const {return state == On;}
   bool volup();
   bool voldown();
   void chanup();
   void chandown();
   void set_mode() {mode = (mode == Antenna)? Cable : Antenna;}
   void set_input() {input = (input == TV)? VCR : TV;}
   void settings() const;
```

```
private:
   int state;
   int volume;
   int maxchannel;
   int channel;
   int mode;
   int input;
};
// методы Remote как встроенные функции
inline bool Remote::volup(Tv & t) { return t.volup();}
inline bool Remote::voldown(Tv & t) { return t.voldown();}
inline void Remote::onoff(Tv & t) { t.onoff(); }
inline void Remote::chanup(Tv & t) {t.chanup();}
inline void Remote::chandown(Tv & t) {t.chandown();}
inline void Remote::set_mode(Tv & t) {t.set_mode();}
inline void Remote::set_input(Tv & t) {t.set_input();}
inline void Remote::set_chan(Tv & t, int c) {t.channel = c;}
#endif
```

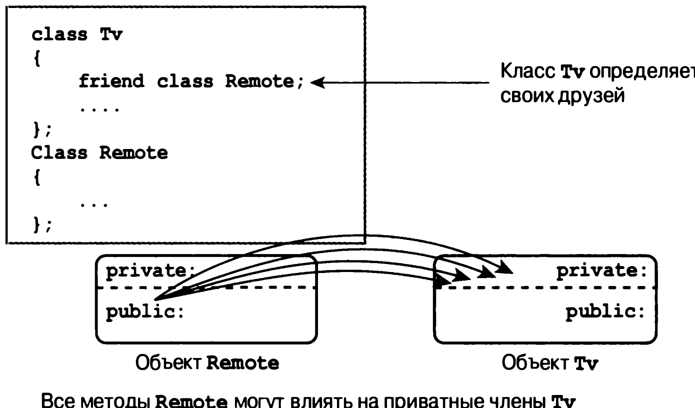
Если включить `tvfm.h` вместо `tv.h` в файлах `tv.cpp` и `use_tv.cpp`, результирующая программа будет вести себя так же, как исходная. Разница лишь в том, что теперь только один метод класса `Remote`, а не все методы, будет дружественным классу `Tv`. На рис. 15.1 показана эта разница.

Вспомните, что встроенные функции имеют внутреннее связывание; это значит, что определения функций должны находиться в том же файле, где они используются. Здесь `inline`-определения содержатся в заголовочном файле, поэтому включение этого файла в файл, использующий эти функции, будет правильным решением. Можно включить определения в файл реализации. В этом случае потребуется удалить ключевое слово `inline`, что приведет к использованию внешнего связывания.

Кстати, если сделать другом весь класс `Remote`, то станет ненужным объявление `forward`, поскольку выражение `friend` само определяет `Remote` как класс.

Другие дружественные отношения

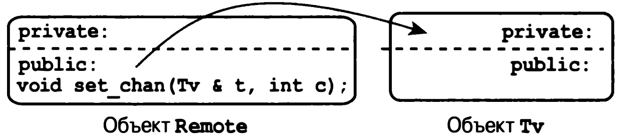
Кроме случаев, рассмотренных в этой главе, возможны и другие комбинации друзей и классов. Рассмотрим вкратце некоторые из них. Предположим, что современная технология использует интерактивное дистанционное управление. Например, устройство интерактивного дистанционного управления позволяет вам ввести ответ на один из вопросов, заданных в телевизионной программе, и телевизор должен активизировать звуковой сигнал в устройстве дистанционного управления, если ваш ответ неверен. Не обращая внимание на то, может ли телевизор использовать такое устройство управления, рассмотрим лишь программные аспекты C++. Теперь система выигрывает от взаимных дружественных связей: некоторые методы класса `Remote` могут воздействовать на объект `Tv`, как и ранее, и некоторые методы класса `Tv` могут воздействовать на объект `Remote`. Этого можно добиться, сделав классы дружественными друг другу. То есть `Tv` будет другом для `Remote`, а `Remote` будет другом для `Tv`. Единственно, следует иметь в виду, что хотя метод класса `Tv`, использующий объект `Remote`, может быть объявлен прежде, чем объявлен класс `Remote`, он должен быть определен после объявления `Remote`.



Все методы **Remote** могут влиять на приватные члены **Tv**

```
class Tv;
class Remote
{
    ....
};
Class Tv
{
    friend void Remote::set_chan(Tv & t, int c);
    ...
};
// Методы Remote
...
```

Класс **Tv** определяет своих друзей



Remote::set_chan() может влиять на приватные члены **Tv**

Рис. 15.1. Друзья класса в сравнении с друзьями-членами класса

Это необходимо для корректной компиляции метода. Система будет выглядеть следующим образом:

```
class Tv
{
friend class Remote;
public:
   void buzz(Remote & r);
   ...
};
class Remote
{
friend class Tv;
public:
   void Bool volup(Tv & t) { t.volup(); }
   ...
};
```

```
inline void Tv::buzz(Remote & r)
{
   ...
}
```

Поскольку объявление Remote сделано после объявления Tv, Remote::volup() может быть определена в объявлении класса. Однако метод Tv::buzz() должен быть определен вне объявления класса Tv, то есть это определение должно следовать за объявлением класса Remote. Если нет необходимости объявлять buzz() как встроенный метод, нужно определить его в отдельном файле методов.

Общие друзья

Другой пример использования друзей — когда функция требует доступа к приватным данным двух разных классов. Логично было бы, если бы такая функция могла быть функцией-членом каждого из этих классов, но это невозможно. Она может быть членом одного класса и другом — другого. Но иногда есть смысл сделать ее дружественной обоим классам. Предположим, что имеется класс Probe, представляющий некий программируемый измерительный прибор, и класс Analyzer, представляющий программируемый анализатор. У каждого прибора есть внутренние часы и их нужно синхронизировать. Можно использовать следующие строки кода:

```
class Analyzer; // опережающее объявление
class Probe
{
   friend void sync(Analyzer & a, const Probe & p); // синхронизация a с p
   friend void sync(Probe & p, const Analyzer & a); // синхронизация p с a
   ...
};
class Analyzer
{
   friend void sync(Analyzer & a, const Probe & p); // синхронизация a с p
   friend void sync(Probe & p, const Analyzer & a); // синхронизация p с a
   ...
};
// определение дружественных функций
inline void sync(Analyzer & a, const Probe & p)
{
   ...
}
inline void sync(Probe & p, const Analyzer & a)
{
   ...
}
```

Когда компилятор достигает определения друга в определении класса Probe, объявление forward уведомляет компилятор, что Analyzer является типом.

Вложенные классы

В C++ можно расположить объявление класса внутри другого класса. Класс, объявленный внутри другого класса, называется *вложенным классом*. Вложенные классы

вводят новую область видимости в пределах класса и позволяют избежать конфликта имен. Функции-члены класса, содержащие это определение, могут создавать и использовать объекты вложенных классов. Извне вложенные классы можно использовать, только если они объявлены в общедоступной части класса и только с использованием операции разрешения контекста. (Ранние версии C++ не поддерживают вложенные классы, а если поддерживают, то далеко не полностью.)

Вложение классов — это не то, чем является включение. Напомним, что включение предполагает наличие объекта класса в качестве члена другого класса. Вложение класса не создает члена класса. Вместо этого создается локальный тип для класса, в котором содержится объявление вложенного класса.

Обычно причиной создания вложенного класса является необходимость содействия в создании другого класса и желание избежать конфликта имен. В примере с классом `Queue` в листинге 12.10 главы 12 представлен скрытый случай вложенного класса в виде вложения определения структуры:

```
class Queue
{
// объявления с областью видимости класса
   // Node -это вложенное определение структуры, локальное для класса
   struct Node {Item item; struct Node * next;};
   ...
};
```

Поскольку структура является классом, чьи члены общедоступны по умолчанию, `Node` — действительно вложенный класс. Однако такое определение не дает преимуществ характерных для класса. В частности, оно нуждается в явном конструкторе. Восполним этот недостаток.

Сначала нужно найти, где в примере `Queue` создается объект `Node`. Просматривая объявление класса (см. листинг 12.10) и определения методов (см. листинг 12.11) обнаружим, что единственным местом, где создается объект `Node`, является метод `queue()`:

```
bool Queue::enqueue(const Item & item)
{
   if (isfull())
      return false;
   Node * add = new Node; // создать узел
   if (add == NULL)
      return false; // выйти, если ничего не доступно
   add->item = item; // установить указатели узлов
   add->next = NULL;
   ...
}
```

В этом коде после создания `Node` его членам явно присваиваются значения. Такую работу лучше проделывать в конструкторе.

Зная, где и как конструктор будет использоваться, можно задать соответствующее определение:

```
class Queue
{
// объявления с областью видимости класса
```

```
    // Node является вложенным классом, локальным для этого класса
    class Node
    {
    public:
       Item item;
       Node * next;
       Node(const Item & i) : item(i), next(0) { }
    };
    ...
};
```

В этом конструкторе член item инициализируется значением i и указатель next устанавливается в 0; это один из способов создания null-указателя в C++. (Для использования значения NULL нужно включить в проект заголовочный файл, в котором оно определено.) Поскольку все узлы, созданные классом Queue, имеют член next, изначально установленный в null-указатель, то одного конструктора для этого класса вполне достаточно. Дальше необходимо переписать enqueue() с использованием конструктора:

```
bool Queue::enqueue(const Item & item)
{
   if (isfull())
      return false;
   Node * add = new Node(item); // создает и инициализирует узел
   if (add == 0)
      return false;            // выход, если ничего нет
   ...
}
```

Это сделает код enqueue() чуть короче и немного надежнее, поскольку инициализация проводится автоматически и программисту не нужно помнить, что в этом случае он должен делать. В следующем примере определяется конструктор в объявлении класса. Предположим, что нужно определить его в файле методов. В этом случае определение должно отображать тот факт, что класс Node определен внутри класса Queue. Это делается с помощью операции разрешения контекста:

```
Queue::Node::Node(const Item & i) : item(i), next(0) { }
```

Вложенные классы и доступ

С вложенными классами используются два типа доступа. В первом случае то место, где вложенный класс был объявлен, определяет область видимости вложенного класса; то есть, он определяет, какие части программы могут создавать объекты этого класса. Во втором случае, как и для любого другого класса, доступ к членам класса определяют общедоступный, приватный и защищенный разделы класса. Где и как вложенный класс может быть использован, зависит как от области видимости класса, так и от управления доступом. Рассмотрим эти утверждения ниже.

Область видимости

Если вложенный класс определен в приватном разделе другого класса, он виден только внутри этого класса. Это относится, например, к классу Node, вложенному в определение класса Queue в рассмотренном примере. (Может показаться, что Node

был определен перед приватным разделом, но следует помнить, что приватный доступ назначается классу по умолчанию.) Следовательно, члены класса Queue могут использовать объекты Node и указатели на объекты Node, но другие части программы даже не знают о существовании класса Node. Если понадобиться породить класс от Queue, то Node для этого класса будет также невидим, поскольку производный класс не имеет прямого доступа к приватной части базового класса.

Если вложенный класс объявлен в защищенном разделе другого класса, он будет видим для этого класса, но не видим извне. Поскольку вложенный класс имеет область видимости базового класса, извне его нужно использовать с квалификатором класса. Предположим, что есть следующее объявление:

```
class Team
{
public:
   class Coach { ... };
   ...
};
```

Теперь предположим, что есть безработный тренер, не имеющий команды. Для создания объекта Coach вне класса Team, можно поступить следующим образом:

```
Team::Coach forhire; // создает объект Coach за пределами класса Team
```

Подобные же рассуждения об области видимости применимы и к вложенным структурам и перечислимым типам. В действительности многие программисты используют общедоступные перечислимые типы для задания констант класса, которые могут быть использованы клиентскими программами. Например, как уже было показано, многие реализации классов для поддержки возможности iostream используют эту технику для создания различных вариантов форматирования (более подробно этот вопрос рассматривается в главе 17). В табл. 15.1 сведены свойства области видимости для вложенных классов, структур и перечислимых типов.

Таблица 15.1. Свойства области видимости для вложенных классов, структур и перечислимых типов

Где определен во вложенном классе	Доступен для вложенного класса	Доступен для класса производного от вложенного класса	Доступен извне
Приватный раздел	Да	Нет	Нет
Защищенный раздел	Да	Да	Нет
Общедоступный раздел	Да	Да	Да, с квалификатором класса

Управление доступом

После того как класс оказывается в области видимости, принимаются во внимание правила управления доступом. Для вложенных классов действуют те же правила доступа, что и для обычных классов. Объявление класса Node в определении класса Queue не дает классу Queue никаких специальных привилегий доступа к классу Node, также как и классу Node — никаких специальных привилегий доступа к классу Queue. Так, объект класса Queue имеет доступ только к общедоступным членам объекта

Node. По этой причине в примере с Queue все члены класса Node сделаны общедоступными. Это нарушает обычную практику создания членов данных приватными, но класс Node — внутренняя реализация класса Queue, и он не виден извне. Это потому, что класс Node объявлен в приватной части класса Queue. Таким образом, методы Queue могут иметь непосредственный доступ к членам Node, а клиенты, использующие класс Queue, не могут этого делать.

Короче говоря, расположение объявления класса определяет область видимости этого класса. Если определенный класс находится в области видимости, обычные правила доступа (общедоступный, приватный, защищенный, дружественный) определяют доступ программы к членам вложенного класса.

Вложение в шаблон

Мы уже видели, что шаблоны хорошо подходят для создания контейнерных классов вроде класса Queue. Было бы удивительно, если при преобразовании в шаблон определения класса Queue возникла бы хоть одна проблема. Но этого не произойдет. В листинге 15.5 показано, как выполнить такое преобразование. Как обычно для шаблонных классов, заголовочный файл включает шаблон класса и функции-шаблоны методов.

Листинг 15.5. queuetp.h

```
// queuetp.h -- шаблон очереди с вложенным классом
#ifndef QUEUETP_H_
#define QUEUETP_H_

template <class Item>
class QueueTP
{
private:
   enum {Q_SIZE = 10};
   // Node представляет собой определение вложенного класса
   class Node
   {
   public:
      Item item;
      Node * next;
      Node(const Item & i):item(i), next(0){ }
   };
   Node * front;     // указатель на начало очереди
   Node * rear;      // указатель на конец очереди
   int items;        // текущее количество элементов в очереди
   const int qsize; // максимальное количество элементов в очереди
   QueueTP(const QueueTP & q) : qsize(0) {}
   QueueTP & operator=(const QueueTP & q) { return *this; }
public:
   QueueTP(int qs = Q_SIZE);
   ~QueueTP();
   bool isempty() const
   {
      return items == 0;
   }
```

```
   bool isfull() const
   {
      return items == qsize;
   }
   int queuecount() const
   {
      return items;
   }
   bool enqueue(const Item &item); // добавление элемента в конец
   bool dequeue(Item &item); // удаление элемента с начала
};
// методы QueueTP
template <class Item>
QueueTP<Item>::QueueTP(int qs) : qsize(qs)
{
   front = rear = 0;
   items = 0;
}
template <class Item>
QueueTP<Item>::~QueueTP()
{
   Node * temp;
   while (front != 0)        // пока очередь не пуста
   {
      temp = front;          // сохраним адрес начального элемента
      front = front->next;   // установим указатель на следующий элемент
      delete temp;           // удалим формирователь начала
   }
}
// Добавление элемента в очередь
template <class Item>
bool QueueTP<Item>::enqueue(const Item & item)
{
   if (isfull())
      return false;
   Node * add = new Node(item); // создаем узел node
   if (add == NULL)
      return false;          // выход, если ничего нет
   items++;
   if (front == 0)           // если очередь пуста,
      front = add;           // добавим элемент в начало
   else
      rear->next = add;      // иначе добавим в конец
   rear = add;               // возьмем последний элемент в новый узел
   return true;
}
// Помещение начального элемента в переменную и удаление его из очереди
template <class Item>
bool QueueTP<Item>::dequeue(Item & item)
{
   if (front == 0)
      return false;
   item = front->item; // установим элемент в первый элемент в очереди
```

```
    items--;
    Node * temp = front;    // сохраним расположение первого элемента
    front = front->next;    // установим начало на следующий элемент
    delete temp;            // удалим формирователь первого элемента
    if (items == 0)
        rear = 0;
    return true;
}
#endif
```

Интересный момент в листинге 15.5 связан с тем, что `Node` определен как общий тип `Item`. Таким образом, объявление

```
QueueTp<double> dq;
```

приводит к тому, что `Node` будет определен для значений типа `double`, тогда как

```
QueueTp<char> cq;
```

приводит к тому, что `Node` будет определен для значений типа `char`. Эти два класса `Node` определены для разных классов `QueueTP`, поэтому между ними не существует конфликта имен. То есть один `Node` имеет тип `QueueTP<double>::Node`, другой — тип `QueueTP<char>::Node`.

В листинге 15.6 предлагается короткая программа для тестирования нового класса.

Листинг 15.6. `nested.cpp`

```
// nested.cpp -- использование очереди, имеющей вложенный класс
#include <iostream>
#include <string>
#include "queuetp.h"
int main()
{
    using std::string;
    using std::cin;
    using std::cout;
    QueueTP<string> cs(5);
    string temp;
    while(!cs.isfull())
    {
        cout << "Пожалуйста, введите свое имя и фамилию. Вы будете "
            "обслужены в порядке прибытия.\n"
            "Имя и фамилия: ";
        getline(cin, temp);
        cs.enqueue(temp);
    }
    cout << "Очередь полна. Начало обслуживания!\n";
    while (!cs.isempty())
    {
        cs.dequeue(temp);
        cout << "Обслуживается " << temp << "...\n";
    }
    return 0;
}
```

Замечание по совместимости

Некоторые устаревшие компиляторы корректно не поддерживают `getline()`. В этом случае можно заменить `getline(cin, temp)` на `cin >> temp`. Следует отметить, что это ограничивает каждый элемент ввода до одного слова вместо целой строки.

Ниже показан вывод программы из листингов 15.5 и 15.6:

```
Пожалуйста, введите свое имя и фамилию. Вы будете обслужены в порядке
прибытия.
Имя и фамилия: Kinsey Millhone
Пожалуйста, введите свое имя и фамилию. Вы будете обслужены в порядке
прибытия.
Имя и фамилия: Adam Dalgliesh
Пожалуйста, введите свое имя и фамилию. Вы будете обслужены в порядке
прибытия.
Имя и фамилия: Andrew Dalziel
Пожалуйста, введите свое имя и фамилию. Вы будете обслужены в порядке
прибытия.
Имя и фамилия: Kay Scarpetta
Пожалуйста, введите свое имя и фамилию. Вы будете обслужены в порядке
прибытия.
Имя и фамилия: Richard Jury
Очередь полна. Начало обслуживания!
Обслуживается Kinsey Millhone...
Обслуживается Adam Dalgliesh...
Обслуживается Andrew Dalziel...
Обслуживается Kay Scarpetta...
Обслуживается Richard Jury...
```

Исключения

В программах иногда встречаются ситуации, препятствующие их нормальному выполнению. Например, программа может попытаться открыть недоступный файл, или может запросить памяти больше, нежели доступно в данный момент, или встретить значение, которое она не ожидает. Обычно программисты стараются предвидеть такие события. Исключения в C++ предлагают мощный и гибкий механизм обработки таких ситуаций. Поскольку исключения являются одним из последних расширений языка C++, некоторые устаревшие компиляторы не поддерживают их. В некоторых компиляторах эта возможность отключена по умолчанию, поэтому для использования исключений эту возможность необходимо включить.

Перед тем как заняться исключениями, рассмотрим некоторые простые методы, доступные программистам. В качестве теста рассмотрим функцию, которая рассчитывает среднее гармоническое значение. *Среднее гармоническое* двух чисел определяется как инверсия среднего значения инверсий. Это можно записать в виде следующего выражения:

$$2.0 \times x \times y \ / \ (x + y)$$

Обратите внимание, что если y равно x с обратным знаком, вычисление по формуле приведет к делению на ноль — весьма нежелательная ситуация. Многие новые компиляторы контролируют деление на ноль путем введения специальной величины

в формате с плавающей точкой, обозначающей бесконечность; cout отображает эту величину как Inf, inf или INF либо что-то подобное. Другие компиляторы генерируют программы, которые при выполнении деления на ноль аварийно завершаются. Лучше всего создавать код, который ведет себя предсказуемо в любой системе.

Вызов abort()

Одним из способов выхода из создавшейся ситуации является создание функции, вызывающей функцию abort(), если один аргумент равен другому с обратным знаком. У функции abort() имеется прототип в заголовочном файле cstdlib (или stdlib.h). В типичной реализации в стандартный поток ошибок посылается сообщение вроде "abnormal program termination" ("аварийное завершение программы", то есть то же самое, что и при использовании cerr) и программа останавливается.

При этом в зависимости от реализации, операционной системе или родительскому процессу возвращается значение, свидетельствующее о крахе программы. Очищает ли функция abort() файловые буферы (промежуточную память, выделенную для хранения данных при операциях с файлами) или нет, зависит от реализации. Можно использовать функцию exit(), которая очищает файловые буферы и не выводит сообщения. В листинге 15.7 показана короткая программа, использующая abort().

Листинг 15.7. error1.cpp

```
//error1.cpp -- использование функции abort()
#include <iostream>
#include <cstdlib>
double hmean(double a, double b);
int main()
{
   double x, y, z;
   std::cout << "Введите два числа: ";
   while (std::cin >> x >> y)
   {
      z = hmean(x,y);
      std::cout << "Среднее гармоническое " << x << " и " << y
                << " равно " << z << std::endl;
      std::cout << "Введите следующий набор чисел <q для завершения>: ";
   }
   std::cout << "Всего наилучшего!\n";
   return 0;
}
double hmean(double a, double b)
{
   if (a == -b)
   {
      std::cout << "непригодные аргументы для hmean()\n";
      std::abort();
   }
   return 2.0 * a * b / (a + b);
}
```

Ниже приведен вывод этой программы:

```
Введите два числа: 3 6
Среднее гармоническое 3 и 6 равно 4
Введите следующий набор чисел <q для завершения>: 10 -10
непригодные аргументы для hmean()
abnormal program termination
```

Обратите внимание, что вызов функции `abort()` из `hmean()` сразу же останавливает программу без возврата в `main()`. (В общем, разные компиляторы выдают разные сообщения об остановке программы.) Можно избежать остановки программы, проверяя значения `x` и `y` перед вызовом функции `hmean()`. Но не очень-то надежно полагаться на программиста, который еще должен знать (или заботиться) о том, как выполнить такую проверку.

Возврат кода ошибки

Для определения возникшей проблемы более гибким подходом, нежели прерывание программы, является использование возвращаемой функцией величины. Например, член `get(void)` класса `ostream` обычно возвращает ASCII-код очередного введенного символа, однако он возвращает специальное значение `EOF`, если обнаруживает конец файла. Этот подход не работает для `hmean()`. Поскольку любое числовое значение является допустимым возвращаемым значением, не существует специальной значения для отражения возникшей проблемы. В этой ситуации можно в качестве аргумента функции использовать указатель или ссылку, для того чтобы вернуть значение обратно в вызывающую программу, и применять это значение для определения успешности выполнения функции. Семейство `istream` перегруженных операций >> использует разновидность такой техники. Информируя вызывающую программу об успешности или неудаче выполнения фрагмента кода, можно предпринять действия, отличные от аварийного завершения программы. В листинге 15.8 показан пример такого подхода. В нем `hmean()` переопределена как функция, которая возвращает значение `bool`, показывающее, успешно ли выполнена функция. В ней также добавлен третий аргумент для получения ответа.

Листинг 15.8. `error2.cpp`

```
//error2.cpp -- возврат кода ошибки
#include <iostream>
#include <cfloat> // (или float.h) для DBL_MAX
bool hmean(double a, double b, double * ans);
int main()
{
   double x, y, z;
   std::cout << "Введите два числа: ";
   while (std::cin >> x >> y)
   {
      if (hmean(x,y,&z))
         std::cout << "Среднее гармоническое " << x << " и " << y
                   << " равно " << z << std::endl;
      else
         std::cout << "Одно значение не может быть негативной "
                   << "копией другого — повторите ввод.\n";
```

```
      std::cout << "Введите следующий набор чисел <q для завершения>: ";
   }
   std::cout << "Всего наилучшего!\n";
   return 0;
}
bool hmean(double a, double b, double * ans)
{
   if (a == -b)
   {
      *ans = DBL_MAX;
      return false;
   }
   else
   {
      *ans = 2.0 * a * b / (a + b);
      return true;
   }
}
```

Вот как выглядит вывод программы:

```
Введите два числа: 3 6
Среднее гармоническое 3 и 6 равно 4
Введите следующий набор чисел <q для завершения>: 10 -10
Одно значение не может быть негативной копией другого — повторите ввод.
Введите следующий набор чисел <q для завершения>: 1 19
Среднее гармоническое 1 и 19 равно 1.9
Введите следующий набор чисел <q для завершения>: q
Всего наилучшего!
```

Замечания по программе

Дизайн программы в листинге 15.8 позволяет пользователю продолжить программу, обойдя ситуацию, возникшую из-за неправильного ввода. Конечно, программа основана на проверке пользователем значений возвращаемых функцией, то есть на том, что не всегда делают программисты. Например, для сокращения размеров программ в большинстве листингов этой книги не проверяется, возвращает ли нулевой указатель функция `new()` и успешно ли завершился вывод с помощью `cout`.

Можно использовать указатель или ссылку в качестве третьего аргумента. Многие программисты предпочитают применять указатели на аргументы встроенных типов, поскольку тогда понятно, какой аргумент использовался для ответа.

Другой вариант идеи сохранения возвращаемых значений предусматривает применение глобальных переменных. Функция, имеющая потенциальную проблему, в случае возникновения проблемы может установить глобальную переменную в определенное значение, а вызывающая программа может проверить эту переменную. Этот метод реализован в стандартной математической библиотекой языка С, которая для этих целей использует глобальную переменную `errno`. Конечно, нужно быть уверенным, что какая-нибудь другая функция не использует глобальную переменную с таким же именем для других целей.

Механизм исключений

Давайте посмотрим, как можно управлять проблемными ситуациями, используя механизм исключений. В C++ *исключение* — это реакция на нештатную ситуацию, возникающую во время выполнения программы, например, при делении на ноль. Исключения позволяют передать управление из одной части программы в другую.

Управление исключениями предусматривает три компонента:

- Генерация исключения.
- Перехват исключения обработчиком.
- Использование блока try.

Программа генерирует исключение, когда возникает проблемная ситуация. Достаточно легко модифицировать hmean() из листинга 15.7 для того, чтобы генерировать исключения вместо вызова функции abort(). В сущности, оператор throw является своеобразным переходом, поскольку он приводит к переходу программы к оператору, расположенному в другом месте. Ключевое слово throw является признаком генерации исключения. После него идет величина, например, символьная строка или объект, обозначающая природу исключения. Программа перехватывает исключение с помощью *обработчика исключений* в том месте программы, где исключение необходимо обработать. Ключевое слово catch означает перехват исключения. Обработчик исключения начинается с ключевого слова catch, за которым следует объявление типа (в круглых скобках), представляющее тип исключения, коему оно соответствует. Потом, по порядку, в фигурных скобках идет блок кода, представляющий действия, которые нужно предпринять. Ключевое слово catch вместе с типом исключения выполняет роль метки, определяющей точку в программе, куда должно быть передано управление при возникновении исключения. Обработчик исключения называется также *блоком перехвата* (catch-блоком).

Блок try представляет собой блок кода, для которого может быть сгенерировано определенное исключение. За ним следуют один или несколько catch-блоков. Блок try начинается с ключевого слова try, а за ним в фигурных скобках находится код, для которого определяется исключение.

Проще всего увидеть взаимодействие этих трех элементов, рассмотрев короткий пример, приведенный в листинге 15.9.

Листинг 15.9. error3.cpp

```
// error3.cpp -- использование исключения
#include <iostream>
double hmean(double a, double b);

int main()
{
   double x, y, z;
   std::cout << "Введите два числа: ";
   while (std::cin >> x >> y)
   {
      try { // начало блока try
         z = hmean(x,y);
      } // конец блока try
```

```
      catch (const char * s) // начало обработчика исключений
      {
         std::cout << s << std::endl;
         std::cout << "Введите новую пару чисел: ";
         continue;
      } // конец обработчика исключений
      std::cout << "Среднее гармоническое " << x << " и " << y
                << " равно " << z << std::endl;
      std::cout << "Введите следующий набор чисел <q для завершения>: ";
   }
   std::cout << "Всего наилучшего!\n";
   return 0;
}
double hmean(double a, double b)
{
   if (a == -b)
      throw "Неправильные аргументы hmean(): a = -b не допускается";
   return 2.0 * a * b / (a + b);
}
```

Ниже показан вывод программы:

```
Введите два числа: 3 6
Среднее гармоническое 3 и 6 равно 4
Введите следующий набор чисел <q для завершения>: 10 -10
Неправильные аргументы hmean(): a = -b не допускается
Введите новую пару чисел: 1 19
Среднее гармоническое 1 и 19 равно 1.9
Введите следующий набор чисел <q для завершения>: q
Всего наилучшего!
```

Замечания по программе

Блок `try` в листинге 15.9 выглядит следующим образом:

```
try { // начало блока try
   z = hmean(x,y);
} // конец блока try
```

Если какой-то оператор в этом блоке приведет к генерации исключения, то обработка исключения произойдет в блоках `catch`, следующих за этим `try`-блоком. Если программа вызовет `hmean()` где-нибудь вне этого (или любого другого) `try`-блока, она не сможет обработать исключение.

Генерация исключения выглядит следующим образом:

```
if (a == -b)
   throw "Неправильные аргументы hmean(): a = -b не допускается";
```

В данном случае генерируемое исключение является строкой `"Неправильные аргументы hmean(): a = -b не допускается"`. Исключение может иметь строковый тип, как в данном примере, или любой другой тип C++. Часто исключение может иметь тип класса, что иллюстрируется примерами в этой главе. Выполнение оператора `throw` похоже на выполнение оператора возврата в функции, в том смысле, что

он завершает выполнение функции. Однако вместо возврата управления вызывающей программе оператор throw заставляет программу возвращаться назад по текущей цепочке вызовов функций до тех пор, пока не будет найден try-блок. В листинге 15.9 такой функцией является вызывающая функция. Ниже мы рассмотрим пример с возвратом более чем на один уровень вызова функций. В данном случае оператор throw передает управление обратно в main(). Здесь программа ищет обработчик исключения (следующий за блоком try), который соответствует типу сгенерированного исключения.

Обработчик или catch-блок выглядит следующим образом:

```
catch (char * s) // начало обработчика исключений
{
   std::cout << s << std::endl;
   sdt::cout << "Введите новую пару чисел: ";
   continue;
} // конец обработчика
```

Блок catch похож на определение функции, но это — не функция. Ключевое слово catch говорит о том, что это — обработчик, а выражение char* s означает, что обработчик соответствует исключению, являющемуся строкой. Такое объявление s работает как определение аргумента функции, в котором соответствующее сгенерированное исключение присваивается s. Если исключение не соответствует обработчику, программа выполняет код внутри фигурных скобок. Если программа выполняет все операторы внутри try-блока без возникновения исключений, то она пропускает все catch-блоки и переходит к выполнению операторов, следующих за обработчиками исключений. Когда программа из листинга 15.9 обрабатывает значения 3 и 6, она переходит прямо к оператору вывода и отображает результат.

Проследим события, возникающие в примере, после того как значения 10 и -10 будут обработаны функцией hmean(). Проверка if заставляет hmean() сгенерировать исключение. Выполнение hmean() прерывается. Просматривая код назад, программа определяет, что hmean() была вызвана внутри try-блока в main(). Затем программа ищет catch-блок с типом, который соответствует типу возникшего исключения. Единственный существующий catch-блок имеет параметр char *, поэтому он соответствует исключению. Найдя соответствие, программа присваивает переменной s значение "Неправильные аргументы hmean(): a = -b не допускается". Затем программа выполняет код обработчика. Сначала она печатает s, идентифицирующее исключение. Затем программа печатает инструкцию, требующую от пользователя ввода новых данных.

И, наконец, программа выполняет оператор continue, который прекращает цикл while и передает управление на его начало. То, что оператор continue переводит программу в начало цикла, говорит о том, что обработчик является частью цикла, и что строки catch служат метками, определяющими дальнейшее выполнение программы (рис. 15.2). Возникает вопрос, что произойдет, если функция сгенерирует исключение и не будет определено ни одного try-блока или соответствующего обработчика. По умолчанию программа вызовет функцию abort(), но такое поведение программы можно изменить. Мы вернемся к этой теме позже в этой главе.

```
...
while (std::cin >> x >> y)
{
  try { // начало блока try
      z = hmean(x,y);
  } // конец блока try
  catch (const char * s) // начало обработчика исключений
  {
      std::cout << s << std::endl;
      std::cout << "Введите новую пару чисел: ";
      continue;
  } // конец обработчика исключений
  std::cout << "Среднее гармоническое " << x << " и " << y
          << " равно " << z << std::endl;
  std::cout << "Введите следующий набор чисел <q для завершения>: ";
}
...
double hmean(double a, double b)
{
  if (a == -b)
    throw "Неправильные аргументы hmean(): a = -b не допускается";
  return 2.0 * a * b / (a + b);
}
```


1. Программа вызывает **hmean()** внутри try-блока.
2. **hmean()** генерирует исключение, передает управление в catch-блок и присваивает переменной **s** значение строки исключения.
3. catch-блок передает управление назад, в начало цикла.

Рис. 15.2. Выполнение программы с исключениями

Использование объектов в качестве исключений

Обычно программы, которые генерируют исключения, создают объекты. Преимущества такого подхода состоит в том, что можно для разделения различных функций и ситуаций, генерирующих исключения, применять разные типы исключений. Объект может управлять собственными данными, и эту информацию можно использовать для определения причин, вызвавших исключение. Используя эту же информацию, catch-блок может определить, какие действия следует предпринять. Ниже показан один из возможных вариантов исключений, генерируемых функцией `hmean()`:

```
class bad_hmean
{
private:
   double v1;
   double v2;
public:
   bad_hmean(int a = 0, int b = 0) :v1(a), v2(b){}
   void mesg();
};
inline void bad_hmean::mesg()
{
   std::cout << "hmean(" << v1 << ", " << v2 <<"): "
             << "Недопустимые аргументы: a = -b\n";
}
```

Объект bad_hmean можно инициализировать значением, передаваемым в hmean(), а метод mesg() можно использовать для сообщения о возникшей проблеме. В функции hmean() можно использовать следующий код:

```
if (a == -b)
   throw bad_hmean(a,b);
```

Здесь вызывается конструктор bad_hmean, который инициализирует объект для хранения значения аргумента. Для того чтобы определить тип возникшего исключения, можно квалифицировать определение *функции спецификацией исключения*. Для этого потребуется добавить спецификацию исключения, которая состоит из ключевого слова throw, за которым в круглых скобках через запятую следуют типы исключений:

```
double hmean(double a, double b) throw(bad_hmean);
```

Такое определение делает две вещи. Во-первых, оно говорит компилятору, какой тип исключения генерирует функция. Если функция в дальнейшем сгенерирует какой-то другой тип исключений, программа среагирует на это вызовом функции abort(). (Позже в этой главе мы детально рассмотрим такое поведение программы и то, как его можно изменить.) Во-вторых, наличие спецификации исключения напоминает пользователю, что данная конкретная функция генерирует исключение и что можно определить try-блок и обработчик исключения. В функции, которая генерирует несколько видов исключений, можно задать разделенный запятыми список типов исключений; синтаксис аналогичен списку аргументов для прототипа функции. Например, следующий прототип представляет функцию, которая может генерировать исключения типов char* и double:

```
double multi_err(double z) throw(const char *, double);
```

Та же информация, что присутствует в прототипе, должна присутствовать и в определении функции:

```
double hmean(double a, double b) throw(bad_hmean)
{
   if (a == -b)
      throw bad_hmean(a,b);
   return 2.0 * a * b / (a + b);
}
```

Если в спецификации исключения используются пустые скобки, это значит, что функция не генерирует исключений:

```
double simple(double z) throw(); // не генерирует исключений
```

В листингах 15.10 и 15.11 добавляется другой класс исключений bad_gmean и другая функция gmean(), которая генерирует исключение bad_gmean. Функция gmean() рассчитывает среднее геометрическое двух чисел, являющееся квадратным корнем из их произведения. Эта функция определена, если оба аргумента не отрицательны. Поэтому она генерирует исключение, если обнаруживает отрицательные аргументы. Листинг 15.10 — это заголовочный файл, содержащий определение класса исключения, а листинг 15.11 — пример программы, использующей этот файл. Обратите внимание, что после блока try идут два последовательных catch-блока:

```
try { // начало блока try
   ...
}// конец блока try
catch (bad_hmean & bg) // начало блока catch
{
   ...
}
catch (bad_gmean & hg)
{
   ...
} // конец catch блока
```

Если hmean() сгенерирует исключение bad_hmean, первый catch-блок перехватит его. Если gmean() сгенерирует исключение bad_gmean, то оно пройдет через первый catch-блок и будет перехвачено вторым catch-блоком.

Листинг 15.10. exc_mean.cpp

```
// exc_mean.h -- классы исключений для hmean() и gmean()
#include <iostream>
class bad_hmean
{
private:
   double v1;
   double v2;
public:
   bad_hmean(double a = 0, double b = 0) : v1(a), v2(b){}
   void mesg();
};
inline void bad_hmean::mesg()
{
   std::cout << "hmean(" << v1 << ", " << v2 <<"): "
              << "неправильные аргументы: a = -b\n";
}
class bad_gmean
{
public:
   double v1;
   double v2;
   bad_gmean(double a = 0, double b = 0) : v1(a), v2(b){}
   const char * mesg();
};
inline const char * bad_gmean::mesg()
{
   return "Аргументы gmean() должны быть >= 0\n";
}
```

Листинг 15.11. error4.cpp

```
// error4.cpp -- использование классов исключений
#include <iostream>
#include <cmath>    // или math.h, пользователям unix может
                    // потребоваться указать флаг -lm
```

```
#include "exc_mean.h"
// прототипы функции
double hmean(double a, double b) throw(bad_hmean);
double gmean(double a, double b) throw(bad_gmean);
int main()
{
   using std::cout;
   using std::cin;
   using std::endl;
   double x, y, z;
   cout << "Введите два числа: ";
   while (cin >> x >> y)
   {
      try { // начало блока try
         z = hmean(x,y);
         cout << "Среднее гармоническое " << x << " и " << y
              << " равно " << z << endl;
         cout << "Среднее геометрическое " << x << " и " << y
              << " равно " << gmean(x,y) << endl;
         cout << "Введите следующий набор чисел <q для завершения>: ";
      }// конец  блока try
      catch (bad_hmean & bg) // начало блока catch
      {
         bg.mesg();
         cout << "Повторите ввод.\n";
         continue;
      }
      catch (bad_gmean & hg)
      {
         cout << hg.mesg();
         cout << "Используемые значения: " << hg.v1 << ", "
              << hg.v2 << endl;
         cout << "Извините, стараться больше не нужно.\n";
         break;
      } // конец блока catch
   }
   cout << "Всего наилучшего!\n";
   return 0;
}
double hmean(double a, double b) throw(bad_hmean)
{
   if (a == -b)
      throw bad_hmean(a,b);
   return 2.0 * a * b / (a + b);
}
double gmean(double a, double b) throw(bad_gmean)
{
   if (a < 0 || b < 0)
      throw bad_gmean(a,b);
   return std::sqrt(a * b);
}
```

Ниже приведен пример вывода программы из листингов 15.10 и 15.11, который прерывается неправильным вводом для функции gmean():

```
Введите два числа: 4 12
Среднее гармоническое 4 и 12 равно 6
Среднее геометрическое 4 и 12 равно 6.9282
Введите следующий набор чисел <q для завершения>: 5 -5
hmean(5, -5): неправильные аргументы: a = -b
Повторите ввод.
5 -2
Среднее гармоническое 5 и -2 равно -6.66667
Аргументы gmean() должны быть >= 0
Используемые значения: 5, -2
Извините, стараться больше не нужно.
Всего наилучшего!
```

Следует отметить, что обработчик bad_hmean использует оператор continue, а обработчик bad_gmean — оператор break. Таким образом, неправильный ввод в функции hmean() приводит к тому, что программа пропускает остаток цикла и переходит к его началу. Неправильный ввод в функции gmean() прерывает цикл. Это показывает, как программа определяет, какое исключение возникло (по типу исключения), и выбирает реакцию на исключение. Необходимо также отметить, что техники, используемые в bad_gmean и bad_hmean, различны. В частности, в bad_gmean используются общедоступные данные и метод, возвращающий строку C-стиля.

Раскручивание стека

Предположим, что блок try не содержит непосредственного вызова функции, генерирующей исключение, но потом он вызывает функцию, которая, в свою очередь, обращается к функции, генерирующей исключение. Управление передается из функции, в которой возникает исключение, в функцию, содержащую блок try и обработчик. Это называется *раскручиванием стека*, именно ее мы сейчас обсуждаем. Посмотрим, как обычно C++ обрабатывает вызовы функций и возвраты из них. C++ обычно обрабатывает вызов функции, располагая информацию в стеке (см. главу 9). В частности, в стеке сохраняется адрес инструкции вызова функции (*адрес возврата*). Когда вызываемая функция завершает свою работу, программа использует этот адрес для определения точки, с которой нужно продолжить выполнение программы. Аргументы функции также сохраняются в стеке и трактуются как автоматические переменные. Если вызванная функция создает новые автоматические переменные, то они тоже сохраняются в стеке. Если вызванная функция вызывает другую функцию, то ее информация также добавляется в стек, и так далее. Если выполнение функции прерывается, выполнение программы начинается с адреса, сохраненного в момент вызова этой функции, и вершина стека освобождается. То есть функция возвращает управление в функцию, которая ее вызвала, при этом каждая функция при завершении освобождает свои автоматические переменные. Если автоматической переменной является объект класса, то вызывается соответствующий деструктор класса.

Предположим, что функция вместо нормального завершения прерывает свою работу с генерацией исключения. При этом программа, как и ранее, очищает стек. Но вместо того, чтобы остановиться на первом адресе возврата в стеке, программа продолжает очищать стек, пока не достигнет адреса возврата, который находится в

блоке try (рис. 15.3). Затем управление передается в обработчики исключения в конце блока, а не оператору, следующему за оператором вызова функции. Это процесс и называется раскручиванием стека. Очень важной особенностью механизма throw (генерации исключений) является то, что, как и при возврате из функции, деструкторы вызываются для всех автоматических объектов класса в стеке. Тем не менее, при возврате из функции обрабатываются объекты, помещенные в стек при вызове функции, а оператор throw обрабатывает объекты, помещенные в стек целой цепочкой вызовов функций между блоком try и оператором throw. Без раскручивания стека оператор throw не вызвал бы деструкторы автоматических объектов, помещенных в стек промежуточными вызовами функций.

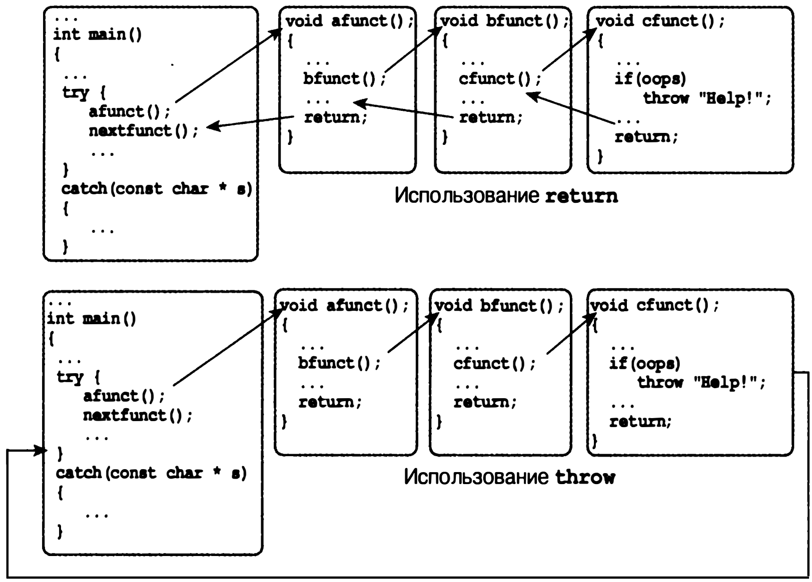


Рис. 15.3. Сравнение операторов throw и return

В листинге 15.12 показан пример раскручивания стека. В нем функция main() вызывает means(), которая, в свою очередь, вызывает hmean() и gmean(). Функция means() за неимением ничего лучшего вычисляет значения арифметического, гармонического и геометрического средних. Обе функции main() и means() создают объекты типа demo ("разговорчивый" класс, анонсирующий, когда используются его конструктор и деструктор), так что можно видеть, что происходит с этими объектами при генерации исключений. Блок try в функции main() перехватывает оба исключения bad_hmean и bad_gmean, а блок try в функции means() перехватывает только исключение bad_hmean. Код блока catch выглядит следующим образом:

```
catch (bad_hmean & bg) // начало блока catch
{
   bg.mesg();
   std::cout << "Перехвачено в means()\n";
   throw; // повторная генерация исключения
}
```

После реакции на исключение и отображения соответствующих сообщений, код снова генерирует исключение, что, в данном случае, означает передачу исключения выше, в функцию `main()`. (В общем случае gjdnjhyfz генерация исключения приводит программу к новой комбинации `try-catch`, которая перехватывает конкретный тип исключения. Если обработчик исключения отсутствует, программа прерывает свое выполнение.) В листинге 15.12 используется тот же заголовочный файл, что и в листинге 15.11.

Листинг 15.12. error5.cpp

```
//error5.cpp -- раскручивание стека
#include <iostream>
#include <cmath> // или math.h, пользователям unix может потребоваться
указать флаг -lm
#include <cstring>
#include "exc_mean.h"
class demo
{
private:
   char word[40];
public:
   demo (const char * str)
   {
      std::strcpy(word, str);
      std::cout << "demo " << word << " создан\n";
   }
   ~demo()
   {
      std::cout << "demo " << word << " уничтожен\n";
   }
   void show() const
   {
      std::cout << "demo " << word << " жив!\n";
   }
};
// прототипы функций
double hmean(double a, double b) throw(bad_hmean);
double gmean(double a, double b) throw(bad_gmean);
double means(double a, double b) throw(bad_hmean, bad_gmean);
int main()
{
   using std::cout;
   using std::cin;
   using std::endl;
   double x, y, z;
   demo d1(", используемый в main(), ");
   cout << "Введите два числа: ";
   while (cin >> x >> y)
   {
      try { // начало блока try
         z = means(x,y);
            cout << "Среднее всех средних " << x << " и " << y
                 << " равно " << z << endl;
```

```
            cout << "Введите следующую пару: ";
        } // конец блока try
        catch (bad_hmean & bg) //начало блока catch
        {
            bg.mesg();
            cout << "Повторите ввод.\n";
            continue;
        }
        catch (bad_gmean & hg)
        {
            cout << hg.mesg();
            cout << "Используемые значения: " << hg.v1 << ", "
                 << hg.v2 << endl;
            cout << "Извините, стараться больше не нужно.\n";
            break;
        } // конец блока catch
    }
    d1.show();
    cout << "Всего наилучшего!\n";
    return 0;
}
double hmean(double a, double b) throw(bad_hmean)
{
    if (a == -b)
        throw bad_hmean(a,b);
    return 2.0 * a * b / (a + b);
}
double gmean(double a, double b) throw(bad_gmean)
{
    if (a < 0 || b < 0)
        throw bad_gmean(a,b);
    return std::sqrt(a * b);
}
double means(double a, double b) throw(bad_hmean, bad_gmean)
{
    double am, hm, gm;
    demo d2(", используемый в means(),");
    am = (a + b) / 2.0;         // арифметическое среднее
    try
    {
        hm = hmean(a,b);
        gm = gmean(a,b);
    }
    catch (bad_hmean & bg)      // начало блока catch
    {
        bg.mesg();
        std::cout << "Перехвачено в means()\n";
        throw;                  // повторная генерация исключения
    }
    d2.show();
    return (am + hm + gm) / 3.0;
}
```

Вот как выглядит вывод программы, представленной в листингах 15.10 и 15.12:

```
demo, используемый в main(), создан
Введите два числа: 6 12
demo, используемый в means(), создан
demo, используемый в means(), жив!
demo, используемый в means(), уничтожен
Среднее всех средних 6 и 12 равно 8.49509
6 -6
demo, используемый в means(), создан
hmean(6, -6): неправильные аргументы: a = -b
Перехвачено в means()
demo, используемый в means(), уничтожен
hmean(6, -6): неправильные аргументы: a = -b
Повторите ввод.
6 -8
demo, используемый в means(), создан
demo, используемый в means(), уничтожен
Аргументы gmean() должны быть >= 0
Используемые значения: 6, -8
Извините, стараться больше не нужно.
demo, используемый в main(), жив!
Всего наилучшего!
demo, используемый в main(), уничтожен
```

Замечания по программе

Давайте проследим выполнение программы из предыдущего раздела. Сначала в main() с помощью конструктора создается объект demo. Дальше вызывается means() и создается другой объект demo. Функция means() передает значения 6 и 12 в hmean() и gmean(); эти функции возвращают значение в means(), которая рассчитывает результат и возвращает его. Перед тем, как вернуть результат, means() вызывает d2.show(). Вернув результат, means() завершается и деструктор для d2 вызывается автоматически:

```
demo, используемый в means(), жив!
demo, используемый в means(), уничтожен
```

Следующий цикл ввода передает значения 6 и -6 в means(), после чего means() создает новый объект demo и передает значения в hmean(). Функция hmean() генерирует исключение bad_hmean, которое в means() перехватывается блоком catch, что видно из следующего фрагмента:

```
hmean(6, -6): неправильные аргументы: a = -b
Перехвачено в means()
```

Оператор throw завершает выполнение means() и передает исключение в main(). То, что d2.show не вызывается, показывает, что means() завершена. Обратите внимание, что деструктор для d2 вызывается:

```
demo, используемый в means(), уничтожен
```

Это демонстрирует очень важный аспект исключений: когда программа раскручивает стек, чтобы обнаружить, где исключение будет перехвачено, она очищает переменные класса, автоматически сохраненные в стеке. Если переменная является

объектом класса, то автоматически вызывается деструктор этого класса. Повторно сгенерированное исключение достигает main(), где обрабатывается соответствующим блоком catch:

```
hmean(6, -6): неправильные аргументы: a = -b
Повторите ввод.
```

Третий цикл ввода передает в means() значения 6 и -8. И снова maeans() создает новый объект demo. Он передает 6 и -8 в функцию hmean(), которая обрабатывает их без проблем. Затем hmean() передает 6 и -8 в функцию gmean(), которая генерирует исключение bad_gmean. Поскольку means() не перехватывает исключение, она передает его дальше в main(). И опять-таки, поскольку программа раскручивает стек, она освобождает локальные автоматические переменные и вызывает деструктор для d2:

```
demo, используемый в means(), уничтожен
```

Наконец, обработчик bad_gmean в main() перехватывает исключение и завершает цикл:

```
Аргументы gmean() должны быть >= 0
Используемые значения: 6, -8
Извините, стараться больше не нужно.
```

Далее программа нормально завершается, отображает несколько сообщений и автоматически вызывает конструктор для d1. Если используемый блок catch вызовет exit(EXIT_FAILURE) вместо break, программа завершится немедленно, и вы не увидите сообщений:

```
demo, используемый в main(), жив!
Всего наилучшего!
```

Однако будет выдано сообщение:

```
demo, используемый в main(), уничтожен
```

И снова, механизм исключений будет освобождать автоматические переменные в стеке.

Обратите внимание на спецификацию исключения для means():

```
double means(double a, double b) throw(bad_hmean, bad_gmean);
```

Видно, что means() может генерировать оба исключения — bad_hmean и bad_gmean. Но единственным исключением, которое явно генерируется в means(), является bad_hmean, которое повторно генерируется в обработчике bad_hmean. Спецификация исключения должна содержать не только исключение, генерируемое самой функцией, но и исключения, генерируемые функциями, вызываемыми этой функцией, и так далее. Таким образом, поскольку means() вызывает gmeans(), она должна объявлять, что может передавать исключения, генерируемые функцией gmeans().

Что произойдет, если убрать bad_gmean & из спецификации исключения, а gmean() сгенерирует исключение? Это пример непредвиденного исключения, он будет рассмотрен ниже в этой главе, в разделе “Потеря исключений”.

Дополнительные свойства исключений

Хотя механизм throw-catch имеет много общего с аргументами и механизмом возврата функций, существуют и отличия. Одно, уже рассмотренное нами, заключается в том, что оператор возврата функции fun() передает управление функции, из которой была вызвана fun(). А оператор throw передает управление по цепочке вверх в первую функцию, содержащую комбинацию try-catch, которая перехватывает исключение. Например, в листинге 15.12, когда hmean() генерирует исключение, управление передается наверх в means(), но когда gmean() генерирует исключение, управление передается наверх в main().

Другое отличие состоит в том, что когда компилятор генерирует исключение, он всегда создает временную копию, даже если спецификатор исключения и блок catch специфицируют ссылку:

```
class problem {...};
...
void super() throw (problem)
{
   ...
   if (oh_no)
   {
      problem oops; // конструктор объекта исключения
      throw oops;   // сгенерируем его
   ...
}
...
try {
   super();
}
catch(problem & p)
{
   // операторы
}
```

Здесь p ссылается на копию oops, а не на сам oops. Это хорошо, потому что oops не будет существовать, после того, как завершится super(). Кстати, проще скомбинировать создание с оператором throw:

```
throw problem(); // конструирует и генерирует объект problem по умолчанию
```

Удивительно, зачем в коде используется ссылка, если throw генерирует копию. В конце концов, целью применения ссылочных возвращаемых величин является отсутствие необходимости создания копии объекта. Ответ заключается в том, что ссылка обладает другим важным свойством: ссылка на базовый класс может ссылаться на объект производного класса. Предположим, что существует коллекция типов исключений, которые связаны наследованием. В этом случае спецификация исключения требует только списка ссылок на базовый тип, и будет соответствовать любому исключению, сгенерированному производными объектами.

Допустим, что имеется иерархия классов исключений и нужно обрабатывать разные типы исключений по-разному. Ссылка на базовый класс может перехватывать все объекты семейства, но объект производного класса может перехватить только этот объект и объекты классов производных от этого класса. Сгенерированный объект

будет перехвачен первым же соответствующим блоком catch. Это наводит на мысль расположить блоки catch в обратном порядке:

```
class bad_1 {...};
class bad_2 : public bad_1 {...};
class bad_3 : public bad_2 {...};
...
void duper() throw (bad_1) //соответствует базовым и производным объектам
{
   ...
   if (oh_no)
      throw bad_1();
   if (rats)
      throw bad_2();
   if (drat)
      throw bad_3();
}
...
try {
   duper();
}
catch(bad_3 &be)
{ // операторы }
catch(bad_2 &be)
{ // операторы}
catch(bad_1 &be)
{ // операторы }
```

Если обработчик bad_1 & будет первым, он перехватит исключения bad_1, bad_2 и bad_3. При обратном порядке расположения обработчиков исключение bad_3& будет перехвачено обработчиком bad_3&.

Совет

Если существует иерархия наследования классов исключений, необходимо расположить блоки catch в таком порядке, чтобы исключение более позднего в цепочке наследования класса было перехвачено первым, а исключение базового класса — последним.

Расположение блоков catch в соответствующем порядке позволяет определить, как будут обрабатываться исключения различных типов. Но иногда невозможно предвидеть, исключение какого типа ожидать. Предположим, что создается функция, которая вызывает другую функцию, и мы не знаем, генерирует ли эта функция исключение. Можно перехватить исключение, даже не зная его типа. Хитрость заключается в использовании троеточия при указании типа исключения:

```
catch (...) { // операторы } // перехватывает любой тип исключений
```

Если тип некоторых исключений известен, можно расположить их в конце блока catch, подобно default в операторе switch:

```
try {
   duper();
}
catch(bad_3 &be)
{ // опрераторы }
```

```
catch(bad_2 &be)
{ // операторы}
catch(bad_1 &be)
{ // операторы }
catch(bad_hmean & h)
{ // операторы }
catch (...) // перехват всего, что осталось
{ // операторы }
```

Вместо ссылки для перехвата объекта можно предусмотреть обработчик. Объект базового класса будет перехватывать объекты производного класса, но характеристики производного объекта при этом будут скрыты. В результате будут использоваться виртуальные методы базового класса.

Класс exception

Главная цель введения исключений в C++ — создать средства на уровне языка для разработки надежных программ. Таким образом, исключения упрощают обработку ошибок в программах, при этом отпадает необходимость применения менее гибких методов обработки ошибок. Гибкость и относительное удобство исключений поощряют программистов к использованию этого механизма в своих разработках. Короче говоря, исключения — это будущее, которое, как и классы, может изменить сам подход к программированию. В новых компиляторах исключения входят в состав языка. Например, в заголовочном файле `exception` (точнее, `exception.h` и `except.h`) определен класс `exception`, который служит в C++ в качестве базового класса для других классов исключений. Ваш код может сгенерировать объект `exception` или использовать класс `exception` в качестве базового класса. Для реализации зависимостей используется виртуальная функция-член `what()`, возвращающая строку. Поскольку этот метод виртуальный, его можно переопределить в производном классе:

```
#include <exception>
class bad_hmean : public std::exception
{
public:
   const char * what() { return "Ошибочные аргументы для hmean()"; }
...
};
class bad_gmean : public std::exception
{
public:
   const char * what() { return "Ошибочные аргументы для gmean()"; }
   ...
};
```

Если нет необходимости отдельно обрабатывать это производное исключение, можно перехватить его в том же обработчике базового класса:

```
try {
   ...
}
catch(std::exception & e)
{
   cout << e.what() << endl;
   ...
}
```

Или можно перехватывать различные типы исключений по отдельности.

В библиотеке C++ определено много типов исключений, основанных на exception. В заголовочном файле exception определено bad_exception, которое используется функцией unexpected(). Эта функция рассматривается ниже в разделе “Потеря исключений”.

Классы исключений stdexcept

В заголовочном файле stdexcep определено еще несколько классов исключений. Во-первых, в нем определены классы logic_error и runtime_error, оба общедоступно порожденные от exception:

```
class logic_error : public exception {
public:
explicit logic_error(const string& what_arg);
...
};
class domain_error : public logic_error {
public:
explicit domain_error(const string& what_arg);
...
};
```

Обратите внимание, что конструкторы принимают объект string в качестве аргумента; этот аргумент с помощью метода what() возвращает символьные данные в виде строки C-стиля. Эти два новых класса служат основой для двух семейств производных классов. Семейство logic_error, как и можно было ожидать, описывает типичные логические ошибки. В принципе, таких ошибок можно избежать, но практически они все же возникают. По имени класса можно определить вид ошибок, для которых он предназначен:

```
domain_error
invalid_argument
length_error
out_of_bounds
```

У каждого класса имеется конструктор, как у logic_error, который возвращает строку с помощью метода what().

Возможно, будет полезно некоторое уточнение. Математическая функция имеет область определения и диапазон значений. Область определения состоит из значений, для которых функция определена, а диапазон значений — из значений, которые функция возвращает. Например, область определения функции синуса — от минус бесконечности до плюс бесконечности, поскольку синус определен для всех вещественных чисел. Но диапазон значений функции синуса — от –1 до +1, поскольку это максимальные значения синуса для угла. С другой стороны, область определения обратной функции, арксинуса, соответствует от –1 до +1, в то время как диапазон возвращаемых значений — от $-\pi$ до $+\pi$. Если написать функцию, которая передает аргумент в функцию std::sin(), то эта функция может сгенерировать объект domain_error, если аргумент будет вне области определения от –1 до +1.

Исключение invalid_argument предупреждает, что функции было передано непредвиденное значение. Например, если функция ожидает получить строку, в ко-

торой каждый символ является '1' или '0', она может сгенерировать исключение invalid_argument, если в строке встретится другой символ.

Исключение length_error используется, если для ожидаемого действия недостаточно пространства памяти. Например, класс string имеет метод append(), который генерирует исключение length_error, если результирующая строка больше допустимой величины.

Исключение out_of_bounds обычно служит для обозначения ошибок индексации. Например, можно определить класс массива, для которого operator()[] сгенерирует исключение out_of_bounds в том случае, если используемый индекс является недопустимым для этого массива.

Семейство runtime_error предназначено для ошибок, которые могут возникнуть во время выполнения программы, но не могут быть заранее предсказаны и предупреждены. Имя каждого класса определяет вид ошибок, для которых он предназначен:

```
range_error
overflow_error
underflow_error
```

У каждого класса имеется конструктор, как у runtime_error, который позволяет задать строку, возвращаемую методом what().

Ошибка потери значимости может возникать в вычислениях с плавающей точкой. В общем случае — это наименьшая величина, которая может быть представлена типом с плавающей точкой. Вычисления, использующие меньшие значения, приведут к генерации исключения потери значимости. Ошибка переполнения возникает для целых типов или типов с плавающей точкой, если величина результата превышает максимально возможное значение для этих типов. Результат вычисления может лежать вне допустимого диапазона без потери значимости или переполнения, в этом случае можно использовать исключение range_error.

Семейство logic_error может применяться для фрагментов программ, в которых появление ошибки runtime_error практически неизбежно. Оба эти класса ошибок обладают сходными характеристиками. Различие в том, что разные имена классов позволяют обрабатывать разные типы исключений индивидуально. С другой стороны, наследственные связи позволяют при необходимости объединить эти классы воедино. Например, следующий код перехватывает исключение out_of_bounds индивидуально, трактует семейство исключений logic_error как группу, а остальные объекты exception, семейство объектов runtime_error и объекты, производные от exception, обрабатывает коллективно:

```
try {
...
}
catch(out_of_bounds & oe) // перехват ошибки out_of_bounds
{...}
catch(logic_error & oe) //перехват остальных ошибок семейства logic_error
{...}
catch(exception & oe) // перехват runtime_error, объектов exception
{...}
```

Если какой-либо из этих библиотечных классов не соответствует вашим требованиям, от logic_error или runtime_error можно породить новый класс исключения, который войдет в общую иерархию.

Исключение bad_alloc и операция new

В языке C++ существуют две возможности управления распределением памяти при использовании операции new. Первая возможность — возврат операцией new нулевого (null) указателя, если она не в состоянии выделить требуемую память. Вторая возможность — генерация исключения bad_alloc. Заголовочный файл для new (обычно new.h) включает определение класса bad_alloc, общедоступно унаследованного от класса exception. Некоторые компиляторы предлагают только одну возможность, другие — возможность выбора подходящего метода, используя опции компилятора. В листинге 15.13 предпринята попытка реализации обеих возможностей. Если исключение перехватывается, программа отображает зависящее от реализации сообщение, возвращаемое унаследованным методом what(), и завершается. В противном случае она пытается определить, является ли возвращаемое значение null-указателем. (Здесь преследуется цель продемонстрировать технику выбора методов обработки ошибок распределения памяти, а не необходимость применения обоих методов.)

Листинг 15.13. newexcp.cpp

```
// newexcp.cpp -- исключение bad_alloc
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;
struct Big
{
   double stuff[20000];
};
int main()
{
   Big * pb;
   try {
      cout << "Попытка получить крупный блок памяти:\n";
      pb = new Big[10000]; // 1 600 000 000 байт
      cout << "Получено в результате последнего запроса new:\n";
   }
   catch (bad_alloc & ba)
   {
      cout << "Перехвачено исключение!\n";
      cout << ba.what() << endl;
      exit(EXIT_FAILURE);
   }
   if (pb != 0)
   {
      pb[0].stuff[0] = 4;
      cout << pb[0].stuff[0] << endl;
   }
   else
      cout << "pb — нулевой указатель\n";
   delete [] pb;
   return 0;
}
```

Ниже показан вывод скомпилированной программы из листинга 15.13, которая не генерирует исключение:

```
Попытка получить крупный блок памяти:
Получено в результате последнего запроса new:
pb — нулевой указатель
```

А это — вывод программы, когда генерируется исключение:

```
Попытка получить крупный блок памяти:
Перехвачено исключение!
bad allocation
```

В этом случае метод `what()` возвращает строку `"bad allocation"` (“ошибочное распределение”).

Если программа выполняется без ошибок распределения памяти, можно попробовать увеличить объем запрашиваемой памяти.

Исключения, классы и наследование

Исключения, классы и наследование взаимодействуют несколькими путями. Во-первых, можно породить один класс исключений от другого класса, как это сделано в стандартной библиотеке C++. Во-вторых, можно добавить исключения в классы путем встраивания объявления класса исключений в определение класса. В-третьих, такое вложенное объявление может быть унаследовано и оно само может служить базовым классом.

Код в листинге 15.14 начинается с исследования одной из таких возможностей. В этом заголовочном файле объявляется простой класс `Sales`, содержащий значение года, и массив из 12 ежемесячных объемов продаж. Класс `LabeledSales` порожден от класса `Sales` и дополнительно содержит метку данных.

Листинг 15.14. sales.h

```
// sales.h -- исключения и наследование
#include <stdexcept>
#include <cstring>
class Sales
{
public:
   enum {MONTHS = 12}; // может быть статической константой
   class bad_index : public std::logic_error
   {
   private:
      int bi; // недопустимое значение индекса
   public:
      explicit bad_index(int ix,
         const char * s = "Ошибка индекса в объекте Sales\n");
      int bi_val() const {return bi;}
   };
   explicit Sales(int yy = 0);
   Sales(int yy, const double * gr, int n);
   virtual ~Sales() { }
   int Year() const { return year; }
   virtual double operator[](int i) const throw(std::logic_error);
```

```
   virtual double & operator[](int i) throw(std::logic_error);
private:
   double gross[MONTHS];
   int year;
};
class LabeledSales : public Sales
{
public:
   static const int STRLEN = 50; // может быть числовым значением
   class nbad_index : public Sales::bad_index
   {
   private:
      char lbl[STRLEN];
   public:
      nbad_index(const char * lb, int ix,
         const char * s = "Ошибка индекса в объекте LabeledSales\n");
      const char * label_val() {return lbl;}
   };
   explicit LabeledSales(const char * lb = "none", int yy = 0);
   LabeledSales(const char * lb, int yy, const double * gr, int n);
   virtual ~LabeledSales() { }
   const char * Label() const {return label;}
   virtual double operator[](int i) const throw(std::logic_error);
   virtual double & operator[](int i) throw(std::logic_error);
private:
   char label[STRLEN];
};
```

Рассмотрим некоторые детали кода в листинге 15.14. Во-первых, символьная константа MONTHS расположена в защищенном разделе Sales; это делает ее значение доступным для производного класса, такого как LabeledSales.

Далее, класс bad_index находится в общедоступном разделе Sales; это делает его доступным в качестве типа для клиентского блока catch. Отметим, что внешнее окружение требует, чтобы тип был идентифицирован как Sales::bad_index. Этот класс является производным от стандартного класса logic_error. Класс bad_index имеет возможность сохранять и уведомлять о значениях, выходящих за пределы диапазона, для массива индексов.

Класс nbad_index расположен в общедоступном разделе LabeledSales и доступен в клиентском коде как LabeledSales::nbad_index. Он порожден от bad_index и добавляет возможность хранения и отображения метки объекта LabeledSales. Поскольку bad_index порожден от logic_error, nbad_index также порожден от logic_error.

Оба класса имеют перегруженный метод operator[](), предназначенный для доступа к хранимым в объекте отдельным элементам массива и генерации исключения, если индекс массива выходит за допустимые пределы. Обратите внимание на спецификацию исключения:

```
// версия Sales
virtual double operator[](int i) const throw(std::logic_error);
// версия LabeledSales
virtual double operator[](int i) const throw(std::logic_error);
```

Поскольку тип спецификации исключения соответствует производному классу, тип `std::logic_error` соответствует обоим типам `bad_index` и `nbad_index`. Следует помнить, что при переопределении методов базового класса метод производного класса имеет то же имя, а возвращаемый тип может изменяться в зависимости от того, прямо или непрямо наследуется возвращаемый тип метода базового класса. Те же правила действуют и для спецификации исключений. Производный метод будет иметь ту же спецификацию исключения, что и базовый метод, или же он должен иметь тип, унаследованный (прямо или непрямо) от типа, используемого в спецификации исключения базового метода. Таким образом, можно использовать `bad_index` для типа исключения в `Sales::operator[]()`, и `nbad_index` – для `LabeledSales`. Однако на практике не все компиляторы поддерживают упомянутые возможности.

В листинге 15.15 показана реализация методов, которые не были определены как встроенные в листинге 15.14. Отметим, что вложенные классы требуют использования операции разрешения контекста несколько раз. Отметим также, что функция `operator[]()` генерирует исключения в случае выхода индекса массива за допустимые пределы.

Листинг 15.15. sales.cpp

```
// sales.cpp -- реализация Sales
#include "sales.h"

Sales::bad_index::bad_index(int ix, const char * s )
   : std::logic_error(s), bi(ix)
{
}
Sales::Sales(int yy)
{
   year = yy;
   for (int i = 0; i < MONTHS; ++i)
      gross[i] = 0;
}
Sales::Sales(int yy, const double * gr, int n)
{
   year = yy;
   int lim = (n < MONTHS)? n : MONTHS;
   int i;
   for (i = 0; i < lim; ++i)
      gross[i] = gr[i];
   // for i > n and i < MONTHS
   for ( ; i < MONTHS; ++i)
      gross[i] = 0;
}
double Sales::operator[](int i) const throw(std::logic_error)
{
   if(i < 0 || i >= MONTHS)
      throw bad_index(i);
   return gross[i];
}
double & Sales::operator[](int i) throw(std::logic_error)
{
```

```
   if(i < 0 || i >= MONTHS)
      throw bad_index(i);
   return gross[i];
}
LabeledSales::nbad_index::nbad_index(const char * lb, int ix,
   const char * s ) : Sales::bad_index(ix, s)
{
   std::strcpy(lbl, lb);
}
LabeledSales::LabeledSales(const char * lb, int yy)
   : Sales(yy)
{
   std::strcpy(label, lb);
}
LabeledSales::LabeledSales(const char * lb, int yy, const double * gr, int n)
   : Sales(yy, gr, n)
{
   std::strcpy(label, lb);
}
double LabeledSales::operator[](int i) const throw(std::logic_error)
{
   if(i < 0 || i >= MONTHS)
      throw nbad_index(Label(), i);
   return Sales::operator[](i);
}
double & LabeledSales::operator[](int i) throw(std::logic_error)
{
   if(i < 0 || i >= MONTHS)
      throw nbad_index(Label(), i);
   return Sales::operator[](i);
}
```

Код в листинге 15.16 использует эти классы и при этом сначала пытается выйти за пределы массива объекта sales2 типа LabeledSales, а затем — за пределы массива объекта sales1 типа Sales. Поскольку эти попытки предпринимаются в двух разных блоках try, можно протестировать отдельно каждый тип исключений.

Листинг 15.16. use_sales.cpp

```
// use_sales.cpp -- вложенные исключения
#include <iostream>
#include "sales.h"

int main()
{
   using std::cout;
   using std::cin;
   using std::endl;
   double vals1[12] =
   {
      1220, 1100, 1122, 2212, 1232, 2334,
      2884, 2393, 3302, 2922, 3002, 3544
   };
```

```
double vals2[12] =
{
   12, 11, 22, 21, 32, 34,
   28, 29, 33, 29, 32, 35
};
Sales sales1(2004, vals1, 12);
LabeledSales sales2("Blogstar",2005, vals2, 12 );
cout << "Первый блок try:\n";
try
{
   int i;
   cout << "Год = " << sales1.Year() << endl;
   for (i = 0; i < 12; ++i)
   {
      cout << sales1[i] << ' ';
      if (i % 6 == 5)
         cout << endl;
   }
   cout << "Год = " << sales2.Year() << endl;
   cout << "Метка = " << sales2.Label() << endl;
   for (i = 0; i <= 12; ++i)
   {
      cout << sales2[i] << ' ';
      if (i % 6 == 5)
         cout << endl;
   }
   cout << "Конец первого блока try.\n";
}
catch(LabeledSales::nbad_index & bad)
{
   cout << bad.what();
   cout << "Компания: " << bad.label_val() << endl;
   cout << "Неверный индекс: " << bad.bi_val() << endl;
}
catch(Sales::bad_index & bad)
{
   cout << bad.what();
   cout << "Неверный индекс: " << bad.bi_val() << endl;
}
cout << "\nСледующий блок try:\n";
try
{
   sales2[2] = 37.5;
   sales1[20] = 23345;
   cout << "Конец второго блока try.\n";
}
catch(LabeledSales::nbad_index & bad)
{
   cout << bad.what();
   cout << "Компания: " << bad.label_val() << endl;
   cout << "Неверный индекс: " << bad.bi_val() << endl;
}
```

```
    catch(Sales::bad_index & bad)
    {
        cout << bad.what();
        cout << "Неверный индекс: " << bad.bi_val() << endl;
    }
    cout << "Готово.\n";
    return 0;
}
```

Ниже приведен вывод программ из листингов 15.14, 15.15 и 15.16:

```
Первый блок try:
Год = 2004
1220 1100 1122 2212 1232 2334
2884 2393 3302 2922 3002 3544
Год = 2005
Метка = Blogstar
12 11 22 21 32 34
28 29 33 29 32 35
Ошибка индекса в объекте LabeledSales
Компания: Blogstar
Неверный индекс: 12
Следующий блок try:
Ошибка индекса в объекте Sales
Неверный индекс: 20
done
```

Потеря исключений

После того как исключение сгенерировано, доступны два пути решения возникшей проблемы. Если исключение возникло в функции, имеющей спецификацию исключения, оно должно соответствовать одному из типов в списке спецификации. (Напомним, что в иерархии наследования тип класса соответствует объекту этого класса и типам, производным от этого класса.) Если исключение не соответствует спецификации, оно называется *непредвиденным исключением* и по умолчанию приводит к останову программы. Если исключение преодолевает этот первый барьер (или избегает его, поскольку функция требует спецификации исключения), дальше оно должно быть перехвачено. Если не преодолевает, что может произойти при отсутствии блока `try` или соответствующего блока `catch`, то такое исключение называется *неперехваченным исключением*. По умолчанию неперехваченное исключение также приводит к останову программы. Тем не менее, можно изменить реакцию программы на непредвиденное или неперехваченное исключение. Посмотрим, как это делается, и начнем с неперехваченного исключения.

Неперехваченное исключение не приводит к немедленному останову программы. Взамен программа сначала вызывает функцию `terminate()`. По умолчанию функция `terminate()` вызывает функцию `abort()`. Можно изменить поведение функции `terminate()`, *зарегистрировав* функцию, которую `terminate()` будет вызывать вместо `abort()`. Для того чтобы это сделать, необходимо вызвать функцию `set_terminate()`. Обе функции, `terminate()` и `set_terminate()`, объявлены в заголовочном файле `exception`:

```
typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler f) throw();
void terminate();
```

Здесь typedef делает terminate_handler именем типа указателя на функцию, не принимающую аргументов и не возвращающую значение. Функция set_terminate() принимает в качестве аргумента имя функции (то есть ее адрес), которая не имеет аргументов и имеет тип возврата void. Функция set_terminate() возвращает адрес ранее зарегистрированной функции. Если обратиться к функции set_terminate() более одного раза, terminate() вызовет функцию, установленную самым последним вызовом set_terminate().

Рассмотрим пример. Предположим, необходимо иметь неперехватываемое исключение, которое приводит к печати сообщения об этом событии, затем вызывает функцию exit() и устанавливает значение состояние завершения в 5. Сначала потребуется включить в проект заголовочный файл exception. Объявления этого файла можно сделать доступными с помощью директивы using, подходящих объявлений using или с использованием квалификатора std::.

```
#include <exception>
using namespace std;
```

Далее нужно создать функцию, которая выполняет два требуемых действия и имеет подходящий прототип:

```
void myQuit()
{
   cout << "Завершение по причине неперехватываемого исключения\n";
   exit(5);
}
```

Наконец, в начале программы необходимо обозначить эту функцию как выбранную вами процедуру завершения программы:

```
set_terminate(myQuit);
```

Теперь, если исключение будет сгенерировано и не перехвачено, программа вызовет terminate(), a terminate() вызовет MyQuit().

Теперь рассмотрим непредвиденные исключения. Спецификация исключений в функции дает возможность пользователям функции узнать, какие исключения перехватывать. Предположим, что имеется следующий прототип:

```
double Argh(double, double) throw(out_of_bounds);
```

Тогда можно использовать функцию следующим образом:

```
try {
   x = Argh(a, b);
}
catch(out_of_bounds & ex)
{
   ...
}
```

Хорошо бы знать, какие исключения перехватывать; вспомним, что неперехваченное исключение по умолчанию приводит к останову программы.

Однако есть темы, о которых можно поговорить. В принципе, спецификация исключений должна включать исключения, генерируемые функциями, которые вызываются функцией в запросе.

Например, если `Argh()` вызывает функцию `Duh()`, которая генерирует объект исключения `retort`, то `retort` должен содержаться в спецификации исключений обеих функций `Argh()` и `Duh()`. До тех пор пока вы внимательно не реализуете все функции, не будет гарантии, что все будет работать корректно. Например, можно использовать устаревшие коммерческие библиотеки, не содержащие спецификаций исключений. В этом случае нужно очень внимательно смотреть, что получится, если функция сгенерирует исключение, которого нет в спецификации.

Ситуация очень похожа на неперехваченное исключение. Если возникает непредвиденное исключение, программа вызывает функцию `unexpected()`. (Вы не ожидали функции `unexpected()`? Никто не ожидает функции `unexpected()`!) Эта функция снова вызывает `terminate()`, которая по умолчанию вызывает `abort()`. Так же как существует функция `set_terminate()`, изменяющая поведение `terminate()`, доступна функция `set_unexpected()`, изменяющая поведение `unexpected()`. Эти новые функции также объявлены в заголовочном файле `exception`.

```
typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler f) throw();
void unexpected();
```

Однако, поведение функции, которую можно установить для `set_unexpected()`, более управляемо. В частности, функция `unexpected_handler` предоставляет следующие варианты:

- Она может завершить программу, вызвав `terminate()` (поведение по умолчанию), `abort()` или `exit()`.
- Она может сгенерировать исключение.

Результат генерации исключения зависит от исключения, генерируемого замененной функцией `unexpected_handler`, и исходной спецификации исключений для функции, которая сгенерировала непредвиденный тип исключения:

- Если вновь сгенерированное исключение соответствует спецификации исключений, то программа выполняется нормально; то есть она ищет блок `catch`, который соответствует возникшему исключению. По сути, при таком подходе непредвиденный тип исключения меняется на ожидаемый тип.
- Если вновь сгенерированное исключение не соответствует спецификации исключений и если спецификация исключений не содержит типа `std::bad_exception`, программа вызывает `terminate()`. Тип `bad_exception` является производным от типа `exception`, а его объявление содержится в заголовочном файле `exception`.
- Если вновь сгенерированное исключение не соответствует спецификации исключений и если спецификация исключений содержит тип `std::bad_exception`, непредвиденное исключение меняется на тип `std::bad_exception`.

Короче говоря, если необходимо перехватывать все исключения, ожидаемые и непредвиденные, можно поступить следующим образом. Во-первых, нужно сделать доступным заголовочный файл исключений:

```
#include <exception>
using namespace std;
```

Дальше следует создать функцию замены, которая преобразует непредвиденные исключения в тип bad_exception и имеет соответствующий прототип:

```
void myUnexpected()
{
   throw std::bad_exception(); // или просто throw;
}
```

Применение throw без указания исключения приведет к повторной генерации первоначального исключения. При этом исключение будет заменено объектом bad_exception, если спецификация исключений содержит этот тип.

Далее, в начале программы нужно определить следующую функцию в качестве реакции на непредвиденное исключение:

```
set_unexpected(myUnexpeced);
```

И, наконец, нужно включить тип bad_exception в спецификацию исключений и цепочку блоков catch:

```
double Argh(double, double) throw(out_of_bounds, bad_exception);
...
try {
   x = Argh(a, b);
}
catch(out_of_bounds & ex)
{
   ...
}
catch(bad_exception & ex)
{
   ...
}
```

Предостережения при использовании исключений

Из предыдущего анализа исключений можно сделать вывод (и совершенно справедливо), что управление исключениями должно быть встроено в программу, а не присоединено к ней как-то сбоку. Создание таких программ связано с рядом трудностей. Например, исключения увеличивают размер программы и уменьшают скорость ее выполнения. Спецификации исключений плохо работают с шаблонами, поскольку шаблонные функции могут генерировать разные исключения в зависимости от конкретной специализации. Исключения и динамическое распределение памяти также не всегда работают хорошо.

Рассмотрим совместную работу динамического распределения памяти и исключений. Во-первых, рассмотрим следующую функцию:

```
void test1(int n)
{
   string mesg("Случился бесконечный цикл");
   ...
```

```
   if (oh_no)
      throw exception();
   ...
   return;
}
```

Класс string использует динамическое распределение памяти. Обычно деструктор string вызывается, когда функция достигает оператора return и завершается. Из-за раскручивания стека оператор throw, даже если он завершает функцию преждевременно, позволяет осуществиться вызову деструктора. То есть в этом случае управление памятью реализуется должным образом.

Теперь рассмотрим следующую функцию:

```
void test2(int n)
{
   double * ar = new double[n];
   ...
   if (oh_no)
      throw exception();
   ...
   delete [] ar;
   return;
}
```

Здесь присутствует проблема. Раскручивание стека удаляет переменную ar из стека. Однако преждевременное завершение функции означает, что оператор delete [] не будет выполнен. Указатель исчезнет, но блок памяти, на который он ссылался, останется нетронутым и недоступным. Короче говоря, эта память будет утеряна.

Потери памяти можно избежать. Например, можно перехватить исключение в той же функции, которая его сгенерировала, добавить очищающий память код в блок catch и сгенерировать исключение заново:

```
void test3(int n)
{
   double * ar = new double[n];
   ...
   try {
      if (oh_no)
         throw exception();
   }
   catch(exception & ex)
   {
      delete [] ar;
      throw;
   }
   ...
   delete [] ar;
   return;
}
```

Очевидно, что такой подход при недостаточной внимательности программиста может породить новые ошибки. Другой способ предусматривает использование шаблона auto_ptr, который рассматривается в главе 16.

Короче говоря, несмотря на исключительную важность управления исключениями для некоторых проектов, все это будет стоить вам усилий по программированию, увеличения размеров программы и времени ее выполнения. Поддержка исключений компиляторами и опыт пользователей еще не достигли должного уровня, поэтому технику исключений следует использовать осмотрительно.

Пример из практики: управление исключениями

В современных библиотеках управление исключениями может появиться для достижения нового уровня сложности — в основном, за счет недокументированных или слабо документированных функций обработчиков исключений. Каждый, кто знаком с современными операционными системами, наверняка сталкивался с ошибками и проблемами, вызываемыми необработанными исключениями. После возникновения таких ошибок программисты часто проделывают тяжелую работу, изучая внутренности библиотек: какое исключение сгенерировано, когда и где оно возникло и как его обработать.

Программисты-новички быстро понимают, что изучение управления исключениями в библиотеках не менее сложно, нежели изучение самого языка; современные библиотеки могут содержать программы и парадигмы, столь же чужеродные и сложные, как и любые нюансы синтаксиса C++. Нахождение выхода из создавшейся ситуации, понимание сложности библиотек и классов для хорошего программирования столь же важно, как и изучение самого языка C++. Знания обработки ошибок и исключений, которые вы извлекли из документации по библиотеке, и исходные коды всегда принесут вам и вашим программам большую пользу.

RTTI

RTTI (Run Time Type Identification) — это аббревиатура для обозначения механизма динамической идентификация типов. Это одно из последних расширений языка C++ и оно не поддерживается многими устаревшими реализациями языка. Некоторые реализации могут иметь установки компилятора, позволяющие включать и отключать RTTI. Главная цель введения RTTI — предоставить программам стандартный механизм для определения типов объектов во время выполнения программы. Во многих библиотеках классов такой механизм, применительно к их собственным классам, существует. Однако без встроенной в C++ поддержки механизмы RTTI от разных поставщиков библиотек не совместимы. Создание стандарта языка для RTTI позволит будущим библиотекам быть совместимыми друг с другом.

Для чего нужен RTTI

Предположим, что существует иерархия классов, производных от общего базового класса. Можно установить указатель базового класса на объект любого класса в этой иерархии. Далее можно вызвать функцию, которая, обработав определенное количество информации, выбирает один из существующих классов, создает объект этого типа и возвращает его адрес, который получает значение указателя базового класса. Как теперь определить, на какой тип объекта он указывает?

Перед тем как ответить на этот вопрос, потребуется понять, зачем вообще знать тип. Возможно, требуется просто вызвать корректную версию метода класса? В действительности, в этом случае не нужно знать тип объекта, так как эта функция является виртуальной и известна всем членам иерархии класса. Но может оказаться, что производный объект имеет собственный, а не унаследованный метод. В этом случае

только некоторые объекты смогут использовать этот метод. Или, для целей отладки, необходимо проследить, какого типа был сгенерирован объект. Для двух последних случаев ответ дает RTTI.

Как работает RTTI?

В C++ есть три компонента, поддерживающие RTTI:

- Операция dynamic_cast, если возможно, создает указатель на производный класс из указателя на базовый класс. В противном случае оператор возвращает 0, то есть null-указатель.
- Операция typeid возвращает величину, идентифицирующую точный тип объекта.
- Структура type_info содержит информацию о конкретном типе.
- Можно использовать RTTI с иерархией классов, имеющих виртуальные функции. Причина в том, что это — единственная иерархия классов, для которой можно назначать адреса производных объектов указателям базового класса.

Внимание!

RTTI работает только для классов, имеющих виртуальные функции.

Давайте исследуем три компонента RTTI.

Операция dynamic_cast

Операция dynamic_cast — наиболее трудный в использовании компонент RTTI. Она не отвечает на вопрос, на какой тип объекта указывает указатель. Вместо этого она дает ответ на вопрос, можно ли назначить адрес объекта указателю определенного типа. Посмотрим, что это значит. Предположим, что существует следующая иерархия:

```
class Grand { // имеет виртуальные методы };
class Superb : public Grand { ... };
class Magnificent : public Superb { ... };
```

Далее, предположим, что есть следующие указатели:

```
Grand * pg = new Grand;
Grand * ps = new Superb;
Grand * pm = new Magnificent;
```

Кроме того, имеются следующие приведения типов:

```
Magnificent * p1 = (Magnificent *) pm;   // #1
Magnificent * p2 = (Magnificent *) pg;   // #2
Superb * p3 = (Magnificent *) pm;        // #3
```

Какие из этих приведений безопасны? В зависимости от объявления класса, все они могут быть безопасными. Однако единственным гарантированно безопасным приведением является такое, в котором указатель имеет тот же тип, что и сам объект, либо тот же тип, что и тип прямого или непрямого базового типа. Например, приведение типов #1 надежно, потому что оно устанавливает указатель типа Magnificent на указатель объекта типа Magnificent. Приведение типов #2 не надежно, поскольку

оно устанавливает адрес объекта базового класса (Grand) на указатель производного класса (Magnificent). Таким образом, программа будет ожидать, что объект базового класса содержит указатели производного класса, что в общем случае неверно. Объект Magnificent, например, может содержать члены-данные, которые нужны объекту Grand. Приведение типов #3, тем не менее, надежно, поскольку в нем адрес производного объекта назначается указателю базового класса. То есть, благодаря общедоступному наследованию, объект Magnificent также является объектом Superb (прямой базовый класс) и объектом Grand (непрямой базовый класс). Поэтому допустимо назначить их адреса указателям всех трех типов. Виртуальные функции гарантируют, что использование указателей всех трех типов с объектом Magnificent приведет к вызову методов Magnificent.

Следует отметить, что вопрос, является ли преобразование типов безопасным, более важен, нежели вопрос, на какой тип объекта указывает указатель. Обычно знать тип объекта нужно тогда, когда требуется знать, безопасным ли будет вызов конкретного метода. Для вызова метода не обязательно иметь точное совпадение типов. Типом может быть базовый тип, для которого определен виртуальный метод. Следующий пример демонстрирует это.

Но сначала рассмотрим синтаксис dynamic_cast. Операция используется следующим образом, где pg указывает на объект:

```
Superb * pm = dynamic_cast<Superb *>(pg);
```

Спрашивается, может ли тип указателя pg быть безопасно приведен (как рассматривалось выше) к типу Superb *. Если да, операция возвращает адрес объекта. В противном случае она возвращает 0, то есть null-указатель.

На память!

В общем случае, выражение dynamic_cast<Type >(pt) преобразует указатель pt в указатель типа Type*, если указываемый объект имеет тип Type или унаследован прямо или непрямо от типа Type. В противном случае выражение возвращает 0, то есть null-указатель.

В листинге 15.17 иллюстрируется описанный процесс. Сначала создаются три класса — Grand, Superb и Magnificent. В классе Grand определяется виртуальная функция Speak(), которую все остальные классы переопределяют. В классе Superb() определяется виртуальная функция Say(), переопределяемая в классе Magnificent (рис. 15.4). В программе определяется функция GetOne(), которая случайным образом создает и инициализирует объект одного из трех классов и возвращает адрес как указатель типа Grand *. (Функция GetOne() имитирует интерактивное принятие решения пользователем.) В цикле этот указатель назначается переменной pg типа Garnd *, и затем pg используется для вызова функции Speak(). Поскольку эта функция виртуальная, код корректно вызывает версию Speak(), соответствующую указываемому объекту:

```
for (int i = 0; i < 5; i++)
{
   pg = GetOne();
   pg->Speak();
   ...
}
```

Листинг 15.17. `rtti1.cpp`

```
// rtti1.cpp -- использование RTTI-операции dynamic_cast
#include <iostream>
#include <cstdlib>
#include <ctime>
using std::cout;
class Grand
{
private:
   int hold;
public:
   Grand(int h = 0) : hold(h) {}
   virtual void Speak() const { cout << "Я — великий класс!\n";}
   virtual int Value() const { return hold; }
};
class Superb : public Grand
{
public:
   Superb(int h = 0) : Grand(h) {}
   void Speak() const {cout << "Я — изумительный класс!!\n"; }
   virtual void Say() const
      { cout << "Я храню изумительное значение " << Value() << "!\n";}
};
class Magnificent : public Superb
{
private:
   char ch;
public:
   Magnificent(int h = 0, char c = 'A') : Superb(h), ch(c) {}
   void Speak() const {cout << "Я — превосходный класс!!!\n";}
   void Say() const {cout << "Я храню символ " << ch <<
      " и целое " << Value() << "!\n"; }
};
Grand * GetOne();
int main()
{
   std::srand(std::time(0));
   Grand * pg;
   Superb * ps;
   for (int i = 0; i < 5; i++)
   {
      pg = GetOne();
      pg->Speak();
      if( ps = dynamic_cast<Superb *>(pg))
         ps->Say();
   }
   return 0;
}
Grand * GetOne() // случайным образом генерирует три типа объектов
{
   Grand * p;
   switch( std::rand() % 3)
```

```
    {
       case 0: p = new Grand(std::rand() % 100);
            break;
       case 1: p = new Superb(std::rand() % 100);
            break;
       case 2: p = new Magnificent(std::rand() % 100, 'A' + std::rand() % 26);
            break;
    }
    return p;
}
```

Тем не менее, нельзя использовать этот явный подход для вызова функции Say(); она не определена для класса Grand(). С помощью операции dynamic_cast можно определить, может ли тип pg быть приведен к указателю на Superb. Это было бы возможно, если бы объект имел тип Superb или Magnificent. В таком случае можно безопасно вызвать функцию Say():

```
if (ps = dynamic_cast<Superb *>(pg))
ps->Say();
```

Напомним, что значением в выражении присваивания является величина, расположенная слева. Таким образом, значением выражения if является ps. Если приведение типа выполнится успешно, ps будет не равно нулю, или true. Если приведение типа неудачно, что произойдет, если pg указывает на объект Grand, ps будет равно нулю, или false. В листинге 15.17 показан полный код. (Кстати, некоторые компиляторы, зная что программисты часто используют операцию == в условии if, могут выдать предупреждение о некорректном присваивании.)

Замечание по совместимости

Даже если ваш компилятор поддерживает RTTI, эта возможность по умолчанию может быть отключена. Если это так, то программа может нормально компилироваться, но приводить к ошибкам времени выполнения. Если вы считаете, что проблема в этом, проконсультируйтесь с документацией или исследуйте опции меню. В среде Microsoft Visual C++ 7.1 выберите команду меню Project (Проект), затем свойства проекта *proj* (где *proj* — имя вашего проекта), перейдите на вкладку C/C++, щелкните на Language (Язык) и измените установку Enable Run-Time Type Info (Включить динамическое получение информации о типах) на Yes (Да).

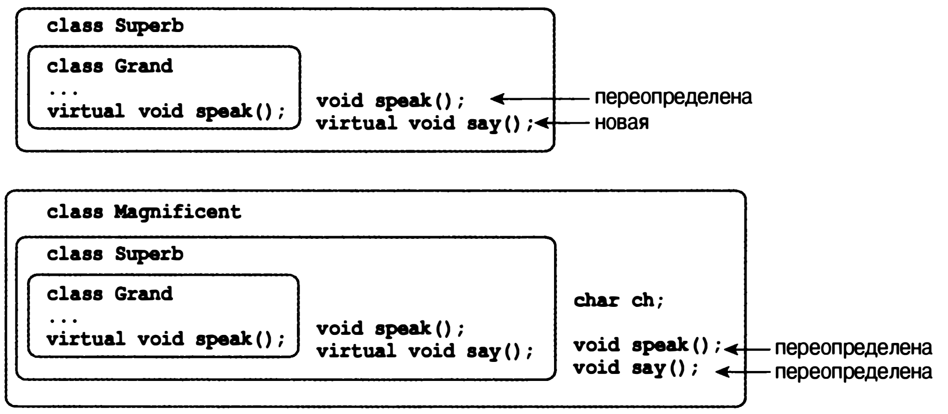


Рис. 15.4. Семейство классов Grand

Программа в листинге 15.17 иллюстрирует важную мысль. Нужно всегда использовать виртуальные функции, когда это возможно, а RTTI — только когда это необходимо. Ниже приведен вывод программы примера:

```
Я — изумительный класс!!
Я храню изумительное значение 68!
Я — превосходный класс!!!
Я храню символ R и целое 68!
Я — превосходный класс!!!
Я храню символ D и целое 12!
Я — превосходный класс!!!
Я храню символ V и целое 59!
Я — великий класс!
```

Как видите, методы Say() вызываются только для классов Superb и Magnificent. (Вывод изменяется от запуска к запуску, поскольку для выбора объектов программа использует функцию rand().)

Можно также применять dynamic_cast к ссылкам. Использование слегка отличается; не существует значения ссылки, соответствующей типу null-указателя, поэтому не существует специального значения, которое обозначает крах программы. Вместо этого, принуждаемая некорректным запросом, операция dynamic_cast генерирует исключение bad_cast, производное от класса exception и определенное в заголовочном файле typeinfo. Таким образом, эту операцию можно использовать везде, где rg является ссылкой на объект Grand:

```
#include <typeinfo> // для bad_cast
...
try {
   Superb & rs = dynamic_cast<Superb &>(rg);
   ...
}
catch(bad_cast &){
   ...
};
```

Операция typeid и класс type_info

Операция typeid позволяет определить, имеют ли объекты один и тот же тип. Похожая на sizeof, она принимает два аргумента:

- Имя класса.
- Выражение, которое вычисляется для объекта.

Операция typeid возвращает ссылку на объект type_info, где type_info — класс, определенный в заголовочном файле typeinfo (обычно typeinfo.h). Операция type_info перегружает операции == и !=, так что эти операции можно использовать для сравнения типов. Например, выражение

```
typeid(Magnificent) == typeid(*pg)
```

возвращает bool-значение true, если pg ссылается на объект Magnificent, и false — в противном случае. Если pg окажется null-указателем, программа сгенерирует исключение bad_typeid. Этот тип исключения является производным от exception и

определен в заголовочном файле typeinfo. Реализация класса type_info варьируется от одного поставщика компилятора к другому, тем не менее, она включает член name(), который возвращает зависящую от реализации строку, являющуюся обычно именем класса. Например, оператор

```
cout << "Обработка типа " << typeid(*pg).name() << ".\n";
```

отображает строку для класса объекта, на который ссылается указатель pg.

Листинг 15.18 представляет собой модифицированную версию листинга 15.17, в нем используются операция typeid и функция-член name(). Отметим, что они применяются в ситуациях, которые не могут быть разрешены с помощью dynamic_cast и виртуальных функций. Проверка typeid используется для выбора действия, которое даже не является методом класса, поэтому ее нельзя вызвать с помощью указателя класса. Метод name() демонстрирует, как можно использовать метод для выполнения отладки. Также отметим, что в программу включен заголовочный файл typeinfo.

Листинг 15.18. rtti2.cpp

```
// rtti2.cpp -- использование dynamic_cast, typeid и type_info
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <typeinfo>
using namespace std;
class Grand
{
private:
   int hold;
public:
   Grand(int h = 0) : hold(h) {}
   virtual void Speak() const { cout << "Я — великий класс!\n";}
   virtual int Value() const { return hold; }
};
class Superb : public Grand
{
public:
   Superb(int h = 0) : Grand(h) {}
   void Speak() const {cout << "Я — изумительный класс!!\n"; }
   virtual void Say() const
      { cout << "Я храню изумительное значение " << Value() << "!\n";}
};
class Magnificent : public Superb
{
private:
   char ch;
public:
   Magnificent(int h = 0, char cv = 'A') : Superb(h), ch(cv) {}
   void Speak() const {cout << "Я — превосходный класс!!!\n";}
   void Say() const {cout << "Я храню символ " << ch <<
      " и целое " << Value() << "!\n"; }
};

Grand * GetOne();
```

```
int main()
{
   srand(time(0));
   Grand * pg;
   Superb * ps;
   for (int i = 0; i < 5; i++)
   {
      pg = GetOne();
      cout << "Обработка типа " << typeid(*pg).name() << ".\n";
      pg->Speak();
      if( ps = dynamic_cast<Superb *>(pg))
         ps->Say();
      if (typeid(Magnificent) == typeid(*pg))
         cout << "Да, вы действительно превосходны.\n";
   }
   return 0;
}
Grand * GetOne()
{
   Grand * p;
   switch( rand() % 3)
   {
      case 0: p = new Grand(rand() % 100);
         break;
      case 1: p = new Superb(rand() % 100);
         break;
      case 2: p = new Magnificent(rand() % 100, 'A' + rand() % 26);
         break;
   }
   return p;
}
```

Вот как выглядит вывод программы из листинга 15.18:

```
Обработка типа Magnificent.
Я — превосходный класс!!!
Я храню символ P и целое 52!
Да, вы действительно превосходны.
Обработка типа Superb.
Я — изумительный класс!!
Я храню изумительное значение 37!
Обработка типа Grand.
Я — великий класс!
Обработка типа Superb.
Я — изумительный класс!!
Я храню изумительное значение 18!
Обработка типа Grand.
Я — великий класс!
```

Как и в предыдущем примере, вывод программы изменяется от запуска к запуску, поскольку в программе для выбора типов используется функция `rand()`.

Неправильное использование RTTI

Внутри сообщества C++ у RTTI много критиков. Они считают RTTI бесполезным, к тому же потенциальным источником неэффективности программ и плохого стиля программирования.

Не углубляясь в дебаты вокруг RTTI, рассмотрим некоторые приемы программирования, которых следует избегать.

Взглянет на основную часть листинга 15.17:

```
Grand * pg;
Superb * ps;
for (int i = 0; i < 5; i++)
{
   pg = GetOne();
   pg->Speak();
   if( ps = dynamic_cast<Superb *>(pg))
      ps->Say();
}
```

Используя `typeid` и игнорируя `dynamic_cast` и виртуальные функции, можно переписать код следующим образом:

```
Grand * pg;
Superb * ps;
Magnificent * pm;
for (int i = 0; i < 5; i++)
{
   pg = GetOne();
   if (typeid(Magnificent) == typeid(*pg))
   {
      pm = (Magnificent *) pg;
      pm->Speak();
      pm->Say();
   }
   else if (typeid(Superb) == typeid(*pg))
   {
      ps = (Superb *) pg;
      ps->Speak();
      ps->Say();
   }
   else
      pg->Speak();
}
```

Мало того, что этот код длиннее исходного, он имеет серьезный недостаток, заключающийся в явном именовании каждого класса. Предположим, что требуется породить класс `Insufferable` от класса `Magnificent`. Новый класс переопределит `Speak()` и `Say()`. В версии с использованием `typeid` для явной проверки каждого типа придется модифицировать цикл `for`, добавив новый раздел `else if`. Первоначальная версия не требует изменений вообще. Оператор

```
pg->Speak();
```

работает для всех классов, производных от `Grand`, а оператор

```
if( ps = dynamic_cast<Superb *>(pg))
   ps->Say();
```

применим для всех классов, производных от Superb.

Совет

Если вам приходится использовать typeid в длинной цепочке операторов if else, проверьте, не лучше ли применять виртуальные функции и операцию dynamic_cast.

Операции приведения типов

По мнению Бьерна Страуструпа, операция приведения типа в языке C слишком нестрогая. Например, рассмотрим следующий фрагмент кода:

```
struct Data
{
   double data[200];
};
struct Junk
{
   int junk[100];
};
Data d = {2.5e33, 3.5e-19, 20.2e32};
char * pch = (char *) (&d);// приведение типа #1 — преобразование в строку
char ch = char (&d);       // приведение типа #2 — преобразование в символ
Junk * pj = (Junk *) (&d);// приведение типа #3 — преобразование
                          // в указатель на Junk
```

Во-первых, какие из этих приведений типов имеют смысл? Пока вы не захотите невероятного, ни одно из этих приведений не имеет смысла. Во-вторых, какие из этих приведений допустимы? В языке C допустимы все эти приведения. Реакцией Страуструпа на такую вольность было четкое определение того, что является допустимым для общих приведений типов, и введение четырех операций приведения типов, ужесточающих дисциплину приведения:

```
dynamic_cast
const_cast
static_cast
reinterpret_cast
```

Вместо общего приведения типа можно использовать операцию, которая более подходит для конкретной цели. Это фиксирует цель приведения и дает возможность компилятору проверить, что вы делаете то, что хотите сделать.

Вы уже встречали операцию dynamic_cast. Подытоживая, предположим, что High и Low — два класса, что ph имеет тип High *, а pl — тип Low *. Оператор

```
pl = dynamic_cast<Low *> ph;
```

присваивает указатель Low * переменной pl, только если Low является доступным базовым классом (прямым или непрямым) для High. В противном случае оператор назначает pl нулевой указатель. В общем случае операция dynamic_cast имеет следующий синтаксис:

dynamic_cast < *имя-типа* > (*выражение*)

Назначение этой операции — разрешить приведения внутри иерархии классов (такие приведения типов будут безопасными по причине наличия отношения *is-a*) и запретить другие приведения.

Операция `const_cast` предназначена исключительно для приведения типа, если значение имеет тип `const` или `volatile`. Она имеет такой же синтаксис, как у `dynamic_cast`:

```
const_cast < имя-типа > (выражение)
```

Результатом такого приведения типа будет ошибка, если любые другие аспекты типов не совпадают. То есть *имя-типа* и *выражение* должны быть одного типа, за исключением того, что они могут отличаться только наличием или отсутствием `const` или `volatile`. Предположим, что `High` и `Low` — следующие два класса:

```
High bar;
const High * pbar = &bar;
...
High * pb = const_cast<High *> (pbar); // верно
const Low * pl = const_cast<const Low *> (pbar); // неверно
```

Первое приведение делает `*pb` указателем, который можно использовать для изменения значения объекта `bar`; оно удаляет метку `const`. Второе приведение неверно, поскольку оно пытается изменить тип с `const High *` на `const Low *`.

Эта операция предназначена для того случая, когда нужно иметь величину, которая большую часть времени постоянна, но иногда может изменяться. В этом случае можно объявить величину как `const` и использовать `const_cast`, кода требуется изменить ее значение. Это же можно проделать, используя общее приведение типа, но при этом одновременно изменится тип величины:

```
High bar;
const High * pbar = &bar;
...
High * pb = (High *) (pbar); // верно
Low * pl = (Low *) (pbar);   // тоже верно
```

Поскольку одновременное изменение типа и изменение константы может оказаться непреднамеренной программной ошибкой, безопаснее воспользоваться операцией `const_cast`.

Операция `const_cast` не во всем хороша. Она может изменить указатель на величину, но эффект от изменения значения, заданного как `const`, не определен. Проясним ситуацию с этой операwbtq, рассмотрев короткий пример, приведенный в листинге 15.19.

Листинг 15.19. constcast.cpp

```
// constcast.cpp -- использование const_cast<>
#include <iostream>
using std::cout;
using std::endl;
void change(const int * pt, int n);
int main()
{
   int pop1 = 38383;
```

```
    const int pop2 = 2000;
    cout << "pop1, pop2: " << pop1 << ", " << pop2 << endl;
    change(&pop1, -1);
    change(&pop2, -1);
    cout << "pop1, pop2: " << pop1 << ", " << pop2 << endl;
    return 0;
}
void change(const int * pt, int n)
{
    int * pc;
    if (n < 0)
    {
        pc = const_cast<int *>(pt);
        *pc = 100;
    }
}
```

Операция const_cast может удалить const из const int * pt, что позволяет компилятору в функции change() воспринять следующий оператор:

```
*pc = 100;
```

Однако, поскольку pop2 объявлен как const, компилятор может защитить его от изменения, как показано в следующем выводе программы:

```
pop1, pop2: 38383, 2000
pop1, pop2: 100, 2000
```

Легко заметить, что вызов change() изменяет pop1, но не pop2. (Компилятор в данном случае создает временную копию pop2 и присваивает ее адрес pc, но, как уже говорилось, в стандарте C++ данная ситуация называется неопределенной.)

Операция static_cast имеет такой же синтаксис, как и другие операции:

static_cast < *имя-типа* > (*выражение*)

Она допустима только в том случае, если *имя-типа* может быть неявно преобразовано в тип, который имеет *выражение*, или наоборот. В любом другом случае приведение типа вызывает ошибку. Предположим, что High является базовым классом для Low, а Pond — несвязанный класс. Тогда приведение High к Low и обратно допустимо, а приведение Low к Pond — нет:

```
High bar;
Low blow;
...
High * pb = static_cast<High *> (&blow); // верное восходящее
                                         // преобразование вверх
Low * pl = static_cast<Low *> (&bar); // верное нисходящее преобразование
Pond * pmer = static_cast<Pond *> (&blow); // неверно, Pond не связан
```

Первое преобразование верно, потому что восходящее преобразование может быть выполнено явно. Второе преобразование, из указателя базового класса в указатель производного класса, не может быть выполнено без явного приведения типа. Но поскольку преобразование в обратном направлении может быть выполнено без приведения типа, можно использовать static_cast для нисходящего преобразования.

Аналогично, поскольку числовое значение может быть преобразовано в интегральный тип без приведения, интегральный тип может быть преобразован в числовой с помощью static_cast. Также можно использовать static_cast для приведения double к int, float к long и выполнения многих других числовых преобразований.

Операция reinterpret_cast предназначена для рискованных приведений. Она не позволит преобразовать const, но может сделать другие нехорошие вещи. Иногда программистам приходится делать зависящие от реализации нехорошие штуки, и применение reinterpret_cast позволяет им отслеживать этот процесс. Эта операция имеет такой же синтаксис, что и остальные три операции:

```
reinterpret_cast < имя-типа > (выражение)
```

Ниже приведен пример ее использования:

```
struct dat {short a; short b};
long value = 0xA224B118;
dat * pd = reinterpret_cast< dat *) (&value);
cout << pd->a; // отображает два первых байта величины
```

Обычно такое приведение типа применяется для низкоуровневого, зависящего от реализации программирования, которое является непереносимым. Например, код рассмотренного примера создает разный вывод на IBM-совместимых компьютерах и компьютерах Macintosh, поскольку эти системы хранят байты в многобайтных целых типах в разном порядке. Тем не менее, операция reinterpret_cast не позволяет делать все что угодно. Например, можно привести тип указателя к целому типу, который достаточно велик, чтобы хранить указатель, но нельзя привести указатель к меньшему целому типу или к типу с плавающей точкой. Еще одно ограничение состоит в том, что нельзя привести указатель на функцию к указателю на данные и наоборот.

Явное приведение типов в C++ также ограничено. В основном, оно может делать то же, что и другие приведения, плюс некоторые комбинации вроде static_cast или reinterpret_cast, за которыми следует const_cast. Но это, пожалуй, и все. Так, приведение типа

```
char ch = char (&d); // приведение #2 — преобразование адреса в символ
```

допустимо в языке C, но обычно недопустимо в C++, потому что в большинстве реализаций C++ тип char слишком мал, чтобы содержать указатель. Такие ограничения имеют смысл, но если вы считаете подобные меры предосторожности излишними, вам нужно работать на языке C.

Резюме

Друзья позволяют разрабатывать для классов более гибкий интерфейс. Класс может иметь другие функции, другие классы и функции-члены других классов в качестве друзей. В некоторых случаях, для того чтобы друзья корректно взаимодействовали, нужно использовать объявление forward и сосредоточивать внимание на порядке расположения объявлений классов и методов.

Вложенными классами называются классы, которые объявлены внутри других классов. Вложенные классы упрощают создание вспомогательных классов, которые обслуживают другие классы, но не являются частью общедоступного интерфейса.

Механизм исключений в C++ предоставляет гибкое средство для работы с нежелательными программными событиями, такими как недопустимые значения, неудачные попытки ввода-вывода и тому подобное. Генерация исключения прерывает выполнение текущей функции и передает управление в соответствующий блок catch. Блок catch следует сразу за блоком try. Для того чтобы исключение было перехвачено, функция, которая прямо или косвенно приводит к возникновению этого исключения, должна быть расположена внутри блока try.

Затем программа выполняет блок catch. Этот код может попытаться решить возникшую проблему или завершить программу. Класс может быть создан с вложенными классами исключений, которые генерируется при возникновении соответствующей проблемы. Функция может содержать спецификацию исключений, определяющую исключения, которые могут быть сгенерированы этой функцией. Неперехваченные исключения (не имеющие соответствующего блока catch) по умолчанию прерывают выполнение программы. Это же делают непредвиденные исключения (не соответствующие спецификации исключений).

Средство RTTI позволяет программам определять типы объектов. Операция dynamic_cast используется для приведения указателя на производный класс к указателю на базовый класс. Его главная цель — убедиться, что все нормально при вызове виртуальной функции. Операция typeid возвращает объект type_info. Для определения, имеет ли объект определенный тип, можно сравнить два значения, возвращенные typeid. Возвращенный объект type_info можно использовать для получения информации о самом объекте. Операции dynamic_cast, static_cast, const_cast и reinterpret_cast предоставляют более надежный и лучше документированный механизм приведения типов, нежели общий механизм приведения.

Вопросы для самоконтроля

1. Что неверно в следующей попытке создания друзей:

а.
```
class snap {
   friend clasp;
   ...
};
class clasp { ... };
```
б.
```
class cuff {
public:
   void snip(muff &) { ... }
   ...
};
class muff {
   friend void cuff::snip(muff &);
   ...
};
```
в.
```
class muff {
   friend void cuff::snip(muff &);
   ...
};
```

```
class cuff {
public:
   void snip(muff &) { ... }
   ...
};
```

2. Вы видели, как создаются взаимные друзья класса. Можете ли вы создать более ограниченную форму дружественной связи, при которой только некоторые члены класса B являются друзьями для класса A и некоторые члены класса A — друзьями для B. Приведите объяснения.
3. Какие проблемы могут возникнуть в следующем объявлении вложенного класса?

```
class Ribs
{
private:
   class Sauce
   {
      int soy;
      int sugar;
   public:
      Sauce(int s1, int s2) : soy(s1), sugar(s2) { }
   };
   ...
};
```

4. Чем throw отличается от return?
5. Предположим, что имеется иерархия классов исключений, порожденная от базового класса исключений. В каком порядке вы расположите блоки catch?
6. Рассмотрим классы Grand, Superb и Magnificent, определенные в настоящей главе. Допустим, что pg является указателем Grand *, которому присвоен адрес одного из объектов этих трех классов, а ps — указателем Superb *. В чем разница в поведении кода этих двух примеров?

```
if (ps = dynamic_cast<Superb *>(pg))
   ps->say();                 // пример #1
if (typeid(*pg) == typeid(Superb))
   (Superb *) pg)->say();    // пример #2
```

7. Чем отличается операция static_cast от операции dynamic_cast?

Упражнения по программированию

1. Измените классы Tv и Remote следующим образом:

   а. Сделайте их взаимными друзьями.

   б. Добавьте в класс Remote переменную-член, описывающую режим дистанционного управления — нормальный или интерактивный.

   в. Добавьте метод Remote, который отображает режим.

   г. Снабдите класс Tv методом для переключения нового члена Remote.

   Напишите короткую программу для тестирования новых возможностей.

2. Модифицируйте листинг 15.11 таким образом, чтобы два типа исключений были классами, производными от класса `logic_error`, определенного в заголовочном файле `<stdexcept>`. Сделайте, чтобы каждый метод `what()` отображал имя функции и суть проблемы. Объект исключения не должен содержать значение ошибки, а должен просто поддерживать метод `what()`.
3. Это упражнение подобно упражнению 2 за исключением того, что исключения должны быть производными от базового класса (являющегося производным от `logic_error`), который хранит два значения аргумента. Исключения должны иметь метод, отображающий эти значения и имя функции, и единственный блок `catch`, который используется для обоих исключений и, в том числе, для исключений, прерывающих цикл обработки.
4. В листинге 15.16 используются два блока `catch` после каждого блока `try`, поэтому исключение `nbad_index` приводит к вызову `label_val()`. Измените программу так, что бы она использовала один блок `catch` после каждого блока `try` и применяла RTTI для управления вызовом `label_val()`, когда это необходимо.

ГЛАВА 16

Класс string и стандартная библиотека шаблонов

В этой главе:

- Стандартный класс string в C++
- Шаблон auto_ptr
- Стандартная библиотека шаблонов (STL)
- Контейнерные классы
- Итераторы
- Объекты функций (функторы)
- Алгоритмы STL

Теперь, после изучения предыдущих глав, идея повторного использования кода в C++ должна быть более понятной. Одно из главных преимуществ — повторное использование кода, написанного другими программистами. И здесь на первый план выходят библиотеки классов. Существует множество библиотек классов C++, как отдельно распространяемых на платной основе, так и поставляемых в составе компилятора C++. Например, благодаря наличию заголовочного файла ostream можно использовать классы ввода-вывода. В этой главе рассматриваются вопросы повторного использования программного кода для различных нужд разработчика.

Класс string уже встречался ранее в этой книге. В этой главе он будет рассмотрен более детально. После этого будет описан класс auto_ptr — шаблон "интеллектуального указателя", облегчающий управление динамической памятью. И, наконец, будет представлена стандартная библиотека шаблонов (Standard Template Library — STL) — набор шаблонов для поддержки различных видов контейнерных объектов. STL иллюстрирует распространенную на данный момент идеологию программирования — *обобщенное программирование*.

Класс string

Обработка строк нужна в большинстве приложений. Язык C предоставляет некоторые функции для обработки строк в заголовочном файле string.h (cstring в C++), и множество ранних реализаций C++ предлагало "самодельные" классы для поддержки строк. Класс string стандарта ANSI/ISO C++ описан в главе 4. В главе 12 показаны некоторые аспекты разработки классов для поддержки строк.

Класс string поддерживается заголовочным файлом string (следует отметить, что заголовочные файлы string.h и cstring поддерживают библиотеку функций для работы со строками в стиле языка C, но не класс string). Для использования

класса нужно знать, какой интерфейс он предлагает. Класс string предоставляет обширный набор методов для работы со строками. В этот набор входят несколько конструкторов, перегруженных операций для доступа к строкам, конкатенации и сравнения строк, доступа к отдельным элементам строки, а также средства поиска символов и подстрок в строке. И это далеко не полный перечень возможностей класса string.

Создание объекта string

Рассмотрим конструкторы класса string. Опции, используемые при создании класса — одна из самых важных вещей, которые нужно знать о классе. В листинге 16.1 используются все шесть конструкторов класса string (они помечены комментариями ctor — это стандартная аббревиатура *конструктора* в C++). В табл. 16.1 кратко описаны все конструкторы, используемые в программе. В этой программе применяется упрощенное использование конструкторов и здесь скрыт тот факт, что на самом деле string — это определение типа (typedef) для шаблона basic_string<char>. Кроме того, здесь опущен необязательный параметр, относящийся к управлению памятью. Этот момент будет рассмотрен далее в этой главе, а также в приложении Ж. Тип size_type является внутренним типом, описанным в заголовочном файле string, и зависит от конкретной реализации объекта. В классе определяется параметр string::npos в качестве максимально возможной длины строки. Обычно он будет равен максимальному значению типа unsigned int. Также в таблице используется общепринятая аббревиатура для строк, завершающихся с нулевым символом — NBTS (null-byte-terminated string). Это обычные строки C, которые заканчиваются символом с кодом 0.

Листинг 16.1. str1.cpp

```
// str1.cpp -- введение в класс string
#include <iostream>
#include <string>
// использование различных конструкторов класса string
int main()
{
  using namespace std;
  string one("Lottery Winner!");      // ctor #1
  cout << one << endl;                // перегруженная <<
  string two(20, '$');                // ctor #2
  cout << two << endl;
  string three(one);                  // ctor #3
  cout << three << endl;
  one += " Oops!";                    // перегруженная +=
  cout << one << endl;
  two = "Sorry! That was ";
  three[0] = 'P';
  string four;                        // ctor #4
  four = two + three;                 // перегруженные +, =
  cout << four << endl;
  char alls[] = "All's well that ends well";
```

```
    string five(alls,20);             // ctor #5
    cout << five << "!\n";
    string six(alls+6, alls + 10);    // ctor #6
    cout << six  << ", ";
    string seven(&five[6], &five[10]); // снова ctor #6
    cout << seven << "...\n";
    return 0;
}
```

Таблица 16.1. Конструкторы класса string

Конструктор	Описание
`string(const char * s)`	Создает объект типа `string`, где `s` — указатель на NBTS.
`string(size_type n, char c)`	Создает объект типа `string` из `n` элементов, в каждый из которых заносится символ `c`.
`string(const string & str, string size_type n = npos)`	Инициализирует объект типа `string`, устанавливает параметр `pos` типа `size_type` в `0`. Объект `str` указывает на строку, начиная с позиции, заданной в параметре `pos`, и до конца строки, либо на `n` символов, независимо от того, какие символы были в начале строки.
`string()`	Создает объект типа `string` с параметрами по умолчанию и нулевой длиной.
`string(const char * s, size_type n)`	Объект типа `string` будет представлять NBTS, указанную переменной `s` и длиной `n` символов, даже если `n` больше, чем длина NBTS.
`template<class Iter> string(Iter begin, Iter end)`	Объект типа `string` будет представлять NBTS со значениями в диапазоне [`begin`, `end`). `begin` и `end` являются указателями начала и конца диапазона. Диапазон начинается и включает в себя элемент по адресу `begin` и продолжается до элемента по адресу `end`, но не включает его.

В программе также используется перегруженная операция += для сложения строк, перегруженная операция = для присваивания одной переменной типа `string` второй переменной такого же типа, перегруженная операция << для вывода содержимого объекта `string` на экран и перегруженная операция `[]` для доступа к отдельным элементам строки.

Замечание по совместимости

Некоторые ранние реализации класса `string` не поддерживают конструктор #6 из табл. 16.1. (Напоминаем, что *ctor* — это принятое в C++ сокращение для понятия "конструктор".)

Ниже показан результат работы программы из листинга 16.1:

```
Lottery Winner!
$$$$$$$$$$$$$$$$$$$$
Lottery Winner!
Lottery Winner! Oops!
Sorry! That was Pottery Winner!
All's well that ends!
well, well...
```

Замечания по программе

Первый конструктор создает объект string с именем one как строку C-стиля. Вывод на экран осуществляется с помощью перегруженной операции <<:

```
string one("Lottery Winner!"); // ctor #1
cout << one << endl;       // перегруженная <<
```

Следующий конструктор создает объект string с именем two и заполняет строку 20 символами $:

```
string two(20, '$');       // ctor #2
```

Следующий конструктор создает объект string с именем three, который является копией string-объекта one:

```
string three(one);         // ctor #3
```

Перегруженная операция += добавляет строку " Oops!" к строке one:

```
one += " Oops!";           // перегруженная +=
```

В этом примере строка в стиле языка C добавляется к объекту string. Однако в шаблоне объекта string операция += перегружена несколько раз, и с ее помощью можно добавлять как объекты string, так и отдельные символы:

```
one += two;                // добавляет объект string(в приведенной выше
                           // программе этой строки нет)
one += '!';                // добавляет значение типа char (в приведенной
                           // выше программе этой строки нет)
```

Операция = тоже перегружена. Таким образом, можно присвоить один объект string или значение типа char другому объекту string:

```
two = "Sorry! That was "; // присваивается строка в стиле C
two = one;                 // присваивается объект string (в приведенной
                           // выше программе этой строки нет)
two = '?';                 // присваивается значение типа char
                           // (в приведенной выше программе этой строки нет)
```

Перегрузка операции [], как показано в примере из главы 12, позволяет обращаться к отдельным элементам строки. В этом случае доступ к элементу стоки аналогичен доступу к элементу массива:

```
three[0] = 'P';
```

Конструктор по умолчанию создает пустую строку, которой впоследствии может быть присвоено значение:

```
string four;               // ctor #4
four = two + three;        // перегруженные +, =
```

Во второй строке используется перегруженная операция + для создания временного объекта string, который потом присваивается объекту four с помощью перегруженной операции =. Операция + объединяет два операнда в один объект string. В этой операции используется множественная перегрузка, и вторым операндом может быть как объект string, так и строка в стиле языка C, или отдельный символ.

Пятый конструктор использует параметры — строку в стиле языка C и целочисленное значение, которое указывает количество копируемых символов:

```
char alls[] = "All's well that ends well";
string five(alls,20);                    // ctor #5
```

Как видно из результата работы программы, при создании объекта five были использованы первые 20 элементов массива alls[] ("All's well that ends"). В табл. 16.1 было отмечено, что можно задать количество символов, превышающее длину строки С-стиля. В этом случае строка из параметра тоже используется при создании объекта. Если в предыдущем примере заменить 20 на 40, то в результате 15 последних элементов five будут заполнены содержимым следующих за строкой "All's well that ends well" 15 байтов памяти.

Конструктор sixth использует шаблон в качестве аргумента:

```
template<class Iter> string(Iter begin, Iter end);
```

begin и end выступают в роли указателей на начало и конец диапазона памяти. (В общем случае, begin и end могут быть итераторами — обобщениями указателей, активно использующимися в STL.) Конструктор применяет значения элементов памяти между begin и end для инициализации объекта six. Запись [begin, end), позаимствованная из математики, означает, что диапазон включает в себя begin, но не включает end. Таким образом, end указывает на значение, следующее за последним требуемым значением.

```
string six(alls + 6, alls + 10);         // ctor #6
```

Поскольку имя массива является указателем, значения alls + 6 и alls + 10 будут иметь тип char *, то в шаблоне тип Iter тоже будет char *. Первый аргумент указывает на первый элемент (w) в массиве alls, второй аргумент — на пробел после первого слова well. Итак, объект six инициализирован строкой "wells". На рис. 16.1 показано, как работает конструктор.

char alls[] = "All's well that ends well";
string six(alls + 6, alls + 10);

Диапазон = [**alls** + 6, **alls** + 10)

A	l	l	'	s		w	e	l	l		t	h	a	t		e	n	d	s		w	e	l	l	\0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

w	e	l	l

Рис. 16.1. Конструктор объекта **string** *использует диапазон*

Предположим, что нужно инициализировать объект частью строки другого объекта string. Представленный ниже код работать не будет:

```
string seven(five + 6, five + 10);
```

Причина в том, что имя объекта, в отличие от имени массива, не является адресом объекта. Следовательно, five не указатель и выражение five + 6 не имеет смысла. Однако, five[6] имеет тип char и в качестве аргумента можно использовать выражение &five[6], поскольку это адрес:

```
string seven(&five[6], &five[10]);       // снова ctor #6
```

Ввод для класса string

Для строк в стиле языка C существуют три варианта ввода данных:

```
char info[100];
cin >> info;                // читать слово
cin.getline(info, 100);     // читать строку, без \n
cin.get(info, 100);         // читать строку, оставить \n в очереди
```

У объекта типа string есть два варианта:

```
string stuff;
cin >> stuff;               // читать слово
getline(cin, stuff);        // читать строку, без \n
```

Оба варианта вызова функции getline() допускают использование необязательного аргумента — символа, завершающего ввод:

```
cin.getline(info,100,':');  // читать до тех пор, пока не встретится
                            // символ ':', символ ':' отбросить
getline(stuff, ':');        // читать до тех пор, пока не встретится
                            // символ ':', символ ':' отбросить
```

Основное отличие функции getline() при использовании объекта string связано с автоматическим изменением размера объекта string для хранения вводимых данных:

```
char fname[10];
string lname;
cin >> fname;               // если вводимых символов будет больше 9,
                            // то программа может работать некорректно
cin >> lname;               // можно считать очень длинные строки
cin.getline(fname, 10);     // вводимая строка ограничена 10 символами
getline(cin, fname);        // нет ограничений ввода
```

Такая особенность работы функции getline() позволяет отказаться от использования параметра, который указывает максимальную длину строки.

Средства ввода для строк в стиле C являются методами класса istream, а для объектов string — отдельными функциями. Таким образом, cin будет вызывающим объектом для функций ввода данных, и аргументом функции для объектов string. Это относится и к записи вида >>, и явно видно при записи кода в виде вызова функций:

```
cin.operator>>(fname);      // метод класса ostream
operator>>(cin, lname);     // обычный вызов функции
```

Рассмотрим более детально работу функций ввода. Обе функции, как отмечалось ранее, устанавливают размер строки для того, чтобы в нее поместились вводимые данные. Существует несколько ограничений на длину строки. Первый ограничивающий фактор — максимально допустимая длина строки, задаваемая константой string::npos. Обычно это максимальная величина типа unsigned int, и на практике этого размера хватает для обычного интерактивного ввода. Проблемы могут возникнуть, если попытаться прочитать содержимое файла в одну строку. Второй ограничивающий фактор — размер памяти, доступной программе.

Функция getline() для класса string будет читать данные из входного потока и сохранять их в строке до тех пор, пока не произойдет одно из трех событий:

- Достигнут конец файла. В этом случае во входном потоке будет установлен флаг eofbit и обе функции fail() и eof() будут возвращать значение true.

- Во входном потоке прочитан разделительный символ (\n по умолчанию). Этот символ удаляется из входного потока, но не сохраняется.
- Прочитано максимально возможное количество символов (меньше значения константы string::npos и меньше количества доступных байтов памяти). В этом случае во входном потоке будет установлен флаг failbit и функция fail() будет возвращать значение true.

(В объекте, обеспечивающем работу с входным потоком, имеется система учета и контроля за ошибками при работе с потоком. В этой системе установленный флаг eofbit означает достижение конца файла, failbit — ошибку при чтении из потока, badbit — нераспознанный сбой, например, аппаратный сбой, и goodbit — все работает нормально и без ошибок. Подробнее эта система рассматривается в главе 17.)

Функция operator>>() для класса string ведет себя аналогичным образом, за исключением того, что она считывает данные из потока до тех пор, пока не встретится разделитель (символ пробела, новой строки, табуляции или, в общем, любой символ, для которого функция isspace() вернет true), и оставляет его в очереди ввода.

Далее в этой книге будут приведены примеры консольного ввода для класса string. Поскольку функции ввода для объектов string работают с потоками и распознают достижение конца файла, их можно использовать для чтения строк из файла. В листинге 16.2 показан способ чтения строк из файла. Предполагается, что файл содержит строки, разделенные двоеточиями, и в методе getline() указан этот разделитель. Программа подсчитывает количество строк и выводит их на экран.

Листинг 16.2. strfile.cpp

```
// strfile.cpp -- чтение строк из файла
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
int main()
{
  using namespace std;
  ifstream fin;
  fin.open("tobuy.txt");
  if (fin.is_open() == false)
  {
    cerr << "Невозможно открыть файл. Завершение.\n";
    exit(EXIT_FAILURE);
  }
  string item;
  int count = 0;
  getline(fin, item, ':');
  while (fin)  // до тех пор, пока нет ошибок чтения
  {
    ++count;
    cout << count <<": " << item << endl;
    getline(fin, item,':');
  }
  cout << "Готово.\n";
  fin.close();
  return 0;
```

Предположим, что на вход программы подается файл tobuy.txt:

```
sardines:chocolate ice cream:pop corn:leeks:
cottage cheese:olive oil:butter:tofu:
```

Обычно если не указывать путь к текстовому файлу, программа будет искать его в той папке, в которой находится исполняемый файл, или, в ряде случаев, в той папке, в которой находится файл проекта. Можно задать полный путь к файлу. В системах Windows или DOS последовательность символов \\ в строке стиля C интерпретируется как один символ \:

```
fin.open("C:\\CPP\\Progs\\tobuy.txt"); // файл = C:\CPP\Progs\tobuy.txt
```

Ниже показан результат работы программы из листинга 16.2:

```
1: sardines
2: chocolate ice cream
3: pop corn
4: leeks
5:
cottage cheese
6: olive oil
7: butter
8: tofu
9:

Готово.
```

Обратите внимание, что при использовании символа : в качестве разделителя символ перевода строки стал просто еще одним обычным символом. Поэтому символ перевода строки в конце первой строки файла стал первым символом строки, которая начинается с "cottage cheese". А символ перевода строки в конце второй строки файла стал единственным элементом девятой строки.

Работа со строками

До настоящего момента были рассмотрены способы создания объектов string, отображения содержимого объекта, чтение данных объектом, добавление, сложение и присваивание объектов string. Что же еще можно делать со строками?

Строки можно сравнивать. Все шесть операций отношения перегружены для объектов string таким образом, что один объект будет считаться меньше другого, если он находится раньше в последовательности сортировки. Если последовательность сортировки основана на коде ASCII, то цифры будут меньше, чем заглавные символы, а заглавные символы меньше, чем прописные символы. Каждая операция отношения перегружена тремя способами, так что можно сравнивать объект string с другим объектом string, объект string с строкой стиля C, и строку стиля C с объектом string:

```
string snake1("cobra");
string snake2("coral");
char snake3[20] = "anaconda";
if (snake1 < snake 2)        // operator<(const string &, const string &)
...
if (snake1 == snake3)        // operator==(const string &, const char *)
...
if (snake3 != snake2)        // operator!=(const char *, const string &)
...
```

Существуют два метода класса string для определения размера строки — size() и length() — которые возвращают количество символов в строке:

```
if (snake1.length() == snake2.size())
   cout << "Строки имеют одну и ту же длину.\n"
```

Оба метода работают абсолютно одинаково. Функция length() пришла из ранних версий класса string, а size() добавлена для совместимости с STL.

Поиск подстроки или символа в строке можно провести несколькими способами. В табл. 16.2 кратко описаны четыре варианта использования метода find().

Таблица 16.2. Использование перегруженного метода find()

Прототип метода	Описание
`size_type find( const string & str, size_type pos= 0) const`	Поиск первого вхождения подстроки str, начиная с символа по адресу pos в исходной строке. Возвращает индекс первого символа найденной подстроки или string::npos, если подстрока не найдена.
`size_type find(const char * s, size_type pos= 0) const`	Поиск первого вхождения подстроки str, начиная с символа по адресу pos в исходной строке. Возвращает индекс первого символа найденной подстроки или string::npos, если подстрока не найдена.
`size_type find(const char * s, size_type pos= 0, size_type n)`	Поиск первого вхождения подстроки, состоящей из первых n символов строки s, начиная с символа по адресу pos в исходной строке. Возвращает индекс первого символа найденной подстроки или string::npos, если подстрока не найдена.
`size_type find(char ch, size_type pos= 0) const`	Поиск первого вхождения символа ch, начиная с символа по адресу pos в исходной строке. Возвращает индекс найденного символа или string::npos, если символ не найден.

Библиотека string предлагает также методы rfind(), find_first_of(), find_last_of(), find_first_not_of() и find_last_not_of(), каждый из которых обладает тем же набором перегруженных параметров, что и метод find(). Метод rfind() находит последнее вхождение подстроки или символа. Метод find_first_of() находит первое вхождение в строке любого из символов, переданных в аргументах метода. Например, оператор

```
int where = snake1.find_first_of("hark");
```

вернет позицию символа r в строке "cobra" (3), поскольку это первое вхождение любого из символов строки "hark" в строке "cobra". Метод find_last_of() работает аналогично, только находит последнее вхождение. Итак,

```
int where = snake1.find_last_of("hark");
```

вернет позицию символа a в строке "cobra". Метод find_first_not_of() находит первое вхождение символа в строке, которого нет в аргументах вызова метода. Таким образом

```
int where = snake1.find_first_not_of("hark");
```

вернет позицию символа c в "cobra", поскольку c не найден в "hark". (В упражнениях в конце этой главы будут приведены примеры использования find_last_not_of().)

Существует множество других методов, однако описанных выше достаточно для создания игровой программы — упрощенной версии игры "Палач" (детская игра в слова; при неправильном ответе игрок рисует одну за другой части виселицы с повешенным. — *Прим. ред.*). Программа хранит список слов в массиве объектов `string`, выбирает одно слово случайным образом и предлагает игроку угадать буквы в слове. Шесть неудачных попыток означают проигрыш. В программе используется функция `find()` для проверки попыток и операция += для создания объекта `string`, в котором хранятся неудачные попытки. Чтобы отследить удачные попытки, в программе создается слово такой же длины, что и загаданное, но состоящее из дефисов. При удачной попытке дефис заменяется угаданной буквой. Текст программы приведен в листинге 16.3.

Листинг 16.3. hangman.cpp

```
// hangman.cpp -- использование методов работы со строками
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include <cctype>

using std::string;

const int NUM = 26;
const string wordlist[NUM] = {"apiary", "beetle", "cereal",
   "danger", "ensign", "florid", "garage", "health", "insult",
   "jackal", "keeper", "loaner", "manage", "nonce", "onset",
   "plaid", "quilt", "remote", "stolid", "train", "useful",
   "valid", "whence", "xenon", "yearn", "zippy"};

int main()
{
  using std::cout;
  using std::cin;
  using std::tolower;
  using std::endl;

  std::srand(std::time(0));
  char play;
  cout << "Поиграем в игру в слова? <y/n> ";
  cin >> play;
  play = tolower(play);

  while (play == 'y')
  {
    string target = wordlist[std::rand() % NUM];
    int length = target.length();
    string attempt(length, '-');
    string badchars;
    int guesses = 6;
    cout << "Угадайте мое секретное слово. Оно содержит " << length
         << " букв, и вы можете угадывать\n"
         << "одну букву за раз. Вам предоставляется " << guesses
         << " неправильных угадываний.\n";
    cout << "Ваше слово: " << attempt << endl;
```

```
    while (guesses > 0 && attempt != target)
    {
      char letter;
      cout << "Угадайте букву: ";
      cin >> letter;
      if (badchars.find(letter) != string::npos
          || attempt.find(letter) != string::npos)
      {
        cout << "Вы уже задавали ее. Повторите попытку.\n";
        continue;
      }
      int loc = target.find(letter);
      if (loc == string::npos)
      {
        cout << "Неправильно!\n";
        --guesses;
        badchars += letter; // добавить к строке
      }
      else
      {
        cout << "Правильно!\n";
        attempt[loc]=letter;
        // проверить, не появляется ли буква еще раз
        loc = target.find(letter, loc + 1);
        while (loc != string::npos)
        {
          attempt[loc]=letter;
          loc = target.find(letter, loc + 1);
        }
      }
      cout << "Ваше слово: " << attempt << endl;
      if (attempt != target)
      {
        if (badchars.length() > 0)
        cout << "Неправильные варианты: " << badchars << endl;
        cout << guesses << " неправильных вариантов осталось\n";
      }
    }
    if (guesses > 0)
       cout << "Угадано!\n";
    else
       cout << "Очень жаль, но было задумано слово: " << target << ".\n";
    cout << "Играем еще? <y/n> ";
    cin >> play;
    play = tolower(play);
  }
  cout << "Удачи!\n";
  return 0;
}
```

Ниже приведен пример работы программы 16.3:

```
Поиграем в игру в слова? <y/n> y
Угадайте мое секретное слово. Оно содержит 6 букв, и вы можете угадывать
одну букву за раз. Вам предоставляется 6 неправильных угадываний.
Ваше слово: ------
Угадайте букву: e
Неправильно!
Ваше слово: ------
Неправильные варианты: e
5 неправильных вариантов осталось
Угадайте букву: a
Правильно!
Ваше слово: a--a—
Неправильные варианты: e
5 неправильных вариантов осталось
Угадайте букву: t
Неправильно!
Ваше слово: a--a--
Неправильные варианты: et
4 неправильных вариантов осталось
Угадайте букву: r
Правильно!
Ваше слово: a--ar-
Неправильные варианты: et
4 неправильных вариантов осталось
Угадайте букву: y
Правильно!
Ваше слово: a--ary
Неправильные варианты: et
4 неправильных вариантов осталось
Угадайте букву: i
Правильно!
Ваше слово: a-iary
Неправильные варианты: et
4 неправильных вариантов осталось
Угадайте букву: p
Правильно!
Ваше слово: apiary
Угадано!
Играем еще? <y/n> n
Удачи!
```

Замечания по программе

В программе 16.3 перегрузка операций отношения позволяет работать со строками так же, как с числовыми переменными:

```
while (guesses > 0 && attempt != target)
```

Такой подход проще, нежели использование, например, функции `strcmp()` со строками в стиле С.

Программа применяет `find()` для проверки на повторное использование символа; если символ уже вводился, то он будет присутствовать либо в строке `badchars` (неудачные попытки), либо в строке `attempt` (удачные попытки):

```
if (badchars.find(letter) != string::npos
     || attempt.find(letter) != string::npos)
```

Переменная npos — это статическое свойство класса string. Вспомните, что это максимально возможное количество символов в объекте string. Поскольку нумерация символов в строке начинается с нуля, то это значение на единицу больше, чем максимально возможная позиция символа, и может использоваться для индикации неудачного поиска символа в строке.

Также в программе используется перегруженная операция += для добавления символов к строке:

```
badchars += letter;  // добавить символ к объекту string
```

Основной цикл работы программы начинается с проверки на наличие введенного символа в загаданном слове:

```
int loc = target.find(letter);
```

Если в переменной loc находится правильное значение (меньшее npos), буква подставляется на нужное место в строке ответа:

```
attempt[loc]=letter;
```

Однако введенный символ может встречаться в загаданном слове несколько раз, поэтому программе приходится продолжать проверку. В программе используется необязательный аргумент функции find(), который указывает на позицию в строке, с которой необходимо начать поиск. Буква была найдена на позиции loc, поэтому следующий поиск должен начаться с loc+1. Цикл while будет повторять поиск до тех пор, пока не будет найдено больше ни одного значения. Обратите внимание, что find() сообщит об ошибке, если loc будет больше длины строки:

```
// проверить, не появляется ли буква еще раз
loc = target.find(letter, loc + 1);
while (loc != string::npos)
{
  attempt[loc]=letter;
  loc = target.find(letter, loc + 1);
}
```

Дополнительные возможности класса string

Библиотека string поддерживает множество дополнительных возможностей для работы со строками. Среди этих возможностей, например, удаление части или всей строки, замена части или всей строки частью другой строки (или всей строкой), добавление и удаление данных из строки, сравнение частей строк и строк целиком, извлечение подстроки из строки, копирование одной строки в другую, обмен содержимого двух строк. Большинство этих функций перегружено и может работать со строками в стиле С так же, как и с объектами string. В приложении Ж функции библиотеки string описаны более детально, но здесь тоже будут рассмотрены несколько особенностей.

Во-первых, обратимся к возможности автоматического изменения размера строки. Что происходит, когда в программе 16.3 добавляется символ к строке? Просто увеличить размер зачастую невозможно, поскольку соседние блоки памяти уже заняты. Поэтому нужно выделить новый блок памяти и скопировать туда содержимое строки. Частое повторение такой процедуры скажется на скорости работы программы, поэтому большинство реализаций С++ выделяют для строки блок памяти боль-

ший, чем сама строка, дабы строке было куда увеличиваться. Когда строка вырастает до размера блока, программа выделяет новый блок, вдвое больший, чем текущий размер строки. Такой подход уменьшает количество операций изменения размера. Метод `capacity()` возвращает размер текущего блока, а метод `reserve()` позволяет запросить минимальный размер блока. В листинге 16.4 демонстрируется использование этих методов:

Листинг 16.4. `str2.cpp`

```
// str2.cpp -- использование capacity() и reserve()
#include <iostream>
#include <string>
int main()
{
  using namespace std;
  string empty;
  string small = "bit";
  string larger = "Elephants are a girl's best friend";
  cout << "Размеры:\n";
  cout << "\tempty: " << empty.size() << endl;
  cout << "\tsmall: " << small.size() << endl;
  cout << "\tlarger: " << larger.size() << endl;
  cout << "Запасы:\n";
  cout << "\tempty: " << empty.capacity() << endl;
  cout << "\tsmall: " << small.capacity() << endl;
  cout << "\tlarger: " << larger.capacity() << endl;
  empty.reserve(50);
  cout << "Запасы после empty.reserve(50): "
       << empty.capacity() << endl;
  return 0;
}
```

Ниже показан пример работы программы из листинга 16.4 для одной из реализаций C++:

```
Размеры:
        empty: 0
        small: 3
        larger: 34
Запасы:
        empty: 15
        small: 15
        larger: 47
Запасы после empty.reserve(50): 63
```

В этой реализации C++ под строку резервируется минимум 15 байт и, похоже, что при увеличении размера используется инкремент 16. В других реализациях языка результаты работы программы могут отличаться.

Как поступить, если есть объект `string`, а нужна строка в стиле C? Например, требуется открыть файл, имя которого хранится в объекте `string`:

```
string filename;
cout << "Введите имя файла: ";
cin >> filename;
ofstream fout;
```

Проблема в том, что метод open() требует в качестве аргумента строку в стиле C. Однако существует метод c_str(), который возвращает указатель на строку с тем же содержимым, что и сам объект. Поэтому можно воспользоваться следующим подходом:

```
fout.open(filename.c_str());
```

Пример из практики: перегрузка функций C для использования объектов string
Для сравнения объектов string можно использовать перегруженную операцию ==. Однако эта операция выполняет проверку на равенство, чувствительную к регистру символов. В некоторых случаях такая проверка не подходит — например, в программе выполняется сравнение значений, введенных пользователем, с некоторой константой, а пользователь может использовать не тот регистр. В листинге 16.3 используется один из вариантов обхода этой проблемы — перевод всего ввода в нижний регистр. Существует и другой способ. Большинство библиотек языка C содержат функцию stricmp() или _stricmp(), которая выполняет сравнение строк, не зависящее от регистра символов. (Однако эта функция не входит в стандарт языка C, поэтому в некоторых реализациях ее может и не оказаться.) Создав перегруженный вариант этой функции, можно решить проблему сравнения строк, не зависящего от регистра символов:

```
#include <cstring>            // в большинстве реализаций stricmp()
                              // находится в библиотеке cstring
#include <string>             // объект string
inline bool stricmp( const std::string& strA,
const std::string& strB )    // перегруженная функция
{
  return stricmp( strA.c_str(), strB.c_str() ) == 0;    // функция языка C
}
string strA;
cin >> strA; // предположим, что пользователь вводит Maplesyrup
string strB = "mapleSyrup";  // константа для сравнения
bool bStringsAreEqual = stricmp( strA, strB );
```

Теперь можно сравнивать строки независимо от регистра символов. В более общем случае можно использовать метод c_str(), который позволяет конвертировать функции для работы с строками в стиле C в функции для работы с объектами string.

В этом разделе класс string рассматривается как класс, основанный на типе char. В действительности, как было отмечено ранее, библиотека string использует шаблонный класс:

```
template<class charT, class traits = char _traits<charT>,
        class Allocator = allocator<charT> >
basic_string {...};
```

В класс включено два определения типов (typedef):

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

Эти определения позволяют использовать строки на основе типа wchar_t так же успешно, как и строки на основе типа char. Более того, можно разработать свой класс на базе символьных типов и использовать шаблон класса basic_string, при условии, что новый класс удовлетворяет определенным требованиям.

Класс traits содержит описание выбранного символьного типа, например, способы сравнения значений. Существуют заранее определенные спецификации шаблона char_traits для типов char и wchar_t, и эти спецификации являются значениями по умолчанию для класса traits.

Класс Allocator используется для управления памятью. Существуют заранее определенные спецификации шаблона Allocator для типов char и wchar_t, и эти спецификации являются значениями по умолчанию. Шаблон применяет операции new и delete обычным способом, однако можно зарезервировать область памяти и создать собственные методы для управления памятью.

Класс auto_ptr

Класс auto_ptr — это шаблонный класс для контроля над распределением динамической памяти. Рассмотрим этот класс более детально. В следующем примере:

```
void remodel(string & str)
{
  string * ps = new string(str);
  ...
  str = ps;
  return;
}
```

работа программы довольно прозрачна. При каждом вызове функции происходит выделение памяти в куче, однако память не освобождается. Это приводит к утечкам памяти. Решение проблемы очевидно — не забыть освободить память, добавив следующую строку:

```
delete ps;
```

перед оператором return. Однако, решение, которое содержит фразу “не забыть”, не является идеальным. Иногда разработчик не желает помнить об этом. Или разработчик помнит, но случайно удалил или закомментировал строку. И даже в том случае, когда это решение используется, могут возникать проблемы. Например, как в этом примере:

```
void remodel(string & str)
{
  string * ps = new string(str);
  ...
  if (weird_thing())
     throw exception();
  str = *ps;
  delete ps;
  return;
}
```

При возникновении исключительной ситуации операция delete вообще не будет достигнута, и снова появится утечка памяти.

Такие ошибки исправимы, как было показано в главе 14, но желательно было бы иметь лучшее решение. Что же для этого нужно? Когда функция вроде remodel() завершает работу, успешно или после исключительной ситуации, локальные переменные удаляются из стека — таким образом, память, занятая указателем ps, тоже

освобождается. Было бы неплохо, если бы память, на которую указывал ps, также была освобождена. Это означает, что программа должна выполнять некоторые дополнительные действия после удаления указателя. В базовых типах подобной функциональности нет. Для классов такая возможность предоставлена через деструкторы. Таким образом, проблема состоит в том, что ps — это обычный указатель, а не объект класса. Если бы это был объект, то деструктор бы освобождал память, на которую указывал объект, при удалении объекта. Это и есть основная идея использования auto_ptr.

Использование auto_ptr

Шаблон auto_ptr определяет объект, которому присваивается адрес области памяти, полученный (прямо или косвенно) вызовом операции new. Когда объект auto_ptr удаляется, его деструктор использует delete для освобождения памяти. Таким образом, присвоив адрес, возвращаемый операцией new, объекту auto_ptr, не нужно беспокоиться о последующем высвобождении памяти. Это будет сделано автоматически, при удалении объекта. На рис. 16.2 показаны различия в работе auto_ptr и обычного указателя:

Для создания объекта auto_ptr необходимо включить заголовочный файл memory, в котором содержится шаблон auto_ptr. После этого, используя обычный синтаксис шаблонов, можно создавать указатели нужного типа.

Шаблон включает в себя следующее:

```
template<class X> class auto_ptr {
public:
  explicit auto_ptr(X* p =0) throw();
...};
```

(Запись throw() означает, что конструктор не должен вызывать исключения.) Итак, указывая при создании объекта тип X, разработчик получает объект auto_ptr, который указывает на значение типа X:

```
auto_ptr<double> pd(new double);  // объект auto_ptr, указывающий
                                  // на значение типа double
                                  // (используется вместо double *)
auto_ptr<string> ps(new string);  //объект auto_ptr, указывающий на строку
                                  // (используется вместо string *)
```

В этом примере new double — это указатель (возвращаемый new) на новый выделенный участок памяти. new double используется в качестве аргумента при вызове конструктора auto_ptr<double>. Этот аргумент соответствует формальному параметру p в прототипе класса. Аналогично, new string тоже будет аргументом для конструктора класса.

Итак, чтобы использовать auto_ptr() в функции remodel(), потребуется выполнить три шага:

1. Подключить заголовочный файл memory.
2. Заменить указатель на тип string объектом auto_ptr, который указывает на string.
3. Убрать строку с delete.

```
void demo1()
{
    double * pd = new double; // #1
    *pd = 25.5;               // #2
    return;                   // #3
}
```

#1: Создается хранилище для **pd** и значения типа **double**, адрес сохраняется:

#2: Значение копируется в динамическую память:

#3: **pd** удаляется, значение остается в памяти:


```
void demo2()
{
    auto_ptr<double> ap(new double); // #1
    *ap = 25.5;                      // #2
    return;                          // #3
}
```

#1: Создается хранилище для **pd** и значения типа **double**, адрес сохраняется:

#2: Значение копируется в динамическую память:

#3: **ap** удаляется и деструктор объекта **ap** освобождает память.

***Рис. 16.2. Обычный указатель и класс* `auto_ptr`**

Теперь функция `remodel()` будет выглядеть так:

```
#include <memory>
void remodel(string & str)
{
  auto_ptr<string> ps (new string(str));
  ...
  if (weird_thing())
     throw exception();
  str = *ps;
  // delete ps;  В ЭТОМ НЕТ НЕОБХОДИМОСТИ
  return;
}
```

В конструкторе `auto_ptr` присутствует директива `explicit`, поэтому неявного преобразования типов из указателя в объект `auto_ptr` не происходит:

```
auto_ptr<double> pd;
double *p_reg = new double;
pd = p_reg;                         // недопустимо (неявное преобразование)
pd = auto_ptr<double>(p_reg);       // допустимо (явное преобразование)
auto_ptr<double> pauto = p_reg;     // недопустимо (неявное преобразование)
auto_ptr<double> pauto(p_reg);      // допустимо (явное преобразование)
```

Заметим, что шаблон позволяет инициализировать объект auto_ptr обычным указателем, используя конструктор.

auto_ptr является примером *интеллектуального указателя*, то есть объекта, ведущего себя как указатель и обладающего дополнительными свойствами. Класс auto_ptr определен таким образом, что в большинстве случаев объект auto_ptr работает подобно обычному указателю. Например, если ps — это объект auto_ptr, то его можно разыменовать (*ps), инкрементировать (++ps), использовать для доступа к членам структуры (ps->puffIndex) и присваивать обычному указателю на тот же тип, что и объект. Также можно присвоить один объект auto_ptr другому того же типа, но здесь возникают моменты, которые будут рассмотрены в следующем разделе.

Соображения по использованию auto_ptr

auto_ptr — это не панацея. Например, в следующем примере:

```
auto_ptr<int> pi(new int [200]);// ТАК ДЕЛАТЬ НЕЛЬЗЯ!
```

delete используется только в паре с new, а delete [] — только с new []. В шаблоне auto_ptr применяется delete, а не delete [], поэтому он может использоваться только с new. Эквивалента auto_ptr для динамических массивов нет.

Можно скопировать шаблон auto_ptr из заголовочного файла memory, переименовать его в auto_arr_ptr и изменить delete на delete []. В этом случае понадобится добавить поддержку операции [].

А что насчет такого подхода:

```
string vacation("I wandered lonely as a cloud.");
auto_ptr<string> pvac(&vacation);  // ТАК ДЕЛАТЬ НЕЛЬЗЯ!
```

Здесь операция delete применяется к нераспределенной памяти, что совершенно неприемлемо.

Внимание!

Объект auto_ptr должен использоваться только для памяти, выделенной операцией new. Память, выделенная с помощью new[] или простого объявления переменной, не подходит.

Рассмотрим следующий пример:

```
auto_ptr<string> ps (new string("I reigned lonely as a cloud."));
auto_ptr<string> vocation;
vocation = ps;
```

Чем завершится оператор присваивания? Если бы ps и vocation были обычными указателями, то в результате получилось бы два указателя на один объект string. Это недопустимо, поскольку программа будет вести себя непредсказуемо, пытаясь удалить один объект дважды — при удалении ps и при удалении vocation. Вот способы обхода этой проблемы:

- Определить операцию присваивания так, чтобы она создавала точную копию объекта. Тогда два указателя будут указывать на разные объекты.
- Использовать концепцию *владения*, когда определенным объектом может владеть только один интеллектуальный указатель. Деструктор будет удалять объект только тогда, когда интеллектуальный указатель владеет объектом. Затем можно использовать операцию присваивания, которая будет передавать право владения объектом. Эта стратегия используется для auto_ptr.
- Создать еще более интеллектуальный указатель, который будет отслеживать, сколько интеллектуальных указателей ссылается на определенный объект. Эта стратегия называется *подсчетом ссылок*. Присваивание, например, будет увеличивать счетчик на единицу, удаление указателя — уменьшать счетчик. delete будет вызвана только тогда, когда будет удаляться последний указатель.

Эти же стратегии применимы и к конструкторам копирования.

В разных ситуациях применяется различные подходы. Ниже приведен пример, который может работать с объектами auto_ptr некорректно:

```
auto_ptr<string> films[5] =
{
  auto_ptr<string> (new string("Fowl Balls")),
  auto_ptr<string> (new string("Duck Walks")),
  auto_ptr<string> (new string("Chicken Runs")),
  auto_ptr<string> (new string("Turkey Errors")),
  auto_ptr<string> (new string("Goose Eggs"))
};
auto_ptr<string> pwin(films[2]);
int i;
cout << "The nominees for best avian baseball film are\n";
for (i = 0; i < 5; i++)
    cout << *films[i] << endl;
cout << "The winner is " << *pwin << "!\n";
```

Проблема в том, что передача права владения от films[2] к pwin может привести к тому, что films[2] больше не будет указывать на строку. Действительно, после передачи права владения объект auto_ptr может стать непригодным для использования. Так это будет или нет — зависит от конкретной реализации.

Интеллектуальные указатели

Объект auto_ptr из библиотеки C++ является примером интеллектуального указателя. *Интеллектуальный указатель* — это класс, разработанный таким образом, чтобы объекты этого класса имели свойства, аналогичные указателям. Например, интеллектуальный указатель может хранить адрес памяти, выделенной операцией new, и может быть разыменован. Поскольку интеллектуальный указатель — это класс, то он может изменять поведение указателя, а также расширять возможности использования указателя. Например, в интеллектуальном указателе может быть использован *подсчет ссылок*, что позволит нескольким объектам ссылаться на одну и ту же область памяти. Когда количество объектов, использующих эту область памяти, станет равно нулю, интеллектуальный указатель может удалить это значение. Интеллектуальные указатели позволяют использовать память более эффективно и могут предотвратить утечки памяти, однако они требуют знакомства с новыми технологиями программирования.

Стандартная библиотека шаблонов (STL)

Библиотека STL предоставляет набор шаблонов, представляющих контейнеры, итераторы, объекты функций и алгоритмы. Контейнер — это структура данных (подобная массиву), которая может хранить несколько значений. Контейнеры STL однородны по структуре и хранят только однотипные значения. Алгоритмы используются для выполнения определенных задач, например, сортировки массива или поиска значения в списке. Итераторы — это объекты, позволяющие перемещаться внутри контейнера подобно тому, как с помощью указателей можно перемещаться по массиву; они являются обобщениями указателей. Объекты функций — это объекты, которые ведут себя как функции; они могут быть объектами класса или указателями на функции (включая имена функций, поскольку имя функции это и есть указатель). STL позволяет создавать различные контейнеры, содержащие массивы, очереди и списки, и осуществлять множество операций, включая поиск, сортировку и тасование в случайном порядке.

Алексей Степанов (Alex Stepanov) и Менг Ли (Meng Lee) разработали STL в лабораториях Hewlett-Packard в 1994 году. Комитет по стандарту ISO/ANSI C++ проголосовал за внедрение библиотеки в стандарт C++. STL не является примером объектно-ориентированного программирования. Здесь используется другая идеология программирования — *обобщенное программирование*. Поэтому STL интересна как предоставляемыми возможностями, так и способами их достижения. Полный обзор всех возможностей и технологий STL слишком велик для одной главы, поэтому здесь будут представлены только примеры использования, позволяющие ощутить дух подхода обобщенного программирования. После этого, когда будет достигнуто достаточное понимание работы контейнеров, итераторов и алгоритмов, будет рассмотрена философия архитектуры библиотеки и общий обзор STL. В приложении Ж приведен обзор различных функций и методов STL.

Шаблон `vector`

В вычислительных процессах термином *вектор* обозначается массив, в отличие от векторов, используемых в математике (которые обсуждались в главе 11). С точки зрения математики, *N*-мерный вектор может быть представлен набором из *N* компонент, то есть математический вектор похож на *N*-мерный массив. Однако математический вектор обладает дополнительными свойствами, такими как скалярное и векторное произведение, каковыми компьютерный вектор может и не обладать. Компьютерный вектор — это набор однотипных значений, к которым можно обращаться в любом порядке. То есть, можно получить непосредственный доступ к 10-му элементу вектора без необходимости считывать 9 предыдущих элементов. Итак, класс `vector` должен предоставлять возможности, аналогичные классам `valarray` и `ArrayTP`, представленным в главе 14. То есть, можно создать объект `vector`, присвоить один объект `vector` другому и применять операцию `[]` для доступа к отдельным элементам. Чтобы сделать этот класс обобщенным, нужно реализовать его в виде шаблона. Именно так и поступает STL, определяя шаблон `vector` в заголовочном файле `vector` (ранее — `vector.h`).

При создании объекта шаблона `vector` используется запись `<type>` для указания используемого типа. Шаблон `vector` применяет динамическое выделение памяти, и в качестве аргумента при инициализации можно указать количество элементов вектора:

```
#include vector
using namespace std;
vector<int> ratings(5);     // вектор из 5 значений типа int
int n;
cin >> n;
vector<double> scores(n);   // вектор из n значений типа double
```

После создания объекта vector перегрузка операции [] позволяет обращаться к элементам вектора так же, как к элементам массива:

```
ratings[0] = 9;
for (int i = 0; i < n; i++)
    cout << scores[i] << endl;
```

Еще раз о распределителях

Так же, как и в классе string, различные контейнеры шаблонов STL принимают необязательный аргумент, который указывает, какой объект-распределитель будет использоваться для управления памятью. Например, шаблон vector начинается так:

```
template <class T, class Allocator = allocator<T> >
class vector {...
```

Если не передавать значение этого аргумента, контейнер шаблона будет использовать класс allocator<T> по умолчанию. Этот класс обычно использует new и delete.

Применение класса vector демонстрируется в примере 16.5. Здесь создается два объекта vector, один содержит элементы типа int, второй — string. В каждом объекте находится по 5 элементов.

Листинг 16.5. vect1.cpp

```
// vect1.cpp -- пример работы с шаблоном vector
#include <iostream>
#include <string>
#include <vector>
const int NUM = 5;
int main()
{
  using std::vector;
  using std::string;
  using std::cin;
  using std::cout;
  using std::endl;
  vector<int> ratings(NUM);
  vector<string> titles(NUM);
  cout << "Вы должны делать в точности то, что описано. Вы должны ввести\n"
       << NUM << " названий книг, а также их рейтинги (0-10).\n";
  int i;
  for (i = 0; i < NUM; i++)
  {
    cout << "Введите название книги #" << i + 1 << ": ";
    getline(cin,titles[i]);
    cout << "Укажите свой рейтинг (0-10): ";
    cin >> ratings[i];
    cin.get();
  }
```

```
  cout << "Спасибо. Вы ввели следующую информацию:\n"
       << "Рейтинг\tКнига\n";
  for (i = 0; i < NUM; i++)
  {
    cout << ratings[i] << "\t" << titles[i] << endl;
  }
  return 0;
}
```

Замечание по совместимости

Ранние реализации STL использовали `vector.h` вместо заголовочного файла `vector`. Хотя порядок, в котором перечислены подключаемые библиотеки, не важен, некоторые ранние версии g++ требуют, чтобы заголовочный файл `string` появился раньше, чем заголовочные файлы STL. Ранние версии Microsoft Visual C++ содержат ошибку в функции `getline()`, которая нарушает синхронизацию ввода и вывода.

Ниже приведен пример работы программы 16.5:

```
Вы должны делать в точности то, что описано. Вы должны ввести
названий книг, а также их рейтинги (0-10).
Введите название книги #1: The Cat Who Knew C++
Укажите свой рейтинг (0-10): 6
Введите название книги #2: Felonious Felines
Укажите свой рейтинг (0-10): 4
Введите название книги #3: Warlords of Wonk
Укажите свой рейтинг (0-10): 3
Введите название книги #4: Don't Touch That Metaphor
Укажите свой рейтинг (0-10): 5
Введите название книги #5: Panic Oriented Programming
Укажите свой рейтинг (0-10): 8
Спасибо. Вы ввели следующую информацию:
Рейтинг Книга
6       The Cat Who Knew C++
4       Felonious Felines
3       Warlords of Wonk
5       Don't Touch That Metaphor
8       Panic Oriented Programming
```

Все, что делает эта программа — это использует шаблон `vector` для создания динамического массива. Ниже приведены более развернутые примеры.

Что еще можно делать с помощью векторов

Что же еще, кроме выделения памяти, может предложить разработчику шаблон `vector`? У всех контейнеров STL имеется набор базовых методов вроде следующих: `size()` — возвращает количество элементов в контейнере, `swap()` — обменивает содержимое двух контейнеров, `begin()` — возвращает итератор, ссылающийся на первый элемент в контейнере, и `end()` — возвращает итератор, ссылающийся на область памяти, следующую за последним элементом в контейнере.

Что представляет собой итератор? Это обобщение указателей. Итератор может быть указателем. Или он может быть объектом, для которого определены операции над указателями — разыменование (например, `operator*()`) и инкремент (например, `operator++()`). Как станет видно из дальнейших примеров, обобщение указателей

позволяет STL предоставлять одинаковый интерфейс для множества классов-контейнеров, включая те, для которых обычные указатели не работают. В каждом классе-контейнере определяется соответствующий итератор. Типом этого итератора будет определение типа (typedef), действующее в области конкретного класса, и называемое iterator. Например, для объявления итератора для класса vector типа double применяется следующий синтаксис:

```
vector<double>::iterator pd;   // pd - это итератор
```

Предположим, что scores — это объект vector<double>:

```
vector<double> scores;
```

Теперь можно использовать итератор pd, например, так:

```
pd = scores.begin();            // pd ссылается на первый элемент
*pd = 22.3; // разыменование pd и присваивание значения первому элементу
++pd;       // pd указывает на следующий элемент
```

Как видно, итератор ведет себя подобно указателю.

Зачем нужен итератор, возвращаемый методом end()? Он указывает на область памяти, следующую за последним элементом в контейнере. Идея применения этого итератора сходна с идеей использования символа с ASCII-кодом, равным 0, в строках в стиле C, за исключением того, что нулевой символ — это значение элемента, а end() возвращает указатель (или итератор) на этот элемент. Если установить итератор на первый элемент контейнера и увеличивать его, то, в конце концов, будет достигнут элемент, на который указывает итератор, возвращаемый end(), то есть все содержимое контейнера было пройдено. Итак, если scores и pd определены так, как в предыдущем примере, то все содержимое контейнера можно отобразить так:

```
for (pd = scores.begin(); pd != scores.end(); pd++)
   cout << *pd << endl;;
```

У всех контейнеров имеются методы, описанные выше. В шаблоне vector есть также некоторые методы, которые присутствуют не во всех контейнерах STL. Один из полезных методов — push_back() — добавляет элемент в конец вектора. При выполнении этой операции осуществляется дополнительное выделение памяти, и размер вектора увеличивается, чтобы в него поместились добавляемые элементы. Этот метод позволяет писать, например, такой код:

```
vector<double> scores;  // создание пустого вектора
double temp;
while (cin >> temp && temp >= 0)
   scores.push_back(temp);
cout << "Вы ввели " << scores.size() << " оценок.\n";
```

На каждом шаге цикла к вектору scores добавляется один элемент. Не нужно заботиться о количестве элементов во время создания или запуска программы. До тех пор, пока у программы есть доступ к необходимому количеству памяти, размер scores будет при необходимости увеличиваться.

Метод erase() удаляет из вектора заданный диапазон. В качестве аргументов принимаются два итератора, указывающих границы диапазона. Важно понимать, как STL определяет диапазон, заданный итераторами. Первый итератор указывает на начало диапазона, второй — на элемент, следующий за концом диапазона.

Например:

```
scores.erase(scores.begin(), scores.begin() + 2);
```

удалит первый и второй элементы — те, на которые ссылается begin() и begin()+1. (Поскольку vector предоставляет непосредственный доступ к любому элементу, такие действия, как begin() + 2, определены для итераторов класса vector.) В литературе по STL используется также запись вида [p1, p2) (где p1 и p2 — итераторы), описывающая диапазон, начинающийся с p1 и продолжающийся, но не включающий p2. Таким образом, диапазон [begin(), end()) охватывает все содержимое контейнера (рис. 16.3). Кроме того, диапазон [p1, p1) — это пустой диапазон. (Запись вида [) не является частью языка C++, поэтому в коде она не используется и присутствует только в документации.)

На память!

Диапазон [it1, it2) указан двумя итераторами it1 и it2; он начинается с it1 и продолжается, но не включает в себя it2.

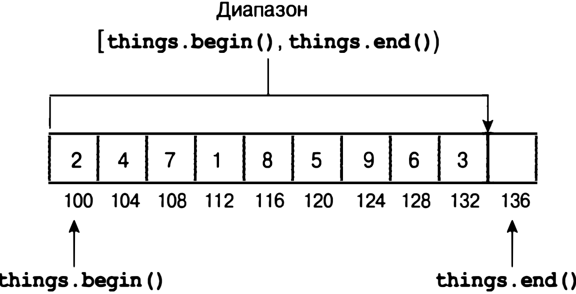


Рис. 16.3. Диапазоны в STL

Метод insert() дополняет erase(). В качестве аргументов принимаются три итератора. Первый указывает на позицию, после которой будут добавляться новые элементы. Второй и третий итераторы описывают диапазон, который будет добавлен. Этот диапазон обычно является частью другого контейнера. Например, следующий код:

```
vector<int> old;
vector<int> new;
...
old.insert(old.begin(), new.begin() + 1, new.end());
```

добавит все элементы вектора new, кроме первого, после первого элемента вектора old. Кстати, здесь может пригодиться метод end() для облегчения операции добавления в конец вектора. В коде

```
old.insert(old.end(), new.begin() + 1, new.end());
```

новые данные добавляются в позицию old.end(), *после* последнего элемента вектора.

В примере 16.6 демонстрируется применение size(), begin(), end(), push_back(), erase() и insert(). Для упрощения работы с данными компоненты rating и title объединены в одну структуру Review, а функции FillReview() и ShowReview() предоставляют возможности ввода и вывода для объектов Review.

Листинг 16.6. vect2.cpp

```
// vect2.cpp -- методы и итераторы
#include <iostream>
#include <string>
#include <vector>
struct Review {
   std::string title;
   int rating;
};
bool FillReview(Review & rr);
void ShowReview(const Review & rr);
int main()
{
  using std::cout;
  using std::vector;
  vector<Review> books;
  Review temp;
  while (FillReview(temp))
  books.push_back(temp);
  int num = books.size();
  if (num > 0)
  {
    cout << "Спасибо. Вы ввели следующую информацию:\n"
    << "Рейтинг\tКнига\n";
    for (int i = 0; i < num; i++)
       ShowReview(books[i]);
    cout << "Повторение:\n"
          << "Рейтинг\tКнига\n";
    vector<Review>::iterator pr;
    for (pr = books.begin(); pr != books.end(); pr++)
       ShowReview(*pr);
    vector <Review> oldlist(books); //используется конструктор копирования
    if (num > 3)
    {
      // 2 элемента удаляются
      books.erase(books.begin() + 1, books.begin() + 3);
      cout << "После подчистки:\n";
      for (pr = books.begin(); pr != books.end(); pr++)
          ShowReview(*pr);
      // 1 элемент добавляется
      books.insert(books.begin(), oldlist.begin() + 1,
      oldlist.begin() + 2);
      cout << "После вставки:\n";
      for (pr = books.begin(); pr != books.end(); pr++)
          ShowReview(*pr);
    }
    books.swap(oldlist);
    cout << "Обмен oldlist и books:\n";
    for (pr = books.begin(); pr != books.end(); pr++)
       ShowReview(*pr);
  }
  else
   cout << "Ничего не введено, ничего и не сделано.\n";
  return 0;
}
```

```
bool FillReview(Review & rr)
{
  std::cout << "Введите название книги (quit для завершения): ";
  std::getline(std::cin,rr.title);
  if (rr.title == "quit")
   return false;
  std::cout << "Введите рейтинг книги: ";
  std::cin >> rr.rating;
  if (!std::cin)
   return false;
  std::cin.get();
  return true;
}
void ShowReview(const Review & rr)
{
  std::cout << rr.rating << "\t" << rr.title << std::endl;
}
```

Замечание по совместимости

Ранние реализации C++ использовали `vector.h` вместо заголовочного файла `vector`. Хотя порядок, в котором перечислены подключаемые библиотеки, не важен, некоторые ранние версии g++ требуют, чтобы заголовочный файл `string` появился раньше, чем заголовочные файлы STL. Ранние версии Microsoft Visual C++ 6.0 содержат ошибку в функции `getline()`, из-за которой вывод данных на экран не происходит до тех пор, пока не будет введено что-либо еще. (В этом случае пользователю придется нажать клавишу <Enter> два раза после ввода заголовка книги.)

Ниже показан пример работы программы из листинга 16.6:

```
Введите название книги (quit для завершения): The Cat Who Knew Vectors
Введите рейтинг книги: 5
Введите название книги (quit для завершения): Candid Canines
Введите рейтинг книги: 7
Введите название книги (quit для завершения): Warriors of Wonk
Введите рейтинг книги: 4
Введите название книги (quit для завершения): Quantum Manners
Введите рейтинг книги: 8
Введите название книги (quit для завершения): quit
Спасибо. Вы ввели следующую информацию:
Рейтинг Книга
5       The Cat Who Knew Vectors
7       Candid Canines
4       Warriors of Wonk
8       Quantum Manners
Повторение:
Рейтинг Книга
5       The Cat Who Knew Vectors
7       Candid Canines
4       Warriors of Wonk
8       Quantum Manners
После подчистки:
5       The Cat Who Knew Vectors
8       Quantum Manners
После вставки:
7       Candid Canines
```

```
5       The Cat Who Knew Vectors
8       Quantum Manners
Обмен oldlist и books:
5       The Cat Who Knew Vectors
7       Candid Canines
4       Warriors of Wonk
8       Quantum Manners
```

Дополнительные возможности векторов

Существует множество задач, которые часто выполняются над массивами, такие как поиск, сортировка, тасование и так далее. Предлагает ли шаблон `vector` соответствующие методы? Нет! STL использует более обобщенный подход, определяя *глобальные* функции (или функции – не члены класса) для таких операций. Вместо того чтобы определять отдельный метод `find()` для каждого класса, в библиотеке определяется одна глобальная функция `find()`, которая может использоваться для всех классов. Такой подход позволяет уйти от повторения однотипных функций. Например, пусть имеется 8 контейнеров и 10 операций над ними. Если бы в каждом классе определялась своя собственная функция для каждой операции, то потребовалось бы определять 8×10, или 80, функций. С применением идеологии STL потребуется всего лишь 10 определений глобальных функций. И, создав новый класс-контейнер с применением соответствующих указаний, можно использовать уже существующие глобальные функции для сортировки, поиска и тому подобного.

Рассмотрим более детально три типичных функции STL: `for_each()`, `random_shuffle()` и `sort()`. `for_each()` работает с любым классом-контейнером. Функция имеет три аргумента. Первые два аргумента – это итераторы, определяющие диапазон, третий аргумент – указатель на функцию (точнее – функтор, далее будет рассмотрен детальнее). Функция `for_each()` применяет функцию, указанную в аргументе, ко всем элементам указанного диапазона. Функция, указанная в аргументе, не должна влиять на значение элементов контейнера. `for_each()` может использоваться вместо цикла `for`. Например, код:

```
vector<Review>::iterator pr;
for (pr = books.begin(); pr != books.end(); pr++)
   ShowReview(*pr);
```

можно заменить следующим кодом:

```
for_each(books.begin(), books.end(), ShowReview);
```

Теперь код очищен от чрезмерного использования переменных итераторов.

Функция `random_shuffle()` в качестве аргументов принимает два итератора – указателя границ диапазона и тасует элементы в этом диапазоне случайным образом. Например, выражение

```
random_shuffle(books.begin(), books.end());
```

изменяет случайным образом расположение всех элементов вектора `books`. В отличие от функции `for_each`, которая работает с любым классом-контейнером, функция `random_shuffle` требует от контейнера возможности непосредственного доступа к произвольному элементу. Класс `vector` удовлетворяет этим требованиям.

Функция `sort()` также требует от контейнера возможности непосредственного доступа к произвольному элементу. Существует две версии этой функции. Первая ис-

пользует два итератора, определяющих границы диапазона, и сортирует элементы этого диапазона с помощью операции <, определенной для типа элемента, хранящегося в контейнере. Например, выражение

```
vector<int> coolstuff;
...
sort(coolstuff.begin(), coolstuff.end());
```

сортирует содержимое coolstuff по возрастанию, используя встроенную операцию < для сравнения значений.

Если элементами контейнера являются объекты, типы которых определены пользователем, то для этого типа объекта должна быть определена функция operator<(), в противном случае функция sort() работать не будет. Например, можно сортировать вектор, содержащий объекты Review, если будет создан либо метод класса Review, либо глобальная функция для operator<(). Поскольку Review — это структура (struct), ее члены доступны для любого пользователя класса, и в этом случае можно применять глобальную функцию:

```
bool operator<(const Review & r1, const Review & r2)
{
  if (r1.title < r2.title)
      return true;
  else if (r1.title == r2.title && r1.rating < r2.rating)
      return true;
  else
      return false;
}
```

Используя эту функцию, можно отсортировать вектор с объектами Review (такими как books):

```
sort(books.begin(), books.end());
```

Приведенный выше вариант функции operator<() сортирует структуры в алфавитном порядке по полю title. Если у двух объектов поля title совпадают, то объекты сортируются по полю rating.

Предположим, что требуется провести сортировку не по названию книг, а по рейтингу. В этом случае используется второй вариант функции sort() с тремя аргументами. Первые два — это снова итераторы, определяющие диапазон. Третий аргумент — указатель на функцию (точнее — функтор), которая будет использоваться вместо operator<() для сравнения. Функция должна возвращать значение, которое можно преобразовать в тип bool, где false означает, что аргументы функции расположены не по порядку. Вот пример такой функции:

```
bool WorseThan(const Review & r1, const Review & r2)
{
  if (r1.rating < r2.rating)
      return true;
  else
      return false;
}
```

Используя эту функцию, можно отсортировать вектор books, состоящий из объектов Review, по возрастанию рейтинга:

```
sort(books.begin(), books.end(), WorseThan);
```

Обратите внимание, что функция WorseThan() сортирует объекты Review не совсем так, как operator<(). Если у двух объектов поле title одинаково, то operator<() будет сортировать по полю rating. Однако если у обоих объектов значения в полях rating будут совпадать, функция WorseThan() будет считать эти объекты одинаковыми. Первый тип сортировки называется *полным упорядочением*, а второй — *строгим квазиупорядочением*. При полном упорядочении, если оба выражения $a < b$ и $b < a$ ложны, то a должно быть идентично b. При строгом квазиупорядочении это не так. Объекты могут быть полностью идентичными, а могут совпадать только по одному критерию, такому как поле rating в примере с функцией WorseThan(). Поэтому при строгом квазиупорядочении лучше говорить, что объекты *эквивалентны*, а не идентичны.

Код в листинге 16.7 иллюстрирует использование функций STL.

Листинг 16.7. vect3.cpp

```
// vect3.cpp -- использование функций STL
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
struct Review {
  std::string title;
  int rating;
};

bool operator<(const Review & r1, const Review & r2);
bool worseThan(const Review & r1, const Review & r2);
bool FillReview(Review & rr);
void ShowReview(const Review & rr);

int main()
{
  using namespace std;

  vector<Review> books;
  Review temp;
  while (FillReview(temp))
    books.push_back(temp);
  cout << "Спасибо. Вы ввели следующие "
       << books.size() << " рейтингов:\n"
       << "Рейтинг\tКнига\n";
  for_each(books.begin(), books.end(), ShowReview);
  sort(books.begin(), books.end());
  cout << "Отсортировано по названию:\nРейтинг\tКнига\n";
  for_each(books.begin(), books.end(), ShowReview);
  sort(books.begin(), books.end(), worseThan);
  cout << "Отсортировано по рейтингу:\nРейтинг\tКнига\n";
  for_each(books.begin(), books.end(), ShowReview);
  random_shuffle(books.begin(), books.end());
  cout << "После тасования:\nРейтинг\tКнига\n";
  for_each(books.begin(), books.end(), ShowReview);
  cout << "Всего наилучшего!\n";
  return 0;
}
```

```
bool operator<(const Review & r1, const Review & r2)
{
  if (r1.title < r2.title)
    return true;
  else if (r1.title == r2.title && r1.rating < r2.rating)
    return true;
  else
    return false;
}
bool worseThan(const Review & r1, const Review & r2)
{
  if (r1.rating < r2.rating)
    return true;
  else
    return false;
}
bool FillReview(Review & rr)
{
  std::cout << "Введите название книги (quit для завершения): ";
  std::getline(std::cin,rr.title);
  if (rr.title == "quit")
    return false;
  std::cout << "Введите рейтинг книги: ";
  std::cin >> rr.rating;
  if (!std::cin)
    return false;
  std::cin.get();
  return true;
}
void ShowReview(const Review & rr)
{
  std::cout << rr.rating << "\t" << rr.title << std::endl;
}
```

Замечание по совместимости

Ранние реализации C++ использовали `vector.h` вместо заголовочного файла `vector` и `algo.h` вместо заголовочного файла `algorithm`. Хотя порядок, в котором перечислены подключаемые библиотеки, не важен, g++ 2.7.1 требует, чтобы заголовочный файл `string` упоминался раньше заголовочных файлов STL. Ранние версии Microsoft Visual C++ 6.0 содержат ошибку в функции `getline()`, при которой вывод данных на экран не происходит до тех пор, пока не будет введено что-либо еще. Также ранние версии Microsoft Visual C++ требуют определения `operator==()` в дополнение к `operator<()`.

Borland C++Builder X требует добавления в глобальное пространство имен оператора

```
using std::rand;
```

Ниже показан пример работы программы из листинга 16.7:

```
Введите название книги (quit для завершения): The Cat Who Can Teach You
Weight Loss
Введите рейтинг книги: 8
Введите название книги (quit для завершения): The Dogs of Dharma
Введите рейтинг книги: 6
Введите название книги (quit для завершения): The Wimps of Wonk
Введите рейтинг книги: 3
```

```
Введите название книги (quit для завершения): Farewell and Delete
Введите рейтинг книги: 7
Введите название книги (quit для завершения): quit
Спасибо. Вы ввели следующие 4 рейтингов:
Рейтинг Книга
8       The Cat Who Can Teach You Weight Loss
6       The Dogs of Dharma
3       The Wimps of Wonk
7       Farewell and Delete
Отсортировано по названию:
Рейтинг Книга
7       Farewell and Delete
8       The Cat Who Can Teach You Weight Loss
6       The Dogs of Dharma
3       The Wimps of Wonk
Отсортировано по рейтингу:
Рейтинг Книга
3       The Wimps of Wonk
6       The Dogs of Dharma
7       Farewell and Delete
8       The Cat Who Can Teach You Weight Loss
После тасования:
Рейтинг Книга
7       Farewell and Delete
3       The Wimps of Wonk
6       The Dogs of Dharma
8       The Cat Who Can Teach You Weight Loss
Всего наилучшего!
```

Обобщенное программирование

Теперь, когда вы получили некоторый опыт использования STL, давайте взглянем на лежащую в его основе философию. STL — это пример *обобщенного программирования.* Объектно-ориентированное программирование сосредоточено на данных, в то время как обобщенное программирование — на алгоритмах. Общее у этих двух подходов — абстрагирование и создание повторно используемого кода, но философия несколько отличается.

Цель обобщенного программирования — написание кода, не зависящего от типов данных. Шаблоны — это инструмент C++ для создания обобщенных программ. Они позволяют вам определить функцию или класс в терминах обобщенного типа. STL идет еще дальше, представляя обобщенное представление алгоритмов. Шаблоны делают это возможным, но при условии тщательного и согласованного дизайна. Чтобы увидеть, как работает эта смесь шаблонов и дизайна, давайте рассмотрим, зачем нужны итераторы.

Зачем нужны итераторы?

Понимание итераторов — возможно, ключ к пониманию STL в целом. Так же, как шаблоны обеспечивают независимость алгоритмов от типа хранимых данных, итераторы обеспечивают независимость от типа используемых контейнеров. Таким образом, они являются существенным компонентом обобщенного подхода STL.

Чтобы увидеть, зачем нужны итераторы, давайте взглянем, как вы можете реализовать функцию find для двух различных представлений данных и как потом можно обобщить подход. Во-первых, рассмотрим функцию, которая осуществляет поиск некоторого значения в обычном массиве элементов типа double. Вы можете написать функцию следующим образом:

```
double *find_ar(double *ar, int n, const double &val)
{
   for (int i=0; i < n; i++)
      if (ar[i] == val)
         return &ar[i];
   return 0;
}
```

Если функция находит значение в массиве, она возвращает адрес в массиве, по которому находится указанное значение, в противном случае она возвращает нулевой указатель. Она использует индексную нотацию для перемещения по массиву. Вы можете применить шаблон, чтобы обобщить эту функцию для работы с массивами любого типа, к которым применима операцию ==. Однако этот алгоритм все еще применим только для определенной структуры данных — массива.

Поэтому давайте рассмотрим поиск в другой структуре — связном списке. (В главе 12 связный список используется для реализации класса Queue.) Список состоит из связанных элементов типа структуры Node:

```
struct Node
{
   double item;
   Node *p_next;
};
```

Предположим, что имеется указатель на первый элемент списка. Указатель p_next в каждом узле списка указывает на следующий элемент, а p_next последнего элемента установлен в 0.

Вы можете написать функцию find_ll() следующим образом:

```
Node *find_ll(Node *head, const double &val)
{
   Node *start;
   for (start = head; start != 0; start = start->next)
      if (start->item == val)
         return start;
   return 0;
}
```

Опять же, вы можете использовать шаблон, чтобы обобщить этот алгоритм для списков данных, поддерживающих операцию ==. Однако он останется применимым только к определенной структуре данных — связному списку.

Если вы посмотрите на детали реализации, то увидите, что эти две функции find реализуют разные алгоритмы: одна использует индексирование массивов для перемещения по списку значений, а вторая — переставляет start на start->p_next. Но, грубо говоря, эти алгоритмы делают одно и то же: сравнивают искомое значение последовательно со значением каждого элемента контейнера до тех пор, пока не найдется совпадение.

Целью обобщенного программирования в данном случае может быть получение единственной функции `find`, которая бы работала с массивами, или со связными списками, или с любым другим контейнерным типом. То есть, функция не только должна быть независимой от типа данных, хранимых в контейнере, но и от структур данных самого контейнера. Шаблоны представляют обобщенное представление типа данных, хранимых в контейнере. Что нам необходимо — так это обобщенное представление процесса перемещения по элементам контейнера. Итератор и является таким обобщенным представлением.

Какие свойства должен иметь итератор, чтобы реализовать функцию `find`? Вот краткий список:

- Вы должны иметь возможность разыменовывать итератор для того, чтобы получить доступ к значению, на которое он ссылается. То есть, если `p` — итератор, то должно быть определено `*p`.
- Вы должны иметь возможность присваивать один итератор другому. То есть, если `p` и `q` — итераторы, то должно быть определено выражение `p = q`.
- Вы должны иметь возможность сравнивать один итератор с другим. То есть, если `p` и `q` — итераторы, то должны быть определены выражения `p == q` и `p != q`.
- Вы должны иметь возможность перемещать итератор по элементам контейнера. Это может быть сделано определением `++p` и `p` для итератора `p`.

Есть и другие вещи, которые может делать итератор, но больше нет ничего такого, что он должен делать — по крайней мере, для обеспечения функции `find`. В действительности STL определяет несколько уровней итераторов с возрастающими возможностями, и мы вернемся к этой теме позднее. Кстати, отметим, что обычный указатель соответствует требованиям, предъявляемым к итераторам. А потому вы можете переписать `find_arr()` следующим образом:

```
typedef double * iterator;
iterator find_ar(iterator ar, int n, const double & val)
{
   for (int i = 0; i < n; i++, ar++)
      if (*ar == val)
         return ar;
   return 0;
}
```

Затем вы можете изменить список параметров функции так, чтобы она принимала указатель на начало массива и указатель на элемент, следующий за концом массива, в качестве аргументов, задающих диапазон. (В листинге 7.8 в главе 7 делается нечто подобное.) Функция может возвращать конечный указатель в качестве признака того, что значение не найдено. Следующая версия `find_arr()` включает эти изменения:

```
typedef double * iterator;
iterator find_ar(iterator begin, iterator end, const double & val)
{
   iterator ar;
   for (ar = begin; ar != end; ar++)
      if (*ar == val)
         return ar;
   return end; // признак того, что значение не найдено
}
```

Для функции find_ll() вы можете определить класс итератора, в котором определены операции * и ++:

```
struct Node
{
   double item;
   Node * p_next;
};
class iterator
{
   Node * pt;
public:
   iterator() : pt(0) {}
   iterator (Node * pn) : pt(pn) {}
   double operator*() { return pt->item;}
   iterator& operator++() // для ++it
   {
      pt = pt->p_next;
      return *this;
   }
   iterator operator++(int) // для it++
   {
      iterator tmp = *this;
      pt = pt->p_next;
      return tmp;
   }
   // ... operator==(), operator!=() и тому подобное
};
```

(Чтобы различать префиксную и постфиксную версии операции ++, в C++ предусмотрено соглашение о том, что operator++() — это префиксная форма, а operator++(int) — постфиксная; аргумент никогда не используется, поэтому не нуждается в именовании.)

Главный момент здесь — не то, как в подробностях реализован класс iterator, а то, что с его помощью функция find может быть переписана следующим образом:

```
iterator find_ll(iterator head, const double & val)
{
   iterator start;
   for (start = head; start!= 0; ++start)
      if (*start == val)
         return start;
   return 0;
}
```

Это очень похоже на find_ar(). Разница только в том, как обе функции определяют достижение конца списка значений при поиске. Функция find_ar() использует итератор, указывающий на псевдозначение, расположенное за последним элементом массива, в то время как find_ll() применяет нулевое значение, сохраненное в последнем узле связного списка. Если исключить эту разницу, то можно сделать эти функции идентичными. Например, вы можете потребовать, чтобы связный список имел один дополнительный элемент за последним реальным элементом. То есть, вы можете иметь и в массиве и в связном списке элемент, находящийся "за концом", и вы можете использовать в качестве признака завершения поиска достижение итера-

тором этого элемента. Тогда и find_arr(), и find_ll() будут одинаково определять конец данных, и применять идентичные алгоритмы поиска. Отметим, что требование дополнительного элемента, находящегося "за концом", вытекает из требований к итераторам, которые, в свою очередь, предъявляют требования к контейнерным классам.

STL следует описанному выше подходу. Во-первых, каждый контейнерный класс (vector, list, deque и так далее) определяет соответствующий тип итератора. Для одного класса итератор может быть указателем, для другого — объектом. Независимо от реализации, каждый итератор представляет необходимые операции — такие, как * и ++. (Некоторые классы нуждаются в большем количестве операций, чем другие.) Далее, каждый контейнерный класс имеет маркер, находящийся "за концом", который представляет собой значение, присваиваемое итератору, когда он выходит за последнее значение контейнера. Каждый контейнерный класс имеет методы begin() и end(), которые возвращают итераторы, указывающие, соответственно, на первый элемент и на элемент, находящийся за последним. И каждый контейнерный класс будет иметь операцию ++, перемещающую итератор от первого элемента до элемента, находящегося за последним, посещая по пути каждый из элементов контейнера.

Чтобы использовать контейнерный класс, вам не нужно знать, ни как реализованы итераторы, ни как реализован элемент, находящийся "за концом". Достаточно знать, что у него есть итераторы, что begin() возвращает итератор, указывающий на первый элемент, а end() — возвращает итератор, указывающий на элемент, находящийся за последним. Например, предположим, что вы хотите напечатать значения из объекта vector<double>. В этом случае вы можете использовать следующий код:

```
vector<double>::iterator pr;
for (pr = scores.begin(); pr != scores.end(); pr++)
   cout << *pr << endl;
```

Здесь строка

```
vector<double>::iterator pr;
```

идентифицирует тип итератора, определенный для класса vector<double>. Если вы используете вместо этого шаблон класса list<double> для хранения счетов, то можете использовать следующий код:

```
list<double>::iterator pr;
for (pr = scores.begin(); pr != scores.end(); pr++)
   cout << *pr << endl;
```

Единственное отличие — в объявленном типе pr. То есть, определяя для каждого класса соответствующие итераторы и проектируя классы в унифицированной манере, STL позволяет вам писать один и тот же код для контейнеров, которые имеют совершенно разное внутреннее представление.

На самом деле из соображений стиля лучше избегать непосредственного применения итераторов; вместо этого, если возможно, вы должны использовать функции STL вроде for_each(), которые позаботятся о деталях за вас.

Таким образом, чтобы прочувствовать подход STL, вы начинаете с алгоритма обработки контейнера. Выражаете его настолько обобщенным образом, насколько возможно, обеспечивая независимость от типа данных и типа контейнера. Для обеспечения работы алгоритма со специфическими случаями, определяете итераторы, отвечающие нуждам алгоритма, и закладываете требования в дизайн контейнеров.

То есть, базовые свойства итераторов и свойства контейнеров вытекают из требований, заложенных в алгоритм.

Виды итераторов

Разные алгоритмы предъявляют разные требования к итераторам. Например, алгоритм find требует, чтобы была определена операция ++, дабы итератор мог пошагово проходить весь контейнер. Ему нужен доступ к контейнеру для чтения, но не нужен доступ для записи. (Он просто просматривает данные, но не изменяет их.) Обычный алгоритм sort, с другой стороны, требует произвольного доступа, чтобы иметь возможность обменивать значениями два несоседних элемента. Если iter — итератор, то вы можете получить произвольный доступ, определив операцию +, чтобы можно было написать выражение вроде iter + 10. К тому же алгоритм sort должен иметь возможность как читать, так и писать данные.

STL определяет пять видов итераторов и описывает алгоритмы в терминах итераторов, которые им необходимы. Эти пять видов итераторов следующие: входной итератор, выходной итератор, однонаправленный итератор, двунаправленный итератор и итератор произвольного доступа. Например, прототип find() выглядит так:

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

Это говорит о том, что алгоритм требует входного итератора. Аналогично, следующий прототип:

```
template<class RandomAccessIterator, class T>
RandomAccessIterator find(RandomAccessIterator first,
RandomAccessIterator last,const T& value);
```

Все пять видов итераторов могут быть разыменованы (то есть, для них определена операция *), и могут быть сравнены на эквивалентность (используя операцию ==, возможно, перегруженную) и неэквивалентность (используя операцию !=, возможно, перегруженную). Если два сравниваемых итератора эквивалентны, то разыменование одного должно порождать то же значение, что и разыменование другого. То есть, если истинно выражение

```
iter1 == iter2
```

то также истинно следующее выражение:

```
*iter1 == *iter2
```

Конечно, эти свойства остаются истинными и для встроенных операций и указателей, поэтому эти требования могут служить руководством для вас в том, что нужно сделать для перегрузки этих операций в классах итераторов. Теперь давайте взглянем на другие свойства итераторов.

Входные итераторы

Термин *входной* (input) используется с точки зрения программы. То есть, информация, поступающая из контейнера в программу, рассматривается как входящая — как и информация, поступающая в программу от клавиатуры. Поэтому *входной итератор* — это тот, который программа может использовать для чтения значений из контейнера. В частности, разыменование входного итератора должно позволить программе

читать значение из контейнера, но не позволять ей изменять это значение. Поэтому алгоритмы, которые требуют входных итераторов, являются алгоритмами, не модифицирующими значений, хранящихся в контейнере.

Входной итератор должен обеспечивать вам доступ к значениям контейнера. Он делает это за счет поддержки операции ++, как в префиксной, так и в постфиксной форме. Если вы устанавливаете входной итератор на первый элемент контейнера и затем увеличиваете его до тех пор, пока он не окажется в положении за последним элементом, то таким образом переберете все элементы — по одному за шаг. Кстати, нет гарантий, что второй проход по элементам контейнера с помощью входного итератора даст ту же последовательность элементов. Также не существует гарантий, что после того, как входной итератор увеличивается на единицу, то предыдущее его значение может быть разыменовано. Любые алгоритмы, базирующиеся на входном итераторе, должны быть однопроходными, и не полагаться на значения итератора из предыдущего прохода либо на старые значения итератора из того же прохода.

Отметим, что входной итератор — однонаправленный, он может увеличиваться, перемещаясь вперед по контейнеру, но не может возвращаться назад.

Выходные итераторы

Термин *выходной* (output) в STL означает, что итератор используется для передачи значений из программы в контейнер. (То есть, вывод программы является вводом для контейнера.) Выходной итератор похож на входной, за исключением того, что разыменование его гарантированно дает программе возможность изменять значение контейнера, но не читать его. Если возможность писать без возможности чтения кажется вам странной, вспомните, что то же касается вывода на дисплей: `cout` может модифицировать поток символов, отправленный на дисплей, но не может с дисплея читать. STL обеспечивает достаточно общий инструментарий, чтобы его контейнеры представляли и выходные устройства, поэтому вы можете столкнуться с этим при работе с обычными контейнерами. К тому же, если алгоритм модифицирует содержимое контейнера (например, генерируя новые значения, которые должны быть сохранены) без чтения этого содержимого, нет причин требовать, чтобы он это содержимое читал.

Короче говоря, вы можете использовать входной итератор для одиночного прохода с доступом только для чтения, а выходной итератор — для одного прохода с доступом только для записи.

Однонаправленные итераторы

Как входные и выходные итераторы, *однонаправленные* (forward) итераторы используют только операцию ++ для навигации по контейнеру. Однонаправленные итераторы, таким образом, могут перемещаться только вперед, на один элемент за раз. Однако в отличие от входных и выходных итераторов, последовательность значений, которые проходит однонаправленный итератор, при каждом использовании одна и та же. Также, после того, как вы увеличиваете однонаправленный итератор операцией инкремента, вы можете разыменовать предыдущее значение итератора, если вы его сохранили, и получить то же самое значение. Это позволяет реализовать многопроходные алгоритмы.

Однонаправленный итератор может позволить вам и читать, и модифицировать данные, либо только читать их:

```
int *pirw;          // итератор чтения-записи
const int *pir;     // итератор только для чтения
```

Двунаправленные итераторы

Предположим, что имеется алгоритм, которому нужно проходить контейнер в обоих направлениях. Например, обратная функция может обменивать значения первого и последнего элемента, инкрементировать указатель на первый элемент, декрементировать указатель на второй элемент, и повторять этот процесс. *Двунаправленный* (bidirectional) итератор обладает всеми свойствами однонаправленного итератора и добавляет к ним поддержку двух операций декремента (префиксной и постфиксной).

Итераторы произвольного доступа

Некоторые алгоритмы, такие как стандартная сортировка и бинарный поиск, требуют возможности перескакивать непосредственно на произвольный элемент контейнера. Это называется *произвольным доступом* и требует итератора *произвольного доступа* (random access). Итератор этого типа обладает всеми свойствами двунаправленного итератора, плюс к тому добавляет операции (вроде сложения указателей с целым), которые поддерживают произвольный доступ и операции отношения для обращения к элементам. В табл. 16.3 перечислены операции итераторов произвольного доступа, которые добавлены к нему сверх имеющихся у двунаправленных итераторов. В этой таблице `a` и `b` — это значения итератора, `n` — целое, а `r` — переменная произвольного итератора или ссылка.

Таблица 16.3. Операции итераторов произвольного доступа

Выражение	Описание
`a + n`	Указывает на `n`-й элемент после того, на который указывает `a`.
`n + a`	То же самое, что `a + n`.
`a - n`	Указывает на `n`-й элемент перед тем, на который указывает `a`.
`r += n`	Эквивалентно `r = r + n`.
`r -= n`	Эквивалентно `r = r - n`.
`a[n]`	Эквивалентно `*(a + n)`.
`b - a`	Такое значение `n`, что `b = a + n`.
`a < b`	Истинно, если `b - a > 0`.
`a > b`	Истинно, если `b < a`.
`a >= b`	Истинно, если `a` не меньше `b`.
`a <= b`	Истинно, если `a` не больше `b`.

Выражения вроде `a + n` корректны, только если и `a`, и `a + n` лежат в диапазоне контейнера (включая и элемент, находящийся "за концом").

Иерархия итераторов

Возможно, вы уже заметили, что виды итераторов образуют иерархию. Однонаправленный итератор имеет все свойства входного и выходного итераторов, плюс

свои собственные возможности. Двунаправленный итератор обладает всеми свойствами однонаправленного, плюс своими собственными возможностями. Итератор произвольного доступа имеет все свойства однонаправленного итератора, плюс свои собственные возможности. В табл. 16.3 суммируются основные свойства итераторов. В этой таблице `i` — это итератор, а `n` — целое.

Таблица 16.4. Возможности итераторов

Свойство итератора	Входной	Выходной	Однонаправленный	Двунаправленный	Произвольного доступа
Разыменующее чтение	Да	Нет	Да	Да	Да
Разыменующая запись	Нет	Да	Да	Да	Да
Фиксированный и повторяющийся порядок	Нет	Нет	Да	Да	Да
`++i, i++`	Да	Да	Да	Да	Да
`--i, i--`	Нет	Нет	Нет	Да	Да
`i[n]`	Нет	Нет	Нет	Нет	Да
`i + n`	Нет	Нет	Нет	Нет	Да
`i - n`	Нет	Нет	Нет	Нет	Да
`i += n`	Нет	Нет	Нет	Нет	Да
`i -= n`	Нет	Нет	Нет	Нет	Да

Алгоритмы, написанные в терминах определенного вида итераторов, могут использовать этот итератор или любой другой, обладающий нужными свойствами. Поэтому контейнер, скажем, с итератором произвольного доступа может использовать алгоритм, написанный для входного итератора.

Зачем нужны все эти разные виды итераторов? Идея состоит в том, чтобы написать алгоритм, использующий итератор с минимально возможными требованиями, что позволит ему быть использованным с максимальным числом контейнеров. То есть, функция `find()`, используя входной итератор начального уровня, может быть использована с любым контейнером, который содержит читаемые значения. Однако функция `sort()`, которая требует итераторов произвольного доступа, может применяться только с контейнерами, поддерживающими этот вид итераторов.

Отметим, что различные виды итераторов не определяют типов. Скорее, это концептуальные характеристики. Как упоминалось ранее, каждый контейнерный класс определяет в контексте класса `typedef` по имени `iterator`. Поэтому класс `vector<int>` имеет итератор типа `vector<int>::iterator`. Но документация для класса говорит вам, что итераторы вектора — это итераторы произвольного доступа. Это, в свою очередь, позволяет вам использовать алгоритмы, базирующиеся на итераторах любого типа, потому что итератор произвольного доступа обладает свойствами всех итераторов. Аналогично класс `list<int>` имеет итераторы типа `list<int>::iterator`. STL реализует двунаправленные связные списки, поэтому он использует двунаправленный итератор. Поэтому он не может применять алгоритмы на базе итераторов произвольного доступа, но может использовать алгоритмы на базе менее требовательных итераторов.

Концепции, уточнения и модели

STL имеет некоторые средства, такие как виды итераторов, невыразимые на языке C++. То есть, хотя вы можете спроектировать, скажем, класс, имеющий свойства однонаправленного итератора, вы не можете наложить ограничение компилятора на алгоритм, использующий только этот класс. Причина в том, что однонаправленный итератор — это набор требований, а не тип. Требования могут быть удовлетворены классом итератора, который вы проектируете, но они также могут быть удовлетворены обычным указателем. Алгоритм STL работает с любой реализацией итераторов, которая отвечает требованиям. Литература по STL использует слово *концепция* для описания набора требований. То есть, существует концепция входного итератора, концепция однонаправленного итератора и так далее. Кстати, если вам нужны итераторы, скажем, для контейнерного класса, который вы разрабатываете, вы можете посмотреть на STL, который включает шаблоны итераторов всех стандартных видов.

Концепции могут иметь между собой отношения, подобные отношениям наследования. Например, двунаправленный итератор наследует свойства однонаправленного итератора. Однако вы не можете применить механизм наследования C++ к итераторам. Например, вы можете реализовать однонаправленный итератор как класс, а двунаправленный — как обычный указатель. Поэтому в терминах языка C++ этот конкретный двунаправленный итератор, будучи встроенного типа, не может наследоваться от класса. Концептуально, однако, он его наследует. Некоторая литература по STL использует термин *усовершенствование* (refinement) для обозначения такого концептуального наследования. То есть, двунаправленный итератор — усовершенствование концепции однонаправленного итератора.

Конкретная реализация концепции называется *моделью*. То есть, обычный указатель на целое — это модель концепции итератора произвольного доступа. Это также модель однонаправленного итератора, потому что она удовлетворяет всем требованиям этой концепции.

Указатель как итератор

Итераторы — это обобщения указателей, и указатели отвечают всем требованиям, предъявляемым к итераторам. Итераторы формируют интерфейс алгоритмов STL, и указатели являются итераторами, поэтому алгоритмы STL могут применять указатели для операций с не-STL-контейнерами, которые базируются на указателях. Например, вы можете применять алгоритмы STL к массивам. Предположим, Receipts — массив значений типа double, и вы хотите отсортировать его в порядке возрастания:

```
const int SIZE = 100;
double Receipts[SIZE];
```

Вспомним, что функция STL по имени sort() принимает в качестве аргументов итераторы, указывающие на первый элемент контейнера, и итератор, указывающий на элемент, следующий за последним. Хорошо, &Receipts[0] (или просто Receipts) — это адрес первого элемента, а &Receipts[SIZE] (или просто Receipts + SIZE) — это адрес элемента, следующего за последним элементом массива. То есть вызов функции

```
sort(Receipts, Receipts + SIZE)
```

отсортирует массив. C++ гарантирует, что выражение Receipts + n определено до тех пор, пока результат лежит в пределах массива или на один элемент за его концом. То есть C++ поддерживает концепцию "за концом" для указателей на массив, и это дает возможность применять алгоритмы STL к обычным массивам. То есть, тот факт, что указатели являются итераторами, и этот алгоритм основан на итераторах, позволяет применять алгоритмы STL к обычным массивам. Аналогично, вы можете применять алгоритмы STL к спроектированным вами формам данных, если вы предусмотрите соответствующие итераторы (которые могут быть указателями на объекты), а также индикаторы положения "за концом".

copy(), ostream_iterator и istream_iterator

STL предлагает ряд предопределенных итераторов. Чтобы понять — почему, давайте проясним некоторые фундаментальные понятия. Существует алгоритм по имени copy() для копирования данных из одного контейнера в другой. Этот алгоритм выражен в терминах итераторов, поэтому он может копировать из одного вида контейнеров в другой, и даже из или в массив, потому что вы можете использовать указатели массива в качестве итераторов. Например, следующий фрагмент копирует массив в вектор:

```
int casts[10] = {6, 7, 2, 9, 4, 11, 8, 7, 10, 5};
vector<int> dice[10];
copy(casts, casts + 10, dice.begin()); // копировать массив в вектор
```

Первые два итератора-аргумента copy() представляют диапазон, который следует скопировать, а конечный аргумент-итератор — местоположение, с которого необходимо начинать копировать. Первые два аргумента должны быть входными итераторами (или более мощными), а заключительный аргумент — выходным итератором (или более мощным). Функция copy() переписывает существующие данные в контейнере назначения, и этот контейнер должен быть достаточно велик, чтобы вместить копируемые элементы. Поэтому вы не можете использовать copy() для помещения данных в пустой вектор — по крайней мере, без пересортировки — трюка, описанного далее в настоящей главе.

Теперь предположим, что вы хотите копировать информацию на дисплей. Вы можете применить copy(), если существует итератор, представляющий выходной поток. STL предусматривает такой итератор в виде шаблона ostream_iterator. Применяя терминологию STL, этот шаблон является *моделью* концепции выходного итератора. Он также является примером *адаптера* — класса или функции, которая преобразует некоторый другой интерфейс в интерфейс, используемый STL. Вы можете создать итератор этого вида, включив заголовочный файл iterator (бывший iterator.h) и указав объявление:

```
#include <iterator>
...
ostream_iterator<int, char> out_iter(cout, " ");
```

Итератор out_iter теперь становится интерфейсом, который позволяет вам использовать cout для отображения информации. Первый аргумент шаблона (в данном случае int) обозначает тип данных, отправляемый в выходной поток. Второй аргумент шаблона (в данном случае char) обозначает символьный тип, используемый выходным потоком (другим допустимым значением мог быть wchar_t). Первый аргу-

мент конструктора (в данном случае cout) идентифицирует используемый выходной поток. Он также может быть потоком файлового вывода. Последний строковый аргумент — это разделитель, который должен отображаться после каждого элемента, отправленного в выходной поток.

Внимание!

Старые реализации C++ используют только первый шаблонный аргумент для ostream_iterator:

```
ostream_iterator<int> out_iter(cout, " "); // старая реализация
```

Вы можете использовать итераторы следующим образом:

```
*out_iter++ = 15; // работает подобно cout << 15 << " ";
```

Для обычного указателя это означает присваивание значения 15 переменной, находящейся по адресу out_iter, с последующим приращением этого указателя. Однако для ostream_iterator это предложение означает отправить значение 15 и строку, состоящую из пробела, в выходной поток, управляемый cout. Затем он должен подготовиться к следующей операции вывода. Вы можете использовать итератор с copy() следующим образом:

```
copy(dice.begin(), dice.end(), out_iter); //копировать вектор в выходной поток
```

Это должно означать копирование полного содержимого контейнера dice в выходной поток — то есть, отобразить содержимое контейнера.

Или же вы можете пропустить создание именованного итератора и вместо него сконструировать неименованный итератор. То есть, вы можете использовать адаптер вроде следующего:

```
copy(dice.begin(), dice.end(), ostream_iterator<int, char>(cout, " ") );
```

Аналогично, заголовочный файл iterator определяет шаблон istream_iterator для адаптации ввода istream к интерфейсу входного итератора. Это — модель концепции входного итератора. Вы можете использовать два объекта istream_iterator для определения входного диапазона для copy():

```
copy(istream_iterator<int, char>(cin),
   istream_iterator<int, char>(), dice.begin());
```

Подобно ostream_iterator, istream_iterator использует два шаблонных аргумента. Первый обозначает тип данных, подлежащих чтению, а второй — тип символьных данных, используемых в выходном потоке. Применение аргумента конструктора cin означает указание на использование входного потока, управляемого cin. Опускание аргумента конструктора означает ошибку ввода, поэтому приведенный выше код означает чтение из входного потока до метки конца файла, до несоответствия типа либо до какой-то другой ошибки ввода.

Другие полезные итераторы

Заголовочный файл iterator предлагает и ряд других предопределенных итераторов специального назначения в дополнение к ostream_iterator и istream_iterator. Среди них: reverse_iterator, back_insert_iterator, front_insert_iterator и insert_iterator.

Начнем с того, что посмотрим, что делает обратный итератор (reverse_iterator). По сути дела, инкремент этого итератора вызывает его декремент. Почему бы про-

сто не применить декремент обычного итератора? Главная причина — упрощение использования существующих функций. Предположим, что требуется отобразить на дисплее содержимое контейнера dice. Как вы только что видели, для этого можно использовать copy() и ostream_iterator для копирования содержимого в выходной поток:

```
ostream_iterator<int, char> out_iter(cout, " ");
copy(dice.begin(), dice.end(), out_iter); // отобразить в прямом порядке
```

Теперь предположим, что вы хотите печатать элементы в обратном порядке (Возможно, вы формируете обратную съемку.) Существует несколько подходов, которые не работают, и вместо того, чтобы в них барахтаться, давайте обратимся к тому, который работает. Класс vector имеет функцию-член по имени rbegin(), которая возвращает обратный итератор, указывающий на элемент, находящийся за последним, и функцию-член rend(), которая возвращает обратный итератор, указывающий на первый элемент. Поскольку приращение обратного итератора осуществляет его декремент, вы можете применить следующий оператор:

```
copy(dice.rbegin(), dice.rend(), out_iter); //отобразить в обратном порядке
```

чтобы отобразить содержимое в обратном порядке. Вам даже не потребуется объявлять обратный итератор.

На память!

Как rbegin(), так и end() возвращают одно и то же значение (находящееся за последним элементом), но другого типа (reverse_iterator против iterator). Аналогично, rend() и begin() возвращают одно и то же значение (итератор, указывающий на первый элемент), но другого типа.

Обратные указатели должны иметь специальную “компенсацию”. Предположим, что rp — обратный указатель, инициализированный dice.rbegin(). Чем должен быть *rp? Поскольку rbegin() возвращает элемент, находящийся за последним в контейнере, вы не должны разыменовывать этот адрес. Аналогично, если rend() — действительное местоположение первого элемента, copy() останавливается за один элемент до первого элемента контейнера, потому что диапазон по определению не включает в себя последний элемент. Обратные указатели решают обе проблемы за счет того, что выполняют декремент перед разыменованием. То есть, *rp разыменовывает значение итератора, непосредственно предшествующее текущему значению *rp. Если rp указывает на шестую позицию в контейнере, то *rp — значение из пятой позиции и так далее. В листинге 16.8 иллюстрируется применение copy() с итератором istream и обратным итератором.

Листинг 16.8. copyit.cpp

```
// copyit.cpp -- copy() и итераторы
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
   using namespace std;
   int casts[10] = {6, 7, 2, 9 ,4 , 11, 8, 7, 10, 5};
   vector<int> dice(10);
```

```
    // копировать из массива в вектор
    copy(casts, casts + 10, dice.begin());
    cout << "Выбросим dice на дисплей!\n";
    // создание итератора ostream
    ostream_iterator<int, char> out_iter(cout, " ");
    // копировать вектор в выходной поток
    copy(dice.begin(), dice.end(), out_iter);
    cout << endl;
    cout <<"Неявное применение обратного итератора.\n";
    copy(dice.rbegin(), dice.rend(), out_iter);
    cout << endl;
    cout <<"Явное применение обратного итератора.\n";
    vector<int>::reverse_iterator ri;
    for (ri = dice.rbegin(); ri != dice.rend(); ++ri)
      cout << *ri << ' ';
    cout << endl;
    return 0;
}
```

Замечание по совместимости

Старые реализации C++ могут использовать заголовочные файлы `iterator.h` и `vector.h` вместо `iterator` и `vector`. Также некоторые реализации используют `ostream_iterator<int>` вместо `ostream_iterator<int, char>`.

Вывод программы из листинга 16.8 выглядит следующим образом:

```
Выбросим dice на дисплей!
6 7 2 9 4 11 8 7 10 5
Неявное применение обратного итератора.
5 10 7 8 11 4 9 2 7 6
Явное применение обратного итератора.
5 10 7 8 11 4 9 2 7 6
```

Если перед вами стоит выбор явного объявления итераторов либо использования функций STL для выполнения всей работы внутри, например, передавая значение, возвращаемое `rbegin()`, функции, вам стоит предпочесть второй вариант. В этом случае вам придется делать меньше работы и будет меньше шансов сделать ошибку.

Другие три итератора (`back_insert_iterator`, `front_insert_iterator` и `insert_iterator`) также повышают степень обобщенности алгоритмов STL. Многие функции STL подобны `copy()` в том, что передают свои результаты по адресу, указанному через выходной итератор. Вспомните, что

```
copy(casts, casts + 10, dice.begin());
```

копирует значения по адресу, начинающемуся с `dice.begin()`. Эти значения перезаписывают предыдущее содержимое `dice`, и функция предполагает, что `dice` имеет достаточно места, чтобы вместить все значения. То есть `copy()` не выполняет автоматического изменения размера контейнера назначения, дабы вместить переданную в него информацию. Программа в листинге 16.8 заботится об этой ситуации, объявляя `dice` размером в 10 элементов, но предположим, что вы заранее не знаете, какого размера должен быть `dice`. Или предположим, что вы хотите добавлять элементы в `dice` вместо перезаписи старых значений.

Три итератора вставки решают эти проблемы, преобразуя процесс копирования в процесс вставки. Вставка добавляет новые элементы без перезаписи существующих данных, и она использует автоматическое выделение памяти для обеспечения размещения новой информации. Итератор back_insert_iterator вставляет элементы в конец контейнера, а front_insert_iterator вставляет их в начало. И, наконец, insert_iterator вставляет элементы, начиная с позиции указанной в аргументе, переданном его конструктору. Все эти три итератора являются моделью концепции выходного контейнера.

Однако существуют некоторые ограничения. Так, например, back_insert_iterator может быть использован только с контейнерными типами, которые допускают быструю вставку в конец. (*Быстрая* — означает применение алгоритмов с постоянным временем доступа; в разделе "Концепции контейнеров" далее в главе концепция постоянного времени обсуждается более подробно.) Класс vector позволяет это делать. Итератор front_insert_iterator может быть использован только с контейнерными типами, допускающими вставку в начало за постоянное время. Класс vector не позволяет это делать, а класс queue — позволяет. Итератор insert_iterator не имеет таких ограничений. То есть вы можете применять его для вставки данных в начало вектора. Однако front_insert_iterator делает это быстрее с теми контейнерными типами, которые поддерживают его применение.

Совет

Вы можете использовать insert_iterator для преобразования алгоритма, который копирует данные, в алгоритм, вставляющий их.

Эти итераторы принимают тип контейнера в качестве параметра шаблона и действительный идентификатор контейнера — в качестве аргумента конструктора. То есть, чтобы создать back_insert_iterator для контейнера vector<int> по имени dice, вы делаете вот что:

```
back_insert_iterator<vector<int> > back_iter(dice);
```

Причина, по которой вы должны объявлять тип контейнера, состоит в том, что итератор должен использовать соответствующий метод контейнера. Код конструктора back_insert_iterator предполагает, что метод push_back() определен в переданном ему типе. Функция copy(), будучи автономной (не методом), не имеет права доступа для изменения размера контейнера. Но приведенное выше объявление позволяет back_iter использовать метод vector<int>::push_back(), который имеет соответствующие права доступа.

Объявление front_insert_iterator имеет ту же форму. Объявление insert_iterator предусматривает дополнительный аргумент конструктора для указания позиции вставки:

```
insert_iterator<vector<int> > insert_iter(dice, dice.begin() );
```

Код в листинге 16.9 иллюстрирует применение этих двух итераторов.

Листинг 16.9. inserts.cpp

```
// inserts.cpp -- copy() и итераторы вставки
#include <iostream>
#include <string>
#include <iterator>
#include <vector>
```

```
int main()
{
   using namespace std;
   string s1[4] = {"fine", "fish", "fashion", "fate"};
   string s2[2] = {"busy", "bats"};
   string s3[2] = {"silly", "singers"};
   vector<string> words(4);
   copy(s1, s1 + 4, words.begin());
   ostream_iterator<string, char> out(cout, " ");
   copy (words.begin(), words.end(), out);
   cout << endl;
   // сконструировать анонимный объект типа back_insert_iterator
   copy(s2, s2 + 2, back_insert_iterator<vector<string> >(words));
   copy (words.begin(), words.end(), out);
   cout << endl;
   // сконструировать анонимный объект типа insert_iterator
   copy(s3, s3 + 2, insert_iterator<vector<string> >(words, words.begin()));
   copy (words.begin(), words.end(), out);
   cout << endl;
   return 0;
}
```

Замечание по совместимости

Старые компиляторы могут использовать файлы заголовков `list.h` и `iterator.h`. Кроме того, некоторые компиляторы могут применять `ostream_iterator<int>` вместо `ostream_iterator<int, char>`.

Ниже показан вывод программы из листинга 16.9:

```
fine fish fashion fate
fine fish fashion fate busy bats
silly singers fine fish fashion fate busy bats
```

Первый вызов `copy()` копирует четыре строки из `s1` в `words`. Это работает, потому что `words` объявлен как контейнер емкостью в четыре строки, что как раз равно количеству копируемых строк. Затем `back_insert_iterator` вставляет строки из `s2` в место, находящееся перед концом массива `words`, увеличивая размер `words` до шести элементов. И, наконец, `insert_iterator` вставляет две строки из `s3` перед первым элементом `words`, увеличивая его размер до восьми элементов. Если программа попытается скопировать `s2` и `s3` в `words`, используя `words.end()` и `words.begin()` в качестве итераторов, то в `words` не окажется места для этих новых данных, и тогда программа, вероятно, прервется из-за нарушения доступа к памяти.

Если вы чувствуете себя растерянными из-за обилия вариаций итераторов, имейте в виду, что только практическое применение сделает их знакомыми для вас. Также помните, что эти предопределенные итераторы расширяют степень обобщенности алгоритмов STL. То есть, `copy()` не только может копировать информацию из одного контейнера в другой, но и копировать информацию из контейнера в выходной поток или из входного потока в контейнер. Вы также можете применять `copy()` для вставки данных в другой контейнер. Таким образом, единственная функция может выполнять работу многих. И поскольку `copy()` — одна из нескольких функций STL, которые используют выходной итератор, предопределенные итераторы также умножают возможности всех этих функций.

Виды контейнеров

STL включает в себя как концепции контейнеров, так и типы контейнеров. Концепции — это общие категории с названиями вроде "контейнер", "последовательный контейнер" и "ассоциативный контейнер". Типы контейнеров — это шаблоны, которые вы можете использовать для создания специфических объектов-контейнеров. Всего определено 11 контейнерных типов: deque, list, queue, priority_queue, stack, vector, map, multimap, set, multiset и bitset. (В данной главе не рассматривается bitset, являющийся контейнером для работы с данными на уровне битов.) Поскольку концепции категоризируют типы, с них и начнем.

Концепции контейнеров

С базовой концепцией контейнера не ассоциируется никакой тип, однако концепция описывает элементы, общие для всех контейнерных классов. Это разновидность концептуального абстрактного класса — концептуального потому, что классы контейнеров на самом деле не используют механизм наследования. Или, если посмотреть на это иначе, концепция контейнера устанавливает набор требований, которым должны удовлетворять все классы контейнеров STL.

Контейнер — это объект, который хранит в себе другие объекты одного и того же типа. Хранимые объекты могут быть объектами в смысле объектно-ориентированного программирования либо они могут быть значениями встроенных типов. Данные, сохраненные в контейнере, находятся в его владении. Это означает, что когда завершается жизнь контейнера, то же происходит с его данными. (Однако если данные являются указателями, то, на что они указывают, не обязательно должно исчезать.)

Вы не можете сохранять в контейнере данные любого вида. В частности, тип хранимых объектов должен допускать присваивание и конструирование копированием. Базовые типы удовлетворяют этим требованиям, как и типы классов — если только определение класса не объявляет конструктор копирования и/или операцию присваивания приватными или защищенными.

Базовый контейнер не гарантирует, что его элементы будут сохранены в каком-то определенном порядке, или же что этот порядок не изменится, но уточнение концепции может добавить такие гарантии. Все контейнеры предоставляют определенные средства и операции. В табл. 16.5 суммируются некоторые из этих общих средств. В этой таблице X представляет тип контейнера, такой как vector, T является типом объекта, сохраняемого в контейнере, a и b — значениями типа X, а u — идентификатором типа X.

Столбец "Сложность" в табл. 16.5 описывает время, необходимое для выполнения операции. В таблице встречаются три возможных значения — от самой быстрой операции до самой медленной:

- Время компиляции
- Константная
- Линейная

Если в столбце "Сложность" указано "Время компиляции", это означает, что действие осуществляется во время компиляции и не требует времени при выполнении. "Константная" сложность означает, что операция происходит во время выполнения, но не зависит от количества элементов в объекте.

Таблица 16.5. Некоторые базовые свойства итераторов

Выражение	Возвращаемый тип	Описание	Сложность
X::iterator	Тип итератора, указывающего на T	Итератор любой категории, за исключением выходного итератора	Время компиляции
X::value_type	T	Тип для T	Время компиляции
X u;		Создает контейнер нулевого размера по имени u	Константная
X();		Создает безымянный контейнер нулевого размера	Константная
X u(a);		Конструктор копирования	Линейная
X u = a;		Тот же эффект, что и X u(a);	
(&a)->~X();	void	Применяет деструктор к каждому элементу контейнера	Линейная
a.begin()	iterator	Возвращает итератор, указывающий на первый элемент контейнера	Константная
a.end()	iterator	Возвращает итератор, указывающий на значение, находящееся сразу за последним элементом контейнера	Константная
a.size()	Беззнаковый целый тип	Возвращает количество элементов, эквивалентное a.end() - a.begin()	Константная
a.swap(b)	void	Обменивает значения a и b	Константная
a == b	Преобразуемый в bool	Возвращает true, если a и b имеют одинаковый размер и каждый элемент a эквивалентен соответствующему элементу b (то есть == дает истину)	Линейная
a != b	Преобразуемый в bool	Возвращает !(a == b)	Линейная

"Линейная" сложность означает, что время операции пропорционально количеству элементов. То есть, если a и b являются контейнерами, то сравнение a == b имеет линейную сложность, потому что операция == должна быть применена к каждому элементу контейнера. На самом деле — это худший сценарий. Если два контейнера имеют разный размер, то никаких индивидуальных сравнений элементов выполнять не требуется.

Требования сложности — это характеристика STL. Хотя детали реализации могут быть скрыты, спецификации производительности должны быть открытыми, чтобы вы представляли вычислительные затраты на выполнение определенной операции.

Сложность константного времени и линейного времени

Представьте длинный, узкий ящик, наполненный большими пакетами, выстроенными в линию, причем этот ящик открыт только с одного конца. Предположим, что ваша задача — выгрузить пакет из открытого конца. Это задача, на выполнение которой требуется постоянное время. То есть, будь там 10 или 1000 пакетов за тем, что находится в конце — разницы нет.

Теперь представьте, что ваша задача — извлечь пакет, находящийся в закрытом конце ящика. Это задача, требующая линейного времени. Если в ящике всего 10 пакетов, вам придется выгрузить 10, чтобы добраться до последнего. Если их 100 — вам придется выгрузить 100, чтобы добраться до конца. Даже если вы — неутомимый работник, который может передвигать только по 1 пакету за раз, вторая задача потребует для выполнения в 10 раз больше времени, чем первая.

Теперь предположим, что вам нужно извлечь произвольный пакет. Может оказаться, что он будет первым, попавшим в ваши руки. Однако в среднем количество пакетов, которые вам придется передвинуть, по-прежнему пропорционально общему числу пакетов в контейнере, поэтому данная задача также требует линейного времени.

Замена длинного, узкого ящика подобным, но имеющим открывающиеся боковые стенки, делает время, необходимое для выполнения этого задания, константным, потому что вы можете обратиться непосредственно к нужному пакету, открыв боковую стенку ящика, и вытащить его, не трогая остальных.

Идея сложности по времени касается размера контейнера и времени выполнения операций, но игнорирует другие факторы. Если некий супергерой может выгружать ящики из открытого конца в 1000 раз быстрее, чем вы, задача по-прежнему будет иметь линейную сложность, но в этом случае линейное время доступа супергероя к пакетам закрытого ящика (с одним открытым концом) может оказаться меньше, чем ваше константное время доступа к пакетам открытого ящика, до тех пор, пока в ящиках не слишком много пакетов.

Последовательности

Вы можете уточнить базовую концепцию контейнера, добавив требования. *Последовательность* — это важное уточнение, потому что шесть из контейнерных типов STL (`deque`, `list`, `queue`, `priority_queue`, `stack` и `vector`) являются последовательностями. (Вспомните, что очередь (`queue`) позволяет элементам добавляться к заднему концу и удаляться из переднего. Двусторонняя очередь (double-ended queue), представленная контейнером `deque`, допускает добавление и извлечение из обоих концов.) Концепция последовательности добавляет требование, чтобы итератор был, по меньшей мере, однонаправленным. Это, в свою очередь, гарантирует размещение элементов в определенном порядке, который не меняется от одного цикла итераций к другому.

Последовательность также требует, чтобы элементы были упорядочены в строго линейном порядке. То есть, существует первый элемент, существует последний элемент, и каждый элемент кроме первого и последнего имеет только один элемент непосредственно перед ним, и один — непосредственно после него. Массив и связный список — это примеры последовательностей, в то время как структуры ветвлений (в которых каждый узел указывает на два дочерних) последовательностями не являются.

Поскольку элементы в последовательностях размещены в определенном порядке, становятся возможными такие операции, как вставка значений в определенную позицию и удаление определенного диапазона элементов. В табл. 16.6 перечислены эти, а также другие операции, необходимые последовательностям. В этой таблице используются те же обозначения, что и в табл. 16.5, с добавлением `t`, представляющего зна-

чение типа T — то есть типа значений, хранимых в контейнере, а также n — целого и p, q, i и j, представляющих итераторы.

Таблица 16.6. Требования к последовательностям

Выражение	Возвращаемый тип	Описание
X a(n,t);		Объявляет последовательность a из n копий значения t.
X(n, t)		Создает анонимную последовательность из n копий значения t.
X a(i, j)		Объявляет последовательность a, инициализированную содержимым из диапазона [i, j).
X(i, j)		Создает анонимную последовательность, инициализированную содержимым из диапазона [i, j).
a.insert(p,t)	iterator	Вставляет копию t перед p.
a.insert(p,n,t)	void	Вставляет n копий t перед p.
a.insert(p,i,j)	void	Вставляет копии элементов диапазона [i, j) перед p.
a.erase(p)	iterator	Удаляет элемент, на который указывает p.
a.erase(p,q)	iterator	Удаляет элементы диапазона [p, q).
a.clear()	void	То же самое, что erase(begin(), end()).

Поскольку шаблонные классы deque, list, priority_queue, stack и vector являются моделями концепции последовательности, все они поддерживают операции из табл. 16.6. В дополнение к этому существуют операции, которые доступны некоторым их этих шести моделей. Когда возможно, они имеют постоянное время выполнения. В табл. 16.7 перечислены эти дополнительные операции.

Таблица 16.7. Необязательные требования к последовательностям

Выражение	Возвращаемый тип	Значение	Контейнер
a.front()	T&	*a.begin()	vector, list, deque
a.back()	T&	*--a.end()	vector, list, deque
a.push_front(t)	void	a.insert(a.begin(), t)	list, deque
a.push_back(t)	void	a.insert(a.end(), t)	vector, list, deque
a.pop_front(t)	void	a.erase(a.begin())	list, deque
a.pop_back(t)	void	a.erase(--a.end())	vector, list, deque
a[n]	T&	*(a.begin() + n)	vector, deque
a.at(n)	T&	*(a.begin() + n)	vector, deque

Таблица 16.7 требует нескольких комментариев. Во-первых, обратите внимание, что и a[n], и a.at(n) возвращают ссылку на n-й элемент (нумерация начинается с 0) в контейнере. Разница между ними в том, что a.at(n) выполняет проверку границ и генерирует исключение out_of_range, если n находится вне корректного

диапазона значений для контейнера. Далее, вас может удивить, что, скажем, метод push_front() определен для list и deque, но не для vector. Предположим, что вы хотите вставить новое значение в начало вектора из 100 элементов. Чтобы освободить место, вам нужно переместить 99-й элемент в позицию 100, затем 98-й элемент – в позицию 99 и так далее. Это операция со сложностью линейного времени, потому что перемещение 100 элементов займет в 100 раз больше времени, чем перемещение единственного элемента. Но предполагается, что операции из табл. 16.7 реализованы только в том случае, если они могут быть выполнены за постоянное время. Дизайн списков и двусторонних очередей, однако, позволяет выполнять добавление элементов в начало без необходимости перемещения других элементов в новые позиции, поэтому они могут реализовать push_front() с постоянным временем выполнения. На рис. 16.4 иллюстрируются функции push_front() и push_back().

```
char word[4] = "cow";
 deque<char>dword(word, word+4)

dqword: [c|o|w]

 dqword.push_front('s');

dqword: [s|c|o|w]

 dqword.push_back('l');

dqword: [s|c|o|w|l]
```

Рис. 16.4. Функции push_front() и push_back()

Давайте повнимательнее взглянем на шесть последовательных контейнерных типа.

vector

Вы уже видели несколько примеров, использующих шаблон vector, который объявлен в заголовочном файле vector. Кратко говоря, vector – это представление массива в виде класса. Этот класс предоставляет автоматическое управление памятью, которое позволяет динамически менять размер объекта vector, увеличиваясь и уменьшаясь при добавлении и удалении элементов. Он предоставляет произвольный доступ к элементам. Элементы могут добавляться в конец и удаляться с конца за константное время, но вставка и удаление в начале и средине – это операции, требующие линейного времени.

В дополнение к тому, что vector является последовательностью, данный контейнер также представляет собой модель концепции *обратимого контейнера*. Это добавляет два метода класса: rbegin(), возвращающий итератор на первый элемент обратной последовательности, и rend(), возвращающий итератор на элемент, находящийся в обратной последовательности за последним. Поэтому, если dice – это контейнер типа vector<int>, a Show(int) – функция, отображающая целое значение, то следующий код отображает содержимое dice сначала в прямой, затем в обратной последовательности:

```
for_each(dice.begin(), dice.end(), Show); // отображает по порядку
cout << endl;
for_each(dice.rbegin(), dice.rend(), Show); // отображает в обратном порядке
cout << endl;
```

Итератор, возвращаемый этими двумя методами, относится к классу reverse_iterator. Вспомните, что инкремент этого итератора вызывает движение по обратимому контейнеру в обратном порядке.

Шаблонный класс vector — простейший из типов последовательностей и должен рассматриваться как тип, который следует использовать по умолчанию, если только не обнаружится, что требованиям программы больше удовлетворяют какие-то другие типы контейнеров.

deque

Шаблонный класс deque (объявленный в заголовочном файле deque) представляет собой двустороннюю очередь — тип, называемый кратко "декой". Как он реализован в STL, это контейнер, во многом похожий на vector, с поддержкой произвольного доступа. Основное отличие состоит в том, что вставка и удаление элементов из начала объекта deque — операция, выполняемая за постоянное время, в то время как у vector она занимает линейное время. Поэтому, если большинство операций происходят в начале и конце последовательности, вам стоит рассмотреть возможность применения структуры данных deque.

Цель обеспечения константного времени вставки и удаления на обоих концах deque делает дизайн объекта deque более сложным, чем у vector. Поэтому, хотя оба они предоставляют произвольный доступ к элементам, а также вставку и удаление из середины последовательности за линейное время, контейнер vector все же обеспечивает более скорое выполнение этих операций.

list

Шаблонный класс list (объявленный в заголовочном файле list) представляет собой двусвязный список. Каждый его элемент, за исключением первого и последнего, связан с предшествующим элементом и элементом, следующим за ним, откуда следует, что такой список можно проходить в обоих направлениях. Принципиальная разница между list и vector заключается в том, что list обеспечивает вставку и удаление за константное время в любой позиции списка. (Вспомните, что шаблон vector обеспечивает вставку и удаление за линейное время, если только не в конец последовательности, где время вставки и удаления константно). То есть, vector акцентирует внимание на быстром произвольном доступе, а list — на быстрой вставке и удалении элементов.

Подобно vector, list — обратимый контейнер. В отличие от vector, list не поддерживает нотацию массива и произвольный доступ. В отличие от итераторов vector, итераторы list указывают на одни и те же элементы, даже после вставки или удаления элементов. Например, предположим, что у вас есть итератор, указывающий на пятый элемент контейнера vector. Затем предположим, что вы вставляете элемент в начало контейнера. Все другие элементы должны быть перемещены, чтобы освободить место, поэтому после вставки пятый элемент содержит значение, которое было до вставки четвертым. То есть итератор указывает на ту же позицию, но на другие данные.

Вставка нового элемента в список, однако, не перемещает существующих элементов; она только изменяет информацию связей. Итератор, указывавший на определенный элемент, по-прежнему указывает на него же, но он может быть связан с другими элементами, нежели ранее.

Шаблонный класс `list` имеет некоторые ориентированные на списки функции-члены в дополнение к тем, что относятся ко всем последовательностям и обратимым контейнерам. В табл. 16.8 перечислены многие из них. (Полный список всех функций и методов STL вы можете найти в приложении Ж.) Параметр шаблона `Alloc` обычно не должен вас беспокоить, поскольку для него предусмотрено значение по умолчанию.

Таблица 16.8. Некоторые функции-члены `list`

Функция	Описание
`void merge(list<T, Alloc>& x)`	Объединяет список `x` с вызывающим списком. Оба списка должны быть отсортированы. Результирующий отсортированный список помещается в вызывающий, а `x` остается пустым.
`void remove(const T & val)`	Удаляет все экземпляры `val` из списка. Эта функция имеет линейное время выполнения.
`void sort()`	Сортирует список, применяя операцию <; время выполнения пропорционально `N` log `N` для `N` элементов.
`void splice(iterator pos, list<T, Alloc> x)`	Вставляет содержимое списка `x` перед позицией `pos` и оставляет `x` пустым. Эта функция выполняется за константное время.
`void unique()`	Удаляет повторяющиеся соседние элементы. Выполняется за линейное время.

Код в листинге 16.10 иллюстрирует эти методы, наравне с методом `insert()`, который приходит из классов STL, моделирующих последовательности.

Листинг 16.10. `list.cpp`

```
// list.cpp -- использование списка
#include <iostream>
#include <list>
#include <iterator>
int main()
{
   using namespace std;
   list<int> one(5, 2); // список из 5 двоек
   int stuff[5] = {1,2,4,8, 6};
   list<int> two;
   two.insert(two.begin(),stuff, stuff + 5 );
   int more[6] = {6, 4, 2, 4, 6, 5};
   list<int> three(two);
   three.insert(three.end(), more, more + 6);
   cout << "Список первый: ";
   ostream_iterator<int,char> out(cout, " ");
   copy(one.begin(), one.end(), out);
   cout << endl << "Список второй: ";
```

```
    copy(two.begin(), two.end(), out);
    cout << endl << "Список третий: ";
    copy(three.begin(), three.end(), out);
    three.remove(2);
    cout << endl << "Список третий минус двойки: ";
    copy(three.begin(), three.end(), out);
    three.splice(three.begin(), one);
    cout << endl << "Список третий после слияния: ";
    copy(three.begin(), three.end(), out);
    cout << endl << "Список первый: ";
    copy(one.begin(), one.end(), out);
    three.unique();
    cout << endl << "Список третий после unique: ";
    copy(three.begin(), three.end(), out);
    three.sort();
    three.unique();
    cout << endl << "Список третий после sort и unique: ";
    copy(three.begin(), three.end(), out);
    two.sort();
    three.merge(two);
    cout << endl << "Сортированный второй объединен с третьим: ";
    copy(three.begin(), three.end(), out);
    cout << endl;
    return 0;
}
```

Замечание по совместимости

Старые компиляторы могут использовать файлы заголовков `list.h` и `iterator.h`. Кроме того, некоторые компиляторы могут применять `ostream_iterator<int>` вместо `ostream_iterator<int, char>`.

Вывод программы из листинга 16.10 выглядит следующим образом:

```
Список первый: 2 2 2 2 2
Список второй: 1 2 4 8 6
Список третий: 1 2 4 8 6 6 4 2 4 6 5
Список третий минус двойки: 1 4 8 6 6 4 4 6 5
Список третий после слияния: 2 2 2 2 2 1 4 8 6 6 4 4 6 5
Список первый:
Список третий после unique: 2 1 4 8 6 4 6 5
Список третий после sort и unique: 1 2 4 5 6 8
Сортированный второй объединен с третьим: 1 1 2 2 4 4 5 6 6 8 8
```

Замечания по программе

Программа в листинге 16.10 использует технику, описанную ранее — когда обобщенная функция STL по имени `copy()` вместе с объектом `ostream_iterator` применяется для отображения содержимого контейнера.

Главное различие между `insert()` и `splice()` состоит в том, что `insert()` вставляет копию исходного диапазона в место назначения, в то время как `splice()` перемещает исходный диапазон в место назначения. То есть, после того, как содержимое `one` сливается с `three`, `one` остается пустым. (Метод `splice()` имеет дополнительные прототипы для перемещения отдельных элементов либо их диапазонов.) Метод `splice()` оставляет итераторы корректными. То есть, если вы настроили определен-

ный итератор на указание элемента из one, то он указывает на тот же элемент и после того, как splice() переместит его в three.

Обратите внимание, что unique() лишь удаляет соседние повторяющиеся элементы, оставляя по одному экземпляру. После того, как программа выполнит three.unique(), three будет по-прежнему содержать две четверки и две шестерки, которые не были соседними. Но применение sort(), а затем unique() обеспечит уникальность каждого элемента в списке.

Существует функция-не-член sort() (см. листинг 16.7), но она требует итераторов прямого доступа. Поскольку компромисс в отношении быстрой вставки ускоряет произвольный доступ, вы не можете использовать функцию-не-член sort() для работы со списком. Поэтому класс включает версию функции-члена, которая работает в пределах ограничений класса.

Набор инструментов list

Методы класса list образуют удобный набор инструментов. Предположим, например, что вам нужно организовать два списка почтовой рассылки. Вы должны отсортировать каждый список, объединить их, а затем воспользоваться unique() для удаления дубликатов.

Каждый из методов sort(), merge() и unique() также имеет версию, которая принимает дополнительный аргумент, специфицирующий альтернативную функцию, служащую для сравнения элементов. Аналогично, метод remove() имеет версию с дополнительным аргументом, указывающим функцию, используемую для определения того, что элемент удален. Эти аргументы являются примерами функций-предикатов — темы, к которой мы вернемся позднее.

queue

Шаблонный класс queue (объявленный в заголовочном файле queue, в прошлом queue.h) представляет собой класс-адаптер. Вспомните, что шаблон ostream_iterator — адаптер, который позволяет выходному потоку использовать интерфейс итератора. Аналогично, шаблон queue позволяет стоящему за ним классу (по умолчанию deque) демонстрировать типичный интерфейс очереди.

Шаблонный класс queue более ограничен, чем deque. Он не только не допускает произвольного доступа к элементам очереди, но даже не разрешает выполнять итерацию по ее элементам. Вместо этого, он ограничивает вас базовыми операциями, определяющими очередь. Вы можете добавлять элемент в конец очереди, удалять элемент в ее начале, просматривать значения первого и последнего элементов, проверять количество элементов, а также проверять, не пуста ли очередь. В табл. 16.9 перечислены эти операции.

Таблица 16.9. Операции queue

Метод	Описание
bool empty() const	Возвращает true, если очередь пуста, в противном случае — false.
size_type size() const	Возвращает количество элементов в очереди.
T& front()	Возвращает ссылку на элемент, находящийся в начале очереди.
T& back()	Возвращает ссылку на элемент, находящийся в конце очереди.
void push(const T& x)	Вставляет x в конец очереди.
void pop()	Удаляет элемент в начале очереди.

Следует отметить, что pop() — это метод, удаляющий данные, а не извлекающий их. Если вы хотите использовать значение из очереди, вы должны сначала вызвать метод front() для извлечения значения, а затем pop() — для удаления его из очереди.

priority_queue

Шаблонный класс priority_queue (объявленный в заголовочном файле queue) представляет собой другой класс-адаптер. Он поддерживает те же операции, что и queue. Главное отличие между этими двумя классами состоит в том, что в priority_queue наибольшее значение перемещается в начало очереди. (Жизнь — не всегда рынок, и не всегда — очередь.) Внутреннее отличие в том, что по умолчанию класс, лежащий в его основе — vector. Вы можете изменить операцию сравнения, используемую для того, чтобы определить, что должно находиться в голове очереди, передавая необязательный аргумент конструктору:

```
priority_queue<int> pq1; // версия по умолчанию
priority_queue<int> pq2(greater<int>);  // использовать greater<int>
                                        // для определения порядка
```

Функция greater<>() — это предопределенный функциональный объект, обсуждаемый далее в главе.

stack

Подобно queue, stack (объявленный в заголовочном файле stack, бывшем stack.h) — это класс-адаптер. Он дает лежащему в его основе классу (по умолчанию vector) типичный интерфейс стека.

Шаблонный класс stack более ограничен, чем vector. Он не только не допускает произвольного доступа к элементам стека, но также не позволяет выполнять итерацию по своим элементам. Вы можете заталкивать значение в вершину стека, выталкивать элемент с вершины, просматривать элемент, находящийся на вершине, проверять количество элементов, а также проверять, не пуст ли стек. В табл. 16.10 перечислены эти операции.

Таблица 16.10. Операции stack

Метод	Описание
bool empty() const	Возвращает true, если стек пуст, в противном случае — false.
size_type size() const	Возвращает количество элементов в стеке.
T& top()	Возвращает ссылку на элемент, находящийся на вершине стека.
void push(const T& x)	Вставляет x в вершину стека.
void pop()	Удаляет элемент из вершины стека.

Почти так же, как и с queue, если вы хотите использовать значение из стека, то сначала применяете top() для извлечения значения, а затем — pop() для его удаления из стека.

Ассоциативные контейнеры

Ассоциативный контейнер — это другое уточнение концепции контейнеров. Ассоциативный контейнер связывает значение с ключом и использует ключ для нахождения значения. Например, значения могут быть структурами, представляющими информацию о сотрудниках, такую как имя, адрес, номер офиса, домашний и рабочий телефоны, медицинская карточка и так далее, а ключом может быть уникальный табельный номер сотрудника. Чтобы извлечь информацию о сотруднике, программа должна использовать ключ для нахождения структуры, описывающей сотрудника. Вспомните, что вообще для контейнера `X` выражение `X::value_type` указывает на тип значений, хранимых в контейнере. Для ассоциативного контейнера выражение `X::key_type` означает тип, используемый для ключей.

Мощь ассоциативных контейнеров заключается в том, что они предоставляют быстрый доступ к своим элементам. Подобно последовательности, ассоциативный контейнер позволяет вам вставлять элементы; однако, вы не можете указать определенное местоположение для вставляемых элементов. Причина в том, что ассоциативный контейнер обычно имеет определенный алгоритм для определения того, куда поместить данные, чтобы он мог извлечь их быстро.

STL предоставляет четыре ассоциативных контейнера: `set`, `multiset`, `map` и `multimap`. Первые два типа объявлены в заголовочном файле `set` (в прошлом — раздельно в `set.h` и `multiset.h`), а вторые два типа объявлены в заголовочном файле `map` (в прошлом — раздельно в `map.h` и `multimap.h`).

Простейший контейнер из четырех — `set`; тип значения тот же, что и тип ключа, ключи уникальны, что означает, что в наборе хранится не более одного экземпляра каждого значения ключа. В самом деле, для `set` значение элемента является также его ключом. Тип `multiset` подобен `set`, за исключением того, что он может содержать более одного значения с одним и тем же ключом. Например, если типом ключа и значения является `int`, то объект `multiset` может содержать, скажем, `1, 2, 2, 2, 3, 5, 7` и `7`.

Для типа `map` тип значения элемента отличается от типа ключа, а ключи уникальны — по одному значению на ключ. Тип `multimap` подобен `map`, за исключением того, что один ключ может ассоциироваться с множеством значений.

Об этих типах доступно слишком много информации, чтобы раскрыть ее в настоящей главе (но в приложении Ж перечислены все их методы), поэтому давайте посмотрим на простой пример, использующий `set`, и простой пример, использующий `multimap`.

Пример set

Класс STL по имени `set` моделирует несколько концепций. Это ассоциативный набор, он обратим, отсортирован и его ключи уникальны, поэтому он не может содержать более одного заданного значения. Подобно `vector` и `list`, `set` использует параметр шаблона для указания хранимого типа:

```
set<string> A; // набор строковых объектов
```

Необязательный второй аргумент шаблона может служить для указания функции сравнения или объекта, который может быть использован для упорядочивания ключей. По умолчанию применяется шаблон `less<>` (обсуждаемый ниже). Старые реали-

зации C++ могут не предоставлять значений по умолчанию, а потому требуют явного указания второго параметра шаблона:

```
set<string, less<string> > A; // старая реализация
```

Рассмотрим следующий код:

```
const int N = 6;
string s1[N] = {"buffoon", "thinkers", "for", "heavy", "can", "for"};
set<string> A(s1, s1 + N); // инициализация набора A диапазоном массива
ostream_iterator<string, char> out(cout, " ");
copy(A.begin(), A.end(), out);
```

Как и другие контейнеры, set имеет конструктор (см. табл. 16.6), который принимает диапазон итераторов в качестве аргумента. Это обеспечивает простой способ инициализации набора содержимым массива. Вспомните, что последний элемент диапазона — это на самом деле элемент, лежащий за последним значащим элементом, а s1 + N указывает на одну позицию за концом массива s1. Вывод этого фрагмента кода иллюстрирует, что ключи уникальны (строка "for" появляется дважды в массиве, но один раз — в наборе), а также то, что набор отсортирован:

```
buffoon can for heavy thinkers
```

Математика определяет некоторые стандартные операции для множеств (наборов). Например, объединение двух множеств — это множество, состоящее из содержимого этих двух множеств. Если определенное значение — общее для двух множеств, то оно появляется в их объединении только одни раз, благодаря уникальности ключей. Пересечение двух множеств есть множество, состоящее из элементов, общих для обоих этих множеств. Разность двух множеств — это первое множество минус элементы, общие для обоих.

STL предоставляет алгоритмы, которые поддерживают эти операции. Это обычные функции, а не методы, поэтому они не ограничены объектами типа set. Однако все объекты set автоматически удовлетворяют предварительному условию применения этих алгоритмов — а именно: контейнер должен быть отсортирован. Функция set_union() принимает пять итераторов в качестве аргументов. Первые два определяют диапазон одного набора, вторые два — диапазон второго набора, а последний — это выходной итератор, указывающий местоположение, куда следует копировать результирующий набор. Например, чтобы отобразить объединение наборов A и B, вы можете использовать такой оператор:

```
set_union(A.begin(), A.end(), B.begin(), B.end(),
   ostream_iterator<string, char> out(cout, " "));
```

Предположим, что вы хотите поместить результат в набор C вместо отображения его на дисплее. В этом случае вы должны передать в последнем аргументе итератор на C. Очевидным выбором будет C.begin(), однако это не работает по двум причинам. Первая причина в том, что ассоциативные наборы интерпретируют ключи как константные значения, поэтому итератор, возвращенный C.begin(), будет константным итератором, который не может использоваться в качестве выходного итератора. Вторая причина того, что C.begin() нельзя применять, связана с тем, что set_union(), как и copy(), перезаписывает существующие данные контейнера и требует, чтобы в нем было достаточно места для размещения новой информации. C, будучи пустым, не удовлетворяет этому требованию. Но шаблон insert_iterator,

упомянутый ранее, решает обе проблемы. Ранее вы видели, что он превращает копирование во вставку. Кроме того, он моделирует концепцию выходного итератора, поэтому вы можете использовать его для записи в контейнер. Поэтому вы можете сконструировать анонимный insert_iterator для копирования информации в C. Конструктор, как вы должны помнить, принимает имя контейнера и итератор в качестве аргументов:

```
set_union(A.begin(), A.end(), B.begin(), B.end(),
   insert_iterator<set<string> >(C, C.begin()));
```

Функции set_intersection() и set_difference() находят пересечение и разницу двух наборов и имеют тот же интерфейс, что и set_union().

Два удобных метода set — это lower_boind() и upper_bound(). Метод lower_boind() принимает значение типа ключа в качестве аргумента и возвращает итератор, указывающий на первый член набора, который не меньше ключевого аргумента. Аналогично, upper_bound() принимает ключ в качестве аргумента и возвращает итератор, указывающий на первый член набора, который больше ключевого аргумента. Например, если вы имеете набор строк, вы можете использовать этот метод, чтобы идентифицировать диапазон, включающий все строки в наборе от "b" до "f".

Поскольку сортировка определяет, куда пойдут добавления в наборе, класс имеет методы, которые только специфицируют добавляемый материал без указания позиции. Если, например, A и B — наборы строк, то вы можете использовать следующий код:

```
string s("tennis");
A.insert(s);                   // вставить значение
B.insert(A.begin(), A.end());  // вставить значение
```

В листинге 16.11 иллюстрируется такое применение наборов.

Листинг 16.11. setops.cpp

```
// setops.cpp -- некоторые операции с наборами
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include <iterator>
int main()
{
   using namespace std;
   const int N = 6;
   string s1[N] = {"buffoon", "thinkers", "for", "heavy", "can", "for"};
   string s2[N] = {"metal", "any", "food", "elegant", "deliver","for"};
   set<string> A(s1, s1 + N);
   set<string> B(s2, s2 + N);
   ostream_iterator<string, char> out(cout, " ");
   cout << "Набор A: ";
   copy(A.begin(), A.end(), out);
   cout << endl;
   cout << "Набор B: ";
   copy(B.begin(), B.end(), out);
   cout << endl;
   cout << "Объединение A и B:\n";
   set_union(A.begin(), A.end(), B.begin(), B.end(), out);
```

```
   cout << endl;
   cout << "Пересечение A и B:\n";
   set_intersection(A.begin(), A.end(), B.begin(), B.end(), out);
   cout << endl;
   cout << "Разность A и B:\n";
   set_difference(A.begin(), A.end(), B.begin(), B.end(), out);
   cout << endl;
   set<string> C;
   cout << "Набор C:\n";
   set_union(A.begin(), A.end(), B.begin(), B.end(),
   insert_iterator<set<string> >(C, C.begin()));
   copy(C.begin(), C.end(), out);
   cout << endl;
   string s3("grungy");
   C.insert(s3);
   cout << "Набор C после вставки:\n";
   copy(C.begin(), C.end(),out);
   cout << endl;
   cout << "Отображение диапазона:\n";
   copy(C.lower_bound("ghost"),C.upper_bound("spook"), out);
   cout << endl;
   return 0;
}
```

Замечание по совместимости

Старые реализации STL могут использовать файлы заголовков `set.h`, `iterator.h` и `algo.h`. Кроме того, старые реализации могут потребовать `less<string>` в качестве второго аргумента шаблона `set`. Также старые версии могут применять `ostream_iterator<string>` вместо `ostream_iterator<string, char>`.

Вот как выглядит вывод программы из листинга 16.11:

```
Набор A: buffoon can for heavy thinkers
Набор B: any deliver elegant food for metal
Объединение A и B:
any buffoon can deliver elegant food for heavy metal thinkers
Пересечение A и B:
for
Разность A и B:
buffoon can heavy thinkers
Набор C:
any buffoon can deliver elegant food for heavy metal thinkers
Набор C после вставки:
any buffoon can deliver elegant food for grungy heavy metal thinkers
Отображение диапазона:
grungy heavy metal
```

Подобно большинству примеров в настоящей главе, код листинга 16.11 применяет “ленивый” способ объявления пространства имен `std`:

```
using namespace std;
```

Это делается для упрощения представления. Приведенный пример использует так много элементов пространства имен `std`, что применение директив с операциями разрешения контекста может сделать код несколько вычурным:

```
std::set<std::string> B(s2, s2 + N);
std::ostream_iterator<std::string, char> out(std::cout, " ");
std::cout << "Набор A: ";
std::copy(A.begin(), A.end(), out);
```

Пример multimap

Подобно set, multimap является обратимым, сортированным ассоциативным контейнером. Однако у multimap тип ключа отличается от типа значения, и объект multimap может иметь более одного значения, ассоциированного с определенным ключом.

Базовое объявление multimap специфицирует тип ключа и тип значения, сохраненные в виде аргументов шаблона. Например, следующее объявление создает объект multimap, который применяет int в качестве типа ключа и string — в качестве типа хранимых значений:

```
multimap<int, string> codes;
```

Необязательный третий параметр шаблона может быть использован для указания функции сравнения или объекта, который следует применить для упорядочивания ключа. По умолчанию используется шаблон less<> (описанный ниже) с типом ключа в качестве параметра. Более старые реализации C++ могут требовать явного указания этого параметра шаблона.

Короче говоря, действительный тип значений комбинируется с типом ключа в одну пару. Чтобы сделать это, STL применяет шаблонный класс pair<class T, class U> для сохранения двух видов значений в единственном объекте. Если keytype — тип ключа, а datatype — тип сохраняемых данных, то тип значения будет pair<const keytype, datatype>. Например, тип значения объекта codes есть pair<const int, string>.

Предположим, например, что вы хотите хранить названия городов, используя код региона в качестве ключа. Для этого подходит объявление codes, которое использует int в качестве ключа и string в качестве типа данных.

Одним подходом может быть создание пары и вставка ее в контейнер:

```
pair<const int, string> item(213, "Los Angeles");
codes.insert(item);
```

Либо вы можете создать анонимный объект pair и вставить его в единственном предложении:

```
codes.insert(pair<const int, string> (213, "Los Angeles"));
```

Поскольку элементы сортируются по ключу, нет необходимости указывать позицию вставки.

Имея объект pair, вы можете иметь доступ к его двум компонентам, используя члены first и second:

```
pair<const int, string> item(213, "Los Angeles");
cout << item.first << ' ' << item.second << endl;
```

А как насчет получения информации об объекте multimap? Функция-член count() принимает в качестве аргумента ключ и возвращает количество элементов, имеющих такое значение ключа. Функции-члены lower_bound() и upper_bound() принимают значение ключа и работают точно так же, как с объектами типа set. Также и функ-

ция-член equal_range() принимает в качестве аргумента ключ и возвращает итераторы, представляющие диапазон значений, соответствующих этому ключу. Чтобы вернуть два значения, метод пакует их в один объект pair, на этот раз с обоими аргументами шаблона — итераторами. Например, следующий фрагмент должен напечатать список городов объекта codes с кодом региона 718:

```
pair<multimap<KeyType, string>::iterator,
   multimap<KeyType, string>::iterator> range
      = codes.equal_range(718);
cout << "Города с кодом региона 718:\n";
for (it = range.first; it != range.second; ++it)
    cout << (*it).second << endl;
```

В листинге 16.12 демонстрируется большая часть этой техники. В нем также используется typedef для упрощения кодирования.

Листинг 16.12. multimap.cpp

```
// multimap.cpp -- использование multimap
#include <iostream>
#include <string>
#include <map>
#include <algorithm>
typedef int KeyType;
typedef std::pair<const KeyType, std::string> Pair;
typedef std::multimap<KeyType, std::string> MapCode;

int main()
{
   using namespace std;
   MapCode codes;
   codes.insert(Pair(415, "San Francisco"));
   codes.insert(Pair(510, "Oakland"));
   codes.insert(Pair(718, "Brooklyn"));
   codes.insert(Pair(718, "Staten Island"));
   codes.insert(Pair(415, "San Rafael"));
   codes.insert(Pair(510, "Berkeley"));

   cout << "Количество городов с кодом региона 415: "
        << codes.count(415) << endl;
   cout << "Количество городов с кодом региона 718: "
        << codes.count(718) << endl;
   cout << "Количество городов с кодом региона 510: "
        << codes.count(510) << endl;
   cout << "Код региона   Город\n";
   MapCode::iterator it;
   for (it = codes.begin(); it != codes.end(); ++it)
        cout << "    " << (*it).first << "    "
             << (*it).second << endl;
   pair<MapCode::iterator, MapCode::iterator> range
        = codes.equal_range(718);
   cout << "Города с кодом региона 718:\n";
   for (it = range.first; it != range.second; ++it)
        cout << (*it).second << endl;
   return 0;
```

Замечание по совместимости

Старые реализации STL могут использовать `multimap.h` и `algo.h`. Другие реализации могут требовать передачи `less<Pair>` в качестве третьего аргумента `multimap`. Также старые версии могут применять `ostream_iterator<string>` вместо `ostream_iterator<string, char>`. Borland C++Builder 1.0 требует изъятия `const` из `typedef`-определения `Pair`.

Ниже показан вывод программы из листинга 16.12:

```
Количество городов с кодом региона 415: 2
Количество городов с кодом региона 718: 2
Количество городов с кодом региона 510: 2
Код региона   Город
   415      San Francisco
   415      San Rafael
   510      Oakland
   510      Berkeley
   718      Brooklyn
   718      Staten Island
Города с кодом региона 718:
Brooklyn
Staten Island
```

Функциональные объекты (также известные как функторы)

Многие алгоритмы STL используют *функциональные объекты*, также известные как *функторы*. *Функтор* — это любой объект, который может быть использован с `()`, то есть на манер функции. Это включает нормальные имена функций, указатели на функции и объекты классов, с перегруженной операцией `()` — классы, для которых определен странно выглядящая функция `operator()()`. Например, вы можете определить класс вроде следующего:

```
class Linear
{
private:
   double slope;
   double y0;
public:
   Linear(double _sl = 1, double _y = 0)
      : slope(_sl), y0(_y) {}
   double operator()(double x) {return y0 + slope * x; }
};
```

Перегруженная операция `()` затем позволяет применять объект `Linear` так, как будто это функция:

```
Linear f1;
Linear f2(2.5, 10.0);
double y1 = f1(12.5); // справа - f1.operator()(12.5)
double y2 = f2(0.4);
```

Здесь `y1` вычисляется с использованием выражения `0 + 1 * 12.5`, а `y2` — с использованием выражения `10.0 + 2.5 * 0.4`. В выражении `y0 + slope * x` значения `y0` и `slope` поступают из конструктора объекта, а значение `x` — из аргумента операции `()`.

Помните функцию for_each? Она применяет указанную функцию к каждому элементу диапазона:

```
for_each(books.begin(), books.end(), ShowReview);
```

В общем случае, третий аргумент может быть функтором, а не обычной функцией. На самом деле, это вызывает вопрос: как объявить третий аргумент? Вы не можете объявить его как указатель на функцию, поскольку указатель на функцию подразумевает наличие типов аргументов. Поскольку контейнер может содержать почти любой тип, вы не знаете заранее, какой конкретный тип аргумента должен быть использован. STL решает эту проблему, применяя шаблоны. Прототип for_each выглядит следующим образом:

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

Прототип ShowReview() такой:

```
void ShowReview(const Review &);
```

Это указывает для идентификатора ShowReview тип void (*)(const Review &), поэтому это и есть тип, назначенный аргументу шаблона Function. При разных вызовах функций аргумент Function может представлять тип класса с перегруженной операцией ().

В конечном итоге код for_each() получит выражение f(...). В примере с ShowReview() f — указатель на функцию, и f(...) вызывает эту функцию. Если последний аргумент for_each() будет объектом, то f(...) становится объектом, который вызывает перегруженную операцию ().

Концепции функторов

Подобно тому, как библиотека STL определяет концепции контейнеров и итераторов, также она определяет концепции функторов:

- *Генератор* — это функтор, который может быть вызван без аргументов.
- *Унарная функция* — функтор, который может быть вызван с одним аргументом.
- *Бинарная функция* — функтор, который может быть вызван с двумя аргументами.

Например, функтор, поддерживающий for_each(), должен быть унарной функцией, поскольку он применяется к одному элементу контейнера за раз.

Конечно, приведенные концепции имеют уточнения:

- Унарная функция, которая возвращает bool, представляет собой *предикат*.
- Бинарная функция, которая возвращает bool, представляет собой *бинарный предикат*.

Некоторые функции STL требуют предикатов либо бинарных предикатов в качестве аргументов. Например, в листинге 16.7 используется версия sort(), которая принимает бинарный предикат в качестве третьего аргумента:

```
bool WorseThan(const Review & r1, const Review & r2);
...
sort(books.begin(), books.end(), WorseThan);
```

Шаблон `list` имеет функцию-член `remove_if()`, которая принимает предикат в виде аргумента. Она применяет предикат к каждому члену указанного диапазона, удаляя те элементы, для которых предикат возвращает `true`. Например, следующий код должен удалить из списка `three` все элементы, которые больше 100:

```
bool tooBig(int n){ return n > 100; }
list<int> scores;
...
scores.remove_if(tooBig);
```

Кстати, последний пример показывает, где могут быть полезны классы функторов. Предположим, что вы хотите удалить каждое значение, которое больше 200, из второго списка. Было бы неплохо, если бы вы могли передать граничное значение `tooBig()`, чтобы эта функция вызывалась с разными значениями, но предикат принимает только один аргумент. Если же, однако, вы проектируете класс `TooBig`, то можете использовать члены класса вместо аргументов функции для передачи дополнительной информации:

```
template<class T>
class TooBig
{
private:
   T cutoff;
public:
   TooBig(const T & t) : cutoff(t) {}
   bool operator()(const T & v) { return v > cutoff; }
};
```

Здесь одно значение (`v`) передается как аргумент функции, а второй аргумент (`cutoff`) устанавливается конструктором класса. Имея такое определение, вы можете инициализировать разные объекты `TooBig` для разных граничных значений, чтобы использовать их при вызовах `remove_if()`. В листинге 16.13 демонстрируется эта техника.

Листинг 16.13. `functor.cpp`

```
// functor.cpp -- использование functor
#include <iostream>
#include <list>
#include <iterator>

template<class T> // класс функтора определяет operator()()
class TooBig
{
private:
   T cutoff;
public:
   TooBig(const T & t) : cutoff(t) {}
   bool operator()(const T & v) { return v > cutoff; }
};
int main()
{
   using std::list;
   using std::cout;
   using std::endl;
```

```
    TooBig<int> f100(100); // лимит = 100
    list<int> yadayada;
    list<int> etcetera;
    int vals[10] = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};

    yadayada.insert(yadayada.begin(), vals, vals + 10);
    etcetera.insert(etcetera.begin(), vals, vals + 10);
    std::ostream_iterator<int, char> out(cout, " ");
    cout << "Исходные списки:\n";
    copy(yadayada.begin(), yadayada.end(), out);
    cout << endl;
    copy(etcetera.begin(), etcetera.end(), out);
    cout << endl;

    yadayada.remove_if(f100);//использовать именованный функциональный объект
    etcetera.remove_if(TooBig<int>(200));//конструировать функциональный объект
    cout <<"Усеченные списки:\n";
    copy(yadayada.begin(), yadayada.end(), out);
    cout << endl;
    copy(etcetera.begin(), etcetera.end(), out);
    cout << endl;
    return 0;
}
```

Один функтор (`f100`) — объявленный объект, а второй (`TooBig<int>(200)`) — анонимный объект, созданный вызовом конструктора. Ниже приведен вывод программы из листинга 16.13:

```
Исходные списки:
50 100 90 180 60 210 415 88 188 201
50 100 90 180 60 210 415 88 188 201
Усеченные списки:
50 100 90 60 88
50 100 90 180 60 88 188
```

Замечание по совместимости

Метод `remove_if()` — это шаблонный метод шаблонного класса. Шаблонные методы — недавнее расширение шаблонных средств C++, поэтому старые компиляторы могут их не поддерживать. Однако существует функция-не-член `remove_if()`, которая принимает в качестве аргументов диапазон (два итератора) и предикат.

Предположим, что имеется шаблонная функция с двумя аргументами:

```
template <class T>
bool tooBig(const T & val, const T & lim)
{
   return val > lim;
}
```

Вы можете использовать класс для преобразования ее в одноаргументный функциональный объект:

```
template<class T>
class TooBig2
{
private:
   T cutoff;
```

```
public:
   TooBig2(const T & t) : cutoff(t) {}
   bool operator()(const T & v) { return tooBig<T>(v, cutoff); }
};
```

То есть, вы можете использовать следующий код:

```
TooBig2<int> tB100(100);
int x;
cin >> x;
if (tB100(x)) // то же, что и (tooBig(x,100))
   ...
```

Поэтому вызов `tB100(100)` — это то же самое, что `tooBig(x,100)`, но функция с двумя аргументами преобразуется в функциональный объект с одним аргументом, где второй аргумент используется для конструирования функционального объекта. Короче говоря, функтор `tooBig(x,100)` — это функциональный адаптер, приспосабливающий функцию к требованиям отличающегося интерфейса.

Предопределенные функторы

STL определяет несколько элементарных функторов. Они выполняют такие действия, как сложение двух значений и проверка двух значений на предмет равенства. Они предусмотрены для оказания поддержки тем функциям STL, которые принимают функции в качестве аргументов. Для примера рассмотрим функцию `transform()`. Она имеет две версии. Первая принимает четыре аргумента. Из них первые два — итераторы, которые задают диапазон контейнера. (Теперь вам должен быть знаком такой подход.) Третий — итератор, который указывает, куда копировать результат. И последний — функтор, который применяется к каждому элементу диапазона для генерации каждого нового элемента. Рассмотрим следующий пример:

```
const int LIM = 5;
double arr1[LIM] = {36, 39, 42, 45, 48};
vector<double> gr8(arr1, arr1 + LIM);
ostream_iterator<double, char> out(cout, " ");
transform(gr8.begin(), gr8.end(), out, sqrt);
```

Этот код вычисляет квадратный корень из каждого элемента и посылает результирующее значение в выходной поток. Итератор назначения может быть в пределах исходного диапазона. Например, замена `out` в данном примере на `gr8.begin()` скопирует новые значения поверх старых. Понятно, что используемый функтор должен быть таким, что работает с одним аргументом.

Вторая версия использует функцию, которая принимает два аргумента, применяя функцию к одному элементу из каждого их двух диапазонов. Она принимает дополнительный аргумент, который идет третьим по счету, идентифицируя начало второго диапазона. Например, если `m8` будет вторым объектом `vector<double>` и если `mean(double, double)` будет возвращать среднее из двух значений, то следующий фрагмент кода выведет среднее из каждой пары значений из `g8` и `m8`:

```
transform(gr8.begin(), gr8.end(), m8.begin(), out, mean);
```

Теперь предположим, что вы хотите сложить два массива. Вы не можете применить `+` в качестве аргумента, потому что для типа `double` `+` — встроенная операция,

а не функция. Вы можете определить функцию для сложения двух чисел и воспользоваться ею:

```
double add(double x, double y) { return x + y; }
...
transform(gr8.begin(), gr8.end(), m8.begin(), out, add);
```

Но тогда вам придется определять отдельную функцию для каждого типа. Может быть, лучше было бы определить шаблон, если бы только STL уже не имел его. Заголовочный файл functional (бывший function.h) объявляет несколько шаблонных классов функциональных объектов, включая plus<>().

Применение класса plus<> для простого сложения возможно, хотя и неудобно:

```
#include <functional>
...
plus<double> add;        // создать объект plus<double>
double y = add(2.2, 3.4); // использовать plus<double>::operator()()
```

Но проще представить функциональный объект в виде аргумента:

```
transform(gr8.begin(), gr8.end(), m8.begin(), out, plus<double>() );
```

Здесь вместо создания именованного объекта код использует конструктор plus<double> для конструирования суммирующего функтора. (Скобки указывают на вызов конструктора по умолчанию; то, что передается transform(), является сконструированным функциональным объектом.)

STL предоставляет функторы-эквиваленты всех встроенных арифметических, сравнивающих и логических операций. В табл. 16.11 перечислены имена этих функторов-эквивалентов. Они могут использоваться со встроенными типами C++ или любыми определенными пользователем типами, которые перегружают соответствующую операцию.

Внимание!

Старые реализации C++ используют имя функтора times вместо multiplies.

Таблица 16.11. Операции и их эквиваленты-функторы

Операция	Эквивалент-функтор
+	plus
-	minus
*	multiplies
/	divides
%	modulus
-	negate
==	equal_to
!=	not_equal_to
>	greater
<	less
>=	greater_equal
<=	less_equal
&&	logical_end
||	logical_or
!	logical_not

Адаптируемые функторы и функциональные адаптеры

Предопределенные функторы в табл. 16.11 являются *адаптируемыми*. На самом деле STL имеет пять связанных концепций: адаптируемые генераторы, адаптируемые унарные функции, адаптируемые бинарные функции, адаптируемые предикаты и адаптируемые бинарные предикаты.

Что делает функторы адаптируемыми, так это то, что они используют typedef-члены, идентифицирующие их типы аргументов и тип возвращаемых значений. Эти члены называются result_type, first_argument_type и second_argument_type, и они представляют то, что можно предположить по их названиям. Например, возвращаемый тип объекта plus<int> идентифицируется как plus<int>::result_type, и это должен быть typedef для int.

Значимость адаптируемости функторов состоит в том, что они затем могут быть использованы объектами-адаптерами функций, предполагающими наличие typedef-членов. Например, функция с аргументом, который является адаптируемым функтором, может использовать член result_type для объявления переменной, соответствующей типу возврата функции.

В самом деле, STL предоставляет классы функциональных адаптеров, использующих эти средства. Например, предположим, что вы хотите умножить каждый элемент вектора gr8 на 2.5. Это повод для использования версии transform() с аргументом — унарной функцией, вроде:

```
transform(gr8.begin(), gr8.end(), out, sqrt);
```

из примера, показанного ранее. Функтор multiplies может выполнять умножение, но это — бинарная функция. Поэтому вам нужен адаптер функции, который преобразует функтор, имеющий два аргумента, в такой, который имеет один аргумент. Приведенный ранее пример с TooBig2 демонстрирует один способ сделать это, но STL автоматизирует этот процесс с помощью классов binder1st и binder2nd, которые преобразуют адаптируемые бинарные функции в адаптируемые унарные функции.

Взглянем повнимательнее на binder1st. Предположим, что у вас есть адаптируемый бинарный функциональный объект f2(). Вы можете создать объект binder1st, который привязывает определенное значение, назовем его val, для использования его в качестве первого аргумента f2():

```
binder1st(f2, val) f1;
```

Затем вызов f1(x) с его единственным аргументом возвратит то же значение, что и f2() с первым аргументом val и аргументом функции f1() в качестве второго аргумента. То есть, f1(x) — это эквивалент f2(val, x), за исключением того, что это — унарная функция вместо бинарной. Функция f2() адаптирована. Опять же, такое возможно только в случае, если f2() — адаптируемая функция.

Это может показаться несколько неудобным. Однако STL предлагает функцию bind1st() для упрощения использования класса binder1st. Вы передаете ей имя функции и значение, используемое для конструирования объекта binder1st, а она возвращает объект этого типа. Например, вы можете преобразовать бинарную функцию multiplies() в унарную, которая умножает свой аргумент на 2.5 следующим образом:

```
bind1st(multiplies<double>(), 2.5)
```

Таким образом, чтобы умножить каждый элемент gr8 на 2.5 и отобразить результаты, нужно поступить так:

```
transform(gr8.begin(), gr8.end(), out,
   bind1st(multiplies<double>(), 2.5));
```

Класс binder2nd аналогичен, за исключением того, что он присваивает константу второму аргументу вместо первого. Он имеет вспомогательную функцию bind2nd, которая работает аналогично bind1st.

Совет

Если функция STL вызывает унарную функцию и у вас есть адаптируемая бинарная функция, которая делает требуемые вещи, вы можете применить bind1st() или bind2nd() для того, чтобы адаптировать бинарную функцию к унарному интерфейсу.

В листинге 16.14 собраны некоторые из последних примеров в короткую программу.

Листинг 16.14. fundap.cpp

```
// fundap.cpp -- использование адаптеров функций
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <functional>
void Show(double);
const int LIM = 5;
int main()
{
   using namespace std;
   double arr1[LIM] = {36, 39, 42, 45, 48};
   double arr2[LIM] = {25, 27, 29, 31, 33};
   vector<double> gr8(arr1, arr1 + LIM);
   vector<double> m8(arr2, arr2 + LIM);
   cout.setf(ios_base::fixed);
   cout.precision(1);
   cout << "gr8:\t";
   for_each(gr8.begin(), gr8.end(), Show);
   cout << endl;
   cout << "m8: \t";
   for_each(m8.begin(), m8.end(), Show);
   cout << endl;
   vector<double> sum(LIM);
   transform(gr8.begin(), gr8.end(), m8.begin(), sum.begin(),
         plus<double>());
   cout << "sum:\t";
   for_each(sum.begin(), sum.end(), Show);
   cout << endl;
   vector<double> prod(LIM);
   transform(gr8.begin(), gr8.end(), prod.begin(),
         bind1st(multiplies<double>(), 2.5));
   cout << "prod:\t";
   for_each(prod.begin(), prod.end(), Show);
   cout << endl;
   return 0;
}
```

```
void Show(double v)
{
   std::cout.width(6);
   std::cout << v << ' ';
}
```

Замечание по совместимости

Старые реализации STL могут использовать vector.h, iterator.h, algo.h и function.h. Кроме того, старые реализации могут использовать times вместо multiplies.

Ниже показан вывод программы из листинга 16.14:

```
gr8: 36.0 39.0 42.0 45.0 48.0
m8: 25.0 27.0 29.0 31.0 33.0
sum: 61.0 66.0 71.0 76.0 81.0
prod: 90.0 97.5 105.0 112.5 120.0
```

Алгоритмы

STL включает в себя множество функций-не-членов для работы с контейнерами. Вы уже видели некоторые из них: sort(), copy(), find(), for_each(), random_shuffle(), set_union(), set_intersection(), set_difference() и transform(). Возможно, вы заметили, что у всех них примерно одинаковый дизайн — с применением итераторов для идентификации диапазонов обрабатываемых данных и диапазонов, куда нужно направлять результаты. Некоторые также имеют аргумент — функциональный объект для использования в качестве части обработки данных.

Существует два основных общих компонента в дизайне алгоритма функций. Во-первых, они применяют шаблоны для обобщения типов. Во-вторых, они используют итераторы для обеспечения обобщенного представления доступа к данным в контейнере. То есть функция copy() может работать с контейнером, который содержит значения типа double в массиве, с контейнером, содержащим значения string в связном списке, либо с контейнером, который хранит определенные пользователем объекты в древовидной структуре, как это делает set. Поскольку указатели — это специальный случай итераторов, функции STL вроде copy() могут быть использованы с обычными массивами.

Унифицированный дизайн контейнеров позволяет осуществлять осмысленные отношения между контейнерами разных видов. Например, вы можете применять copy() для копирования значений обычного массива в объект vector, из объекта vector в объект list и из объекта list в объект set. Вы можете применять == для сравнения разных видов контейнеров — например, deque и vector. Это возможно, потому что перегруженная операция == для контейнеров использует итераторы для сравнения содержимого, поэтому объекты deque и vector считаются эквивалентными, если они имеют одно и то же содержимое в одном и том же порядке.

Группы алгоритмов

STL разделяет библиотеку алгоритмов на четыре группы:

- Немодифицирующие последовательные операции.
- Модифицирующие последовательные операции.

- Сортирующие и связанные с ними операции.
- Обобщенные числовые операции.

Первые три группы описаны в заголовочном файле `algorithm` (бывший `algo.h`), а четвертая группа, будучи специально ориентированной на числовые данные, имеет свой собственный заголовочный файл, называемый `numeric` (раньше они также были в `algo.h`).

Немодифицирующие последовательные операции обрабатывают каждый элемент в диапазоне. Эти операции оставляют контейнер неизменным. Например, `find()` и `for_each()` относятся к этой категории.

Модифицирующие последовательные операции также обрабатывают каждый элемент в контейнере. Как следует из их названия, однако, они могут изменить содержимое контейнера. Изменения вносятся в значения либо в порядок, в котором они сохранены. Например, `transform()`, `random_shuffle()` и `copy()` попадают в эту категорию.

Сортирующие и связанные с ними операции включают некоторые сортирующие функции (включая `sort()`), а также многообразие других функций, включая операции с множествами (наборами).

Числовые операции включают функции для суммирования содержимого диапазона, вычисления внутреннего произведения двух контейнеров, подсчета частичных сумм и разностей между соседними элементами. Как правило, эти операции характерны для массивов, поэтому `vector` — это контейнер, который наиболее вероятно будет использован с ними.

В приложение Ж включен полный перечень этих функций.

Основные свойства алгоритмов

Как вы видите снова и снова в этой главе, функции STL работают с итераторами и диапазонами итераторов. Прототипы функций свидетельствуют о предположениях относительно итераторов. Например, функция `copy()` имеет следующий прототип:

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
   OutputIterator result);
```

Поскольку идентификаторы `InputIterator` и `OutputIterator` — это параметры шаблона, они с тем же успехом могли быть `T` и `U`. Однако документация по STL использует имена параметров шаблонов для того, чтобы указать концепции, которые параметры моделируют. Потому эти объявления говорят вам о том, что диапазоны параметров должны быть входными итераторами или выше, и о том, что итератор, указывающий, куда выводить результат, должен быть выходным итератором или выше.

Один способ классификации алгоритмов базируется на том, куда должен быть помещен результат работы алгоритма. Некоторые алгоритмы выполняют свою работу "на месте", другие создают копии. Например, когда завершается функция `sort()`, то результат занимает то же местоположение, что занимали исходные данные. Поэтому `sort()` — *алгоритм "по месту"*. Функция `transform()` может делать и то, и другое. В отличие от `copy()`, `transform()` позволяет выходному итератору указывать позицию внутри входного диапазона, поэтому она может копировать трансформированные значения поверх исходных.

Некоторые алгоритмы поставляются в двух версиях: версия "по месту" и версия копирующая. По соглашению STL, к имени копирующей версии добавляется _copy. Последние версии принимают дополнительный параметр — выходной итератор для указания местоположения, куда следует поместить копию. Например, существует функция replace() с таким прототипом:

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
   const T& old_value, const T& new_value);
```

Она заменяет каждый экземпляр old_value на new_value. Это происходит на месте. Поскольку этот алгоритм и читает и пишет элементы контейнера, тип итератора должен быть ForwardIterator либо мощнее. Копирующая версия имеет следующий прототип:

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
      OutputIterator result,
      const T& old_value, const T& new_value);
```

На этот раз результирующие данные копируются в новое место, заданное параметром result, поэтому только читающий входной итератор подходит для задания значения.

Отметим, что replace_copy() имеет тип возврата OutputIterator. Соглашение относительно копирующих алгоритмов таково, что они возвращают итератор, указывающий на позицию, находящуюся на один шаг за последним скопированным значением.

Другая общая вариация — некоторые функции имеют версию, которая выполняет действие по условию, в зависимости от результата применения функции к элементу контейнера. К именам этих версий обычно добавляется _if. Например, replace_if() заменяет старое значение новым, если применение функции к старому значению возвращает true. Вот ее прототип:

```
template<class ForwardIterator, class Predicate class T>
void replace_if(ForwardIterator first, ForwardIterator last,
      Predicate pred, const T& new_value);
```

(Вспомните, что предикат — это унарная функция, возвращающая значение bool.) Существует также версия под названием replace_copy_if(). Вы можете догадаться, что она делает, и на что похож ее прототип.

Как и InputIterator, Predicate — это имя параметра шаблона, которое с тем же успехом могло быть T или U. Однако STL выбирает имя Predicate, дабы напомнить пользователю, что действительный аргумент должен быть моделью концепции Predicate. Аналогично, STL использует термины вроде Generator и BinaryPredicate, чтобы идентифицировать алгоритмы, которые должны моделировать другие концепции функциональных объектов.

STL и класс string

Класс string, хотя и не является частью STL, спроектирован с STL "в уме". Так, например, он имеет функции-члены begin(), end(), rbegin() и rend(). То есть, он может использовать интерфейсы STL. Код в листинге 16.15 использует STL, чтобы показать все возможные перестановки, которые вы можете сформировать из букв

слова. *Перестановка* — это изменение порядка элементов в контейнере. Алгоритм next_permutation() трансформирует содержимое диапазона в следующую перестановку; в случае строки перестановки выполняются в возрастающем алфавитном порядке. Алгоритм возвращает true в случае успеха, и false — в случае, если диапазон элементов уже пребывает в заключительном порядке из возможных. Чтобы получить все перестановки диапазона, вы должны начать с элементов в наиболее раннем возможном порядке, и программа использует алгоритм STL sort() для этой цели.

Листинг 16.15. strgstl.cpp

```
// strgstl.cpp -- применение STL к строке
#include <iostream>
#include <string>
#include <algorithm>
int main()
{
   using namespace std;
   string letters;
   cout << "Введите группу символов (quit для завершения): ";
   while (cin >> letters && letters != "quit")
   {
     cout << "Перестановка из " << letters << endl;
     sort(letters.begin(), letters.end());
     cout << letters << endl;
     while (next_permutation(letters.begin(), letters.end()))
        cout << letters << endl;
     cout << "Введите следующую последовательность (quit для выхода): ";
   }
   cout << "Готово.\n";
   return 0;
}
```

Вот пример работы программы из листинга 16.15:

```
Введите группу символов (quit для завершения): wed
Перестановка из wed
dew
dwe
edw
ewd
wde
wed
Введите следующую последовательность (quit для завершения): wee
Перестановка из wee
eew
ewe
wee
Введите следующую последовательность (quit для завершения): quit
Готово.
```

Обратите внимание, что алгоритм next_permutation() автоматически обеспечивает генерацию только уникальных перестановок — вот почему для слова *wed* показано больше перестановок, чем для слова *wee*, в котором есть повторяющиеся буквы.

Сравнение функций и методов контейнеров

Иногда у вас возникает ситуация, когда нужно выбрать между применением метода STL и функции STL. Обычно метод — лучший выбор. Во-первых, он может быть лучше оптимизирован для конкретного контейнера. Во-вторых, будучи функцией-членом, он может использовать средства управления памятью шаблонного класса и при необходимости изменять размер контейнера.

Предположим, например, что у вас есть список чисел, и вы хотите удалить из него все экземпляры определенного значение, скажем, 4. Если `la` — это объект `list<int>`, то вы можете использовать метод списка `remove()`:

```
la.remove(4); // удаляет из списка все четверки
```

После вызова этого метода все элементы со значением 4 исключаются из списка и его размер автоматически изменяется.

Существует также алгоритм STL `remove()` (см. приложение Ж). Вместо того чтобы вызываться на объекте, он принимает аргументы, задающие диапазон. Поэтому, если `lb` — это объект `list<int>`, то вызов этой функции будет выглядеть следующим образом:

```
remove(lb.begin(), lb.end(), 4);
```

Однако, поскольку `remove()` — не член класса, он не может изменить размер списка. Вместо этого он обеспечивает, чтобы все оставшиеся элементы списка расположились в его начале, и возвращает итератор на новое значение, находящееся сразу за последним. Вы можете затем применить его, чтобы зафиксировать новый размер списка. Например, вы можете использовать метод `list erase()` для удаления диапазона, который описывает не нужную более часть списка. В листинге 16.16 показано, как работает этот процесс.

Листинг 16.16. `listrmv.cpp`

```
// listrmv.cpp - удаление из списка
#include <iostream>
#include <list>
#include <algorithm>
void Show(int);
const int LIM = 10;
int main()
{
   using namespace std;
   int ar[LIM] = {4, 5, 4, 2, 2, 3, 4, 8, 1, 4};
   list<int> la(ar, ar + LIM);
   list<int> lb(la);
   cout << "Исходное содержимое списка:\n\t";
   for_each(la.begin(), la.end(), Show);
   cout << endl;
   la.remove(4);
   cout << "После вызова метода remove():\n";
   cout << "la:\t";
   for_each(la.begin(), la.end(), Show);
   cout << endl;
   list<int>::iterator last;
   last = remove(lb.begin(), lb.end(), 4);
```

```
    cout << "После вызова функции remove():\n";
    cout << "lb:\t";
    for_each(lb.begin(), lb.end(), Show);
    cout << endl;
    lb.erase(last, lb.end());
    cout << "После вызова метода erase():\n";
    cout << "lb:\t";
    for_each(lb.begin(), lb.end(), Show);
    cout << endl;
    return 0;
}
void Show(int v)
{
    std::cout << v << ' ';
}
```

Вот как выглядит вывод программы из листинга 16.16:

```
Исходное содержимое списка:
4 5 4 2 2 3 4 8 1 4
После вызова метода remove():
la: 5 2 2 3 8 1
После вызова функции remove():
lb: 5 2 2 3 8 1 4 8 1 4
После вызова метода erase():
lb: 5 2 2 3 8 1
```

Как видите, метод remove() уменьшает размер списка la с 10 до 6 элементов. Однако после вызова функции remove() список lb по-прежнему содержит 10 элементов. Последние 4 элемента отбрасываются, потому что каждый из элементов со значением 4 либо с дублированным значением перемещен ближе к началу списка.

Хотя методы обычно лучше подходят, обычные функции более универсальны. Как вы видели, их можно использовать с массивами и объектами string, как и с контейнерами STL. И вы можете применять их с контейнерами смешанных типов, например, чтобы сохранить данные вектора в списке или наборе.

Использование STL

STL — это библиотека, части которой разработаны для совместной работы. Компоненты STL — это инструменты, но они также являются строительными блоками для создания других инструментов. Проиллюстрируем это на примере. Предположим, вы хотите написать программу, которая дает возможность пользователю вводить слова. В конце вы хотели бы записать эти слова, как они были введены, получить список слов алфавитном порядке (игнорируя регистр) и вывести статистику — сколько раз было введено каждое слово. Для упрощения предположим, что пользовательский ввод не будет содержать цифр и знаков препинания.

Ввод и сохранение списка слов достаточно просты. Руководствуясь примерами из листингов 16.5 и 16.6, вы можете создать объект vector<string> и использовать push_back() для добавления введенных слов к вектору:

```
vector<string> words;
string input;
while (cin >> input && input != "quit")
    words.push_back(input);
```

А как насчет получения алфавитного списка слов? Можно применить sort() с последующим unique(), но такой подход перезапишет исходные данные, потому что sort() — алгоритм, работающий "по месту". Существует более простой способ избежать этой проблемы. Вы можете создать объект set<string> и скопировать (используя вставляющий итератор) слова из вектора в этот набор. Набор автоматически сортирует свое содержимое, что означает, что вам не нужно вызывать sort(), кроме того, set хранит только по одной копии каждого ключа, поэтому не придется вызывать и unique(). Подождите! Спецификация требует игнорировать регистр. Один способ обеспечить это — использовать transform() вместо copy() для копирования данных из вектора в набор. В качестве функции трансформации вы можете применить такую, которая преобразует строки в нижний регистр:

```
set<string> wordset;
transform(words.begin(), words.end(),
   insert_iterator<set<string> > (wordset, wordset.begin()), ToLower);
```

Функцию ToLower() написать легко. Вы просто используете transform() для приложения функции tolower() к каждому элементу строки, задавая одну и ту же строку и в качестве источника и в качестве места назначения. Помните, что объекты string также могут применять функции STL. Передача и возврат строк по ссылке означает, что алгоритм работает на исходной строке, не создавая копий. Вот код ToLower():

```
string & ToLower(string & st)
{
   transform(st.begin(), st.end(), st.begin(), tolower);
   return st;
}
```

Возможно, проблемой будет то, что функция tolower() определена как int tolower(int), а некоторые компиляторы требуют функции, соответствующей типу элемента, который у нас char. Решением может быть замена tolower на toLower с представлением следующего определения:

```
char toLower(char ch) { return tolower(ch); }
```

Чтобы получить количество появлений каждого слова, вы можете применить функцию count(). Она принимает в качестве аргументов диапазон и значение, и возвращает количество появлений этого значения в диапазоне. Вы можете воспользоваться объектом vector, чтобы задать диапазон, и объектом set, чтобы получить список слов для подсчета. То есть, для каждого слова в наборе вы считаете, сколько раз оно встречается в векторе. Чтобы сохранить результаты подсчета ассоциированными с соответствующим словом, вы должны сохранить слова и их количества как объекты pair<const string, int> в объекте map. Слово будет ключом (только одна копия), а количество — значением. Это можно сделать в единственном цикле:

```
map<string, int> wordmap;
set<string>::iterator si;
for (si = wordset.begin(); si != wordset.end(); si++)
   wordmap.insert(pair<string, int>(*si, count(words.begin(),
                  words.end(), *si)));
```

Внимание!

Старые реализации STL объявляют `count()` как возвращающую тип `void`. Вместо использования возвращаемого значения вы передаете по ссылке четвертый аргумент, и количество элементов суммируется в нем:

```
int ct = 0;
count(words.begin(), words.end(), *si), ct)); // добавление ct
```

Класс `map` обладает интересным свойством: вы можете использовать нотацию массива с ключами в качестве индексов, чтобы получить доступ к хранимым значениям. Например, `wordmap["the"]` представит значение, ассоциированное с ключом `"the"`, что в нашем случае означает количество строк `"the"` во введенном тексте. Поскольку контейнер `wordset` содержит все ключи, используемые `wordmap`, вы можете применить следующий код как альтернативный и более привлекательный способ сохранения результатов:

```
for (si = wordset.begin(); si != wordset.end(); si++)
   wordmap[*si] = count(words.begin(), words.end(), *si);
```

Так как `si` указывает на строку в контейнере `wordset`, `*si` — это строка, которая может служить ключом для `wordmap`. Этот код помещает ключи и значения в карту `wordmap`.

Аналогично, вы можете использовать нотацию массива для выдачи результатов:

```
for (si = wordset.begin(); si != wordset.end(); si++)
   cout << *si << ": " << wordmap[*si] << endl;
```

Если ключ неверный, то соответствующее ему значение будет равно `0`.

В листинге 16.17 все эти идеи собраны вместе, а также включен код для отображения содержимого трех контейнеров (вектор с вводом, набор со списком слов и карту с подсчетом слов).

Листинг 16.17. usealgo.cpp

```
// usealgo.cpp -- использование некоторых элементов STL
#include <iostream>
#include <string>
#include <set>
#include <map>
#include <iterator>
#include <algorithm>
#include <cctype>
using namespace std;
char toLower(char ch) { return tolower(ch); }
string & ToLower(string & st);
void display(const string & s);
int main()
{
   vector<string> words;
   cout << "Введите слова (quit для завершения):\n";
   string input;
   while (cin >> input && input != "quit")
       words.push_back(input);
   cout << "Вы ввели следующие слова:\n";
   for_each(words.begin(), words.end(), display);
   cout << endl;
```

```
    // поместить слова в набор, преобразуя в нижний регистр
    set<string> wordset;
    transform(words.begin(), words.end(),
          insert_iterator<set<string> > (wordset, wordset.begin()),
          ToLower);
    cout << "\nАлфавитный список слов:\n";
    for_each(wordset.begin(), wordset.end(), display);
    cout << endl;
    // поместить слова и частоты в карту
    map<string, int> wordmap;
    set<string>::iterator si;
    for (si = wordset.begin(); si != wordset.end(); si++)
        wordmap[*si] = count(words.begin(), words.end(), *si);
    // отобразить содержимое карты
    cout << "\nЧастота появления слов:\n";
    for (si = wordset.begin(); si != wordset.end(); si++)
        cout << *si << ": " << wordmap[*si] << endl;
    return 0;
}
string & ToLower(string & st)
{
    transform(st.begin(), st.end(), st.begin(), toLower);
    return st;
}
void display(const string & s)
{
    cout << s << " ";
}
```

Замечание по совместимости

Старые реализации C++ могут использовать заголовочные файлы `vector.h`, `set.h`, `map.h`, `iterator.h`, `algo.h` и `ctype.h`. Кроме того, старые реализации могут потребовать применения в шаблонах `set` и `map` дополнительного параметра `less<string>`. Старые версии также используют функцию `count()` типа `void`, как упоминалось выше.

Ниже показан пример запуска программы из листинга 16.17:

```
Введите слова (quit для завершения):
The dog saw the cat and thought the cat fat
The cat thought the cat perfect
quit
Вы ввели следующие слова:
The dog saw the cat and thought the cat fat The cat thought the cat
perfect
Алфавитный список слов:
and cat dog fat perfect saw the thought
Частота появления слов:
and: 1
cat: 4
dog: 1
fat: 1
perfect: 1
saw: 1
the: 5
thought: 2
```

Мораль всего этого в том, что вам стоит использовать STL для того, чтобы в максимальной степени избежать написания собственного кода. Обобщенный и гибкий дизайн STL сэкономит массу работы. Кроме того, разработчики STL — люди, мыслящие алгоритмически, и сосредоточенные на достижении максимальной эффективности. Поэтому алгоритмы хорошо отлажены и внедрены.

Другие библиотеки

C++ предлагает ряд других библиотек классов, которые в большей степени специализированы, нежели примеры, приведенные в настоящей главе. Заголовочный файл complex предоставляет шаблонный класс complex для комплексных чисел, со специализациями под float, long и long double. Этот класс включает стандартные операции с комплексными числами, наряду со стандартными функциями, которые могут быть применены к комплексным числам.

В главе 14 рассматривался шаблонный класс vararray, поддерживаемый заголовочным файлом vararray. Этот шаблонный класс спроектирован для представления числовых массивов и предлагает поддержку множества операций с числовыми массивами — таких как добавление содержимого одного массива к другому, приложение математических функций к каждому элементу массива и приложение к массивам операций линейной алгебры.

vector и valarray

Возможно, вас удивит, почему C++ имеет два шаблона массивов: vector и valarray. Эти классы были созданы разными группами разработчиков для разных целей. Шаблонный класс vector — это часть системы контейнерных классов и алгоритмов. Класс vector поддерживает контейнерно-ориентированные действия вроде сортировки, вставки, переупорядочивания, поиска и передачи данных в другие контейнеры. Шаблонный класс valarray, с другой стороны, ориентирован на вычислительные операции, и не является частью STL. Например, он не имеет методов push_back() и insert(), но предлагает простой, интуитивный интерфейс для многих математических операций.

Предположим, например, что имеются следующие объявления:

```
vector<double> ved1(10), ved2(10), ved3(10);
valarray<double> vad1(10), vad2(10), vad3(10);
```

К тому же предположим, что ved1, ved2, vad1 и vad2 получили определенные значения. Предположим, вы хотите присвоить сумму первых элементов двух массивов первому элементу третьего массива и так далее. С классом vector вы делаете следующее:

```
transform(ved1.begin(), ved1.end(), ved2.begin(), ved3.begin(),
    plus<double>());
```

Однако класс valarray перегружает все арифметические операции для работы с объектами vararray, поэтому вы должны использовать следующий оператор:

```
vad3 = vad1 + vad2; // + перегружено
```

Аналогично, оператор

```
vad3 = vad1 * vad2; // * перегружено
```

в результате поместит в каждый элемент vad3 произведение соответствующих элементов из vad1 и vad2.

Допустим, вы хотите заменить каждое значение массива им же, но умноженным на 2.5. Подход STL выглядит следующим образом:

```
transform(ved3.begin(), ved3.end(), ved3.begin(),
    bind1st(multiplies<double>(), 2.5));
```

Класс valarray перегружает умножение объекта valarray на отдельное значение, и также перегружает многие операции присваивания, поэтому вы можете использовать любое из следующих выражений:

```
vad3 = 2.5 * vad3;   // * перегружена
vad3 *= 2.5;         // *= перегружена
```

Предположим, что вы хотите получить натуральный логарифм каждого элемента в массиве и сохранить результат в соответствующем элементе второго массива. Вот подход STL:

```
transform(ved1.begin(), ved1.end(), ved3.begin(), log);
```

Класс valarray перегружает обычные математически функции для принятия объекта valarray в качестве аргумента и возврата объекта valarray, поэтому вы можете использовать следующий оператор:

```
vad3 = log(vad1);   // log() перегружена
```

Или вы можете использовать метод apply(), который также работает для неперегруженных функций:

```
vad3 = vad1.apply(log);
```

Метод apply() не изменяет вызывающего объекта; вместо этого он возвращает новый объект, содержащий результирующие значения.

Простота интерфейса valarray еще более заметна, когда выполняются вычисления с multiset:

```
vad3 = 10.0* ((vad1 + vad2) / 2.0 + vad1 * cos(vad2));
```

Разработку той же задачи для STL vector мы оставим в качестве упражнения для заинтересованных читателей.

Класс valarray также предлагает метод sum(), который суммирует содержимое объекта valarray, метод size(), который возвращает количество элементов, метод max(), возвращающий наибольшее значение объекта, и min(), возвращающий наименьшее значение.

Как видите, valarray имеет преимущество в нотации перед vector, если речь идет о математических операциях, однако он менее универсален. Класс valarray включает метод resize(), но не существует автоматического изменения размера того рода, которое обеспечивает метод vector push_back(). Нет методов для вставки значений, выполнения сортировки и поиска и тому подобного. Короче говоря, класс valarray более ограничен, чем класс vector, но его более узкое назначение позволяет ему иметь намного более простой интерфейс.

Приводит ли простой интерфейс, представляемый valarray, к более высокой производительности? В большинстве случаев нет. Простая нотация обычно реализована посредством некоторого рода циклов, которые нужно выполнять над обычными

массивами. Однако некоторые разработчики оборудования обеспечивают параллельное выполнение векторных операций, в которых значения массива загружаются в массив регистров. В принципе, операции valarray могут быть реализованы так, чтобы использовать преимущества такого дизайна.

Можете ли вы применять STL для работы с объектами valarray? Ответ на этот вопрос требует еще раз вспомнить базовые принципы STL. Предположим, что имеется объект valarray<double>, содержащий 10 элементов:

```
valarray<double> vad(10);
```

После того, как массив будет наполнен значениями, можете ли вы, скажем, применить STL для их сортировки? Класс valarray не имеет методов begin() и end(), поэтому вы не можете использовать их в качестве аргументов, задающих диапазон:

```
sort(vad.begin(), vad.end()); // Нельзя, нету begin(), end()
```

Вы не можете имитировать использование обычного массива и указать в качестве диапазона vad и vad + 10, потому что vad — не имя массива; это имя объекта, а потому — не адрес. Поэтому следующий оператор не работает:

```
sort(vad, vad + 10); // НЕТ, vad - это объект, а не адрес
```

Возможно, вы можете воспользоваться операцией взятия адреса:

```
sort(&vad[0], &vad[10]); // может быть?
```

Это выглядит многообещающе; vad имеет тип valarray<double>, vad[0] — тип double, а &vad[0] — тип double *, поэтому его значение может быть присвоено указателю типа double *, который может вести себя как итератор. Далее, давайте посмотрим, удовлетворяет ли этот указатель (назовем его pt) требованиям STL к итераторам произвольного доступа. Во-первых, он может быть разыменован: *pt — это vad[0], значение первого элемента массива. А как насчет pt++? Конечно, указатель может быть инкрементирован, но будет ли он после этого указывать на следующий элемент массива? Описание класса говорит, что &a[i+j] эквивалентно &a[i] + j. В частности, &a[i] + 1 — это &a[i+1]; то есть добавление единицы к адресу дает нам адрес следующего элемента, поэтому инкремент pt делает его указывающим на следующий элемент. Основное правило эквивалентности также означает, что произвольный доступ работает. Пока все хорошо.

Но теперь перейдем к &vad[10]. Здесь возникают две проблемы. Правило о том, что &a[i+j] эквивалентно &a[i] + j, имеет ограничение, что i + j должно быть меньше размера массива. То есть для нашего примера, оно распространяется только до &vad[9]. Более того, эффект от обращения к элементу с индексом, равным или большим, чем размер массива, описан как ведущий к неопределенному поведению программы. Посему результат использования &vad[10]не определен. Это не значит, что sort() не будет работать (фактически, она работала на всех шести компиляторах, где тестировался этот код). Но это значит, что он может не работать. Для того чтобы этот код привел к сбою, возможно, понадобится весьма маловероятное сочетание условий — такое, как размещение массива в конце блока памяти, находящегося вне кучи. Но если потеря 350 миллионов долларов зависит от вашего кода, вы не захотите рисковать получить такой сбой.

Можно обойти эту проблему с выходом за конец, сделав объект valarray на один элемент больше, чем нужно, но это приведет к проблемам с методами sum(), max(), min() и size().

Код в листинге 16.18 иллюстрирует некоторые из относительных преимуществ классов vector и valarray. Он использует push_back() и средства автоматического изменения размера vector для накопления данных. Затем, после сортировки чисел, программа копирует их из объекта vector в объект valarray того же размера и выполняет некоторые математические операции.

Листинг 16.18. valvect.cpp

```
// valvect.cpp -- сравнение vector и valarray
#include <iostream>
#include <valarray>
#include <vector>
#include <algorithm>
int main()
{
   using namespace std;
   vector<double> data;
   double temp;
   cout << "Введите числа (<=0 для завершения):\n";
   while (cin >> temp && temp > 0)
     data.push_back(temp);
   sort(data.begin(), data.end());
   int size = data.size();
   valarray<double> numbers(size);
   int i;
   for (i = 0; i < size; i++)
     numbers[i] = data[i];
   valarray<double> sq_rts(size);
   sq_rts = sqrt(numbers);
   valarray<double> results(size);
   results = numbers + 2.0 * sq_rts;
   cout.setf(ios_base::fixed);
   cout.precision(4);
   for (i = 0; i < size; i++)
   {
     cout.width(8);
     cout << numbers[i] << ": ";
     cout.width(8);
     cout << results[i] << endl;
   }
   cout << "done\n";
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 16.18:

```
Введите числа (<=0 для завершения):
5 21.2 6 8 2 10 14.4 0
2.0000: 4.8284
5.0000: 9.4721
6.0000: 10.8990
8.0000: 13.6569
10.0000: 16.3246
14.4000: 21.9895
21.2000: 30.4087
```

Замечание по совместимости

На момент подготовки этой книги библиотека, используемая по умолчанию в Borland C++Builder X, имела ошибку в реализации математических функций valarray. Для ее обхода следует использовать старую версию STL, скомпилированную со следующими опциями командной строки:

```
-D_USE_OLD_RW_STL
```

Или же, находясь в интегрированной среде разработки, вы должны указать параметр _USE_OLD_RW_STL, выбрав в меню Project (Проект) команду Build Options Explorer (Проводник параметров сборки), щелкнув на bcc32, а затем добавив параметр в текстовое поле Conditional Defines (Условные определения).

Класс valarray имеет множество возможностей помимо тех, что были описаны до сих пор. Например, если numbers — объект valarray<double>, то оператор

```
valarray<bool> vbool = numbers > 9;
```

создает массив значений bool, в котором vbool[i] устанавливается равным значению numbers[i] > 9, то есть true или false.

Существует расширенная версия индексации. Давайте рассмотрим пример — класс slice (срез). Объект класса slice может быть использован в качестве индекса массива — в этом случае он представляет не просто одно значение, а некоторый набор значений. Объект slice инициализируется тремя целочисленными значениями: начало, количество и шаг. *Начало* указывает индекс первого элемента, который должен быть выбран, *количество* задает количество выбираемых элементов, а *шаг* представляет расстояние между соседними элементами. Например, объект, сконструированный как slice(1,4,3) означает — выбрать четыре элемента, с индексами 1, 4, 7 и 10. То есть, начать со стартового элемента, добавить шаг, чтобы получит следующий элемент, и так далее, до тех пор, пока не будет выбрано 4 элемента. Если, скажем, variant — это объект valarray<int>, то следующий оператор присвоит элементам 1, 4, 7 и 10 значение 10:

```
varint[slice(1,4,3)] = 10; // присвоить выбранным элементам значение 10
```

Это специальное средство индексации позволяет применять одномерный объект valarray для представления двумерных данных. Например, предположим, что вы хотите представить массив из 4 строк и 3 столбцов. Вы можете сохранить информацию в объект valarray из 12 элементов. Затем использовать объект slice(0,3,1) в качестве индекса, представляющего элементы 0, 1 и 2 — то есть, первую строку. Аналогично индекс slice(0,4,3) представит элементы 0, 3, 6 и 9 — то есть, первый столбец. Код в листинге 16.19 иллюстрирует некоторые возможности slice.

Листинг 16.19. vslice.cpp

```
// vslice.cpp -- использование срезов valarray
#include <iostream>
#include <valarray>
#include <cstdlib>

const int SIZE = 12;
typedef std::valarray<int> vint; // для упрощения объявлений

void show(const vint & v, int cols);
```

```
int main()
{
   using std::slice; // из <valarray>
   using std::cout;
   vint valint(SIZE); // думаем о 4 строках по 3 элемента

   int i;
   for (i = 0; i < SIZE; ++i)
       valint[i] = std::rand() % 10;
   cout << "Исходный массив:\n";
   show(valint, 3);                    // показать в 3 столбца
   vint vcol(valint[slice(1,4,3)]);  // извлечь 2-й столбец
   cout << "Второй столбец:\n";
   show(vcol, 1);                      // показать в 1 столбец
   vint vrow(valint[slice(3,3,1)]);  // извлечь 2-ю строку
   cout << "Вторая строка:\n";
   show(vrow, 3);
   valint[slice(2,4,3)] = 10;          // присвоить 2-му столбцу
   cout << "Установить последний столбец в 10:\n";
   show(valint, 3);
   cout << "Установить в первом столбце сумму следующих двух:\n";
   // + не определен для slice, поэтому преобразуем в valarray<int>
   valint[slice(0,4,3)] = vint(valint[slice(1,4,3)])
   + vint(valint[slice(2,4,3)]);
   show(valint, 3);
   return 0;
}

void show(const vint & v, int cols)
{
   using std::cout;
   using std::endl;
   int lim = v.size();
   for (int i = 0; i < lim; ++i)
   {
     cout.width(3);
     cout << v[i];
     if (i % cols == cols - 1)
        cout << endl;
     else
        cout << ' ';
   }
   if (lim % cols != 0)
     cout << endl;
}
```

Операция + определена для объектов `valarray`, таких как `valint`, и определена для отдельного элемента `int` вроде `valint[1]`, но, как отмечено в коде листинга 16.19, операция + не определена для проиндексированных `slice` массивов `valarray` наподобие `valint[slice(1,4,3)]`. Поэтому программа конструирует полные объекты из срезов, чтобы обеспечить сложение:

```
vint(valint[slice(1,4,3)]) // вызов конструктора на основе slice
```

Класс `valarray` предусматривает конструкторы для этой цели.

Ниже приведен пример выполнения программы из листинга 16.19:

```
Исходный массив:
1 7 4
0 9 4
8 8 2
4 5 5
Второй столбец:
7
9
8
5
Вторая строка:
0 9 4
Установить последний столбец в 10:
1 7 10
0 9 10
8 8 10
4 5 10
Установить в первом столбце сумму следующих двух:
17 7 10
19 9 10
18 8 10
15 5 10
```

Поскольку значения устанавливаются с использованием rand(), разные реализации rand() в результате дают разные значения.

Есть и еще кое-что, включая класс gslice, для представления многомерных массивов, но сказанного должно быть достаточно, чтобы дать вам понятие того, что собой представляет valarray.

Резюме

Язык C++ включает в себя мощный набор библиотек, предлагающих решения многих часто встречающихся задач, а также инструменты, упрощающие решение множества других задач. Класс string предоставляет удобный способ управления строками как объектами. Класс string обеспечивает автоматическое управление памятью и большой набор методов и функций работы со строками. Например, эти методы и функции позволяют выполнять конкатенацию строк, вставлять одни строки в другие, изменять порядок следования элементов в строках, искать в них символы или подстроки, а также выполнять операции ввода и вывода.

Шаблон auto_ptr упрощает задачу управления памятью, выделенной операцией new. Если вы используете auto_ptr вместо обычного указателя для хранения адреса, возвращенного new, то вам не нужно помнить о необходимости вызова операции delete. Когда объект auto_ptr уничтожается, его деструктор автоматически вызывает операцию delete.

STL — это коллекция шаблонных классов контейнеров, шаблонных классов итераторов, шаблонных функциональных объектов и шаблонных функций алгоритмов, которые имеют унифицированный дизайн, основанный на принципах обобщенного программирования. Алгоритмы используют шаблоны для обеспечения их общности в отношении хранимых типов и интерфейс итераторов — для обеспечения их общности в отношении типа контейнеров. Итераторы — это обобщенные указатели.

STL использует термин *концепции* для определения набора требований. Например, концепция однонаправленных итераторов включает требование, чтобы объект однонаправленного итератора мог быть разыменован для чтения и записи, а также мог быть инкрементирован. Действительная реализация концепции называется *моделью* концепции. Например, концепция однонаправленного итератора может быть смоделирована обыкновенным указателем либо объектом, предназначенным для навигации по связному списку. Концепции, основанные на других концепциях, называются *уточнениями*. Например, двунаправленный итератор — это уточнение концепции однонаправленного итератора.

Контейнерные классы наподобие `vector` и `set` являются моделями контейнерной концепции, такой как контейнеры, последовательности и ассоциативные контейнеры. STL определяет несколько шаблонов контейнерных классов: `vector`, `deque`, `list`, `set`, `multiset`, `map`, `multimap` и `bitset`. Также она определяет шаблоны адаптерных классов: `queue`, `priority_queue` и `stack`; эти классы адаптируют лежащие в их основе контейнерные классы, давая им характеристики и интерфейс, вытекающий из имени соответствующего шаблонного класса-адаптера. То есть `stack`, хотя и основан по умолчанию на `vector`, позволяет вставку и извлечение элементов только с вершины стека.

Некоторые алгоритмы выражены как методы контейнерных классов, но большинство выражены как общие функции-не-члены. Это дает возможность использовать итераторы в качестве интерфейса между контейнерами и алгоритмами. Одно преимущество такого подхода состоит в том, что нужна только одна функция `for_each()` или `copy()` вместо отдельных версий для каждого из контейнеров. Второе преимущество заключается в том, что алгоритмы STL могут быть использованы с не-STL контейнерами, такими как обычные массивы, объекты `string` или же любые классы, которые разработаете вы, но совместимые с итераторами STL и идиомой контейнера.

И контейнеры, и алгоритмы характеризуются типом итераторов, которые они представляют либо в которых нуждаются. Вы должны убедиться, что контейнер оснащен концепцией итератора, служащей нуждам алгоритмов. Например, алгоритм `for_each()` использует входной итератор, чьи минимальные требования удовлетворяются всеми типами контейнерных классов STL. Но `sort()` требует итераторов произвольного доступа, который поддерживают не все контейнерные классы. Контейнерный класс может предоставить специализированный метод — в качестве опции, — если он не отвечает требованиям определенного алгоритма. Например, класс `list` имеет метод `sort()`, основанный на двунаправленных итераторах, поэтому он может его использовать вместо общей функции.

STL также предлагает функциональные объекты, или функторы, которые являются классами с перегруженной операцией `()`, то есть для которых определен метод `operator()()`. Объекты таких классов могут быть вызваны с использованием функциональной нотации, но могут нести в себе дополнительную информацию. Адаптируемые функторы, например, содержат операторы `typedef`, которые идентифицируют типы аргументов и возвращаемые значения функтора. Эта информация может быть использована другими компонентами, такими как адаптеры функций.

Представляя общие контейнерные типы и разнообразие реализаций общих операций с разными алгоритмами, выполняя все в обобщенной манере, STL обеспечивает исключительный по ценности источник повторно используемого кода. Вы можете быть готовы решать программистские задачи непосредственно инструментами STL либо использовать их в качестве строительных блоков для построения нужных вам решений.

Шаблонные классы `complex` и `valarray` поддерживают числовые операции с массивами и комплексными числами.

Вопросы для самоконтроля

1. Имеется следующее объявление класса:

```
class RQ1
{
private:
    char * st; // указатель на строку в стиле C
public:
    RQ1() { st = new char [1]; strcpy(st,""); }
    RQ1(const char * s)
    {st = new char [strlen(s) + 1]; strcpy(st, s); }
    RQ1(const RQ1 & rq)
    {st = new char [strlen(rq.st) + 1]; strcpy(st, rq.st); }
    ~RQ1() {delete [] st};
    RQ & operator=(const RQ & rq);
    // прочее
};
```

   Преобразуйте это в объявление, использующее объект string. Какие методы более не потребуется объявлять явно?

2. Назовите хотя бы два преимущества объектов string перед строками в стиле C в отношении простоты применения.

3. Напишите функцию, которая принимает ссылку на объект string в качестве аргумента и затем преобразует объект string в верхний регистр.

4. Какие из следующих операторов не являются примерами корректного использования (концептуально или синтаксически) объекта auto_ptr? (Предполагается, что все необходимые заголовочные файлы включены.)

```
auto_ptr<int> pia(new int[20]);
auto_ptr<string> (new string);
int rigue = 7;
auto_ptr<int>pr(&rigue);
auto_ptr dbl (new double);
```

5. Если вы можете создать механический эквивалент стека, который хранит клюшки для гольфа вместо номеров, почему это будет (концептуально) плохая сумка для гольфа?

6. Почему контейнер set — плохой выбор для хранения записей о счетах в гольфе?

7. Если указатель — есть итератор, почему разработчики STL просто не используют его вместо итераторов?

8. Почему разработчики STL не определили базовый класс итератора, используя наследование для порождения классов для других типов итераторов, и не выразили алгоритмы в терминах этих классов итераторов?

9. Приведите, как минимум, три примера, показывающих преимущества объекта vector перед обычным массивом.

10. Если в коде из листинга 16.7 использовать list вместо vector, какие части программы станут неверными? Легко ли они могут быть исправлены? Если да, то как?

Упражнения по программированию

1. *Палиндром* – это строка, которая читается одинаково в обоих направлениях. Например, "тот" и "Отто" – короткие палиндромы. Напишите программу, которая запрашивает у пользователя строку и передает `bool`-функции ссылку на нее. Функция должна возвращать `true`, если строка является палиндромом, и `false` – в противном случае. Не беспокойтесь пока о сложностях вроде учета регистра, пробелах и пунктуации. То есть, эта простая версия должна отвергать строки "Отто" и "Madam, I'm Adam". Не стесняйтесь просмотреть список строковых методов в приложении Е, чтобы найти там методы, которые облегчат вашу задачу.

2. Решите предыдущую задачу, но позаботьтесь о сложностях, связанных с учетом регистра, пробелами и пунктуацией. То есть "Madam, I'm Adam" должно распознаваться как палиндром. Например, тестирующая функция может сжимать строку до "madamimadam" и затем проверять, читается ли она наоборот так же, как и слева направо. Не забудьте об удобной библиотеке `cctype`. Вы можете найти одну или две подходящие функции функцию STL, хотя и не обязательно.

3. Переделайте программу из листинга 16.3, чтобы она получала слова из файла. Один подход предусматривает использование объекта `vector<string>` вместо массива объектов `string`. Затем вы можете использовать `push_back()` для копирования стольких слов, сколько их есть в вашем файле данных в объект `vector<string>` и с помощью функции-члена `size()` определять длину списка слов. Поскольку программа должна читать из файла по одному слову за раз, вы должны применять операцию >> вместо `getline()`. Сам файл должен содержать слова разделенные пробелами, знаками табуляции или переводами строки.

4. Напишите функцию с интерфейсом в старом стиле, которая имеет следующий прототип:

   ```
   int reduce(long ar[], int n);
   ```

   Действительные аргументы должны быть именем массива и числом элементов в нем. Функция должна сортировать массив, удалять дублированные значения и возвращать значение, равное количеству элементов в уменьшенном массиве. Напишите эту функцию, используя функции STL. (Если вы решите применить общую функцию `unique()`, обратите внимание, что она возвращает конец результирующего диапазона.) Протестируйте эту функцию в короткой программе.

5. Решите ту же задачу, что и в пункте 4, но с помощью шаблонной функции:

   ```
   template <class T>
   int reduce(T ar[], int n);
   ```

   Протестируйте функцию в короткой программе, используя экземпляры с `long` и `string`.

6. Повторите пример, показанный в листинге 12.15, используя шаблон класса STL по имени `queue` вместо класса `Queue`, описанного в главе 12.

7. Лотерея – весьма популярная. Карточка лотереи имеет нумерованные метки, по которым случайным образом выбирается определенный номер. Напишите функцию `Lotto()`, принимающую два аргумента. Первый должен быть количе-

ством меток на карточке лотереи, а второй — количеством меток, выбранным случайно. Функция должна возвращать объект vector<int>, содержащий в отсортированном порядке случайно выбранные номера. Например, вы можете использовать эту функцию следующим образом:

```
vector<int> winners;
winners = Lotto(51,6);
```

Это должно присваивать winners вектор, содержащий шесть случайно выбранных чисел из диапазона от 1 до 15. Обратите внимание, что простое использование rand() не достаточно для выполнения этого задания, потому что она может генерировать дублированные значения. Совет: пусть функция создает вектор, который содержит все возможные значения, затем используйте random_shuffle(), после чего используйте начало перетасованного вектора для получения значений. Также напишите короткую программу для тестирования разработанной функции.

8. Мэт и Пэт хотят пригласить своих друзей на вечеринку. Они просят вас написать программу, которая делает следующее:
   - Позволяет Мэту ввести список имен его друзей. Имена сохраняются в контейнере и затем отображаются в отсортированном порядке.
   - Позволяет Пэт ввести список ее друзей. Имена сохраняются во втором контейнере и затем отображаются в отсортированном порядке.
   - Создает третий контейнер, который объединяет эти два списка, исключает дубликаты и отображает содержимое этого контейнера.

ГЛАВА 17

Ввод, вывод и файлы

В этой главе:

- Взгляд C++ на ввод и вывод
- Семейство классов `iostream`
- Переадресация
- Методы класса `ostream`
- Форматирование вывода
- Методы класса `istream`
- Состояния потоков
- Файловый ввод-вывод
- Использование класса `ifstream` для ввода из файлов
- Использование класса `ofstream` для вывода в файлы
- Использование класса `fstream` для файлового ввода и вывода
- Обработка командной строки
- Бинарные файлы
- Файлы произвольного доступа
- Встроенное форматирование

Обсуждение ввода и вывода C++ представляет собой проблему. С одной стороны, практически каждая программа использует ввод и вывод, и изучение их применения — одна из первых задач, стоящих перед теми, кто осваивает язык программирования. С другой стороны, C++ применяет многие из своих наиболее развитых языковых средств для реализации этих операций, включая классы, классы-наследники, перегрузку функций, виртуальные функции, шаблоны и множественное наследование. То есть, чтобы действительно понять ввод-вывод C++, вы должны знать об этом языке достаточно много. Чтобы ввести вас в курс дела, в начальных главах книги были обрисованы основные способы использования объектов `cin` класса `istream` и объекта `cout` класса `ostream` для ввода и вывода и (в меньшей степени) применение объектов `ifstream` и `ofstream` для файлового ввода и вывода. Эта глава предлагает более пристальный взгляд на классы ввода и вывода C++, показывая, как они спроектированы, и объясняя управление форматом при выводе. (Если вы пропустили несколько глав, чтобы изучить развитые средства форматирования, то можете сразу читать раздел, посвященный форматированию, обращая внимание на технику и игнорируя объяснения.)

Средства C++, предназначенные для файлового ввода и вывода, базируются на тех же основных определениях классов, что и объекты `cin` и `cout`, поэтому данная глава опирается на описание консольного ввода и вывода (с клавиатуры и на экран) в исследовании файловых операций ввода-вывода.

Комитет по стандартизации ANSI/ISO C++ работал над тем, чтобы сделать ввод-вывод C++ в большей степени совместимым с существующим вводом-выводом языка С, и это привело к некоторым изменениям по сравнению с традиционными приемами C++.

Обзор ввода и вывода в C++

Операции ввода и вывода в большинстве языков программирования встроены в сам язык. Например, если вы просмотрите список ключевых слов таких языков, как BASIC и Pascal, то увидите, что операторы `PRINT`, `writeln` и им подобные являются частью словарей этих языков. Но ни C, ни C++ не имеют операций ввода и вывода, встроенных в сам язык. Если вы посмотрите на ключевые слова этих языков, то найдете там `for` и `if`, но ничего такого, что имело бы отношение к вводу и выводу. Изначально язык C оставлял ввод-вывод на усмотрение производителей компиляторов. Смысл этого состоял в том, чтобы предоставить разработчикам компиляторов определенную свободу в проектировании функций ввода-вывода для обеспечения наилучшего соответствия требованиям оборудования целевой платформы. На практике большинство реализаций ввода-вывода основано на библиотечных функциях, изначально разработанных для среды Unix. ANSI C формализовал спецификации этого пакета в стандарте под названием "Стандартный пакет ввода-вывода" ("Standard Input/Output package"), утвердив его в качестве неотъемлемой составной части стандартной библиотеки C. C++ также включает этот пакет, поэтому если вы знакомы с семейством функций C, объявленных в файле `stdio.h`, то можете использовать их в программах C++. (Более новые реализации используют заголовочный файл `cstdio` для поддержки этих функций).

Однако для организации ввода-вывода C++ полагается на решения C++ вместо решений, предлагаемых C, и это решение представляет собой набор классов, определенных в заголовочных файлах `iostream` (бывший `iostream.h`) и `fstream` (бывший `fstream.h`). Упомянутая библиотека классов не является частью формального определения языка (`cin` и `istream` — не ключевые слова); в конце концов, язык программирования определяет правила того, как делать такие вещи, как создание классов, но не определяет, что именно вы должны создавать, следуя этим правилам. Но так же, как реализации C поставляются со стандартными библиотеками функций, C++ поставляется со стандартными библиотеками классов. Изначально стандартная библиотека классов подчинялась требованиям неформального стандарта, включающего только классы, определенные в заголовочных файлах `iostream` и `fstream`. Комитет по стандартизации ANSI/ISO C++ решил формализовать эту библиотеку в качестве стандартной библиотеки классов и добавить еще несколько стандартных классов, вроде тех, что обсуждались в главе 16. Настоящая глава посвящена стандартному вводу-выводу C++. Но для начала мы ознакомимся с концептуальной структурой ввода-вывода C++.

Потоки и буферы

Программа на C++ воспринимает ввод и вывод как потоки байтов. На вводе программа извлекает байты из входного потока и помещает байты в выходной поток. Для текстовых программ каждый байт может представлять символ. В общем плане байты могут формировать бинарное представление символов и числовых данных. Байты входного потока могут поступать с клавиатуры, но также могут поступать и от устройств хранения, вроде жесткого диска, или же от другой программы. Аналогично байты выходного потока могут передаваться на дисплей, на принтер, на устройство хранения или же отправляться другой программе. Потоки служат посредниками между программой и местом (устройством) назначения потока. Такой подход позволяет

программам на C++ трактовать ввод с клавиатуры в той же манере, что и ввод из файла; программа на C++ просто просматривает поток байт, не нуждаясь в информации о том, откуда эти байты поступают. Аналогично, используя потоки, программа на C++ может обрабатывать вывод независимо от того, куда, собственно, направляются байты. Таким образом, управление вводом включает две стадии:

- Ассоциирование потока с вводом программы.
- Подключение потока к файлу.

Другими словами, входной поток нуждается в двух подключениях — по одному на каждом конце. Подключение со стороны файла представляет собой источник данных для потока, а подключение со стороны программы принимает данные потока в программу. (Подключение со стороны файла может быть файлом, но так же оно может быть устройством — вроде клавиатуры.) Аналогично, управление выводом предусматривает подключение выходного потока к программе и ассоциирование некоторого выходного местонахождения с этим потоком. Это подобно водопроводу с байтами вместо воды (рис. 17.1).

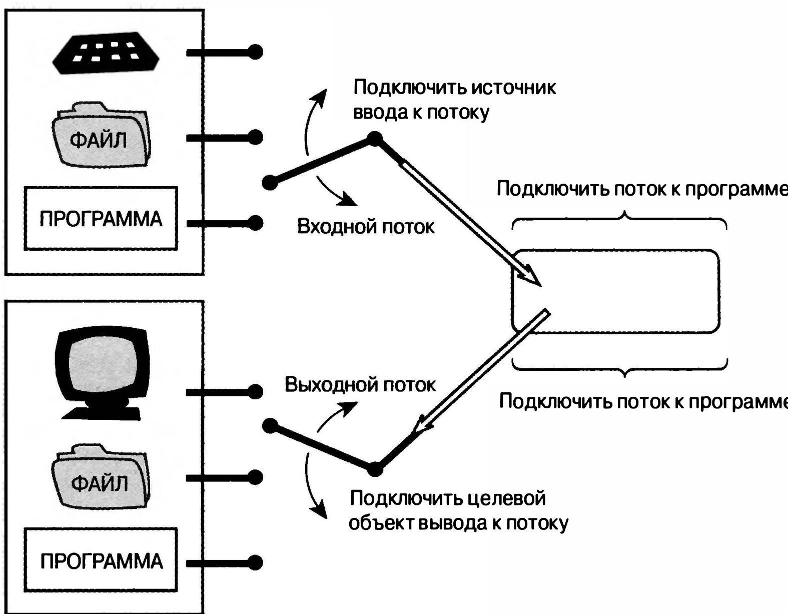


Рис. 17.1. Ввод и вывод в C++

Обычно ввод и вывод могут работать более эффективно, используя буфер. *Буфер* — это блок памяти, служащий в качестве посредника, средство временного хранения при передаче информации от устройства в программу или из программы на устройство. Обычно такие устройства, как приводы дисков, передают информацию блоками размером по 512 байт или более, в то время как программы часто обрабатывают данные по одному байту за раз. Буфер помогает согласовать эти две несоизмеримые пропорции при передаче информации. Например, предположим, что программа должна подсчитать количество символов доллара в файле на жестком диске.

Программа может читать один символ из этого файла, обрабатывать его, читать следующий символ из файла и так далее. Чтение из дискового файла одного символа требует большой аппаратной активности и достаточно медленно. Буферизованный подход заключается в чтении большой порции данных с диска, с сохранением этого куска в буфере и последующим чтением из буфера по одному символу. Поскольку это намного быстрее — читать индивидуальные байты из памяти, чем с диска, такой подход работает намного быстрее и легче для оборудования. Конечно, после того, как программа достигает конца буфера, ей приходится затем читать следующую порцию данных с диска. Принцип подобен использованию резервуара с водой, накапливающего большой объем дождевой воды во время непогоды, из которого затем вода поступает к вам в дом небольшими порциями по мере необходимости (рис. 17.2). Аналогично, при выводе программа может сначала наполнять буфер, а затем передавать блок данных целиком на жесткий диск, очищая буфер для порции новых данных. Это называется *сбросом буфера*. Возможно, вы найдете собственную "водопроводную" аналогию описанному процессу.

Рис. 17.2. Поток с буфером

Клавиатурный ввод поставляет один символ за раз, поэтому в этом случае программа не нуждается в буфере для приведения в соответствие разных уровней скорости передачи данных. Однако буферизованный клавиатурный ввод обеспечивает пользователю возможность вернуться и исправить ввод до того, как он поступит в программу. Обычно программа C++ сбрасывает входной буфер, когда вы нажимаете

клавишу <Enter>. Вот почему примеры в этой книге не начинают обработку данных, пока вы не нажмете клавишу <Enter>. Для вывода на дисплей программа C++ обычно сбрасывает выходной буфер, когда вы передаете символ новой строки. В зависимости от реализации, программа может сбрасывать ввод также и в других случаях — например, при надвигающемся вводе. То есть, когда программа достигает оператора ввода, она сбрасывает любой вывод, находящийся в выходном буфере. Реализации C++, которые согласованы с ANSI C, должны вести себя таким же образом.

Потоки, буферы и файл iostream

Работа по управлению потоками и буферами может оказаться несколько более сложной, но включение заголовочного файла `iostream` (бывшего `iostream.h`) предоставляет в ваше распоряжение несколько классов, предназначенных для реализации и управления потоками и буферами. Новейшая версия ввода-вывода C++ определяет шаблоны классов для поддержки как данных `char`, так и данных `wchar_t`. Используя средство `typedef`, C++ делает специализацию `char` этих шаблонов подобной традиционным, нешаблонным реализациям ввода-вывода. Ниже перечислены некоторые из этих классов (рис. 17.3).

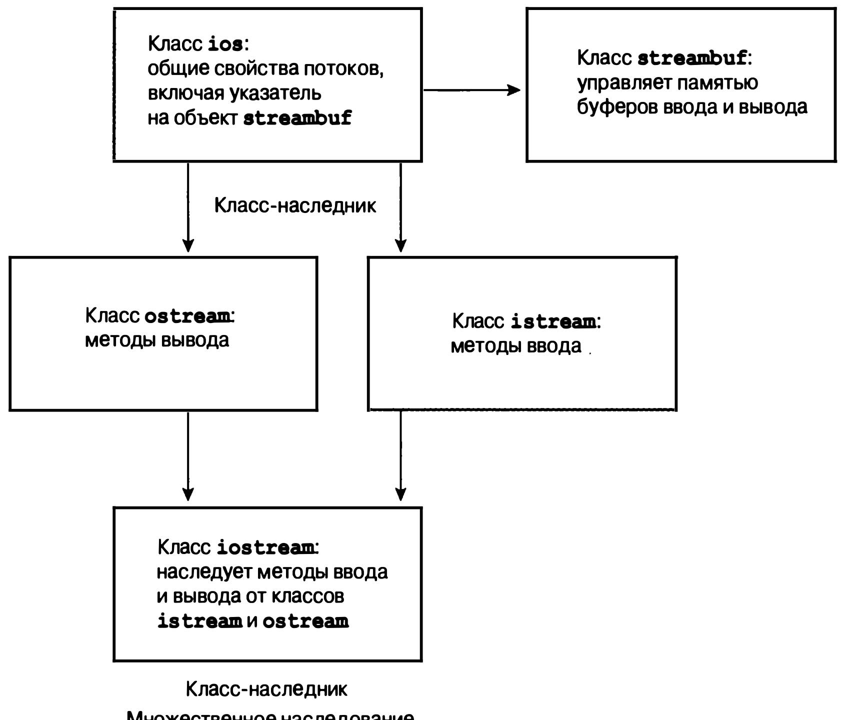


Рис. 17.3. Некоторые классы ввода-вывода

- Класс `streambuf` предоставляет память для буфера, а равно и методы для его наполнения, доступа к содержимому, сброса буфера и управления памятью буфера.

- Класс ios_base предоставляет общие средства потока, такие как признак того, открыт ли поток для чтения, и является ли он бинарным или текстовым.
- Класс ios базируется на ios_base и включает член-указатель на объект класса streambuf.
- Класс ostream наследуется от ios и предоставляет методы вывода.
- Класс istream наследуется от ios и предоставляет методы ввода.
- Класс iostream базируется на классах istream и ostream и потому наследует и методы вывода, и методы ввода.

Чтобы воспользоваться этими средствами, вы применяете объекты соответствующих классов. Например, для организации вывода вы используете объект ostream, подобный cout. Создание такого объекта открывает поток, автоматически создает буфер, и ассоциирует его с потоком. Это также делает доступными для вас функции-члены класса.

Переопределение ввода-вывода

Стандарт ISO/ANSI C++ пересматривает ввод-вывод многими способами. Во-первых, ostream.h заменен ostream, причем ostream помещает классы в пространство имен std. Во-вторых, классы ввода-вывода переписаны. Чтобы быть интернациональным языком, C++ должен уметь обрабатывать наборы символов шириной в 16 бит и больше. Поэтому к традиционному 8-битному представлению символов ("узкому") добавлен символьный тип wchar_t (или "широкий"). Каждому типу требуются свои собственные возможности ввода-вывода. Вместо того чтобы разрабатывать два отдельных набора классов, комитет по стандартизации создал набор классов-шаблонов ввода-вывода, включая basic_istream<charT, traits<charT>> и basic_ostream<charT, traits<charT>>. Шаблон traits<charT>, в свою очередь, представляет собой шаблонный класс, который определяет конкретные характеристики символьного типа, такие как способы сравнения на эквивалентность и значение EOF (end of file — конец файла). Стандарт C++ предлагает специализации char и wchar_t классов ввода-вывода. Например, istream и ostream — это typedef-определения для специализаций char. Аналогично, wistream и wostream — специализации wchar_t. Так, например, существует объект wcout для потокового вывода "широких" символов. Заголовочный файл ostream содержит эти определения.

Определенная независимая от типа информация, которая сохранялась в базовом классе ios, теперь перемещена в новый класс ios_base. Это включает различные константы форматирования вроде ios::fixed, которая теперь называется ios_base::fixed. Также ios_base содержит некоторые опции, которых не было в старом ios.

Библиотека классов C++ iostream заботится о многих деталях за вас. Например, включение в программу файла iostream автоматически создает восемь потоковых объектов (четыре для потоков "узких" символов и четыре — для "широких"):

- Объект cin соответствует стандартному потоку ввода. По умолчанию этот поток ассоциируется со стандартным устройством ввода — как правило, клавиатурой. Объект wcin аналогичен ему, но работает с символами типа wchar_t.
- Объект cout соответствует стандартному потоку вывода. По умолчанию этот поток ассоциируется со стандартным устройством вывода — как правило, монитором. Объект wcout аналогичен ему, но работает с символами типа wchar_t.
- Объект cerr соответствует стандартному потоку ошибок, который вы можете использовать для отображения сообщений об ошибках. По умолчанию этот

поток ассоциируется со стандартным устройством вывода — обычно монитором — и этот поток не буферизуется. Это означает, что информация отправляется непосредственно на экран без ожидания заполнения буфера или передачи символа перевода строки. Объект wcerr аналогичен, но работает с символами типа wchar_t.

- Объект clog также соответствует стандартному потоку ошибок. По умолчанию этот поток ассоциируется с стандартным устройством вывода — обычно монитором — и не буферизуется. Объект wclog аналогичен, но работает с символами типа wchar_t.

Что означает утверждение, что объект представляет поток? Ну, например, когда в файле iostream объявляется объект cout для программы, этот объект получает данные-члены, содержащие информацию относительно вывода, такую как ширина поля для отображения данных, количество знаков после десятичной точки, какая база используется для отображения целочисленных данных, адрес объекта streambuf, который описывает буфер, используемый выходным потоком. Оператор вроде

```
cout << "Bjarne free";
```

помещает символы из строки "Bjarne free" в буфер, управляемый cout через указанный объект streambuf. Класс ostream определяет функцию operator<<(), используемую в этом операторе, и класс ostream также поддерживает данные-члены cout с множеством других методов класса наподобие тех, что обсуждаются в настоящей главе ниже. Более того, C++, видя, что вывод из буфера направляется на стандартный вывод — обычно монитор, предоставленный операционной системой. Короче говоря, один конец потока подключается к программе, а второй — к устройству стандартного вывода, и объект cout с помощью объекта типа streambuf управляет движением байтов в потоке.

Перенаправление

Стандартные потоки ввода и вывода обычно подключаются соответственно к клавиатуре и экрану. Но многие операционные системы, включая Unix, Linux и MS-DOS, поддерживают перенаправление — средство, которое позволяет вам заменять ассоциацию стандартного ввода и стандартного вывода. Предположим, например, что у вас есть исполняемая программа на C++ по имени counter.exe, которая подсчитывает количество символов ввода и выдает результат. (В большинстве версий Windows вы можете выбрать в меню **Start** (Пуск) команду **All Programs** (Все программы) и затем щелкнуть на пиктограмме **MS-DOS Command Prompt** (Командная строка MS-DOS) или **Command Prompt** (Командная строка) для открытия окна MS-DOS.) Пример выполнения counter.exe может выглядеть следующим образом:

```
C>counter
Hello
and goodbye!
Control-Z        <- эмуляция конца файла
Input contained 19 characters.
C>
```

В этом случае ввод поступает с клавиатуры, а вывод направляется на экран.

Используя перенаправление ввода (<) и перенаправление вывода (>), вы можете использовать ту же программу для подсчета количества символов в файле oklahoma и помещать результат в файл cow_cnt:

```
C>counter <oklahoma >cow_cnt
C>
```

Часть команды <oklahoma ассоциирует стандартный ввод с файлом oklahoma, что заставляет cin читать ввод из этого файла вместо клавиатуры. Другими словами, операционная система заменяет подключение начальной части входного потока, в то время как конечная его часть остается подключенной к программе. Часть команды >cow_cnt ассоциирует стандартный вывод с файлом cow_cnt, что заставляет cout направлять свой вывод в этот файл вместо того, чтобы отправлять его на экран. То есть, операционная система заменяет конечную часть выходного потока, оставляя его начальную часть подключенной к программе. Операционные системы DOS (версии 2.0 и более поздние), Linux и Unix автоматически распознают этот синтаксис перенаправления. (Unix, Linux и DOS 3.0 и более поздние версии также допускают необязательные символы пробела между операциями перенаправления и именами файлов.)

Стандартный выходной поток, представленный объектом cout — это нормальный канал вывода программы. Стандартные потоки ошибок (представленные cerr и clog) предназначены для сообщений об ошибках программы. По умолчанию все эти три объекта отправляют информацию на монитор. Но перенаправление стандартного вывода не затрагивает cerr и clog; таким образом, если вы применяете один из этих объектов для печати сообщений об ошибках, программа отобразит их на экране, даже если обычный вывод cout будет перенаправлен куда-то еще. Например, рассмотрим следующий фрагмент кода:

```
if (success)
   std::cout << "Here come the goodies!\n";
else
{
   std::cerr << "Something horrible has happened.\n";
   exit(1);
}
```

Если перенаправление не используется, все сообщения будут выданы на экран. Если же, однако, вывод программы перенаправлен в файл, то первое сообщение, если оно будет выдано, попадет в файл вместо экрана, но если будет выдано второе сообщение, то оно попадет на экран. Кстати, некоторые операционные системы также допускают перенаправление стандартного потока ошибок. Например, в Unix и Linux это делает операция 2>.

Вывод с помощью cout

Как упоминалось ранее, C++ рассматривает вывод как поток байтов. (В зависимости от реализации и платформы это могут быть 16- или 32-битные байты, но все равно они будут байтами.) Однако многие виды данных в программе организованы в более крупные единицы, нежели отдельный байт. Тип int, например, может быть представлен 16- или 32-битным двоичным значением. А значение double может быть представлено 64-битным двоичным значением. Но когда вы отправляете поток бай-

тов на экран, то хотите, чтобы каждый байт представлял символьное значение. То есть, чтобы отобразить на экране число –2.34, вы должны отправить на экран пять символов: –, 2, ., 3 и 4, а не 64 бита двоичных данных. То есть одной из наиболее важных задач, которые стоят перед классом `ostream`, является преобразование числовых типов, таких как `int` или `float`, в поток символов, представляющих их в текстовом виде. То есть, класс `ostream` транслирует внутреннее представление данных как двоичных битовых последовательностей в выходной поток символьных байтов. (Когда-нибудь мы можем получить "бионические имплантанты", которые позволят нам интерпретировать двоичные данные непосредственно. Я оставляю эту разработку в качестве упражнения для любопытных читателей.) Для выполнения такой трансляции в классе `ostream` предусмотрено несколько методов. Мы взглянем на них сейчас, резюмируя методы, используемые на протяжении всей книги и описывая дополнительные методы, обеспечивающие более тонкое управление выводом.

Перегруженная операция <<

Чаще всего в этой книге мы используем `cout` с операцией `<<`, также называемой операцией ***вставки***:

```
int clients = 22;
cout << clients;
```

В языках С и С++ по умолчанию операция `<<` используется в качестве битовой операции сдвига (см. приложение Д). Выражение вроде `x<<3` означает: взять двоичное представление `x` и сдвинуть его на три бита влево. Очевидно, что это не имеет отношения к выводу. Но класс `ostream` переопределяет операцию `<<` для вывода. В этой ипостаси операция `<<` называется операцией вставки, а не операцией сдвига влево. (Операция сдвига влево выполняет эту роль благодаря своему визуальному аспекту, предполагающему направление потока информации влево.) Операция вставки перегружена для применения со всеми базовыми типами С++:

- `unsigned char`
- `signed char`
- `char`
- `short`
- `unsigned short`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `float`
- `double`
- `long double`

Класс `ostream` предлагает определение функции `operator<<()` для каждого из этих типов данных. (Функции, в имени которых присутствует слово *`operator`*, используются для перегрузки операций, как это описано в главе 11.) То есть, если вы используете оператор в форме

```
cout << value;
```

и value будет относиться к одному из перечисленных выше типов, то программа на C++ может найти соответствующую функцию операции с нужной сигнатурой. Например, выражение cout << 88 соответствует следующему прототипу метода:

```
ostream & operator<<(int);
```

Вспомним, что такой прототип указывает, что функция operator<<() принимает один аргумент типа int. И это подходит для параметра 88 из предыдущего оператора. Прототип также указывает, что функция возвращает ссылку на объект ostream. Это свойство делает возможным группировать вывод, как ниже:

```
cout << "I'm feeling sedimental over " << boundary << "\n";
```

Если вы — программист на C и страдаете от многообразия спецификаторов типа % и проблем, связанных с несоответствием спецификатора действительному типу значения, то использование cout покажется вам до неприличия простым. (Конечно же, как и ввод C++ через cin.)

Вывод и указатели

Класс ostream определяет функции операции вставки для следующих типов указателей:

- const signed char *
- const unsigned char *
- const char *
- void *

Не забывайте, что C++ представляет строку как указатель на ее начало. Указатель может иметь форму имени массива char, явного указателя на char или же строки в кавычках. Таким образом, все следующие операторы с cout отображают строки:

```
char name[20] = "Dudly Diddlemore";
char * pn = "Violet D'Amore";
cout << "Hello!";
cout << name;
cout << pn;
```

Методы используют завершающий нулевой символ строки для определения места, где нужно прекратить отображение символов.

C++ понимает указатели любого другого типа как void * , и в этом случае печатает числовое представление адреса. Если вы хотите получить адрес строки, то должны выполнить приведение типа к другому типу, как показано в следующем фрагменте кода:

```
int eggs = 12;
char * amount = "dozen";
cout << &eggs;              // печатает адрес переменной eggs
cout << amount;             // печатает строку "dozen"
cout << (void *) amount;    // печатает адрес строки "dozen"
```

На заметку!

Некоторые старые реализации C++ не имеют прототипа с аргументом void *. В этом случае вы должны выполнить приведение типа указателя к unsigned или же, возможно, к unsigned long, если хотите напечатать значение адреса.

Конкатенация вывода

Все воплощения операции вставки определены с типом возврата ostream &. То есть прототип имеет следующую форму:

```
ostream & operator<<(type);
```

(Здесь *type* задает тип отображаемого значения.) Возвращаемый тип ostream & означает, что использование этой операции возвращает ссылку на объект ostream. Какой объект? Определение функции говорит о том, что это будет ссылка на тот же объект, который использован для вызова операции. Другими словами, функция операции возвращает тот же объект, который вызвал операцию. Например, cout << "potluck" возвращает ссылку на объект cout. Это свойство позволяет выполнять конкатенацию вывода, используя вставку. Например, рассмотрим следующий оператор:

```
cout << "We have " << count << " unhatched chickens.\n";
```

Выражение cout << "We have " отображает строку и возвращает объект cout, сокращая оператор до следующего вида:

```
cout << count << " unhatched chickens.\n";
```

Затем выражение cout << count отображает значение переменной count и возвращает cout, который затем может использоваться для вывода заключительного аргумента в операторе (рис. 17.4). Это действительно изящная возможность, и именно поэтому в примерах перегрузки операции << она бесстыдным образом имитировалась.

Рис. 17.4. Конкатенация вывода

Другие методы ostream

Помимо разнообразных функций operator<<(), класс ostream представляет метод put() для отображения символов и метод write() для отображения строк.

Изначально метод put() имел следующий прототип:

```
ostream & put(char);
```

Текущий стандарт эквивалентен, за исключением того, что он реализован в виде шаблона, чтобы позволить вывод значений типа wchar_t. Вы вызываете его, применяя обычную нотацию вызова метода класса:

```
cout.put('W'); // отображает символ W
```

Здесь cout — вызывающий объект, а put() — функция-член класса. Подобно функции операции <<, эта функция возвращает ссылку на вызывающий объект, поэтому вы можете сцепить вывод с ней:

```
cout.put('I').put('t'); // отображает 'It' двумя вызовами put()
```

Вызов функции cout.put('I') возвращает cout, который затем служит вызывающим объектом для вызова put('t').

Имея правильный прототип, вы можете применять put() с аргументами числовых типов, отличных от char, таких как int, и позволять прототипируемой функции автоматически преобразовывать аргумент к корректному значению типа char. Например, вы можете использовать такие операторы:

```
cout.put(65);   // отображает символ A
cout.put(66.3); // отображает символ B
```

Первый оператор преобразует целочисленное значение 65 в значение типа char и отображает символ, имеющий ASCII-код, равный 65. Аналогично, второй оператор преобразует значение 66.3 типа double в значение 66 типа char и отображает соответствующий символ.

Такое поведение было недоступно в версиях, предшествовавших C++ Release 2.0. В тех версиях язык представлял символьные константы как значения типа int. То есть, оператор вроде

```
cout << 'W';
```

интерпретировал 'W' как значение int, и потому отображал целое число 87, то есть ASCII-код этого символа. Однако оператор

```
cout.put('W');
```

работал правильно. Поскольку современные реализации C++ представляют константы char как значения типа char, теперь вы можете применять оба метода.

Некоторые старые компиляторы ошибочно перегружали put() для трех типов аргументов: char, unsigned char и signed char. Это приводило к неоднозначности при вызове put() с аргументом типа int, поскольку int может быть преобразован к любому из этих трех типов.

Метод write() пишет целую строку и имеет следующий шаблонный прототип:

```
basic_ostream<charT,traits>& write(const char_type* s, streamsize n);
```

Первый аргумент write() представляет адрес строки, которую нужно отобразить, а второй аргумент указывает, сколько символов нужно отобразить. Использование

cout для вызова write() вызывает специализацию char, поэтому возвращаемый тип будет ostream &. В листинге 17.1 показано, как работает метод write().

Листинг 17.1. write.cpp

```
// write.cpp -- использование cout.write()
#include <iostream>
#include <cstring> // или иначе string.h
int main()
{
   using std::cout;
   using std::endl;
   const char * state1 = "Florida";
   const char * state2 = "Kansas";
   const char * state3 = "Euphoria";
   int len = std::strlen(state2);
   cout << "Цикл с возрастанием индекса:\n";
   int i;
   for (i = 1; i <= len; i++)
   {
      cout.write(state2,i);
      cout << endl;
   }
   // конкатенация вывода
   cout << "Цикл с убыванием индекса:\n";
   for (i = len; i > 0; i--)
      cout.write(state2,i) << endl;
   // превышение длины строки
   cout << "Превышение длины строки:\n";
   cout.write(state2, len + 5) << endl;
   return 0;
}
```

Некоторые компиляторы могут обнаружить, что программа объявляет, но не использует массивы state1 и state3. Все в порядке, потому что эти два массива показаны здесь для того, чтобы представить данные, находящиеся в памяти перед и после массивом state2, чтобы вы могли увидеть, что произойдет, если программа неправильно обратится к массиву state2. Вот вывод программы из листинга 17.1:

```
Цикл с возрастанием индекса:
K
Ka
Kan
Kans
Kansa
Kansas
Цикл с убыванием индекса:
Kansas
Kansa
Kans
Kan
Ka
K
Превышение длины строки:
Kansas Euph
```

Отметим, что вызов `cout.write()` возвращает объект `cout`. Это потому, что метод `write()` возвращает ссылку на объект, который его вызвал, и в данном случае это — объект `cout`.

Это позволяет выполнять конкатенацию вывода, поскольку `cout.write()` замещается возвращаемым значением `cout`:

```
cout.write(state2,i) << endl;
```

Также отметим, что метод `write()` не прекращает печать строки по достижении нулевого ограничивающего символа. Он просто печатает столько символов, сколько ему сказано, даже если при этом выходит за пределы конкретной строки! В данном случае программа выводит за строкой `"Kansas"` фрагмент другой строки, находящейся в памяти по соседству. Компиляторы могут по-разному размещать данные в памяти и по-разному выравнивать строки по границам 4 байт, поэтому `"Kansas"` дополняется до 8 байт. Некоторые компиляторы разместят `"Florida"` после `"Kansas"`. Поэтому из-за различий в компиляторах вы можете получить разный результат в последней строке вывода.

Метод `write()` также можно использовать с числовыми данными. Вы должны передать ему адрес числа, приведя его тип к `char *`:

```
long val = 560031841;
cout.write( (char *) &val, sizeof (long));
```

Это не транслирует число в корректные символы; вместо этого такой вызов передаст битовое представление того, что находится в памяти. Например, 4-байтное значение типа `long`, такое как `560031841`, будет передано как 4 отдельных байта. Выходное устройство, например, монитор, может затем попытаться интерпретировать каждый байт, как если бы он был ASCII-кодом (или каким-то другим). Поэтому `560031841` появится на экране в виде некоторой 4-символьной комбинации, скорее всего, бессмысленной (а может, и нет — попробуйте и посмотрите).

Однако `write()` обеспечивает компактный, аккуратный способ сохранения числовых данных в файле. Мы вернемся к этой возможности позднее в настоящей главе.

Сброс содержимого выходного буфера

Посмотрим, что случится, если программа использует `cout` для отправки байтов в стандартный вывод. Поскольку буфер класса `ostream` управляются объектом `cout`, вывод не отправляется немедленно по назначению. Вместо этого он накапливается в буфере до тех пор, пока не наполнит его. Затем программа *сбрасывает* буфер, отправляя по назначению его содержимое и очищая его для приема новых данных. Как правило, размер буфера составляет 512 байт, или же это число, умноженное на некоторое целое. Буферизация — великолепный способ экономии времени, когда стандартный вывод подключен к файлу на жестком диске. Кроме всего, вы не захотите, чтобы программа обращалась к жесткому диску 512 раз, чтобы отправить 512 байт. Гораздо эффективнее накопить эти 512 байт в буфере и записать их на диск за одну операцию.

Однако для экранного вывода первоначальное наполнение буфера нежелательно. В самом деле, было бы не удобно, если бы пришлось ждать появления сообщения вроде “Нажмите любую клавишу для продолжения” до тех пор, пока не буфер не заполнится 512-ю байтами. К счастью, в случае экранного вывода программе не обязательно ждать заполнения буфера. Отправка символа перевода строки, например,

обычно сбрасывает буфер. К тому же, как упоминалось ранее, большинство реализаций C++ сбрасывают буфер, когда ожидается ввод. То есть, предположим, что имеется следующий код:

```
cout << "Введите число: ";
float num;
cin >> num;
```

Тот факт, что программа ожидает ввода, заставляет ее отобразить сообщение cout (то есть сбросить из буфёра сообщение "Введите число: ") немедленно, даже несмотря на то, что выходная строка не содержит символа новой строки. Без этого средства программа бы ожидала ввода, не выдав на cout приглашения.

Если ваша реализация не сбрасывает вывод, когда вы ожидаете этого, можете сделать это принудительно, используя один из двух манипуляторов. Манипулятор flush просто сбрасывает буфер, а манипулятор endl сбрасывает буфер и вставляет символ перевода строки. Эти манипуляторы используются так же, как имена переменных:

```
cout << "Привет, красотка! " << flush;
cout << "Подожди минутку, пожалуйста." << endl;
```

Фактически манипуляторы являются функциями. Например, вы можете сбросить буфер cout, вызвав функцию flush() непосредственно:

```
flush(cout);
```

Однако класс ostream перегружает операцию вставки << таким образом, что выражение

```
cout << flush
```

заменяется вызовом функции flush(cout). То есть вы можете с успехом использовать более удобную нотацию вставки для сброса буфера.

Форматирование с помощью cout

Операции вставки в ostream преобразует значения в текстовую форму. По умолчанию они форматируют значения следующим образом:

- Значение типа char, если оно представляет печатаемый символ, отображается как символ в поле шириной в один символ.
- Целочисленные типы отображаются как десятичные целые в поле с шириной, достаточной для отображения числе и знака минус (если число отрицательное).
- Строки отображаются в поле ширины, равной длине строки.

Поведение по умолчанию для отображения чисел с плавающей точкой изменилось. Ниже описаны отличия между старыми и новыми реализациями C++:

- **Новый стиль.** Типы с плавающей точкой отображаются в шести разрядах (всего), за исключением завершающих нулей, которые не отображаются. (Отметим, что количество отображаемых разрядов не имеет отношения к точности, в которой хранится число.) Число либо отображается в нотации с фиксированной точкой, либо в научной экспоненциальной E-нотации (см. главу 3), в зависимости от значения числа. В частности, E-нотация используется, если экспонента числа больше 6 или меньше –5. Опять-таки, ширина поля выбирается такой,

чтобы вместить все цифры и, если он присутствует, знак минус. Поведение по умолчанию соответствует тому, которое имеет функция библиотеки C `fprintf()` со спецификатором `%g`.

- **Старый стиль**. Типы с плавающей точкой отображаются в шести разрядах справа от десятичной точки, за исключением завершающих нулей, которые не отображаются. (Отметим, что количество отображаемых разрядов не имеет отношения к точности, в которой хранится число.) Число либо отображается в нотации с фиксированной точкой, либо в научной экспоненциальной E-нотации (см. главу 3), в зависимости от значения числа. Опять-таки, ширина поля выбирается такой, чтобы вместить все цифры и, если он присутствует, знак минус.

Поскольку каждое значение отображается в рамках ширины, равной его размеру, пробелы между значениями вам придется указывать явно, в противном случае соседние выводимые значения сольются вместе.

Существует несколько небольших отличий между ранним форматированием C++ и форматированием, принятым в текущем стандарте C++. Все они резюмируются в табл. 17.3 далее в настоящей главе.

В листинге 17.2 иллюстрируются установки вывода по умолчанию. Код в нем отображает двоеточие (`:`) после каждого значения, чтобы вы могли видеть ширину поля, используемую в каждом случае. Программа использует выражение `1.0 / 9.0` для генерации бесконечной дроби, чтобы показать, сколько знаков после запятой будет напечатано.

Замечание по совместимости

Не все компиляторы генерируют вывод, отформатированный в соответствии с текущим стандартом C++. К тому же текущий стандарт допускает региональные вариации. Например, европейские реализации могут следовать континентальному стилю использования запятой вместо точки для отделения дробной части. То есть в них принято `2,54` вместо `2.54`. Библиотека локализации (заголовочный файл `locale`) представляет механизм оформления входного и выходного потоков в определенном стиле, поэтому один компилятор может представлять более одного варианта выбора локальных настроек. В настоящей главе используются установки для США.

Листинг 17.2. `defaults.cpp`

```
// defaults.cpp -- форматы по умолчанию cout
#include <iostream>
int main()
{
   using std::cout;
   cout << "12345678901234567890\n";
   char ch = 'K';
   int t = 273;
   cout << ch << ":\n";
   cout << t << ":\n";
   cout << -t <<":\n";

   double f1 = 1.200;
   cout << f1 << ":\n";
   cout << (f1 + 1.0 / 9.0) << ":\n";

   double f2 = 1.67E2;
   cout << f2 << ":\n";
   f2 += 1.0 / 9.0;
   cout << f2 << ":\n";
```

```
    cout << (f2 * 1.0e4) << ":\n";
    double f3 = 2.3e-4;
    cout << f3 << ":\n";
    cout << f3 / 10 << ":\n";
    return 0;
}
```

Вот результат работы программы из листинга 17.2:

```
12345678901234567890
K:
273:
-273:
1.2:
1.31111:
167:
167.111:
1.67111e+006:
0.00023:
2.3e-005:
```

Каждое значение заполняет свое поле. Обратите внимание, что завершающие нули в значении 1.200 не отображаются, но значения с плавающей точкой без завершающих нулей отображаются с шестью знаками после точки. Кроме того, данная конкретная реализация отображает три разряда в экспоненте; другие могут иметь два.

Изменение основания числа, используемого для отображения

Класс ostream унаследован от класса ios, который наследуется от ios_base. Класс ios_base хранит информацию, описывающую состояние формата. Например, определенные биты в одном члене класса определяют используемое основание числа, в то время как другие — ширину поля. Применяя *манипуляторы*, вы можете управлять основанием чисел для отображения целых. Используя функции-члены ios_base, можно управлять шириной поля и количеством знаков после запятой. Поскольку класс ios_base является непрямым базовым классом для ostream, вы можете использовать эти методы с объектами класса ostream и его наследников, например, с cout.

На заметку!

Члены и методы класса ios_base ранее находились в классе ios. Теперь же ios_base — это базовый класс для ios. В новой системе ios — шаблонный класс со специализациями char и wchar_t, а ios_base содержит нешаблонные средства.

Посмотрим, как устанавливается основание для отображения целых чисел. Чтобы установить отображение целых по основанию 10, 16 или 8, вы можете использовать манипуляторы dec, hex и oct. Например, вызов функции

```
hex(cout);
```

устанавливает шестнадцатеричный формат основания числа для потока cout. После того, как вы это сделаете, программа будет печатать целые значения в шестнадцатеричной форме до тех пор, пока не будет установлено другое основание. Отметим, что манипуляторы не являются функциями-членами, поэтому не должны вызываться объектом.

Хотя манипуляторы на самом деле являются функциями, обычно вы будете применять их следующим образом:

```
cout << hex;
```

Класс ostream перегружает операцию << таким образом, что эта операция эквивалентна вызову функции hex(cout). Манипуляторы находятся в пространстве имен std. В листинге 17.3 иллюстрируется применение этих манипуляторов. Он показывает значение целого числа и его квадрата в трех разных форматах оснований. Обратите внимание, что манипуляторы можно использовать раздельно — как часть последовательности вставок.

Листинг 17.3. manip.cpp

```
// manip.cpp -- использование манипуляторов формата
#include <iostream>
int main()
{
   using namespace std;
   cout << "Введите целое число: ";
   int n;
   cin >> n;
   cout << "n n*n\n";
   cout << n << " " << n * n << " (десятичное)\n";
   // установить шестнадцатеричный режим
   cout << hex;
   cout << n << " ";
   cout << n * n << " (шестнадцатеричное)\n";
   // установить восьмеричный режим
   cout << oct << n << " " << n * n << " (восьмеричное)\n";
   // альтернативный способ вызова манипулятора
   dec(cout);
   cout << n << " " << n * n << " (десятичное)\n";
   return 0;
}
```

Ниже показан пример выполнения этой программы:

```
Введите целое число: 13
n     n*n
13    169 (десятичное)
d     a9 (шестнадцатеричное)
15    251 (восьмеричное)
13    169 (десятичное)
```

Настройка ширины полей

Возможно, вы заметили, что столбец значений, выведенный программой из листинга 17.3, не выровнен. Это потому, что числа имеют разную ширину поля. Вы можете воспользоваться функцией-членом width для размещения чисел в полях постоянной ширины. Этот метод имеет следующие прототипы:

```
int width();
int width(int i);
```

Первая форма возвращает текущую установку ширины поля. Вторая устанавливает ширину поля равной i пробелам и возвращает предыдущее значение ширины. Это позволяет вам сохранять предыдущее значение — на случай, если позднее понадобится восстановить старое значение.

Метод width() касается только следующего элемента, который должен быть отображен, немедленно после этого восстанавливая значение по умолчанию. Например, рассмотрим следующие операторы:

```
cout << '#';
cout.width(12);
cout << 12 << "#" << 24 << "#\n";
```

Поскольку width() — функция-член, вы должны указать объект (в данном случае — cout) для ее вызова. В результате работы этого кода на дисплей будет выведено:

```
#          12#24#
```

Значение 12 помещается в поле 12 символов шириной, в правой части поля. Это называется правосторонним выравниванием. После этого ширина поля возвращается в значение по умолчанию, и два символа # и значение 24 печатаются в полях с шириной, равной их собственным значениям.

На память!

Метод width() касается только следующего отображаемого элемента, после чего ширина поля возвращается к значению по умолчанию.

C++ никогда не усекает данные, поэтому если вы попытаетесь напечатать семизначное число в поле шириной 2, то C++ расширит это поле, чтобы вместить данные. (Некоторые языки заполняют поле звездочками, если значение не помещается в заданную ширину. Философия C/C++ состоит в том, что показать полные данные важнее, чем сохранить красоту столбца; C++ ставит содержимое выше формы.) В листинге 17.4 демонстрирует работу функции-члена width().

Листинг 17.4. width.cpp

```
// width.cpp -- использование метода width
#include <iostream>
int main()
{
   using std::cout;
   int w = cout.width(30);
   cout << "ширина поля по умолчанию = " << w << ":\n";
   cout.width(5);
   cout << "N" <<':';
   cout.width(8);
   cout << "N * N" << ":\n";
   for (long i = 1; i <= 100; i *= 10)
   {
      cout.width(5);
      cout << i <<':';
      cout.width(8);
      cout << i * i << ":\n";
   }
   return 0;
}
```

Здесь — вывод программы из листинга 17.4:

```
       ширина поля по умолчанию = 0:
   N:  N * N:
   1:      1:
  10:    100:
 100: 10000:
```

Вывод отображает значения, выравнивая их по правому краю. При этом они слева дополняются пробелами. То есть cout достигает заданной ширины поля, добавляя пробелы. При выравнивании вправо пробелы вставляются слева от значения. Символ, используемый для этого, называется символом-заполнителем. По умолчанию действует выравнивание вправо.

Обратите внимание, что программа из листинга 17.4 применяет ширину поля 30, а не значение w к строке, отображаемой в первом операторе cout. Дело в том, что метод width() касается только следующего отображаемого элемента. Кроме того, отметим, что значение w равно 0. Это потому, что cout.width(30) возвращает предыдущее значение ширины, а не значение, которое вы устанавливаете. Тот факт, что w равно 0, означает, что значение ширины по умолчанию — 0. Поскольку C++ всегда расширяет поле, чтобы поместились все данные, это значение подходит для всех. И, наконец, программа использует width() для выравнивания заголовков столбцов и данных, применяя ширину в пять символов для первого столбца и восемь символов — для второго.

Символы-заполнители

По умолчанию cout заполняет неиспользуемые части поля пробелами. Вы можете применить функцию-член fill() для изменения этого. Например, вызов

```
cout.fill('*');
```

заменяет символ-заполнитель звездочкой. Это может быть удобно, скажем, для распечатки чеков, чтобы получатель не мог легко добавить разряд или два. В листинге 17.5 демонстрируется применение этой функции-члена.

Листинг 17.5. fill.cpp

```
// fill.cpp -- изменение символа-заполнителя полей

#include <iostream>
int main()
{
   using std::cout;
   cout.fill('*');
   const char * staff[2] = { "Waldo Whipsnade", "Wilmarie Wooper"};
   long bonus[2] = {900, 1350};

   for (int i = 0; i < 2; i++)
   {
      cout << staff[i] << ": $";
      cout.width(7);
      cout << bonus[i] << "\n";
   }
   return 0;
}
```

Вот вывод программы из листинга 17.5:

```
Waldo Whipsnade: $****900
Wilmarie Wooper: $***1350
```

Обратите внимание, что в отличие от ширины поля, новый символ-заполнитель остается в силе до тех пор, пока вы не замените его.

Установка отображаемой точности чисел с плавающей точкой

Значение *точности* плавающей точки зависит от режима вывода. В режиме по умолчанию оно означает общее количество отображаемых разрядов. В фиксированной и научной нотации, которые мы обсудим позднее, *точность* означает количество десятичных разрядов, отображаемых справа от точки. Точность по умолчанию, применяемая в C++, как вы уже видели, равна 6. (Вспомните, однако, что завершающие нули сбрасываются.) Функция-член `precision()` позволяет вам выбрать другие значения. Например, оператор

```
cout.precision(2);
```

устанавливает значение точности `cout` равным 2. В отличие от случая с `width()`, но подобно случаю `fill()`, новое значение точности остается в силе до тех пор, пока не будет сброшено. В листинге 17.6 демонстрируется это.

Листинг 17.6. precise.cpp

```
// precise.cpp -- установка точности
#include <iostream>

int main()
{
   using std::cout;
   float price1 = 20.40;
   float price2 = 1.9 + 8.0 / 9.0;

   cout << "\"Furry Friends\" is $" << price1 << "!\n";
   cout << "\"Fiery Fiends\" is $" << price2 << "!\n";

   cout.precision(2);
   cout << "\"Furry Friends\" is $" << price1 << "!\n";
   cout << "\"Fiery Fiends\" is $" << price2 << "!\n";

   return 0;
}
```

Замечание по совместимости

Старые версии C+ интерпретируют точность для режима по умолчанию, как число разрядов справа от точки, вместо общего числа разрядов.

Вывод программы листинга 17.6:

```
"Furry Friends" is $20.4!
"Fiery Fiends" is $2.78889!
"Furry Friends" is $20!
"Fiery Fiends" is $2.8!
```

Обратите внимание, что третья строка этого вывода не включает завершающей десятичной точки. К тому же, четвертая строка отображает всего два разряда.

Печать завершающих нулей и десятичных точек

Некоторые формы вывода, такие как цены или номера в столбцах, смотрятся лучше, если в них присутствуют завершающие нули. Например, вывод листинга 17.6 будет выглядеть лучше в виде `$20.40`, чем `$20.4`. В семействе классов `iostream` не предусмотрена функция, чье единственное назначение было бы обеспечивать это. Однако класс `ios_base` включает метод `setf()` (то есть *set flag*), управляющий некоторыми средствами форматирования. Этот класс также определяет несколько констант, которые могут использоваться в качестве аргументов этой функции.

Например, вызов функции

```
cout.setf(ios_base::showpoint);
```

заставляет `cout` отображать завершающие десятичные разряды. В формате с плавающей точкой по умолчанию это также принудительно отображает завершающие нули. То есть вместо отображения `2.00` как `2`, `cout` покажет его как `2.00000`, если значение по умолчанию будет установлено равным 6. В листинге 17.7 добавлен этот оператор.

Внимание!

Если ваш компилятор использует заголовочный файл `iostream.h` вместо `iostream`, то вероятно, вы должны будете применять `ios` вместо `ios_base` в аргументах `setf()`.

На случай, если вас удивит нотация `ios_base::showpoint`, заметим, что `showpoint` — это статическая константа контекста класса, определенная в объявлении класса `ios_base`. Контекст (область видимости) класса означает, что вы должны использовать операцию разрешения контекста (`::`) с именем константы, если применяете это имя вне определений функций-членов. Поэтому `ios_base::showpoint` именует константу, определенную в классе `ios_base`.

Листинг 17.7. showpt.cpp

```
// showpt.cpp -- установка точности с показом завершающей точки
#include <iostream>

int main()
{
   using std::cout;
   using std::ios_base;
   float price1 = 20.40;
   float price2 = 1.9 + 8.0 / 9.0;

   cout.setf(ios_base::showpoint);
   cout << "\"Furry Friends\" is $" << price1 << "!\n";
   cout << "\"Fiery Fiends\" is $" << price2 << "!\n";

   cout.precision(2);
   cout << "\"Furry Friends\" is $" << price1 << "!\n";
   cout << "\"Fiery Fiends\" is $" << price2 << "!\n";

   return 0;
}
```

Вывод программы из листинга 17.7 с применением текущего форматирования C++:

```
"Furry Friends" is $20.4000!
"Fiery Fiends" is $2.78889!
"Furry Friends" is $20.!
"Fiery Fiends" is $2.8!
```

Обратите внимание, что теперь завершающие нули не отображаются, но зато показана завершающая десятичная точка.

Дополнительные сведения о setf()

Метод setf() управляет несколькими другими установками формата помимо установки того, где должна отображаться десятичная точка; рассмотрим их повнимательнее. Класс ios_base имеет защищенный член данных, в котором отдельные биты (в данном контексте — *флаги*) управляют различными аспектами форматирования, такими как основание числа и признак необходимости отображения завершающих нулей. Включение такого флага называется *установкой флага* (или бита), и означает установку его значения в 1. (Битовые флаги — эквивалент DIP-переключателей для конфигурирования оборудования компьютера.) Манипуляторы hex, dec и oct, например, настраивают три битовых флага, управляющих выбором основания числа. Функция setf() предоставляет также доступ к другим битовым флагам.

Функция setf() имеет два прототипа. Первый из них выглядит следующим образом:

```
fmtflags setf(fmtflags);
```

Здесь fmtflags — заданный через typedef псевдоним типа bitmask (см. ниже врезку "На заметку!"), применяемый для хранения флагов форматирования. Имя определено в классе ios_base. Эта версия setf() используется для установки информации о формате, управляемой отдельными битами. Аргумент — значение типа fmtflags, указывающее, какие биты следует установить. Возвращаемое значение типа fmtflags — число, отражающее предыдущие значения всех флагов. Вы можете сохранить это значение, если хотите позднее восстановить исходные настройки всех флагов. Какое значение передавать setf()? Если вы хотите установить 11-й бит в 1, то передаете число, у которого 11-й бит выставлен в 1. Возвращаемое значение будет иметь в бите номер 11 старое его значение. Необходимость отслеживать биты кажется утомительной. Однако вы не обязаны выполнять эту работу. Класс ios_base определяет константы, которые представляют битовые значения. В табл. 17.1 показаны некоторые из этих определений.

На заметку!

Тип *битовой маски* предназначен для хранения индивидуальных битовых значений. Он может быть целочисленным, перечислением или контейнером STL по имени bitset. Главная идея в том, что каждый бит доступен индивидуально и имеет свое собственное значение. Пакет iostream использует типы битовых масок для хранения информации о состоянии потока.

Таблица 17.1. Форматирующие константы

Константа	Значение
ios_base::boolalpha	Ввод и вывод значений bool, таких как true и false.
ios_base::showbase	Использовать при выводе префикс основания C++ (0, 0x).
ios_base::showpoint	Показывать завершающую десятичную точку.
ios_base::uppercase	Использовать буквы верхнего регистра для шестнадцатеричного вывода и E-нотации.
ios_base::showpos	Указывать + перед положительными значениями.

Поскольку эти форматирующие константы определены в классе ios_base, вы должны применять с ними операцию разрешения контекста. То есть нужно писать ios_base::uppercase, а не просто uppercase. Если вы не предусматриваете директиву using или объявление using, то должны применять операцию разрешения контекста, чтобы указать, что эти имена принадлежат пространству имен std. То есть, вы можете использовать std::ios_base::showpos и тому подобное. Изменения остаются в силе до тех пор, пока не будут переопределены. В листинге 17.8 демонстрируется применение некоторых из этих констант.

Листинг 17.8. setf.cpp

```
// setf.cpp -- использование setf() для управления форматированием
#include <iostream>
int main()
{
   using std::cout;
   using std::endl;
   using std::ios_base;
   int temperature = 63;
   cout << "Сегодня температура воды: ";
   cout.setf(ios_base::showpos);         // показывать знак плюс
   cout << temperature << endl;
   cout << "Для наших друзей-программистов, это будет \n";
   cout << std::hex << temperature << endl;// применение шестнадцатеричного
                                         // формата
   cout.setf(ios_base::uppercase);       // применение шестнадцатеричного
                                         // формата в верхнем регистре
   cout.setf(ios_base::showbase);        // использование префикса 0X для
                                         // шестнадцатеричных значений
   cout << "или\n";
   cout << temperature << endl;
   cout << "Как " << true << "! ой -- как ";
   cout.setf(ios_base::boolalpha);
   cout << true << "!\n";
   return 0;
}
```

Замечание по совместимости

Некоторые реализации C+ могут применять ios вместо ios_base, и они могут давать сбой при выборе boolalpha.

Вывод программы листинга 17.8 выглядит следующим образом:

```
"Сегодня температура воды: +63
Для наших друзей-программистов, это будет
3f
или
0X3F
Как 0X1! ой -- как true!
```

Обратите внимание, что знак + применяется только с основанием 10. C++ трактует шестнадцатеричные и восьмеричные значения как беззнаковые, поэтому для них указывать знак не нужно. (Хотя некоторые реализации C++ могут отображать и в этом случае знак +.)

Второй прототип `setf()` принимает два аргумента и возвращает предыдущие установки:

```
fmtflags setf(fmtflags, fmtflags);
```

Эта перегруженная форма функции используется для установок форматирования, управляемых более чем 1 битом. Первый аргумент, как и ранее — значение `fmtflags`, которое содержит требуемые установки. Второй аргумент — это значение, которое очищает соответствующие биты. Например, предположим, что установка третьего бита в 1 означает основание 10, установка четвертого бита в 1 означает основание 8, а установка пятого бита — основание 16. Предположим, что вывод идет с основанием 10, а вы хотите переключить его на основание 16. Вам не только нужно установить 5-й бит в единицу, но также 3-й — в ноль (это означает *очистить* бит). Умный манипулятор `hex` выполняет обе задачи автоматически. Применение функции `setf()` требует немного больших усилий, потому что вы используете второй аргумент для указания очищаемых битов, а затем первый аргумент — для указания устанавливаемых битов. Это не так сложно, как кажется, потому что класс `ios_base` определяет для этой цели константы (перечисленные в табл. 17.2). В частности, изменяя основание числа, вы должны применять константу `ios_base::basefield` в качестве второго аргумента, и `ios_base::hex` — в качестве первого аргумента. То есть, вызов функции

```
cout.setf(ios_base::hex, ios_base::basefield);
```

даст тот же эффект, что и применение манипулятора `hex`.

Таблица 17.2. Аргументы для `setf(long, long)`

Второй аргумент	Первый аргумент	Значение
`ios_base::basefield`	`ios_base::dec`	Использовать основание 10.
	`ios_base::oct`	Использовать основание 8.
	`ios_base::hex`	Использовать основание 16.
`ios_base::floatfield`	`ios_base::fixed`	Использовать нотацию с фиксированной точкой.
	`ios_base::scientific`	Использовать научную нотацию.
`ios_base::adjustfield`	`ios_base::left`	Использовать выравнивание влево.
	`ios_base::right`	Использовать выравнивание вправо.
	`ios_base::internal`	Выровненный влево знак и префикс основания, выровненное вправо значение.

Класс ios_base определяет три набора форматирующих флагов, которые могут быть обработаны подобным образом. Каждый набор состоит их одной константы, которая может быть использована как второй аргумент, и двух или трех констант, применяемых в качестве первого аргумента. Второй аргумент очищает пакет связанных битов, затем первый аргумент устанавливает один их этих бит в 1. В табл. 17.2 перечислены имена констант, используемых для второго аргумента setf(), ассоциированный с ними выбор констант для первого аргумента, а также их значения. Например, чтобы выбрать выравнивание влево, вы применяете ios_base::adjustfield для второго аргумента и ios_base::left — для первого. Выравнивание влево означает начать вывод с левой границы поля, а выравнивание вправо — завершить вывод значения на правой границе поля. Внутреннее выравнивание означает размещение знака и префикса основания у левой границы поля, а само значение — у правой его границы (к сожалению, C++ не предоставляет самовыравнивающего режима).

Нотация с фиксированной точкой означает применение стиля 123.4 для значений с плавающей точкой, независимо от размера числа. Если вы знакомы со спецификаторами функции C printf(), это может помочь вам понять, что режим, используемый C++ по умолчанию для вывода чисел с плавающей точкой, соответствует спецификатору %g, режим fixed — спецификатору %f, а scientific — спецификатору %e.

В соответствии со стандартом C++, как фиксированная, так и научная нотация обладают следующими двумя свойствами:

- *Точность* означает количество десятичных разрядов справа от точки, а не общее число разрядов.
- Завершающие нули отображаются.

Ранее завершающие нули не отображались, если только не был установлен флаг ios::showpoint. К тому же, в старых реализациях точность всегда означала количество десятичных разрядов справа от точки, даже в режиме по умолчанию.

Функция setf() — это функция-член класса ios_base. Поскольку это — базовый класс для класса ostream, вы можете вызывать ее с объектом cout. Например, чтобы запросить выравнивание влево, можно воспользоваться вызовом:

```
ios_base::fmtflags old = cout.setf(ios::left, ios::adjustfield);
```

А чтобы восстановить предыдущие настройки, применить такое:

```
cout.setf(old, ios::adjustfield);
```

В листинге 17.9 показаны дополнительные примеры применения setf() с двумя аргументами.

Замечание по совместимости

Программа в листинге 17.9 использует математическую функцию, а некоторые системы C++ не ищут автоматически библиотеку math. Например, некоторые системы Unix требуют, чтобы обращались к компилятору следующим образом:

```
$ CC setf2.C -lm
```

Опция -lm инструктирует компоновщик о том, где искать математическую библиотеку. Аналогично, некоторые системы Linux, использующие g++, требуют того же флага.

Листинг 17.9. setf2.cpp

```
// setf2.cpp -- использование setf() с двумя аргументами для управления
форматированием
#include <iostream>
#include <cmath>
int main()
{
   using namespace std;
   // использовать выравнивание влево, показать знак плюс,
   // показать завершающие пробелы с точностью 3
   cout.setf(ios_base::left, ios_base::adjustfield);
   cout.setf(ios_base::showpos);
   cout.setf(ios_base::showpoint);
   cout.precision(3);
   // использовать е-нотацию и сохранить старые установки формата
   ios_base::fmtflags old = cout.setf(ios_base::scientific,
   ios_base::floatfield);
   cout << "Выравнивание влево:\n";
   long n;
   for (n = 1; n <= 41; n+= 10)
   {
      cout.width(4);
      cout << n << "|";
      cout.width(12);
      cout << sqrt(double(n)) << "|\n";
   }
   // переключиться на внутреннее выравнивание
   cout.setf(ios_base::internal, ios_base::adjustfield);
   // восстановить стиль отображения по умолчанию для плавающей точки
   cout.setf(old, ios_base::floatfield);
   cout << "Внутреннее выравнивание:\n";
   for (n = 1; n <= 41; n+= 10)
   {
      cout.width(4);
      cout << n << "|";
      cout.width(12);
      cout << sqrt(double(n)) << "|\n";
   }
   // использовать правое выравнивание, фиксированную нотацию
   cout.setf(ios_base::right, ios_base::adjustfield);
   cout.setf(ios_base::fixed, ios_base::floatfield);
   cout << "Выравнивание вправо:\n";
   for (n = 1; n <= 41; n+= 10)
   {
      cout.width(4);
      cout << n << "|";
      cout.width(12);
      cout << sqrt(double(n)) << "|\n";
   }
   return 0;
}
```

Ниже показан результат работы программы из листинга 17.9:

```
Выравнивание влево:
+1  |+1.000e+00 |
+11 |+3.317e+00 |
+21 |+4.583e+00 |
+31 |+5.568e+00 |
+41 |+6.403e+00 |
Внутреннее выравнивание:
+  1|+     1.00|
+ 11|+     3.32|
+ 21|+     4.58|
+ 31|+     5.57|
+ 41|+     6.40|
Выравнивание вправо:
 +1|     +1.000|
+11|     +3.317|
+21|     +4.583|
+31|     +5.568|
+41|     +6.403|
```

Обратите внимание, как точность 3 влияет на отображение по умолчанию значений с плавающей точкой (используется в этой программе для внутреннего выравнивания), выводящее всего три разряда, тогда как в режиме фиксированной и научной нотации отображается три разряда после точки. (Количество разрядов, отображенное в экспоненте для E-нотации зависит от реализации.)

Эффект от вызова `setf()` может быть отменен вызовом функции `unsetf()`, которая имеет следующий прототип:

```
void unsetf(fmtflags mask);
```

Здесь `mask` — битовый шаблон. Все биты `mask`, установленные в `1`, соответствуют битам, которые должны быть очищены. То есть, `setf()` устанавливает биты в `1`, а `unsetf()` устанавливает их в `0`. Вот пример:

```
cout.setf(ios_base::showpoint); // показать завершающую десятичную точку
cout.unsetf(ios_base::boolalpha);   // не показывать завершающую
                                    // десятичную точку
cout.setf(ios_base::boolalpha);     // отображать true, false
cout.unsetf(ios_base::boolalpha);   // отображать 1, 0
```

Возможно, вы обратите внимание на то, что не существует специального флага для указания режима по умолчанию для отображения значений с плавающей точкой. Дело в том, что система работает следующим образом: фиксированная нотация применяется, если установлен бит фиксированной нотации, и только этот бит. Научная нотация применяется, если установлен бит научной нотации, и только он. Любая другая комбинация, например, когда не установлен никакой из них, приводит к использованию режима по умолчанию. Поэтому одним из способов включить режим по умолчанию может быть такой:

```
cout.setf(0, ios_base::floatfield); // перейти к режиму по умолчанию
```

Второй аргумент выключает оба бита, а первый аргумент не устанавливает ни одного. Более краткий способ достижения того же эффекта может заключаться в применении `unsetf()` с `ios_base::floatfield`:

```
cout.unsetf(ios_base::floatfield);  // перейти к режиму по умолчанию
```

Если вы знали, что cout находился в режиме фиксированной точки, то можете использовать ios_base::fixed как аргумент unsetf(), но применение ios_base::floatfield работает независимо от текущего состояния cout, поэтому это — лучшее решение.

Стандартные манипуляторы

Применение setf() — не слишком дружественный подход к форматированию, поэтому C++ предлагает несколько манипуляторов, вызывающих setf() за вас, автоматически передавая правильные аргументы. Вы уже видели dec, hex и oct. Эти манипуляторы, большинство которых не были доступны в старых реализациях C++, работают как hex. Например, оператор

```
cout << left << fixed;
```

переключает режим вывода на фиксированную десятичную точку, с левым выравниванием. В табл. 17.3 перечислены некоторые стандартные манипуляторы.

Совет

Если ваша система поддерживает эти манипуляторы, воспользуйтесь их преимуществами. Если же нет, у вас есть возможность применять setf().

Таблица 17.3. Некоторые стандартные манипуляторы

Манипулятор	Вызывает
boolalpha	setf(ios_base::boolalpha)
noboolalpha	unset(ios_base::noboolalpha)
showbase	setf(ios_base::showbase)
noshowbase	unsetf(ios_base::showbase)
showpoint	setf(ios_base::showpoint)
noshowpoint	unsetf(ios_base::showpoint)
showpos	setf(ios_base::showpos)
noshowpos	unsetf(ios_base::showpos)
uppercase	setf(ios_base::uppercase)
nouppercase	unsetf(ios_base::uppercase)
internal	setf(ios_base::internal, ios_base::adjustfield)
left	setf(ios_base::left, ios_base::adjustfield)
right	setf(ios_base::right, ios_base::adjustfield)
dec	setf(ios_base::dec, ios_base::basefield)
hex	setf(ios_base::hex, ios_base::basefield)
oct	setf(ios_base::oct, ios_base::basefield)
fixed	setf(ios_base::fixed, ios_base::floatfield)
scientific	setf(ios_base::scientific, ios_base::floatfield)

Заголовочный файл iomanip

Устанавливая некоторые значения форматирования, такие как ширина поля, неудобно использовать средства iostream. Чтобы облегчить жизнь, в C++ предлагает дополнительные манипуляторы в заголовочном файле iomanip. Они обеспечивают тот же сервис, что и рассмотренный выше, но в более удобной манере. Три из них, которые наиболее часто используются — это setprecision() для установки точности, setfill() для установки символа-заполнителя и setw() для установки ширины поля. В отличие от манипуляторов, которые мы рассмотрели выше, эти принимают аргументы. Манипулятор setprecision() принимает целочисленный аргумент, задающий точность, setfill() принимает аргумент char, указывающий символ-заполнитель, а setw() принимает целочисленный аргумент, устанавливающий ширину поля. Так как все они являются манипуляторами, их можно включать в операторы с cout. В частности, это обеспечивает удобное применение манипулятора setw() при отображении нескольких столбцов значений. Код в листинге 17.10 иллюстрирует это, изменяя ширину поля и символ-заполнитель несколько раз при выводе одной строки. Он также применяет некоторые из новых стандартных манипуляторов.

Замечание по совместимости

Программа в листинге 17.10 использует математическую функцию, а некоторые системы C++ не ищут автоматически библиотеку math. Например, некоторые системы Unix требуют, чтобы вы указывали компилятору:

```
$ CC setf2.C -lm
```

Опция -lm инструктирует компоновщик о том, где искать математическую библиотеку. Некоторые системы Linux, использующие g++, требуют того же флага. Также старые компиляторы могут не распознавать новые стандартные манипуляторы наподобие showpoint. В этом случае применяйте их эквиваленты setf().

Листинг 17.10. iomanip.cpp

```
// iomanip.cpp -- использование манипуляторов из iomanip
// некоторые манипуляторы требуют явной компоновки математической библиотеки
#include <iostream>
#include <iomanip>
#include <cmath>
int main()
{
   using namespace std;
   // использование новых стандартных манипуляторов
   cout << showpoint << fixed << right;
   // использование манипуляторов iomanip
   cout << setw(6) << "N" << setw(14) << "Квадратный корень"
        << setw(15) << "Корень 4-й степени\n";
   double root;
   for (int n = 10; n <=100; n += 10)
   {
      root = sqrt(double(n));
      cout << setw(6) << setfill('.') << n << setfill(' ')
           << setw(12) << setprecision(3) << root
           << setw(14) << setprecision(4) << sqrt(root)
```

Вот как выглядит результат работы программы из листинга 17.10:

```
      N    Квадратный корень    Корень 4-й степени
....10                3.162                1.7783
....20                4.472                2.1147
....30                5.477                2.3403
....40                6.325                2.5149
....50                7.071                2.6591
....60                7.746                2.7832
....70                8.367                2.8925
....80                8.944                2.9907
....90                9.487                3.0801
...100               10.000                3.1623
```

Теперь вы можете получить почти выровненные столбцы. Обратите внимание, что эта программа обеспечивает одинаковое форматирование как в старых, так и в современных реализациях. Использование манипулятора showpoint в старых реализациях вызывает отображение завершающих пробелов, а в современных они отображаются после манипулятора fixed. Применение fixed заставляет отображать значения с фиксированной точкой во всех системах, но в современных при этом точность интерпретируется как количество знаков после запятой, а в старых точность всегда интерпретировалась подобным образом, независимо от режима отображения плавающей точки.

В табл. 17.4 резюмируется некоторая разница между старым форматированием C++ и текущим состоянием.

Мораль этой таблицы в том, что вы не должны чувствовать растерянность, если при запуске программы в своей системе увидите, что формат вывода отличается от приведенного в примере.

Таблица 17.4. Изменения в формате

Средство	Значение в старом C++	Значение в новом C++
precision(n)	Отображается n разрядов после десятичной точки.	Отображается всего n разрядов в режиме по умолчанию и n разрядов справа от десятичной точки в фиксированном и научном режиме.
ios::showpoint	Отображается завершающая десятичная точка и завершающие нули.	Отображается завершающая десятичная точка.
ios::fixed ios::scientific		Отображаются завершающие нули (также см. комментарии о precision(n)).

Ввод с помощью cin

Теперь обратимся к вводу и передаче данных в программу. Объект cin представляет стандартный ввод как поток байтов. Обычно вы генерируете этот поток байтов, пользуясь клавиатурой. Если вы вводите последовательность символов 2005, то объект cin извлекает эти символы из входного потока. Можете представлять себе, что ввод является частью строки, значением типа int, типа float или некоторого другого типа. Таким образом, извлечение символов из потока также предполагает

преобразования типа. Объект cin на основании типа целевой переменной, куда принимается значение, должен использовать свои методы для преобразования последовательности символов в значения соответствующего типа.

Обычно вы используете cin следующим образом:

```
cin >> value_holder;
```

Здесь value_holder идентифицирует местоположение в памяти, куда помещается ввод. Это может быть именем переменной, ссылкой, разыменованным указателем либо членом структуры или класса. Как cin интерпретирует ввод — зависит от типа данных value_holder. Класс istream, определенный в заголовочном файле iostream, перегружает операцию извлечения >> для распознавания следующих базовых типов:

- signed char &
- unsigned char &
- char &
- short &
- unsigned short &
- int &
- unsigned int &
- long &
- unsigned long &
- float &
- double &
- long double &

Их называют *функциями форматированного ввода*, потому что они преобразуют входные данные в соответствии с переменной назначения.

Обычная функция операции имеет прототип следующего вида:

```
istream & operator>> (int &);
```

Как аргумент, так и возвращаемое значение являются ссылками. Когда аргумент — ссылка (см. главу 8), то оператор вроде

```
cin >> staff_size;
```

заставляет функцию operator>>() работать с самой переменной staff_size, а не с ее копией, как это бывает с обычным аргументом. Поскольку тип аргумента является ссылкой, cin в состоянии непосредственно модифицировать переменную-аргумент. Предшествующий оператор, например, напрямую модифицирует значение переменной staff_size. Мы поговорим о значении возврата по ссылке немного позже. Во-первых, давайте исследуем аспект преобразования типа операцией извлечения. Для обработки аргументов каждого типа из приведенного выше списка операция извлечения преобразует символьный тип в значение указанного типа. Например, предположим, что staff_size имеет тип int. В этом случае компилятор ставит в соответствие

```
cin >> staff_size;
```

такой прототип:

```
istream & operator>>(int &);
```

Функция, соответствующая прототипу затем читает поток символов, присланный в программу — скажем, символы 2, 3, 1, 8 и 4. Для систем, использующих 2-байтный тип int, функция преобразует эти символы в 2-байтное двоичное представление значения 23184. С другой стороны, если staff_size будет иметь тип double, то cin использует operator>>(double &), чтобы преобразовать тот же ввод в 8-байтное представление значения с плавающей точкой 23184.0.

Кстати, вы можете использовать манипуляторы hex, oct и dec с cin для указания того, что вводимое целое должно интерпретироваться в шестнадцатеричном, восьмеричном или десятичном формате. Например, оператор

```
cin >> hex;
```

заставляет ввод 12 или 0x12 восприниматься как шестнадцатеричное значение 12, или десятичное 18, а ff или FF читается как десятичное 255.

Класс istream также перегружает операцию извлечения >> для типов указателей на символы:

- signed char *
- char *
- unsigned char *

Для аргументов этих типов операция извлечения читает следующее слово из входного потока и помещает его по указанному адресу, добавляя нулевой байт для ограничения строки. Например, предположим, что у вас есть такой код:

```
cout << "Введите ваше имя:\n";
char name[20];
cin >> name;
```

Если вы ответите на запрос, введя Liz, то операция извлечения поместит символы Liz\0 в массив name. (Как обычно, \0 представляет ограничивающий нулевой символ.) Идентификатор name, будучи именем массива char, действует как адрес первого элемента массива и имеет тип char * (указатель на массив char).

Тот факт, что каждая операция извлечения возвращает ссылку на вызывающий объект, позволяет выполнять конкатенацию ввода, подобно тому, как это делается в отношении вывода:

```
char name[20];
float fee;
int group;
cin >> name >> fee >> group;
```

Здесь, к примеру, объект cin, возвращенный cin >> name, становится тем объектом, который принимает fee.

Как cin >> воспринимает ввод

Различные версии операции извлечения разделяют общий способ просмотра входного потока. Они пропускают пробельные символы (пробелы, переводы строк, табуляции) до тех пор, пока не встретится непробельный символ. Это действительно даже для односимвольных режимов (когда аргумент имеет тип char, unsigned char или signed char), что вовсе не так, если речь идет о функциях ввода языка C (рис. 17.5). В односимвольных режимах операция >> читает символ и присваивает указанному целевому объекту. То есть она читает все, начиная с первого непробельного символа до первого символа, который не соответствует типу объекта назначения.

```
char philosophy[20];
int distance;
char initial;

cin >> philosophy >> distance >> initial;
```

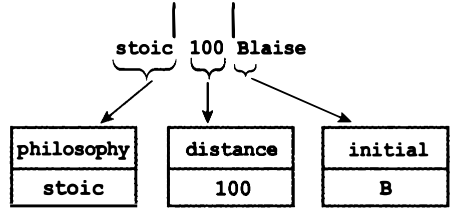


Рис. 17.5. `cin >>` *пропускает пробельные символы*

Например, рассмотрим следующий код:

```
int elevation;
cin >> elevation;
```

Предположим, вы набираете такие символы:

```
-123Z
```

Операция прочитает символы `-`, `1`, `2` и `3`, потому что они являются корректной частью целого числа. Но символ `Z` таковым не является, поэтому последним принятым символом будет `3`. `Z` останется во входном потоке, и следующий оператор `cin` начнет чтение с этой точки. Тем временем, операция преобразует последовательность символов `-123` в целое значение и присвоит его `elevation`.

Может случиться, что ввод не оправдает ожиданий программы. Например, предположим, что вы ввели `Zcar` вместо `-123Z`. В этом случае операция извлечения оставит значение переменной `elevation` без изменений и вернет значение `0`. (Точнее, оператор `if` или `while` оценивает объект `ifstream` как `false`, если будет установлено состояние ошибки; мы подробно обсудим это дальше в этой главе.) Возвращаемое значение `false` позволит программе проверить, отвечает ли ввод требованиям программы, как показано в листинге 17.11.

Листинг 17.11. check_it.cpp

```
// check_it.cpp -- проверка корректности ввода
#include <iostream>

int main()
{
   using namespace std;
   cout << "Введите числа: ";
   int sum = 0;
   int input;
   while (cin >> input)
   {
      sum += input;
   }
```

```
    cout << "Последнее введенное значение = " << input << endl;
    cout << "Сумма = " << sum << endl;
    return 0;
}
```

Замечание по совместимости

Если компилятор не поддерживает манипуляторы showpoint и fixed, вы можете применить эквивалент setf().

Ниже показан вывод программы из листинга 17.11, когда во входной поток проскакивает некоторое неподходящее значение (-123Z).

```
Введите числа: 200
10 -50 -123Z 60
Последнее введенное значение = -123
Сумма = 37
```

Поскольку ввод буферизуется, вторая строка значений, введенных с клавиатуры, не посылается программе до тех пор, пока вы не нажмете клавишу <Enter> в конце строки. Но цикл прекращает обработку на символе Z, потому что он не соответствует формату чисел с плавающей точкой. Несоответствие ввода ожидаемому формату, в свою очередь, приводит к тому, что выражение cin >> input оценивается как false, что прерывает цикл while.

Состояния потока

Посмотрим внимательнее на то, что происходит при несоответствующем вводе. Объект cin или cout содержит член данных (унаследованный от класса ios_base), описывающий *состояние потока*. Состояние потока (определенное как iostate типа битовой маски, описанной ранее) состоит из трех элементов ios_base: eofbit, badbit и failbit.

Каждый элемент — это отдельный бит, который может принимать значение 1 (установлен) или 0 (сброшен). Когда операция cin достигает конца файла, она устанавливает eofbit в значение 1. Когда операция cin не может прочитать ожидаемые символы, как в предыдущем примере, она устанавливает failbit в 1. Сбои ввода-вывода, такие как попытка чтения недоступного файла или попытка записи на защищенный от записи диск, также может установить failbit в 1.

Элемент badbit устанавливается в 1, когда происходит некоторый не поддающийся диагностике сбой, который может повредить поток. (Конкретные реализации не всегда согласны между собой в том, какие события должны устанавливать failbit, и какие — badbit.) Когда все эти три бита состояния установлены в 0, все в порядке. Программы могут проверять состояния потока и использовать эту информацию, чтобы решить, что делать дальше.

В табл. 17.5 перечислены эти биты наряду с некоторыми методами ios_base, которые сообщают о состоянии либо изменяют его. (Старые компиляторы не представляют два метода exceptions().)

Таблица 17.5. Состояния потока

Член	Описание
eofbit	Устанавливается в 1 по достижении конца файла.
badbit	Устанавливается в 1, если поток может быть поврежден; например, в случае ошибки чтения файла.
failbit	Устанавливается в 1, если операция ввода не смогла прочитать ожидаемые символы, или операция вывода не смогла их записать.
goodbit	Просто другой способ выразить 0.
good()	Возвращает true, если поток может быть использован (все биты очищены).
eof()	Возвращает true, если установлен eofbit.
bad()	Возвращает true, если установлен badbit.
fail()	Возвращает true, если установлен badbit или failbit.
rdstate()	Возвращает состояние потока.
exceptions()	Возвращает битовую маску, идентифицирующую флаги, послужившие причиной исключения.
exceptions(iostate ex)	Устанавливает состояния, которые вызовут clear() для генерации исключения; например, если ex — это eofbit, то clear() возбудит исключение, когда eofbit установлен.
clear(iostate s)	Устанавливает состояние потока в s; по умолчанию s есть 0 (goodbit); бросает исключение basic_ios::failure, если rdstate() & exceptions()) != 0
setstate(iostate s)	Вызывает clear(rdstate() | s). Это устанавливает биты состояния в соответствии с теми, что содержит s; остальные биты состояния потока остаются неизменными.

Установка состояния

Два метода из табл. 17.5 — clear() и setstate() — идентичны. Оба сбрасывают состояние, но делают это разными способами. Метод clear() устанавливает состояние в соответствии с переданным ему аргументом. То есть, вызов

```
clear();
```

использует аргумент по умолчанию 0, который очищает все три бита состояния (eofbit, badbit и failbit). Аналогично вызов

```
clear(eofbit);
```

делает состояние эквивалентным eofbit; то есть eofbit устанавливается, а прочие два бита очищаются.

Метод setstate(), однако, касается только тех бит, которые установлены в его аргументе. То есть вызов

```
setstate(eofbit);
```

устанавливает eofbit, не затрагивая остальных битов. Поэтому если failbit уже установлен, он таковым остается.

Зачем вам может понадобиться сбрасывать состояние потока? Для автора программ наиболее частая причина использовать `clear()` без аргументов — необходимость повторного открытия ввода после получения неподходящих данных или конца файла. Когда это имеет смысл делать — зависит от того, чего должна достичь программа. Скоро вы увидите некоторые примеры. Главное назначение `setstate()` в том, чтобы функции ввода и вывода могли изменять состояние. Например, если `num` имеет тип `int`, то вызов

```
cin >> num;  // прочитать int
```

в результате преобразуется в функцию `operator>>(int &)`, использующую `setstate()` для установки `failbit` или `eofbit`.

Ввод-вывод и исключения

Предположим, что функция ввода установила `eofbit`. Должно ли это сгенерировать исключение? По умолчанию ответ — нет. Однако вы можете использовать метод `exceptions()`, чтобы управлять тем, как будут работать исключения.

Сначала некоторая предварительная информация. Метод `exceptions()` возвращает битовое поле с тремя битами, соответствующими `eofbit`, `failbit` и `badbit`. Изменение состояния потока включает как `clear()`, так и `setstate()`, который использует `clear()`. После изменения состояния потока метод `clear()` сравнивает его текущее состояние со значением, возвращенным `exceptions()`. Если бит установлен в возвращаемом значении, и соответствующий бит установлен также в текущем состоянии, то `clear()` возбуждает исключение `ios_base::failure`. Это может случиться, например, если оба значения имеют установленный бит `badbit`. Отсюда следует, что если `exceptions()` возвращает `goodbit`, то никакого исключения не генерируется. Класс `ios_base::failure` унаследован от класса `std::exception`, а потому имеет метод `what()`.

Установкой по умолчанию для `exceptions()` является `goodbit` — то есть никаких исключений не генерировать. Однако перегруженная функция `exceptions(iostate)` дает вам возможность управлять этим поведением:

```
cin.exceptions(badbit); // установка badbit вызывает генерацию исключения
```

Битовая операция ИЛИ (`|`), как описано в приложении Д, позволяет специфицировать более одного бита. Например, оператор

```
cin.exceptions(badbit | eofbit);
```

вызывает генерацию исключения, если оба бита — и `badbit`, и `eofbit` — последовательно установлены.

Код в листинге 17.12 является модификацией кода из листинга 17.11, в результате которой программа возбуждает и перехватывает исключение, если установлен бит `failbit`.

Листинг 17.12. cinexcp.cpp

```
// cinexcp.cpp -- cin, возбуждающий исключения
#include <iostream>
#include <exception>
int main()
{
   using namespace std;
```

```
   // failbit вызовет генерацию исключения
   cin.exceptions(ios_base::failbit);
   cout << "Вводите числа: ";
   int sum = 0;
   int input;
   try {
         while (cin >> input)
         {
            sum += input;
         }
   } catch(ios_base::failure & bf)
   {
      cout << bf.what() << endl;
      cout << "О ужас!\n";
   }
   cout << "Последнее введенное значение = " << input << endl;
   cout << "Сумма = " << sum << endl;
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 17.12; сообщение, выдаваемое `what()`, зависит от реализации:

```
Вводите числа: 20 30 40 pi 6
ios_base failure in clear
О ужас!
Последнее введенное значение = 40.00
Сумма = 90.00
```

Таким образом вы можете использовать исключения при вводе. Но должны ли вы использовать их? Это зависит от контекста. В этом примере ответом будет — нет. Исключения должны отлавливать необычные, непредвиденные ситуации, но эта конкретная программа использует несоответствие типа как способ выхода из цикла. Однако для этой программы может иметь смысл генерировать исключение для `badbit`, когда возникает непредвиденная ситуация. Или же если программа предназначена для чтения данных из файла до момента достижения его конца, тогда, может быть, имеет смысл возбудить исключение для `failbit`, поскольку это будет означать проблему с файлом данных.

Эффекты состояния потока

Проверка `if` или `while` наподобие

```
while (cin >> input)
```

возвращает `true`, только если состояние потока нормальное (все биты очищены). Если проверка возвращает `false`, вы можете использовать функции-члены из табл. 17.5 для точного определения причины. Например, центральную часть листинга 17.11 можно модифицировать следующим образом:

```
while (cin >> input)
{
   sum += input;
}
if (cin.eof())
   cout << "Цикл прерван по причине EOF\n";
```

Установка битов состояния потока имеет очень важное следствие: поток закрывается для дальнейшего ввода или вывода до тех пор, пока бит не будет очищен. Например, следующий код не будет работать:

```
while (cin >> input)
{
   sum += input;
}
cout << "Последнее введенное значение = " << input << endl;
cout << "Сумма = " << sum << endl;
cout << "Введите новое число: ";
cin >> input; // не работает
```

Если вы хотите, чтобы программа продолжала читать ввод после того, как были установлены биты состояния потока, вы должны сбросить его состояние в нормальное. Это можно сделать вызовом метода `clear()`:

```
while (cin >> input)
{
   sum += input;
}
cout << "Последнее введенное значение = " << input << endl;
cout << "Сумма = " << sum << endl;
cout << "Введите новое число: ";
cin.clear();  // сбросить состояние потока
while (!isspace(cin.get()))
   continue;  // освободиться от плохого ввода
cin >> input;  // теперь будет работать
```

Обратите внимание, что сбросить состояние потока недостаточно. Некорректный ввод, который прервал цикл ввода, по-прежнему остается во входной очереди, и программа должна обработать его. Один из способов сделать это — продолжать чтение до достижения пробела. Функция `isspace()` (см. главу 6) — это функция `cctype`, которая возвращает `true`, если ее аргумент является пробельным символом. Или же вы можете отбросить остаток строки, а не следующее слово:

```
while (cin.get() != '\n')
   continue;  // отбросить остаток строки
```

В этом примере цикл прерывается по причине некорректного ввода. Предположим вместо этого, что цикл прерывается по достижению конца файла или по причине аппаратного сбоя. Тогда новый код, отбрасывающий плохой ввод, не имеет смысла. Вы можете справиться с ситуацией, используя метод `fail()`, проверяющий корректность предположения. Поскольку по причинам исторического характера `fail()` возвращает `true`, если оба бита, `failbit` и `eofbit`, установлены, этот код должен исключить последний случай. Следующий фрагмент показывает пример такого исключения:

```
while (cin >> input)
{
   sum += input;
}
cout << "Последнее введенное значение = " << input << endl;
cout << "Сумма = " << sum << endl;
if (cin.fail() && !cin.eof() ) // провал из-за неправильного ввода
{
```

```
      cin.clear(); // сбросить состояние потока
      while (!isspace(cin.get()))
         continue; // отбросить некорректный ввод
   }
   else // иначе - помочь
   {
      cout << "Продолжение невозможно!\n";
      exit(1);
   }
   cout << "Введите новое число: ";
   cin >> input; // теперь будет работать
```

Другие методы класса istream

В главах 3, 4 и 5 обсуждаются методы `get()` и `getline()`. Вспомните, что они обеспечивают следующие дополнительные возможности ввода:

- Методы `get(char &)` и `get(void)` представляют односимвольный ввод, который не пропускает пробельных символов.
- Функции `get(char *, int, char)` и `getline(char *, int, char)` по умолчанию читают строки целиком, а не отдельные слова.

Эти функции называются *функциями неформатированного ввода*, потому что они просто читают символьный ввод, как он есть, не пропуская пробелов и не выполняя никаких преобразований данных.

Рассмотрим эти две группы функций-членов класса `istream`.

Односимвольный ввод

Когда метод `get()` вызываются с аргументом типа `char` или вообще без аргументов, он извлекает следующий символ ввода, даже если это пробел, знак табуляции или символ новой строки. Версия `get(char & ch)` присваивает входящий символ своему аргументу, а версия `get (void)` просто использует входной символ, преобразует его в целочисленный тип (обычно — `int`) и возвращает это значение.

Попробуем для начала `get(char &)`. Предположим, что в программе присутствует следующий цикл:

```
int ct = 0;
char ch;
cin.get(ch);
while (ch != '\n')
{
   cout << ch;
   ct++;
   cin.get(ch);
}
cout << ct << endl;
```

Далее предположим, что вы печатаете следующий оптимистический ввод:

```
I C++ clearly.<Enter>
```

Нажатие клавиши <Enter> посылает эту входную строку программе. Фрагмент программы читает символ `I`, отображает его в `cout` и увеличивает `ct` до 1. Далее он читает символ пробела, следующий за `I`, отображает его и увеличивает `ct` до 2. Это

продолжается до тех пор, пока программа не обработает нажатие клавиши <Enter> как символ новой строки и прервет цикл. Главная идея в том, что, используя `get(ch)`, код читает, отображает и считает пробелы, как и печатные символы.

Предположим теперь, что вместо этого программа использует операцию >>:

```
int ct = 0;
char ch;
cin >> ch;
while (ch != '\n') // неудача
{
   cout << ch;
   ct++;
   cin >> ch;
}
cout << ct << endl;
```

Во-первых, этот код будет пропускать пробелы, таким образом, не считая их, и сжимая вывод:

```
IC++clearly.
```

Плохо, цикл никогда не прервется! Поскольку операция извлечения пропускает символы новой строки, этот код никогда не присвоит такой символ переменной `ch`, поэтому проверка условия цикла `while` никогда не завершит его выполнения.

Функция-член `get(char &)` возвратит ссылку на объект `istream`, использованный для ее вызова. Это значит, что вы выполняете конкатенацию следующих за `get(char &)` извлечений:

```
char c1, c2, c3;
cin.get(c1).get(c2) >> c3;
```

Во-первых, `cin.get(c1)` присвоит первый входной символ `c1` и вернет вызывающий объект, в данном случае — `cin`. Это сокращает код до `cin.get(c2) >> c3`, который присваивает второй входной символ `c2`. Вызов функции возвращает `cin`, уменьшая код до `cin >> c3`. Это, в свою очередь, присваивает следующий непробельный символ переменной `c3`. Отметим, что `c1` и `c2` могут получить значения пробела, но `c3` — нет.

Если `cin.get(char &)` встречает конец файла — реальный или же эмулируемый с клавиатуры (<Ctrl+Z> для DOS, <Ctrl+D> в начале строки — для Unix) — он не присваивает значение своему аргументу. И это достаточно правильно, потому что если программа достигла конца файла, никакого значения присваивать не нужно. Более того, метод вызывает `setstate(failbit)`, что заставляет проверку `cin` вернуть `false`:

```
char ch;
while (cin.get(ch))
{
   // обработать ввод
}
```

До тех пор, пока ввод корректен, возвращаемым значением `cin.get(ch)` будет `cin`, который оценивается как `true`, поэтому цикл продолжает работать. При достижении конца файла возвращаемое значение вычисляется как `false`, что прерывает цикл.

Функция-член get(void) также читает пробелы, но использует свое возвращаемое значения для передачи ввода в программу. Поэтому вы можете использовать ее следующим образом:

```
int ct = 0;
char ch;
ch = cin.get(); // использовать возвращаемое значение
while (ch != '\n')
{
   cout << ch;
   ct++;
   ch = cin.get();
}
cout << ct << endl;
```

Некоторые старые реализации C++ не предусматривают этих функций-членов.

Функция-член get(void) возвращает тип int (или некоторый большой целочисленный тип, в зависимости от набора символов и региональных установок). Это делает следующий фрагмент неправильным:

```
char c1, c2, c3;
cin.get().get() >> c3;    // не верно
```

Здесь cin.get() возвращает значение типа int. Поскольку возвращаемое значение — не объект класса, вы не можете применить к нему операцию членства. То есть получится синтаксическая ошибка. Однако вы можете использовать get() в конце последовательности извлечений:

```
char c1;
cin.get(c1).get();        // правильно
```

Тот факт, что get(void) возвращает тип int, означает, что вы не можете вслед за ним вызвать операцию извлечения. Но поскольку cin.get(c1) возвращает cin, это делает его подходящим префиксом для get(). Этот конкретный код прочтет первый символ ввода, присвоит его c1, затем прочтет второй входной символ и отбросит его.

При достижении конца файла — реального или эмулируемого — cin.get(void) возвратит значение EOF, которое является символической константой, представленной в файле заголовка iostream. Это свойство дизайна позволяет следующей конструкции читать ввод:

```
int ch;
while ((ch = cin.get()) != EOF)
{
   // обработать ввод
}
```

Вы должны использовать тип int для ch вместо типа char, потому что значение EOF не может быть выражено как тип char.

В главе 5 описывает эти функции немного более подробно, а в табл. 17.6 резюмируются средства функций односимвольного ввода.

Таблица 17.6. Сравнение cin, get(ch) и cin.get()

Свойство	cin.get(ch)	ch = cin.get()
Метод доставки входного символа	Присваивает аргументу ch	Использует возвращаемое значение для присвоения ch
Возвращаемое значение для символьного ввода	Ссылка на объект класса istream	Код символа как значение типа int
Возвращаемое значение по концу файла	Преобразуется в false	EOF

Какую форму односимвольного ввода предпочесть?

При выборе между >>, get(char &) и get(void) чему стоит отдать предпочтение? Во-первых, вы должны решить, нужно ли при вводе пропускать пробелы. Если пропуск пробелов допускается, нужно использовать операцию извлечения >>. Например, пропуск пробелов удобен для реализации простого меню:

```
cout << "a. annoy client b. bill client\n"
     << "c. calm client d. deceive client\n"
     << "q.\n";
cout << "Enter a, b, c, d, or q: ";
char ch;
cin >> ch;
while (ch != 'q')
{
   switch(ch)
   {
      ...
   }
   cout << "Enter a, b, c, d, or q: ";
   cin >> ch;
}
```

Чтобы ввести, скажем, ответ b, вы вводите **b** и нажимаете клавишу <Enter>, генерируя двухсимвольный ответ b\n. Если использовать любую из форм get(), то придется добавлять код, обрабатывающий символ \n на каждом шаге цикла, но операция извлечения без труда пропускает его. (Если вы программировали на C, то вероятно, сталкивались с ситуацией, когда символ новой строки появлялся в программе как неверный ответ. Эту проблему легко решить, но все же это — досадная мелочь.)

Если вы хотите, чтобы программа обрабатывала каждый символ, то должны использовать один из методов get(). Например, программа подсчета символов может трактовать пробелы как признак конца слова. Из двух методов get() только get(char &) имеет интерфейс в виде класса. Главное преимущество метода get(void) в том, что он очень похож на стандартную функцию C getchar(), а это значит, что вы можете преобразовать программу на C в C++, включив заголовочный файл iostream вместо stdio.h и глобально заменив getchar() на cin.get(), а putchar(ch) — на cout.put(ch).

Строковый ввод: getline(), get() и ignore()

Теперь рассмотрим функции-члены, описанные в главе 4, служащие для строкового ввода. И функция-член getline(), и строчно-ориентированные версии get() читают строки, и обе они имеют одинаковую сигнатуру (здесь — упрощенная форма по сравнению с более общим объявлением шаблона):

```
istream & get(char *, int, char);
istream & get(char *, int);
istream & getline(char *, int, char);
istream & getline(char *, int);
```

Вспомните, что первый аргумент является адресом, куда помещать вводимую строку. Второй аргумент — на единицу больше, чем максимальное число символов, которое следует прочитать. (Дополнительный символ оставляет место для ограничивающего нулевого символа, чтобы сохранить ввод в виде строки.) Третий аргумент специфицирует символ, служащий разделителем. Каждая функция читает символы до тех пор, пока не будет прочтено максимальное их количество, или до тех пор, пока не встретит символ-разделитель — в зависимости от того, что случится первым.

Например, код

```
char line[50];
cin.get(line, 50);
```

читает символьный ввод в массив символов line. Функция cin.get() прекращает чтение ввода в массив после получения 49 символов или, по умолчанию, после получения символа перевода строки — в зависимости от того, что произойдет раньше. Основное различие между get() и getline() в том, что get() оставляет символ перевода строки во входном потоке, делая его доступным для следующей операции ввода, в то время как getline() отбрасывает символы новой строки из входного потока.

В главе 4 было показано использование двухаргументной формы этих функций-членов. Теперь рассмотрим трехаргументные версии. Третий аргумент — это символ-разделитель. Чтение ввода прекращается, когда встречается разделитель, даже если не прочтено максимальное количество символов. Поэтому по умолчанию оба метода прекращают чтение, когда достигают конца строки прежде, чем прочесть заданное количество символов. Как и в случае по умолчанию, get() оставляет символ-разделитель во входной очереди, а getline() — нет.

Код в листинге 17.13 демонстрирует работу getline() и get(). В нем также представлена функция-член ignore(). ignore() принимает два аргумента: число, означающее максимальное число символов для чтения, и символ, служащий разделителем при вводе. Например, вызов функции

```
cin.ignore(255, '\n');
```

читает и отбрасывает или следующие 255 символов, или вплоть до символа новой строки, в зависимости от того, что произойдет раньше. Этот прототип предусматривает значения по умолчанию 1 и EOF для своих двух аргументов. Функция возвращает тип istream &:

```
istream & ignore(int = 1, int = EOF);
```

(Значение по умолчанию EOF заставляет ignore() читать вплоть до заданного количества символов или до конца файла — что случится раньше.)

Функция возвращает вызывающий объект. Это позволяет выполнять конкатенацию вызовов функции, как в следующем операторе:

```
cin.ignore(255, '\n').ignore(8255, '\n');
```

Здесь первый вызов `ignore()` читает и отбрасывает одну строку, а второй — вторую строку. Вместе эти две функции читают две строки.

Теперь взглянем на код в листинге 17.13.

Листинг 17.13. get_fun.cpp

```
// get_fun.cpp -- использование get() и getline()
#include <iostream>
const int Limit = 255;
int main()
{
   using std::cout;
   using std::cin;
   using std::endl;
   char input[Limit];
   cout << "Введите строку для обработки getline():\n";
   cin.getline(input, Limit, '#');
   cout << "Вы ввели:\n";
   cout << input << "\nЗавершена фаза 1\n";
   char ch;
   cin.get(ch);
   cout << "Следующий введенный символ " << ch << endl;
   if (ch != '\n')
      cin.ignore(Limit, '\n'); // отбросить остаток строки
   cout << "Введите строку для обработки get():\n";
   cin.get(input, Limit, '#');
   cout << "Вы ввели:\n";
   cout << input << "\nЗавершена фаза 2\n";
   cin.get(ch);
   cout << "Следующий введенный символ " << ch << endl;
   return 0;
}
```

Замечание по совместимости

Некоторые старые реализации C++ имеют ошибку в `getline()`, которая вызывает задержку отображения следующей выходной строки до тех пор, пока вы не введете данные, запрошенные неотображаемой строкой.

Вот как выглядит пример выполнения программы из листинга 17.13:

```
Введите строку для обработки getline():
Please pass
me a #3 melon!
Вы ввели:
Please pass
me a
Завершена фаза 1
Следующий введенный символ 3
Введите строку для обработки get():
I still
want my #3 melon!
```

```
Вы ввели:
I still
want my
Завершена фаза 2
Следующий введенный символ #
```

Обратите внимание, что функция getline() отбрасывает ограничивающий символ # во вводе, а функция get() — нет.

Нежелательный строковый ввод

Некоторые формы ввода для get(char *, int) и getline() влияют на состояние потока. Как и с другими функциями ввода, достижение конца файла устанавливает eofbit, а все, что повреждает поток — как например, сбой устройства — устанавливает badbit. Два других специальных случая — это когда нет ввода или когда приходит количество символов, превышающее максимальное число символов, указанное при вызове функции. Давайте рассмотрим эти случаи.

Если любому из методов не удается извлечь никаких символов, во входную строку помещается нулевой символ и применяется setstate() для установки failbit. (Более старые реализации C++ не устанавливают failbit, если никакие символы не прочтены.) Когда методу не удается извлечь никаких символов? Одной из причин может быть немедленное достижение конца файла. Для get(char *, int) другой причиной может быть ввод пустой строки:

```
char temp[80];
while (cin.get(temp,80)) // прерывает на пустой строке
...
```

Интересно, что пустая строка не заставляет getline() устанавливать failbit. Это потому, что getline() извлекает символ новой строки, даже если и не сохраняет его. Если вы хотите, чтобы цикл getline() прерывался на пустой строке, вы можете написать это следующим образом:

```
char temp[80];
while (cin.getline(temp,80) && temp[0] != '\0')//прерывает на пустой строке
```

Теперь предположим, что количество символов во входной очереди соответствует или превышает максимальное, которое указано методу ввода. Для начала возьмем getline() и следующий код:

```
char temp[30];
while (cin.getline(temp,30))
```

Метод getline() прочтет последовательные символы из входной очереди, помещая их один за другим в элементы массива temp до тех пор, пока (для целей тестирования) не будет достигнут конец файла, либо когда следующий символ не окажется символом новой строки, либо пока не будет сохранено 29 байт. Если достигается конец файла, устанавливается eofbit. Если следующий читаемый символ является символом новой строки, он читается и отбрасывается. А если будет прочтено 29 байт, то устанавливается failbit, если только следующий символ — не символ перевода строки. Таким образом, входная строка длиной 30 символов или более прервет ввод.

Теперь рассмотрим метод get(char *, int). Сначала он проверяет количество символов, затем признак конца файла и, наконец — является ли очередной символ символом перевода строки. Независимо от этого, вы можете определить, когда окон-

чание чтения в методе произошло по причине того, что во входном потоке было слишком много символов. Вы можете использовать `peek()` (см. следующий раздел) для проверки следующего символа ввода. Если это перевод строки, то `get()` должен прочитать полную строку. Если это не символ новой строки, то `get()` должен остановиться перед достижением конца. Эта техника не обязательно работает с `getline()`, поскольку `getline()` читает и отбрасывает перевод строки, поэтому взгляд на следующий символ вам ничего не даст. Но если вы применяете `get()`, то у вас есть возможность сделать что-то, если прочитана неполная строка. Следующий раздел включает пример такого подхода. А между тем, в табл. 17.7 резюмируются некоторые различия между старым вводом C++ и текущим стандартом.

Таблица 17.7. Изменения в поведении ввода

Метод	Старый C++	Современный C++
`getline()`	Не устанавливает `failbit`, если ни один символ не прочитан.	Устанавливает `failbit`, если ни один символ не прочитан (но перевод строки считается прочитанным символом).
	Не устанавливает `failbit`, если прочитано максимальное число символов.	Устанавливает `failbit`, если прочитано максимальное количество символов, но в строке еще остаются непрочитанные символы.
`Get(char *, int)`	Не устанавливает `failbit`, если не прочитано ни одного символа.	Устанавливает `failbit`, если ни один символ не прочитан.

Другие методы istream

Другие методы `istream`, помимо описанных до сих пор, включают `read()`, `peek()`, `gcount()` и `putback()`. Функция `read()` читает заданное количество байтов и сохраняет их в указанном месте. Например, оператор

```
char gross[144];
cin.read(gross, 144);
```

читает 144 символа из стандартного ввода и помещает их в массив `gross`. В отличие от `getline()` и `get()`, `read()` не добавляет нулевой символ к вводу, поэтому он не преобразует ввод в строковую форму. Метод `read()` не предназначен для клавиатурного ввода. Вместо этого он чаще применяется в сочетании с функцией `ostream write()` для файлового ввода и вывода. Этот метод возвращает значение типа `istream &`, поэтому может быть конкатенирован следующим образом:

```
char gross[144];
char score[20];
cin.read(gross, 144).read(score, 20);
```

Функция `peek()` возвращает следующий символ ввода без извлечения его из входного потока. То есть он позволяет взглянуть на следующий символ. Предположим, что вы хотите прочитать ввод вплоть до первого перевода строки или точки, в зависимости от того, что произойдет раньше. Вы можете использовать `peek()` для просмотра следующего символа входного потока, дабы оценить, стоит ли продолжать:

```
char great_input[80];
char ch;
```

```
   int i = 0;
   while ((ch = cin.peek()) != '.' && ch != '\n')
      cin.get(great_input[i++]);
   great_input [i] = '\0';
```

Вызов cin.peek() берет следующий входной символ и присваивает его значение ch. Затем условие выхода из цикла while проверяет, не является ли ch точкой или символом новой строки. В этом случае цикл читает символ в массив и обновляет его индекс. Когда цикл завершается, точка или символ новой строки остаются во входном потоке с тем, чтобы быть первым символом, который будет прочитан следующей операцией ввода. Затем код добавляет нулевой символ к массиву, завершая строку.

Метод gcount() возвращает количество символов, прочитанное последним неформатирующим методом извлечения. Это значит символы, прочитанные get(), getline(), ignore() или read(), но не операцией извлечения (>>), которая форматирует ввод для соответствия определенным типам данных. Например, предположим, что вы использовали cin.get(myarray, 80) для чтения строки в массив myarray, и хотите знать, сколько символов было прочитано. Вы можете использовать функцию strlen() для подсчета символов в массиве, но быстрее применить cin.getcount(), чтобы узнать, сколько символов было прочитано только что из входного потока.

Функция putback() вставляет символ обратно в строку ввода. При этом вставленный символ становится первым символом, прочитанным следующим оператором ввода. Метод putback() принимает один аргумент char, представляющий вставляемый символ, и возвращает тип istream &, что позволяет вызов конкатенировать с другими методами istream. Применение peek() подобно вызову get() для чтения символа с последующим putback() с целью помещения прочитанного символа обратно во входной поток. Однако putback() дает возможность вернуть в поток символ, отличающийся от последнего прочитанного.

В листинге 17.14 используются два подхода для чтения и отображения ввода, вплоть до (но не включая) символа #. Первый подход читает до символа # и затем вызывает putback() для вставки этого символа обратно в поток. Второй подход применяет peek() для того, чтобы заглянуть вперед, прежде чем осуществить чтение.

Листинг 17.14. peeker.cpp

```
// peeker.cpp -- некоторые методы istream
#include <iostream>
int main()
{
   using std::cout;
   using std::cin;
   using std::endl;
   // читать и отображать символы до #
   char ch;
   while(cin.get(ch))         // прервать по достижении EOF
   {
      if (ch != '#')
         cout << ch;
      else
      {
         cin.putback(ch);     // повторно вставить символ
         break;
      }
   }
```

```
    if (!cin.eof())
    {
        cin.get(ch);
        cout << endl << ch << " — следующий символ.\n";
    }
    else
    {
        cout << "Достигнут конец файла.\n";
        std::exit(0);
    }
    while(cin.peek() != '#') // заглянуть вперед
    {
        cin.get(ch);
        cout << ch;
    }
    if (!cin.eof())
    {
        cin.get(ch);
        cout << endl << ch << " — следующий символ.\n";
    }
    else
        cout << "Достигнут конец файла.\n";
    return 0;
}
```

Ниже показан пример выполнения этой программы:

```
I used a #3 pencil when I should have used a #2.
I used a
# - следующий символ.
3 pencil when I should have used a
# - следующий символ.
```

Замечания по программе

Рассмотрим код в листинге 17.14 более внимательно. Первый подход использует цикл `while` для чтения ввода:

```
while(cin.get(ch))          // прервать по достижении EOF
{
    if (ch != '#')
        cout << ch;
    else
    {
        cin.putback(ch);    // повторно вставить символ
        break;
    }
}
```

Выражение `cin.get(ch)` возвращает `false` при достижении условия конца файла, поэтому эмуляция конца файла с клавиатуры прерывает цикл. Если раньше появляется символ `#`, то программа поместит его обратно во входной поток и с помощью оператора `break` прервет цикл.

Второй подход проще:

```
while(cin.peek() != '#') // заглянуть вперед
{
   cin.get(ch);
   cout << ch;
}
```

Программа заглядывает на следующий символ. Если это не символ #, программа читает следующий символ, отображает его и заглядывает дальше. Это продолжается до тех пор, пока не появится символ прерывания.

Теперь посмотрим, как и обещалось, на пример, использующий peek() для определения того, была ли прочтена вся строка целиком (см. листинг 17.15). Если только часть строки попадает во входной массив, то программа отбрасывает остаток.

Листинг 17.15. truncate.cpp

```
// truncate.cpp -- использование get() для усечения входной строки,
// если необходимо
#include <iostream>
const int SLEN = 10;
inline void eatline() { while (std::cin.get() != '\n') continue; }
int main()
{
   using std::cin;
   using std::cout;
   using std::endl;
   char name[SLEN];
   char title[SLEN];
   cout << "Введите ваше имя: ";
   cin.get(name,SLEN);
   if (cin.peek() != '\n')
      cout << "Сожалеем, помещается только "
           << name << endl;
   eatline();
   cout << "Ув. " << name << ", введите ваш титул: \n";
   cin.get(title,SLEN);
   if (cin.peek() != '\n')
      cout << "Нам пришлось сократить ваш титул.\n";
   eatline();
   cout << " Имя: " << name
        << "\nТитул: " << title << endl;
   return 0;
}
```

Ниже показан пример запуска программы из листинга 17.15:

```
Введите ваше имя: Ella Fishsniffer
Сожалеем, помещается только Ella Fish
Ув. Ella Fish, введите ваш титул:
Executive Adjunct
Нам пришлось сократить ваш титул.
Имя: Ella Fish
Титул: Executive
```

Обратите внимание, что следующий код имеет разный смысл в зависимости от того, читает ли первый оператор ввода полную строку:

```
while (cin.get() != '\n') continue;
```

Если get() читает полную строку, он все же оставляет на месте символ перевода строки, и этот код читает и отбрасывает символ новой строки. Если же get() читает только часть строки, этот код читает и отбрасывает остаток строки. Если вы не разместили остаток строки, следующий оператор ввода начнет чтение с начала оставшейся части от первой введенной строки. В данном примере это должно привести к чтению строки arpride в массив title.

Файловый ввод и вывод

Большинство компьютерных программ работают с файлами. Текстовые процессоры создают файлы документов. Программы баз данных создают и выполняют поиск в информационных файлах. Компиляторы читают файлы исходного кода и генерируют исполняемые файлы. Сам по себе файл — это группа байтов, сохраненных на некотором устройстве, возможно, магнитной ленте, возможно, оптическом диске, дискете или жестком диске. Как правило, операционная система управляет файлами, отслеживая их местоположение, размеры, дату их создания и тому подобное. Если только вы не программируете на уровне операционной системы, обычно вам не нужно заботиться об упомянутых вещах. Все, что вам нужно знать — это способ подключения программы к файлу, способ прочесть в программе его содержимое, и способ создавать и читать файлы внутри программы.

Перенаправление (описанное ранее в настоящей главе) может предоставить некоторую поддержку файлов, но значительно более ограниченную, чем явный ввод-вывод, осуществляемый из программы. К тому же перенаправление обеспечивается операционной системой, а не C++, поэтому оно доступно не во всех системах. В этой книге мы уже касались темы файлового ввода-вывода, но эта глава освещает данную тему более тщательно.

Пакет классов ввода-вывода C++ управляет файловым вводом и выводом в основном так же, как он делает это со стандартным вводом и выводом. Чтобы записывать в файл, вы создаете объект ofstream и используете такие его методы, как операция вставки << или write(). Чтобы читать из файла, вы создаете объект ifstream и используете методы istream вроде операции извлечения >> и get(). Однако файлы требуют больше внимания, нежели стандартный ввод и вывод. Например, вы должны ассоциировать вновь открытый файл с потоком. Вы можете открыть файла в режиме только для чтения, только для записи либо для чтения-записи. Если вы записываете в файл, то можете пожелать создать новый, заменить старый либо добавить информацию в существующий файл. Или же вы можете вернуться и пройти по файлу заново. Чтобы помочь в выполнении этих задач, C++ определяет несколько новых классов в заголовочном файле fstream (бывший fstream.h), включая класс ifstream для файлового ввода и класс ofstream для файлового вывода. C++ также определяет класс fstream для совместного файлового ввода-вывода. Эти классы унаследованы от классов из заголовочного файла iostream, поэтому объекты этих новых классов могут использовать методы, которые вы уже изучили ранее.

Простой файловый ввод-вывод

Предположим, вы хотите, чтобы ваша программа записывала в файл. Вы должны сделать следующее:

1. Создать объект ofstream для управления выходным потоком.
2. Ассоциировать этот объект с конкретным файлом.
3. Использовать объект тем же способом, как вы используете cout. Единственным отличием будет то, что вывод направляется в файл вместо экрана.

Чтобы достичь этого, вы должны начать с включения заголовка fstream. Его включение в большинстве, хотя и не во всех реализациях, автоматически включает файл iostream, поэтому вам не обязательно включать явно iostream. Затем вы должны объявить объект типа ofstream:

```
ofstream fout;           // создает объект ofstream по имени fout
```

Именем объекта может быть любое допустимое в C++ имя вроде fout, outFile, cgate или didi.

Затем вы должны ассоциировать этот объект с конкретным файлом. Вы можете сделать это с помощью метода open(). Предположим, например, что вы хотите открыть файл jar для вывода. Вы можете сделать это следующим образом:

```
fout.open("jar.txt"); // ассоциировать fout с jar.txt
```

Вы можете совместить эти два шага (создание объекта и ассоциация файла с ним) в одном операторе, используя другой конструктор:

```
ofstream fout("jar.txt"); //создать объект fout, ассоциировать его с jar.txt
```

После того, как вы все это сделаете, вы сможете использовать fout (или любое другое выбранное вами имя) в той же манере, что и cout. Например, если вы захотите поместить слова Dull Data в этот файл, то можете сделать это следующим образом:

```
fout << "Dull Data";
```

В самом деле, поскольку ostream — это базовый класс для класса ofstream, вы вправе использовать все методы ostream, включая разнообразные операции вставки, а также форматирующие методы и манипуляторы. Класс ofstream использует буферизованный вывод, поэтому программа выделяет пространство для выходного буфера, когда создает объект типа ofstream, подобный fout. Если вы создадите два объекта ofstream, то программа создаст два буфера — по одному для каждого объекта. Объект ofstream, такой как fout, принимает от программы вывод — байт за байтом, а затем, когда буфер наполняется, передает его содержимое в файл назначения. Поскольку дисковые приводы спроектированы так, что передают данные крупными порциями, а не байт за байтом, буферизованный подход значительно повышает скорость передачи данных из программы в файл.

Открытие файла для вывода, таким образом, создает новый файл, если файла с указанным именем не существовало. Если же файл с этим именем существовал ранее, то действие по его открытию усекает его до нулевого размера, так что вывод начинается в пустой файл. Позднее в этой главе вы увидите, как открыть существующий файл и сохранить его содержимое.

Внимание!

Открытие файла для вывода в режиме по умолчанию автоматически усекает его до нулевого размера, что уничтожает его предыдущее содержимое.

Требования к чтению файла очень похожи на требования, которые нужно выполнить для записи в файл:

1. Создать объект ifstream для управления входным потоком.
2. Ассоциировать этот объект с конкретным файлом.
3. Использовать объект тем же способом, что используется cin.

Шаги для чтения файла похожи на шаги, которые нужно выполнить для его записи. Во-первых, конечно, вы включаете заголовочный файл fstream. Затем вы объявляете объект ifstream и ассоциируете его с именем файла. Это можно сделать в двух операторах или же в одном:

```
// два оператора
ifstream fin; // создать объект ifstream по имени fin
fin.open("jellyjar.dat");  // открыть jellyjar.dat для чтения
// один оператор
ifstream fis("jamjar.dat");//создать fis и ассоциировать его с jamjar.dat
```

Затем вы можете использовать fin или fis почти так же, как используете cin. Например, можно поступить следующим образом:

```
char ch;
fin >> ch;              // прочесть символ из файла jellyjar.dat
char buf[80];
fin >> buf;             // прочесть слово из файла
fin.getline(buf, 80);  // прочесть строку из файла
string line;
getline(fin, line);    // прочесть из файла в строковый объект
```

Ввод, как и вывод, также буферизуется, поэтому создание объекта ofstream, такого как fin, создает входной буфер, которым управляет объект fin. Как и в случае вывода, буферизация перемещает данные гораздо быстрее, чем передача байт за байтом.

Соединение с файлом закрывается автоматически, когда объекты ввода и вывода уничтожаются, например, по завершении программы. Кроме того, вы можете закрыть соединение с файлом явно, используя для этого метод close():

```
fout.close();           // закрыть вывод, подключенный к файлу
fin.close();            // закрыть ввод, подключенный к файлу
```

Закрытие подключения не уничтожает поток; он просто отключается от файла. Однако средства управления потоком остаются на месте. Например, объект fin продолжает существовать вместе с входным буфером, которым он управляет. Как вы вскоре увидите, этот поток можно подключить заново к тому же файлу либо к другому.

Рассмотрим краткий пример. Программа в листинге 17.16 запрашивает имя файла. Она создает файл с этим именем, пишет некоторую информацию в него и закрывает файл. Закрытие файла сбрасывает буфер, тем самым гарантируя обновление файла. Затем программа открывает тот же файл для чтения и отображает его содержимое. Отметим, что программа использует имена fin и fout в той же манере, что и если бы вы применяли cin и cout. Также программа читает имя файла в объект string и использует метод c_str() для обеспечения аргумента в виде строки стиля C для конструкторов ofstream и ifstream.

Листинг 17.16. `fileio.cpp`

```
// fileio.cpp -- сохранение в файле
#include <iostream> // для многих систем не требуется
#include <fstream>
#include <string>
int main()
{
   using namespace std;
   string filename;
   cout << "Введите имя нового файла: ";
   cin >> filename;
   // создать объект выходного потока для нового файла и назвать его fout
   ofstream fout(filename.c_str());
   fout << "Только для ваших глаз!\n";  // писать в файл
   cout << "Введите секретное число: "; // писать на экран
   float secret;
   cin >> secret;
   fout << "Ваше секретное число " << secret << endl;
   fout.close(); // закрыть файл
   // создать объект входного потока для нового файла и назвать его fin
   ifstream fin(filename.c_str());
   cout << "Вот содержимое " << filename << ":\n";
   char ch;
   while (fin.get(ch)) // читать символы из файла
      cout << ch; // и писать их на экран
   cout << "Готово.\n";
   fin.close();
   return 0;
}
```

Пример запуска программы из листинга 17.16:

```
Введите имя файла: pythag
Введите секретное число: 3.14159
Вот содержимое pythag:
Только для ваших глаз!
Ваше секретное число 3.14159
Готово.
```

Если вы просмотрите каталог, содержащий программу, то найдете там файл по имени `pythag` и, загрузив его в любой текстовый редактор, увидите то же содержимое, что показал вывод программы.

Проверка потока и `is_open()`

Классы файловых потоков C++ наследуют член, описывающий состояние потока, от класса `ios_base`. Этот член, как упоминалось ранее, хранит информацию, отражающую состояние потока, — о том, что все хорошо, что достигнут конец файла, о том, произошел ли сбой операции ввода-вывода, и так далее. Если все хорошо, состояние потока равно нулю (отсутствие новостей — уже хорошая новость). Разнообразные другие состояния описываются установкой конкретных битов в единицу. Классы файловых потоков также наследуют методы `ios_base`, которые сообщают о состоянии потока и перечислены в табл. 17.5. Вы можете проверять состояние потока, чтобы

узнать, успешно ли завершилась самая последняя операция с этим потоком. Для файловых потоков это включает в себя проверку успеха или сбоя операции открытия файла. Например, попытка открыть для ввода несуществующий файл устанавливает failbit. Поэтому вы можете проверить это следующим образом:

```
fin.open(argv[file]);
if (fin.fail())        // попытка открытия не удалась
{
   ...
}
```

Или же, поскольку объект ifstream, подобно istream, преобразуется в тип bool, когда ожидается тип bool, вы можете использовать это:

```
fin.open(argv[file]);
if (!fin)              // попытка открытия не удалась
{
   ...
}
```

Однако новые реализации C++ предлагают лучший способ проверки того, открыт ли файл — метод is_open().

```
if (!fin.is_open())    // попытка открытия не удалась
{
   ...
}
```

Этот способ лучше, потому что он проверяет также наличие некоторых тонких проблем, которых не замечают другие способы, как указано в следующей врезке.

Внимание!

В прошлом проверка успешности открытия файла выполнялась следующим образом:

```
if(fin.fail()) ...      // открыть не удалось
if(!fin.good()) ...     // открыть не удалось
if (!fin) ...           // открыть не удалось
```

Объект fin, когда он проверяется в условии if, преобразуется в false, если fin.good() возвращает false, и в true — в остальных случаях, поэтому данные две формы эквивалентны. Однако эти тесты не могут правильно распознать ситуацию, когда осуществляется попытка открытия файла с неправильным режимом доступа. (См. раздел "Режимы файла" далее в настоящей главе.) Метод is_open() перехватывает ошибки подобного рода, наряду с вызовом из метода good(). Однако старые реализации C++ не предлагали метод is_open().

Открытие нескольких файлов

Иногда может понадобиться, чтобы программа открывала более одного файла. Стратегия открытия множества файлов зависит от того, как они будут использоваться. Если вам нужно, чтобы два файла были открыты одновременно, вы должны создать отдельный поток для каждого файла. Например, программа, которая сравнивает два отсортированных файла и отправляет результат в третий, должна создать два объекта ifstream для двух входных файлов и один объект ofstream — для выходного файла. Количество файлов, которые можно открыть одновременно, зависит от операционной системы, но обычно составляет порядка 20.

Однако вы можете запланировать последовательную обработку файлов. Например, вам нужно подсчитать, сколько раз появляется имя в группе из 10 файлов. В этом случае вы можете открыть единственный поток и по очереди ассоциировать его с

каждым из этих файлов. Чтобы применить такой подход, вы объявляете объект ifstream без его инициализации и затем использовать метод open(), чтобы ассоциировать поток с файлом. Например, вот как можно организовать последовательное чтение двух файлов:

```
ifstream fin;            // создать поток конструктором по умолчанию
fin.open("fat.dat");  // ассоциировать поток с файлом fat.dat
...                      // выполнить какую-то работу
fin.close();             // прервать ассоциацию потока с файлом fat.dat
fin.clear();             // сбросить fin (может не понадобиться)
fin.open("rat.dat");  // ассоциировать поток с файлом rat.dat
...
fin.close();
```

Этот пример мы сейчас рассмотрим, но сначала изучим технику передачи списка файлов программе способом, позволяющим программе применить цикл для их обработки.

Обработка командной строки

Программы, обрабатывающие файлы, часто используют аргументы командной строки для идентификации файлов. *Аргументы командной строки* — это параметры, которые появляются в командной строке после команды. Например, чтобы подсчитать количество слов в некоторых файлах на системе Unix или Linux, вы вводите команду:

```
wc report1 report2 report3
```

Здесь wc — имя программы, а report1, report2 и report3 — имена файлов, переданные программе в качестве аргументов командной строки.

C++ имеет механизм, позволяющий программам, запущенным из среды с командной строкой, получать доступ к аргументам командной строки. Вы можете использовать следующий альтернативный заголовок функции main():

```
int main(int argc, char *argv[])
```

Аргумент argc представляет количество аргументов в командной строке. Счетчик включает имя самой программы. Переменная argv — это указатель на указатель на char. Это звучит несколько абстрактно, но вы можете трактовать argv как массив указателей на аргументы командной строки, причем argv[0] указывает на первый символ строки, содержащей имя самой команды, argv[1] — указатель на первый символ строки, содержащей первый аргумент командной строки, и так далее. То есть argv[0] — первая строка команды и так далее. Например, предположим, что имеется следующая командная строка:

```
wc report1 report2 report3
```

В этом случае argc будет равно 4, argv[0] — wc, argv[1] — report1 и так далее. Следующий цикл напечатает каждый аргумент командной строки в отдельной строке экрана:

```
for (int i = 1; i < argc; i++)
cout << argv[i] << endl;
```

Если начать с i = 1, то будут распечатаны аргументы командной строк, а если начать с i = 0, то будет распечатана вся командная строка.

Аргументы командной строки, конечно, идут рука об руку с командными операционными системами, такими как DOS, Unix и Linux. Другие среды также должны позволять вам использовать аргументы командной строки:

- Многие интегрированные среды разработки (Integrated Development Environments – IDE) в DOS и Windows имеют опцию, позволяющую передавать им аргументы командной строки. Обычно вы должны осуществлять навигацию по серии пунктов меню, чтобы добраться до поля, в которое можно ввести аргументы командной строки. Конкретная последовательность шагов варьируется в зависимости от поставщика и от версии, так что за подробными инструкциями обращайтесь к вашей документации.
- Интегрированные среды IDE в DOS и многие IDE в Windows могут генерировать исполняемые файлы, которые запускаются под управлением DOS или в окне DOS – в обычном режиме командной строки DOS.
- Под управлением Metrowerks CodeWarrior для Macintosh вы можете эмулировать аргументы командной строки, помещая в программу следующий код:

```
...
#include <console.h>           // для эмуляции аргументов командной
строки
int main(int argc, char * argv[])
{
   argc = ccommand(&argv);  // да, ccommand, а не command
   ...
```

  Когда вы запустите программу, функция `ccommand()` выведет на экран диалоговое окно, содержащее поле, в которое вы можете ввести аргументы командной строки. Это также позволяет эмулировать перенаправления ввода-вывода.

В листинге 17.17 комбинируется техника командной строки с потоками для подсчета количества символов в файлах, перечисленных в командной строке.

Листинг 17.17. count.cpp

```
// count.cpp -- подсчет символов в списке файлов
#include <iostream>
#include <fstream>
#include <cstdlib> // или stdlib.h
// #include <console.h> // для Macintosh
int main(int argc, char * argv[])
{
   using namespace std;
   // argc = ccommand(&argv); // для Macintosh
   if (argc == 1) // выйти, если нет аргументов
   {
      cerr << "Использование: " << argv[0] << " имя[имена] файлов\n";
      exit(EXIT_FAILURE);
   }
   ifstream fin; // открыть поток
   long count;
   long total = 0;
   char ch;
   for (int file = 1; file < argc; file++)
   {
```

```
      fin.open(argv[file]); // подключить поток argv[file]
      if (!fin.is_open())
      {
         cerr << "Не удалось открыть " << argv[file] << endl;
         fin.clear();
         continue;
      }
      count = 0;
      while (fin.get(ch))
         count++;
      cout << count << " символов в " << argv[file] << endl;
      total += count;
      fin.clear(); // необходимо для некоторых реализаций
      fin.close(); // отключиться от файла
   }
   cout << total << " символов во всех файлах\n";
   return 0;
}
```

Замечание по совместимости

Некоторые реализации C++ требуют вызова `fin.clear()` в конце программы, другие — нет. Это зависит от того, сбрасывается ли состояние потока автоматически при ассоциации нового файла с объектом типа `ifstream`. Поэтому не помешает использовать `fin.clear()`, даже если в этом нет необходимости.

В системе DOS, например, вы можете скомпилировать листинг 17.17 в исполняемый файл по имени `count.exe`. Пример его запуска выглядит так:

```
C>count
Использование: c:\count.exe имя[имена] файлов
C>count paris rome
3580 символов в paris
4886 символов в rome
8466 символов во всех файлах
C>
```

Отметим, что программа использует `cerr` для вывода сообщений об ошибках. Важный момент: сообщение использует `argv[0]` вместо `count.exe`:

```
cerr << "Использование: " << argv[0] << "имя[имена] файлов\n";
```

Таким образом, если вы измените имя исполняемого файла, программа автоматически использует новое имя.

Программа вызывает метод `is_open()` для того, чтобы проверить, что запрошенный файл удалось открыть. Рассмотрим этот момент подробнее.

Режимы файла

Режим файла описывает, как файл будет использоваться: для чтения, записи, добавления информации и так далее. Когда вы ассоциируете поток с файлом либо инициализацией объекта файлового потока именем файла, либо с помощью метода `open()`, вы можете предоставить второй аргумент, описывающий режим файла:

```
ifstream fin("banjo", mode1);  // конструктор с аргументом режима
ofstream fout();
fout.open("harp", mode2);      // open()с аргументом режима
```

Класс ios_base определяет тип openmode, представляющий режим. Подобно типу fmtflags и iostate, он имеет тип bitmask. (В старые времена он имел тип int.) Вы можете выбрать одну из нескольких констант, определенных в классе ios_base, чтобы специфицировать режим. В табл. 17.8 перечислены константы и представлено их назначение. Файловый ввод-вывод C++ претерпел некоторые изменения, чтобы быть совместимым с файловым вводом-выводом ANSI C.

Таблица 17.8. Константы режима файлов

Константа	Значение
ios_base::in	Открыть файл для чтения.
ios_base::out	Открыть файл для записи.
ios_base::ate	Перейти к концу файла после открытия.
ios_base::app	Добавлять к концу файла.
ios_base::trunc	Усечь файл, если он существует.
ios_base::binary	Бинарный файл.

Если конструкторы ifstream и ofstream, а также методы open() принимают два аргумента, как нам трактовать их вызов с одним аргументом в предшествующих примерах? Как вы можете предположить, прототипы этих функций-членов класса предусматривают значения по умолчанию для второго аргумента (аргумента, описывающего режим файла). Например, метод ifstream по имени open() и конструктор используют в качестве значения по умолчанию для аргумента режима ios_base::in (открыть для чтения), а метод ofstream по имени open() и конструктор применяют в качестве значения по умолчанию ios_base::out | ios_base::trunc (открыть для чтения и усечь файл). Битовая операция ИЛИ (|) служит для комбинирования двух битовых значений в одно, которое может быть использовано для одновременной установки обоих битов. Класс fstream не предусматривает режима по умолчанию, поэтому при создании объектов данного класса вы должны указывать режим явно.

Отметим, что флаг ios_base::trunc означает, что существующий файл должен быть усечен до нулевого размера, прежде чем он начнет принимать вывод из программы; то есть, его предыдущее содержимое отбрасывается. Хотя такое поведение снижает риск переполнения дискового пространства, вы можете представить себе ситуации, в которых вам не нужно стирать содержимое файла при его открытии. Конечно же, в C++ предусмотрены и другие варианты. Если, например, вы хотите предохранить содержимое файла и добавить (дописать) новый материал в его конец, вы можете воспользоваться режимом ios_base::app:

```
ofstream fout("bagels", ios_base::out | ios_base::app);
```

И снова этот код использует операцию | для комбинирования режимов. Поэтому ios_base::out | ios_base::app означает, что нужно включить и режим out, и режим app (рис. 17.6).

Вы можете ожидать некоторых различий между реализациями C++. Например, некоторые из них позволяют пропустить ios_base::out в предыдущем примере, а некоторые — нет. Если вы не используете режим по умолчанию, более безопасный подход — указывать все элементы режим явно.

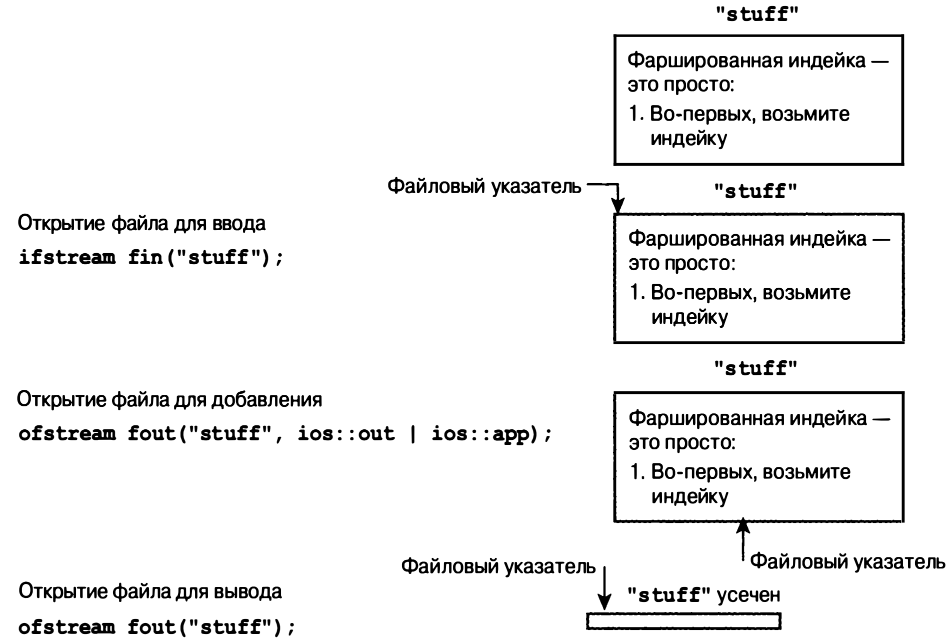


Рис. 17.6. Некоторые режимы открытия файла

Некоторые компиляторы не поддерживают все варианты из таблицы 17.8, а некоторые могут предлагать другие варианты помимо перечисленных в таблице. Следствием этих отличий является то, что возможно, вам придется внести некоторые изменения в последующие примеры, чтобы использовать их в вашей системе. Хорошая новость состоит в том, что разработка стандарта C++ предлагает большую степень согласованности.

Стандарт C++ определяет части файлового ввода-вывода в терминах их эквивалентов из стандарта ввода-вывода ANSI C. Оператор C++ вроде

```
ifstream fin(имя_файла, режим_c++);
```

реализуется, как если бы он использовал функцию C `fopen()`:

```
fopen(имя_файла, режим_c);
```

Здесь *режим_c++* — значение типа `openmode`, такое как `ios_base::in`, а *режим_c* — соответствующая строка режима C вроде `"r"`. В табл. 17.9 показаны соответствия между режимами C++ и C. Отметим, что `ios_base::out` сам по себе вызывает усечение, но не делает этого, если применяется в комбинации с `ios_base::in`. Неперечисленные комбинации, такие как `ios_base::in | ios_base::trunc`, предохраняют файл от открытия. Метод `is_open()` обнаруживает этот сбой.

Следует отметить, что `ios_base::ate` и `ios_base::app` переносит вас (а точнее, файловый указатель) в конец открытого файла. Разница между этими двумя режимами в том, что `ios_base::app` позволяет только добавлять данные в конец файла, в то время как `ios_base::ate` только устанавливает указатель в конец файла.

Таблица 17.9. Режимы открытия файлов C и C++

Режим C++	Режим C	Значение
`ios_base::in`	`"r"`	Открыть для чтения.
`ios_base::out`	`"w"`	То же, что `ios_base::out| ios_base::trunk`.
`ios_base::out | ios_base::trunc`	`"W"`	Открыть для записи с усечением существующего файла.
`ios_base::out | ios_base::app`	`"a"`	Открыть для записи с добавлением.
`ios_base::in | ios_base::out`	`"r+"`	Открыть для чтения и записи с разрешением записи с произвольного места файла.
`ios_base::in | ios_base::out | ios_base::trunc`	`"w+"`	Открыть для чтения и записи с усечением существующего файла.
режим_c++ | `ios_base::binary`	`"`*cmode*`b"`	Открыть в режиме *режим_c++* или соответствующем *режим_c*, в бинарном (не текстовом) режиме. Например, `ios_base::in|ios_base::binary` становится `"rb"`.
режим_c | `ios_base::ate`	`"`*cmode*`"`	Открыть в указанном режиме и перейти в конец файла. C использует отдельный вызов функции вместо кода режима. Например, `ios_base::in | ios_base::ate` транслируется в режим `"r"` с последующим вызовом `fseek(file, 0, SEEK_END)`.

Ясно, что существует множество возможных комбинаций режимов. Мы рассмотрим только наиболее типичные.

Добавление к файлу

Рассмотрим программу, которая дописывает данные в конец файла. Программа поддерживает файл, содержащий список гостей. Когда она начинает выполнение, то отображает текущее содержимое файла, если он уже существует. Она может использовать метод `is_open()` после попытки открытия файла для проверки, существует ли он. Далее программа открывает файл для вывода, используя флаг режима `ios_base::app`. Затем она принимает ввод с клавиатуры, чтобы дописать информацию в файл. И, наконец, программа отображает измененное содержимое файла. Код в листинге 17.18 иллюстрирует, как все это можно реализовать на практике. Обратите внимание, что программа применяет метод `is_open()` для проверки успешности открытия файла.

Замечание по совместимости

Файловый ввод-вывод — возможно, наименее стандартизованный аспект C++ в его ранние годы, и даже сейчас многие компиляторы не полностью соответствуют современному стандарту. Например, некоторые используют такой режим, как `nocreate`, который не входит в современный стандарт. Также лишь некоторые компиляторы требуют вызова `fin.clear()` перед открытием того же файла для чтения второй раз.

Листинг 17.18. append.cpp

```
// append.cpp -- добавление информации в файл
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib> // (или stdlib.h) для exit()
const char * file = "guests.txt";
int main()
{
   using namespace std;
   char ch;
   // показать первоначальное содержимое
   ifstream fin;
   fin.open(file);
   if (fin.is_open())
   {
      cout << "Текущее содержимое файла "
         << file << ":\n";
      while (fin.get(ch))
         cout << ch;
      fin.close();
   }
   // добавить новые имена
   ofstream fout(file, ios::out | ios::app);
   if (!fout.is_open())
   {
      cerr << "Невозможно открыть файл " << file << " для вывода.\n";
      exit(EXIT_FAILURE);
   }
   cout << "Введите имена гостей (или пустую строку для завершения):\n";
   string name;
   while (getline(cin,name) && name.size() > 0)
   {
      fout << name << endl;
   }
   fout.close();
   // показать обновленный файл
   fin.clear(); // не обязательно для некоторых компиляторов
   fin.open(file);
   if (fin.is_open())
   {
      cout << "Новое содержимое файла "
         << file << ":\n";
      while (fin.get(ch))
         cout << ch;
      fin.close();
   }
   cout << "Готово.\n";
   return 0;
}
```

Вот как выглядит пример первого запуска программы из листинга 17.18:

```
Введите имена гостей (или пустую строку для завершения):
Sylvester Ballone
Phil Kates
Bill Ghan
Новое содержимое файла guests.txt:
Sylvester Ballone
Phil Kates
Bill Ghan
Готово.
```

При первом запуске файл `guests.txt` не существовал, поэтому программа не показывает его первоначальное содержимое.

Однако когда программа будет запущена в следующий раз, файл `guests.txt` уже будет существовать, поэтому программа сначала покажет его содержимое. Кроме того, обратите внимание, что новые данные добавляются в конец файла, а не заменяют старые:

```
Текущее содержимое файла guests.txt:
Sylvester Ballone
Phil Kates
Bill Ghan
Введите имена гостей (или пустую строку для завершения):
Greta Greppo
LaDonna Mobile
Fannie Mae
Новое содержимое файла guests.txt:
Sylvester Ballone
Phil Kates
Bill Ghan
Greta Greppo
LaDonna Mobile
Fannie Mae
Готово.
```

Вы можете просмотреть содержимое `guests.txt` в любом текстовом редакторе, включая тот, который используете для написания исходного кода.

Бинарные файлы

Когда вы сохраняете данные в файле, то сохраняете их либо в текстовом, либо в бинарном формате. Текстовая форма означает, что вы храните все в виде текста — даже числа. Например, сохранение значения `2.324216e+07` в текстовой форме означает необходимость хранения 13 байт, используемых для записи этого числа. Это потребует преобразования внутреннего компьютерного представления числа с плавающей точкой в символьную форму, и это как раз то, что делает операция вставки <<. Бинарный формат, с другой стороны, означает хранение внутреннего компьютерного представления значения. То есть, вместо хранения символов компьютер сохраняет (как правило) 64-битное представление значения типа `double`. Для символа бинарное представление совпадает с текстовым представлением — то есть бинарным представлением ASCII-кода (или эквивалента) символа. Для чисел, однако, бинарное представление значительно отличается от символьного (рис. 17.7).

Бинарное представление 0.375

Бит знака	Биты экспоненты	Бинарные биты дробной части
0	0111110	11000000000000000000000

Текстовое представление 0.375

00110000	00101110	00110011	00110111	00110111
Код символа 0	Код символа	Код символа 3	Код символа 7	Код символа 5

Рис. 17.7. Бинарное и текстовое представление числа с плавающей точкой

Каждый формат имеет свои преимущества. Текстовый формат легко читать. С ним вы можете использовать обычный редактор или текстовый процессор для чтения и редактирования текстового файла. Вы можете легко переносить текстовый файл с одной компьютерной системы на другую. Бинарный формат — более точный для представления чисел, потому что сохраняет правильное внутреннее представление значения. Нет никаких ошибок, связанных с преобразованием или округлением. Сохранение данных в бинарном формате может выполняться быстрее, поскольку не происходит никакого преобразования и можно сохранять их более крупными порциями. К тому же, в зависимости от природы данных, бинарный формат требует меньше места. Однако передача их в другую систему может оказаться проблемой, если эта система использует другое внутреннее представление значений. Даже разные компиляторы на одной и той же системе могут использовать различное внутреннее представление структурных слоев. В этих случаях вы (или кто-то другой) можете написать программу для трансляции одного формата данных в другой.

Рассмотрим конкретный пример. Предположим, у нас есть следующее определение и объявление структуры:

```
const int LIM = 20;
struct planet
{
   char name[LIM];     // название планеты
   double population;  // население
   double g;           // ускорение свободного падения
};
planet pl;
```

Чтобы сохранить содержимое структуры pl в текстовом формате, вы можете использовать следующее:

```
ofstream fout("planets.dat", ios_base::out | ios_base::app);
fout << pl.name << " " << pl.population << " " << pl.g << "\n";
```

Отметим, что вам придется указать каждый член структуры явно, используя операцию членства, и вы должны разделять соседние данные для ясности. Если структура содержит, скажем, 30 членов, это может оказаться утомительным.

Чтобы сохранить ту же информацию в бинарной форме, вы можете сделать так:

```
ofstream fout("planets.dat",
ios_base:: out | ios_base::app | ios_base::binary);
fout.write( (char *) &pl, sizeof pl);
```

Этот код сохраняет всю структуру как единое целое, используя внутреннее компьютерное представление данных. Вы не сможете прочитать этот файл как текст, но информация будет сохранена более компактно и точно, чем в виде текста. Этот подход отличается следующими аспектами:

- Он использует бинарный режим файла.
- Он использует функцию-член `write()`.

Посмотрим на эти изменения более внимательно.

Некоторые системы, такие как DOS, поддерживают два формата файлов: текстовый и бинарный. Если вы хотите сохранить данные в бинарной форме, то должны использовать файлы бинарного формата. На C++ вы делаете это, указывая в режиме файла константу `ios_base::binary`. Если вы хотите знать, почему вы должны это делать в системе DOS, прочтите дискуссию в следующей врезке "Бинарные файлы и текстовые файлы".

Бинарные файлы и текстовые файлы

Использование бинарного режима файла заставляет программу переносить данные из памяти в файл или обратно без какого-либо скрытого преобразования. В текстовом режиме по умолчанию это вовсе не обязательно. Например, рассмотрим текстовые файлы DOS. Они представляют новую строку комбинацией двух символов: возврат каретки, перевод строки. Текстовые файлы Macintosh представляют новую строку только возвратом каретки. Файлы Unix и Linux представляют новую строку только переводом строки. C++, который вырос на почве Unix, также представляет новую строку символом перевода строки. Программы на C++ в среде DOS, когда записывают файл в текстовом режиме, автоматически транслируют новую строку C++ в комбинацию "возврат каретки, перевод строки"; программы на C++ в среде Macintosh транслируют новую строку в символ возврата каретки. При чтении текстового файла эти программы обратно преобразуют новую строку в форму, принятую в C++. Текстовый формат файлов может вызвать проблемы с бинарными данными, потому что байт в середине значения типа `double` может иметь тот же битовый шаблон, что и ASCII-код символа перевода строки. К тому же, существует различие в способе обнаружения конца файла. Поэтому вы должны использовать бинарный режим при сохранении данных в бинарном формате. (Системы Unix имеют только один режим файлов, поэтому там бинарный режим — это то же самое, что и текстовый.)

Чтобы сохранить данные в бинарной форме вместо текстовой, вы можете воспользоваться функцией-членом `write()`. Вспомним, что этот метод копирует указанное число байт из памяти в файл. Вы применяли его ранее в этой главе, чтобы копировать текст, но он может также копировать байт за байтом данные любого типа без каких бы то ни было преобразований. Например, если вы передадите ему адрес переменной `long` и укажете скопировать 4 байта, он скопирует в файл 4 байта, содержащие буквальное `long`-значение, не преобразуя его в текст. Единственным неудобством будет то, что придется использовать приведение типа указателя на `long` к указателю на `char`. Тот же подход можно использовать для копирования всей структуры `planet`. Чтобы получить количество байт, которые должны быть записаны, следует применить операцию `sizeof`:

```
fout.write( (char *) &pl, sizeof pl);
```

Этот оператор обращается к адресу структуры pl и копирует 36 байт (значение выражения sizeof pl) , начиная с указанного адреса, в файл, подключенный к fout.

Чтобы восстановить информацию из файла, вы используете соответствующий метод read() с объектом ifstream:

```
ifstream fin("planets.dat", ios_base::in | ios_base::binary);
fin.read((char *) &pl, sizeof pl);
```

Это скопирует sizeof pl байт из файла в структуру pl. Тот же подход можно использовать с классами, у которых нет виртуальных функций. В этом случае будут сохранены только члены-данные, но не методы. Если же класс имеет виртуальные функции, то скрытый указатель на таблицу указателей виртуальных функций также будет скопирован. Поскольку при следующем запуске программы таблица виртуальных функций может быть размещена в другом месте, то копирование из файла старого значения указателя в значение объекта может его разрушить. (См. комментарии к упражнению 6 по программированию.)

Совет

Функции-члены read() и write() дополняют друг друга. Вы используете read() для восстановления данных, которые были записаны в файл методом write().

Код в листинге 17.19 использует эти методы для создания и чтения бинарного файла. По форме эта программа подобна приведенной в листинге 17.18, но применяет write() и read() вместо операции вставки, а также метод get(). Она также применяет манипуляторы для форматирования экранного вывода.

Замечание по совместимости

Хотя концепция бинарного файла является частью ANSI C, некоторые реализации C и C++ не предусматривают поддержку режима бинарного файла. Причина этого упущения в том, что в некоторых системах существует только один тип файлов, поэтому вы можете использовать бинарные операции вроде read() и write() с файлами стандартного формата. Таким образом, если ваша реализация отвергает ios_base::binary как корректную константу, вы можете просто исключить ее из своей программы. Если ваша реализация не поддерживает манипуляторы fixed и right, вы можете применить cout.setf(ios_base::fixed, ios_base::floatfield) и cout.setf(ios_base::right, ios_base::adjustfield). Кроме того, придется заменить ios_base на ios. Другие старые компиляторы могут также иметь собственные отличия.

Листинг 17.19. binary.cpp

```
// binary.cpp -- бинарный файловый ввод-вывод
#include <iostream> // не требуется в большинстве систем
#include <fstream>
#include <iomanip>
#include <cstdlib> // (или stdlib.h) для exit()
inline void eatline() { while (std::cin.get() != '\n') continue; }
struct planet
{
   char name[20];      // название планеты
   double population;  // население
   double g;           // ускорение свободного падения
};
const char * file = "planets.dat";
```

```
int main()
{
   using namespace std;
   planet pl;
   cout << fixed << right;
   // показать начальное содержимое
   ifstream fin;
   fin.open(file, ios_base::in |ios_base::binary); // бинарный файл
   //Примечание: некоторые системы не принимают режим ios_base::binary
   if (fin.is_open())
   {
      cout << "Текущее содержимое файла "
           << file << ":\n";
      while (fin.read((char *) &pl, sizeof pl))
      {
         cout << setw(20) << pl.name << ": "
              << setprecision(0) << setw(12) << pl.population
              << setprecision(2) << setw(6) << pl.g << endl;
      }
      fin.close();
   }
   // добавить новые данные
   ofstream fout(file,
   ios_base::out | ios_base::app | ios_base::binary);
   //Примечание: некоторые системы не принимают режим ios::binary
   if (!fout.is_open())
   {
      cerr << "Не удается открыть файл " << file << " для вывода:\n";
      exit(EXIT_FAILURE);
   }
   cout << "Введите название планеты (или пустую строку для завершения):\n";
   cin.get(pl.name, 20);
   while (pl.name[0] != '\0')
   {
      eatline();
      cout << "Введите численность населения планеты: ";
      cin >> pl.population;
      cout << "Введите ускорение свободного падения планеты: ";
      cin >> pl.g;
      eatline();
      fout.write((char *) &pl, sizeof pl);
      cout << "Введите название планеты (или пустую строку для завершения):\n ";
      cin.get(pl.name, 20);
   }
   fout.close();
   // показать измененный файл
   fin.clear(); // не обязательно в некоторых реализациях, но не помешает
   fin.open(file, ios_base::in | ios_base::binary);
   if (fin.is_open())
   {
      cout << "Новое содержимое файла "
           << file << ":\n";
      while (fin.read((char *) &pl, sizeof pl))
      {
```

```
        cout << setw(20) << pl.name << ": "
             << setprecision(0) << setw(12) << pl.population
             << setprecision(2) << setw(6) << pl.g << endl;
      }
      fin.close();
   }
   cout << "Готово.\n";
   return 0;
}
```

Ниже показан пример первого запуска программы из листинга 17.19:

```
Введите название планеты (или пустую строку для завершения):
Земля
Введите численность населения планеты: 5962000000
Введите ускорение свободного падения планеты: 9.81
Введите название планеты (или пустую строку для завершения):
Новое содержимое файла planets.dat:
Земля: 5932000000 9.81
Готово.
```

А это — пример следующего ее запуска:

```
Текущее содержимое файла planets.dat:
Земля: 5932000000 9.81
Введите название планеты (или пустую строку для завершения):
Планета Билла
Введите численность населения планеты: 23020020
Введите ускорение свободного падения планеты: 8.82
Введите название планеты (или пустую строку для завершения):
Новое содержимое файла planets.dat:
Земля: 5932000000 9.81
Планета Билла: 23020020 8.82
Готово.
```

Вы уже видели основные возможности этой программы, но задержимся на ней еще немного. Программа использует следующий код (в форме встроенной функции `eatlene()`) после чтения значения `g` планеты:

```
while (std::cin.get() != '\n') continue;
```

Это читает и отбрасывает ввод вплоть до символа новой строки. Рассмотрим следующий оператор ввода в цикле:

```
cin.get(pl.name, 20);
```

Если символ новой строки останется на своем месте, этот оператор прочтет его как пустую строку и завершит цикл.

Вы можете спросить: почему бы не воспользоваться объектом `string` вместо массива символов для члена `name` структуры `planet`. Ответ: этого делать нельзя — по крайней мере, без серьезных изменений в дизайне. Проблема в том, что объект `string` в действительности не содержит строку внутри себя. Вместо этого он содержит указатель на область памяти, где хранится строка. Поэтому если вы будете копировать структуру в файл, то при этом скопируете не данные самой строки, а адрес в памяти, по которому она расположена. Когда вы запустите программу снова, этот адрес утратит смысл.

Произвольный доступ

Для нашего последнего примера давайте рассмотрим применение произвольного доступа. *Произвольный доступ* означает возможность перемещения в любую позицию файла вместо последовательного перемещения по нему. Подход с произвольным доступом часто используется в файлах баз данных. Программа будет поддерживать отдельный индексный файл, в котором будет храниться информация о местоположении данных в основном файле. Затем она будет "перепрыгивать" непосредственно в заданное место, читать там данные и, возможно, модифицировать их. Этот подход проще всего реализовать, если файл будет содержать набор записей одинаковой длины. Каждая запись представляет связанный набор данных. Например, в примере, показанном в листинге 17.19, каждая запись файла будет представлять всю информацию об определенной планете. Запись в файле достаточно естественно соответствует структуре или классу программы.

Этот пример основан на программе работы с бинарным файлом из листинга 17.19, при этом используя преимущество того факта, что структура `planet` представляет собой шаблон записи файла. Чтобы добавить творческое начало в программирование, этот пример открывает файл в режиме чтения-записи, так что он имеет возможность и читать и модифицировать записи. Вы можете реализовать это, создав объект `fstream`. Класс `fstream` унаследован от класса `iostream`, который, в свою очередь, базируется на классах `istream` и `ostream`, а потому наследует методы обоих. Он также наследует два буфера — один для чтения и один для записи — и синхронизирует управление этим двумя буферами. То есть, как только программа читает файл или пишет в него, то перемещает одновременно указатель ввода в буфере ввода и указатель вывода в буфере вывода.

Пример делает следующее:

1. Отображает текущее содержимое файла `planets.dat`.
2. Запрашивает, какую запись вы хотите модифицировать.
3. Модифицирует запись.
4. Показывает измененное содержимое файла.

Более претенциозная программа может использовать меню, чтобы дать вам возможность выбирать из списка доступных действий независимо, но эта версия выполняет каждое действие лишь однажды. Этот упрощенный подход позволит вам исследовать несколько аспектов чтения-записи файлов, не слишком погружаясь в проблемы дизайна программы.

Внимание!

Эта программа предполагает, что файл `planets.dat` уже был создан программой `binary.cpp` из листинга 17.19.

Первый вопрос, на который следует ответить — какой режим файла использовать? Чтобы читать файл, вам понадобится режим `ios_base::in`. Для бинарного ввода-вывода понадобится режим `ios_base::binary`. (Опять-таки, в некоторых нестандартных системах вы можете опустить, и даже должны будете опустить этот режим.) Для того чтобы записывать в файл, понадобится режим `ios_base::out` или `ios_base::app`. Однако режим дополнения позволит программе только добавлять данные в конец файла. Остальная часть файла будет доступна только для чтения; то

есть вы сможете читать оригинальные данные, но не модифицировать их — поэтому вы должны использовать ios_base::out. Как показано в табл. 17.9, совместное применение режимов in и out обеспечивает режим чтения-записи, поэтому вам нужно только добавить элемент binary. Как уже упоминалось, режимы комбинируются с помощью операции |. То есть, в результате вам понадобится следующий оператор, чтобы подготовить файл к обработке:

```
finout.open(file,ios_base::in | ios_base::out | ios_base::binary);
```

Далее вам потребуется способ перемещения по файлу. Класс fstream наследует для этого два метода: seekg() перемещает указатель ввода в заданную позицию файла, а seekp() перемещает указатель вывода в заданную позицию файла. (На самом деле, поскольку класс fstream использует буферы для промежуточного хранения данных, эти указатели указывают на положение в буферах, а не в реальном файле.) Вы можете также применять seekg() с объектом ifstream, а seekp() — с объектом ofstream. Прототипы seekg() выглядят так:

```
basic_istream<charT,traits>& seekg(off_type, ios_base::seekdir);
basic_istream<charT,traits>& seekg(pos_type);
```

Как видите, это шаблоны. В этой главе используется специализация шаблона для типа char. Для такой специализации эти два прототипа эквивалентны следующим:

```
istream & seekg(streamoff, ios_base::seekdir);
istream & seekg(streampos);
```

Первый прототип представляет перемещение в позицию файла, измеренную в байтах, как смещение от позиции, заданной вторым аргументом. Второй прототип устанавливает позицию указателя в файле, измеренную в байтах от начала файла.

Повышение типа

Когда C++ был молодым языком, для методов seekg() жизнь была проще. Типы streamoff и streampos были описаны как typedef для некоторого целочисленного типа, такого как long. Однако вопрос о создании переносимого стандарта привел к тому, что обнаружилось, что целочисленный аргумент может предоставить недостаточно информации для некоторых файловых систем, поэтому streamoff и streampos стали структурами или типами классов, позволяющими применять к ним некоторые базовые операции вроде применения целочисленного значения в качестве инициализирующего. Далее, старый класс istream был заменен шаблоном basic_istream, а streamoff и streampos — шаблонными типами pos_type и off_type. Аналогично, вы можете теперь использовать типы wstreamoff и wstreampos, если применяете seekg() с объектом типа wistream.

Посмотрим на аргументы первого прототипа seekg(). Значения типа streamoff используются для измерения смещений в байтах от определенного положения в файле. Аргумент streamoff представляет позицию в файле в байтах, измеренную как смещение от одного из трех возможных положений. (Тип может быть определен как целочисленный или же как класс.) Аргумент seek_dir — это другой целочисленный тип, который определен вместе с тремя возможными значениями в классе ios_base. Константа ios_base::beg означает отсчет смещения от начала файла. Константа ios_base::cur означает отсчет смещения от текущего положения указателя в файле. Константа ios_base::end означает отсчет смещения от конца файла. Вот некоторые примеры вызовов, предполагающие, что fin — объект класса ifstream:

```
fin.seekg(30, ios_base::beg);     // 30 байт от начала
fin.seekg(-1, ios_base::cur);     // назад на 1 байт
fin.seekg(0, ios_base::end);      // перейти в конец файла
```

Теперь посмотрим на второй прототип. Значения типа `streampos` описывают позицию в файле. Это может быть класс, но если так, то класс должен включать конструктор с аргументов `streamoff` и конструктор с целочисленным аргументом, представляя способ преобразования обоих типов в значения `streampos`. Значение `streampos` представляет абсолютную позицию в файле, измеренную от его начала. Вы можете трактовать позицию `streampos`, как если бы она измеряла положение указателя файла в байтах от начала файла, причем первым байтом был бы нулевой. Поэтому оператор

```
fin.seekg(112);
```

устанавливает указатель файла на 112-й байт, который на самом деле является 113-м в файле. Если вы хотите проверить текущую позицию файлового указателя, то можете для этого использовать метод `tellg()` для входных потоков и методы `tellp()` — для выходных. Каждый из них возвращает значение `streampos`, представляющее текущую позицию в байтах, измеренную от начала файла. Когда вы создаете объект `fstream`, указатели ввода и вывода перемещаются в тандеме, поэтому `tellg()` и `tellp()` возвращают одно и то же значение. Но если вы применяете объект `istream` для управления входным потоком и объект `ostream` для управления выходным потоком на одном и том же файле, то указатели ввода и вывода перемещаются независимо друг от друга, и `tellg()` и `tellp()` могут возвращать разные значения. Вы можете использовать `seekg()`, чтобы перейти в начало файла. Ниже показан фрагмент кода, который открывает файл, переходит в его начало и отображает содержимое:

```
fstream finout; // читаем и пишем потоки
finout.open(file,ios::in | ios::out |ios::binary);
// Примечание: некоторые системы Unix требуют опустить | ios::binary
int ct = 0;
if (finout.is_open())
{
   finout.seekg(0); // перейти в начало файла
   cout << "Текущее содержимое файла "
        << file << ":\n";
   while (finout.read((char *) &pl, sizeof pl))
   {
      cout << ct++ << ": " << setw(LIM) << pl.name << ": "
           << setprecision(0) << setw(12) << pl.population
           << setprecision(2) << setw(6) << pl.g << endl;
   }
   if (finout.eof())
      finout.clear(); // очистить флаг eof
   else
   {
      cerr << "Ошибка чтения " << file << ".\n";
      exit(EXIT_FAILURE);
   }
}
else
{
   cerr << file << " не может быть открыт -- завершение.\n";
   exit(EXIT_FAILURE);
}
```

Это похоже на начало листинга 17.19, но с некоторыми изменениями и дополнениями. Во-первых, как только что было описано, программа использует объект fstream с режимом чтения-записи, и применяет seekg() для позиционирования файлового указателя в начало файла. (На самом деле, это не нужно в данном примере, но выступает в качестве демонстрации применения seekg().) Далее программа делает небольшое изменение в нумерации отображаемых записей. После этого вносится существенное дополнение:

```
if (finout.eof())
  finout.clear(); // очистить флаг eof
else
{
  cerr << "Ошибка чтения " << file << ".\n";
  exit(EXIT_FAILURE);
}
```

Проблема в том, что когда программа читает и отображает весь файл, то устанавливает элемент eofbit. Это сообщает программе, что работа с файлом завершена, и делает невозможными любые дальнейшие операции чтения или записи в файл. Применение метода clear() сбрасывает состояние потока, очищая eofbit. Теперь программа опять может обратиться к файлу. Часть else обрабатывает возможную ситуацию, когда программа прекращает чтение файла по некоторой другой причине, чем достижения конца файла — например, по причине аппаратного сбоя.

Следующий шаг состоит в идентификации записи, которая должна быть изменена с последующим изменением ее. Чтобы сделать это, программа запрашивает у пользователя номер записи. Умножение введенного номера на размер записи в байтах дает номер байта, с которого начинается искомая запись. Если record — номер записи, то требуемый номер байта будет record * sizeof pl:

```
cout << "Введите номер записи, которую нужно изменить: ";
long rec;
cin >> rec;
eatline(); // избавиться от символов новой строки
if (rec < 0 || rec >= ct)
{
   cerr << "Неверный номер записи -- завершение.\n";
   exit(EXIT_FAILURE);
}
streampos place = rec * sizeof pl;    // преобразовать в тип streampos
finout.seekg(place);                  // произвольный доступ
```

Переменная ct представляет количество записей; программа завершается, если вы пытаетесь выйти за пределы файла.

Далее программа отображает текущую запись:

```
finout.read((char *) &pl, sizeof pl);
cout << "Ваш выбор:\n";
cout << rec << ": " << setw(LIM) << pl.name << ": "
     << setprecision(0) << setw(12) << pl.population
     << setprecision(2) << setw(6) << pl.g << endl;
if (finout.eof())
   finout.clear(); // очистить флаг eof
```

После отображения записи программа позволяет вам изменить ее:

```
    cout << "Введите название планеты: ";
    cin.get(pl.name, LIM);
    eatline();
    cout << "Введите население планеты: ";
    cin >> pl.population;
    cout << "Введите ускорение свободного падения: ";
    cin >> pl.g;
    finout.seekp(place); // вернуться
    finout.write((char *) &pl, sizeof pl) << flush;
    if (finout.fail())
    {
       cerr << "Ошибка при попытке записи\n";
      exit(EXIT_FAILURE);
    }
```

Программа сбрасывает вывод, чтобы гарантировать, что файл будет обновлен перед тем, как перейти к следующей стадии обработки.

И, наконец, чтобы отобразить измененное содержимое файла, программа применяет `seekg()`, чтобы сбросить указатель файла на начало. В листинге 17.20 показан полный текст программы. Не забудьте, что она предполагает, что файл `planets.dat` уже создан с помощью программы `binary.cpp` и доступен.

Замечание по совместимости

Чем старше реализация, тем более вероятно, что она не соответствует стандарту C++. Некоторые системы не распознают флаг `binary`, манипуляторы `fixed` и `right`, а также `ios_base`.

Листинг 17.20. random.cpp

```
// random.cpp -- произвольный доступ к бинарному файлу
#include <iostream>    // для большинства систем не требуется
#include <fstream>
#include <iomanip>
#include <cstdlib>     // (или stdlib.h) для exit()
const int LIM = 20;
struct planet
{
    char name[LIM];     // название планеты
    double population;  // население
    double g;           // ускорение свободного падения
};
const char * file = "planets.dat"; // ПРЕДПОЛАГАЕТСЯ, ЧТО СУЩЕСТВУЕТ
                                   // (пример binary.cpp)
inline void eatline() { while (std::cin.get() != '\n') continue; }
int main()
{
    using namespace std;
    planet pl;
    cout << fixed;
    // показать начальное содержимое
    fstream finout;     // читать и писать потоки
    finout.open(file,
    ios_base::in | ios_base::out | ios_base::binary);
    //ПРИМЕЧАНИЕ: некоторые системы Unix требуют опустить | ios::binary
```

```
int ct = 0;
if (finout.is_open())
{
   finout.seekg(0); // перейти в начало
   cout << "Текущее содержимое файла "
        << file << ":\n";
   while (finout.read((char *) &pl, sizeof pl))
   {
      cout << ct++ << ": " << setw(LIM) << pl.name << ": "
           << setprecision(0) << setw(12) << pl.population
           << setprecision(2) << setw(6) << pl.g << endl;
   }
   if (finout.eof())
      finout.clear(); // очистить флаг eof
   else
   {
      cerr << "Ошибка чтения " << file << ".\n";
      exit(EXIT_FAILURE);
   }
}
else
{
   cerr << file << " файл не может быть открыт -- завершение.\n";
   exit(EXIT_FAILURE);
}
// изменить запись
cout << "Введите номер записи, которую нужно изменить: ";
long rec;
cin >> rec;
eatline(); // избавиться от символов новой строки
if (rec < 0 || rec >= ct)
{
   cerr << "Неверный номер записи -- завершение.\n";
  exit(EXIT_FAILURE);
}
streampos place = rec * sizeof pl; // преобразовать в тип streampos
finout.seekg(place);               // произвольный доступ
if (finout.fail())
{
   cerr << "Ошибка при попытке перемещения указателя\n";
   exit(EXIT_FAILURE);
}
finout.read((char *) &pl, sizeof pl);
cout << "Ваш выбор:\n";
cout << rec << ": " << setw(LIM) << pl.name << ": "
     << setprecision(0) << setw(12) << pl.population
     << setprecision(2) << setw(6) << pl.g << endl;
if (finout.eof())
   finout.clear(); // очистить флаг eof
cout << "Введите название планеты: ";
cin.get(pl.name, LIM);
eatline();
cout << "Введите население планеты: ";
cin >> pl.population;
cout << "Введите ускорение свободного падения: ";
```

```
    cin >> pl.g;
    finout.seekp(place);  // вернуться назад
    finout.write((char *) &pl, sizeof pl) << flush;
    if (finout.fail())
    {
       cerr << "Ошибка при попытке записи\n";
       exit(EXIT_FAILURE);
    }
    // показать измененный файл
    ct = 0;
    finout.seekg(0);      // перейти в начало файла
    cout << "Новое содержимое файла " << file
         << ":\n";
    while (finout.read((char *) &pl, sizeof pl))
    {
       cout << ct++ << ": " << setw(LIM) << pl.name << ": "
            << setprecision(0) << setw(12) << pl.population
            << setprecision(2) << setw(6) << pl.g << endl;
    }
    finout.close();
    cout << "Готово.\n";
    return 0;
}
```

Ниже представлен пример запуска программы из листинга 17.20 на основе файла planets.dat, в который добавилось несколько новых записей с тех пор, как вы его видели последний раз:

```
Текущее содержимое файла planets.dat:
0: Земля: 5333000000 9.81
1: Планета Билла: 23020020 8.82
2: Трантор: 58000000000 15.03
3: Триллан: 4256000 9.62
4: Фристоун: 3845120000 8.68
5: Таанагоот: 350000002 10.23
6: Марин: 232000 9.79
Введите номер записи, которую нужно изменить: 2
Ваш выбор:
2: Трантор: 58000000000 15.03
Введите название планеты: Trantor
Введите население: 59500000000
Введите ускорение свободного падения: 10.53
Новое содержимое файла planets.dat:
0: Земля: 5333000000 9.81
1: Планета Билла: 23020020 8.82
2: Трантор: 58000000000 10.53
3: Триллан: 4256000 9.62
4: Фристоун: 3845120000 8.68
5: Таанагоот: 350000002 10.23
6: Марин: 232000 9.79
Готово.
```

Используя технику, представленную в этой программе, вы можете расширить ее так, чтобы она позволяла вам добавлять новый материал и удалять записи. Если вы хотите расширить программу, то будет хорошей идеей реорганизовать ее, применив

классы и функции. Например, вы можете преобразовать структуру planet в определение класса, затем перегрузить операцию вставки << так, чтобы cout << pl отображало данные класса, отформатированные, как в приведенном примере. К тому же приведенный пример не заботится о проверке корректности ввода, поэтому вы можете добавить код, проверяющий корректность вводимых числовых данных.

Пример из практики: работа с временными файлами

При разработке приложений часто требуется использовать временные файлы, чье время жизни кратко и должно управляться программой. Думали ли вы когда-нибудь о том, как это реализовать на C++? В действительности достаточно просто создать временный файл, скопировать его содержимое в другой файл и уничтожить его. Прежде всего, вам нужно продумать схему именования ваших временных файлов. Но подождите, как вы можете гарантировать, что каждому из них будет присвоено уникальное имя? Стандартная функция tmpnam(), объявленная в cstdio, поможет вам:

```
char *tmpnam(char * pszName);
```

Функция tmpnam() создает временное имя и помещает его в строку в стиле C, на которую указывает pszName. Обе константы — L_tmpnam и TMP_MAX — определены в cstdio, ограничивая количество символов в имени файла и максимальное число вызовов tmpnam() без генерации повторяющихся имен файлов в текущем каталоге. Следующий пример генерирует 10 временных имен:

```
#include <cstdio>
#include <iostream>
int main()
{
    using namespace std;
    cout << "В этой системе можно генерировать вплоть до " << TMP_MAX
         << " временных имен, содержащих до " << L_tmpnam
         << " символов.\n";
    char pszName[ L_tmpnam ] = {'\0'};
    cout << "Десять таких имен:\n";
    for( int i=0; 10 > i; i++ )
    {
       tmpnam( pszName );
       cout << pszName << endl;
    }
    return 0;
}
```

Вообще говоря, используя tmpnam(), вы можете теперь генерировать TMP_MAX уникальных имен, каждое длиной до L_tmpnam символов. Сами имена зависят от компилятора. Вы можете запустить эту программу, чтобы посмотреть, какие имена сгенерирует ваш компилятор.

Внутреннее форматирование

Семейство iostream поддерживает ввод-вывод между программой и терминалом. Семейство fstream использует тот же интерфейс для обеспечения операций ввода-вывода между программой и файлом. Библиотека C++ также предлагает семейство sstream, которое обеспечивает тот же интерфейс для организации ввода-вывода между программой и объектом string. То есть вы можете использовать те же методы ostream, которые применяли с cout для форматирования информации в объекте

string, а также использовать методы istream, такие как getline(), чтобы читать информацию из объекта string. Процесс чтения форматированной информации из объекта string и записи форматированной информации в объект string называется *внутренним форматированием*. Взглянем кратко на эти возможности. (Семейство sstream поддержки string замещает семейство strstream.h, поддерживающее символьные массивы.)

Заголовочный файл sstream определяет класс ostringstream, который наследуется от класса ostream. (Существует также класс wostringstream, базирующийся на wostream, — для расширенных символьных наборов.) Если вы создаете объект ostringstream, то можете писать в него информацию, которая сохраняется. Вы можете применять с объектом ostringstream те же методы, что и с объектом cout. То есть вы можете делать нечто вроде:

```
ostringstream outstr;
double price = 281.00;
char * ps = " за копию стандарта ISO/EIC C++!";
outstr.precision(2);
outstr << fixed;
outstr << "Заплатите всего $" << price << ps << end;
```

Форматированный текст отправляется в буфер, и объект использует динамическое выделение памяти для расширения размера буфера при необходимости. Класс ostringstream имеет функцию-член по имени str(), которая возвращает объект string, инициализированный содержимым буфера:

```
string mesg = outstr.str();//возвращает строку с форматированной информацией
```

Применение str() "замораживает" объект, и вы более не можете записывать в него. В листинге 17.21 предлагается короткий пример внутреннего форматирования.

Листинг 17.21. strout.cpp

```
// strout.cpp -- внутреннее форматирование (вывод)
#include <iostream>
#include <sstream>
#include <string>
int main()
{
   using namespace std;
   ostringstream outstr; // manages a string stream
   string hdisk;
   cout << "Какова марка вашего жесткого диска? ";
   getline(cin, hdisk);
   int cap;
   cout << "Какова его емкость в Гбайт? ";
   cin >> cap;
   // записать форматированную информацию в строковый поток
   outstr << "Жесткий диск " << hdisk << " имеет емкость "
          << cap << " Гбайт.\n";
   string result = outstr.str(); // сохранить результат
   cout << result; // показать содержимое
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 17.21:

```
Каково имя вашего жесткого диска? Dataraptur
Какова его емкость в Гбайт? 250
Жесткий диск Dataraptur имеет емкость 250 Гбайт.
```

Класс `istringstream` позволяет использовать семейство методов `istream` для чтения данных из объекта `istringstream`, который может быть инициализирован объектом `string`. Предположим, что `facts` — это объект типа `string`. Чтобы создать объект `istringstream`, ассоциированный с этой строкой, вы можете использовать следующее:

```
istringstream instr(facts); // использует facts для инициализации потока
```

Затем вы применяете методы `istream` для чтения данных из `instr`. Например, если `instr` содержит ряд целых числе в строковом формате, вы можете прочесть их следующим образом:

```
int n;
int sum = 0;
while (instr << n)
sum += num;
```

В листинге 17.22 применяется перегруженная операция >> для чтения содержимого строки по одному слову за раз.

Листинг 17.22. strin.cpp

```
// strin.cpp -- форматированное чтение из символьного массива
#include <iostream>
#include <sstream>
#include <string>
int main()
{
   using namespace std;
   string lit = "It was a dark and stormy day, and "
      " the full moon glowed brilliantly. ";
   istringstream instr(lit);   // использовать buf для ввода
   string word;;
   while (instr >> word)       // читать по одному слову
      cout << word << endl;
   return 0;
}
```

Вывод программы из листинга 17.22 выглядит следующим образом:

```
It
was
a
dark
and
stormy
day,
and
the
full
moon
glowed
brilliantly.
```

Короче говоря, классы istringstream и istringstream предоставляют в ваше распоряжение всю мощь методов классов istream и ostream для управления символьными данными, сохраненными в строках.

Что теперь?

Если вы внимательно работали над материалом этой книги, то уже должны ухватить основные правила C++. Однако все это — лишь начало изучения языка. Вторая стадия изучения — использовать язык эффективно, и это — длинное путешествие. Лучше всего попасть в рабочую среду, в которой вам пришлось бы иметь дело с хорошим кодом C++ и опытными программистами. Также теперь, когда вы знаете основы C++, вы можете читать книги, которые концентрируются на более сложных темах и объектно-ориентированном программировании. В приложении З перечислены некоторые полезные источники.

Одной из целей ООП является облегчение разработки и повышение надежности больших проектов. Одним из главных действий при подходе ООП является разработка классов, которые представляют ситуацию (называемую *проблемной областью*), которую вы моделируете. Поскольку реальные проблемы часто сложны, выбор правильного множества классов может оказаться достаточно трудным. Создание сложной системы "с нуля" обычно не работает. Вместо этого лучше использовать итеративный, эволюционный подход. Чтобы достичь этого, профессионалы в этой области разработали ряд технологий и стратегий. В частности, важно пройти как можно больше итерационных шагов эволюции на стадии анализа и проектирования вместо того, чтобы писать и переписывать код.

Две общепринятые технологии, которые служат этому — анализ случаев использования (use-case analysis) и CRC-карты. При анализе случаев использования команда разработчиков составляет перечень общих способов, или сценариев использования будущей системы, идентифицируя при этом элементы, действия и ответственности, которые предполагают возможные классы и их оснащение. Использование CRC-карт (сокращение от "Class/Responsibilities/Collaborations" — "Класс/ответственность/взаимодействие", или событийное взаимодействие классов) — простой способ анализа таких сценариев. Команда разработчиков создает карту-указатель для каждого класса. На карте указывается имя класса, его назначение (зона ответственности), такое как представляемые данные и выполняемые действия, и коллаборанты класса — в виде списков классов, с которыми должен взаимодействовать данный. Затем команда проходит по сценарию, используя интерфейсы, описанный в CRC-картах. Это может привести к уточнению предположений относительно классов, перемещения ответственности и так далее.

На более высоком уровне находится методики систематизации работы со всем проектом. Наиболее современная из них — универсальный язык моделирования (Unified Modeling Language — UML). UML — это не язык программирования, а скорее, язык для представления анализа и проектирования программного проекта. Он был разработан Гради Бучем (Grady Booch), Джимом Рамбо (Jim Rumbaugh) и Айваром Якобсоном (Ivar Jacobson), которым принадлежит авторство трех более ранних языков моделирования: метод Буча (Booch Method), OMT (Object Modeling Technique — техника объектного моделирования) и OOSE (Object-Oriented Software Engineering — разработка объектно-ориентированного программного обеспечения) соответственно. UML — наследник этих трех методик.

В дополнение к повышению вашего уровня понимания C++ в целом, возможно, вы захотите изучить специфические библиотеки классов. Например, Microsoft, Borland и Metrowerks предлагают богатые библиотеки классов, облегчающие программирование в среде Windows, а Metrowerks также предлагает подобные средства для программирования в среде Macintosh.

Резюме

Поток — это течение данных в или из вашей программы. Буфер — область памяти для временного хранения, служащая посредником между программой и файлом или другими устройствами ввода-вывода. Информация может быть передана между буфером и файлом, используя большие порции данных, имеющие размер, который позволяет наиболее эффективно работать с такими устройствами, как дисковые приводы. Информация также может передаваться между буфером и программой байт за байтом, что часто гораздо более удобно для обработки в программе. C++ обрабатывает ввод, подключая буферизованный поток к программе и источнику данных. Аналогично, C++ обрабатывает вывод, подключая буферизованный поток к программе и целевому устройству или файлу. Заголовочные файлы `iostream` и `fstream` предоставляют доступ к библиотекам классов ввода-вывода, включающим богатый набор классов управления потоками. Программы C++, которые включают файл `iostream`, автоматически открывают восемь потоков, управляя ими через восемь объектов. Объект `cin` управляет стандартным потоком ввода, который по умолчанию подключен к стандартному устройству ввода — как правило, клавиатуре. Объект `cout` управляет стандартным выходным потоком, который по умолчанию подключен к стандартному устройству вывода — обычно, монитору. Объекты `cerr` и `clog` управляют, соответственно, небуферизованным и буферизованным потоками, подключенными к стандартному устройству ошибок — обычно, монитору. Эти четыре объекта имеют четыре дополнения, работающие с широкими символами, а именно: `wcin`, `wcout`, `wcerr` и `wclog`.

Библиотека классов ввода-вывода представляет широкий набор удобных методов. Класс `istream` определяет версии операций извлечения (>>), которые распознают все базовые типы C++ и преобразуют символьный ввод в эти типы. Семейство методов `get()` и `getline()` представляют дальнейшую поддержку односимвольного и строкового ввода. Аналогично, класс `ostream` определяет версии операций вставки (<<), которые распознают все базовые типы C++ и преобразуют их в соответствующий символьный вывод. Метод `put()` предлагает дальнейшую поддержку односимвольного вывода. Классы `wistream` и `wostream` предоставляют аналогичную поддержку "широких" символов.

Вы можете управлять тем, как программа форматирует вывод, используя методы класса `ios_base` и применяя манипуляторы (функции, которые могут быть сцеплены с операциями вставки), определенные в файлах `iostream` и `iomanip`. Эти методы и манипуляторы позволяют управлять основанием чисел, шириной поля, количеством отображаемых десятичных разрядов при выводе значений с плавающей точкой, а также другими элементами.

Файл `fstream` представляет определения классов, которые расширяют методы `iostream` для файлового ввода-вывода. Класс `ifstream` унаследован от класса `istream`. Ассоциируя объект `ifstream` с файлом, вы можете использовать методы

iostream для чтения файла. Аналогично, ассоциация объекта ofstream с файлом позволяет применять методы ostream для записи в файл. А вот ассоциация объекта fstream с файлом позволяет применять с файлом и методы ввода, и методы вывода.

Чтобы ассоциировать файл с потоком, вы можете указать имя файла при инициализации объекта файлового потока, либо сначала создать объект файлового потока, а затем применить метод open() для ассоциации его с файлом. Метод close() разрывает соединение между потоком и файлом. Конструкторы классов и метод open() принимают не обязательный второй параметр, указывающий режим файла. Режим файла определяет такие вещи, как возможность чтения и/или записи файла, должно ли происходить усечение файла при открытии его для записи, будет ли считаться ошибкой попытка открыть несуществующий файл, а также использовать текстовый или бинарный формат.

Текстовый файл хранит всю информацию в символьной форме. Например, числовые значения преобразуются в символьное представление. Обычные операции вставки и извлечения, наряду с методами get() и getline(), поддерживают этот формат. Бинарный файл сохраняет всю информацию, используя внутреннее ее представление в компьютере. Бинарные файлы хранят данные — в частности, значения с плавающей точкой — более точно и компактно, чем текстовые файлы, но при этом они менее переносимы. Методы read() и write() поддерживают бинарный ввод и вывод.

Функции seekg() и seekp() обеспечивают произвольный доступ C++ к файлам. Эти методы класса позволяют позиционировать указатель файла относительно начала файла, его конца или текущей позиции. Методы tellg() и tellp() сообщают текущую позицию указателя в файле.

Заголовочный файл sstream определяет классы istringstream и istringstream, позволяющие использовать методы istream и ostream для извлечения информации из строки и форматирования информации, помещаемой в строку.

Вопросы для самоконтроля

1. Какую роль играет файл iostream во вводе-выводе C++?
2. Почему ввод числа 121 с клавиатуры требует от программы выполнения преобразования?
3. Какая разница между потоком стандартного вывода и стандартным потоком ошибок?
4. Почему cout может отображать различные типы C++ без необходимости явного указания каждого типа?
5. Какое свойство определений методов вывода позволяет выполнять конкатенацию вывода?
6. Напишите программу, которая запрашивает целое число и затем отображает его в десятичной, восьмеричной и шестнадцатеричной формах. Отобразите все формы в одной и той же строке, в полях шириной 15 символов, с применением C++ префиксов оснований числа.
7. Напишите программу, которая запрашивает следующую информацию и форматирует ее, как показано ниже:

```
Введите ваше имя: Billy Gruff
Ваш почасовой заработок: 12
Количество отработанных часов: 7.5
Первыйформат:
              Billy Gruff: $     12.00:  7.5
Второй формат:
Billy Gruff             : $12.00     :7.5
```

8. Имеется следующая программа:

```
//rq17-8.cpp
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   int ct1 = 0;
   cin >> ch;
   while (ch != 'q')
   {
      ct1++;
      cin >> ch;
   }
   int ct2 = 0;
   cin.get(ch);
   while (ch != 'q')
   {
      ct2++;
      cin.get(ch);
   }
   cout << "ct1 = " << ct1 << "; ct2 = " << ct2 << "\n";
   return 0;
}
```

Что она напечатает, если получит такой ввод:

```
I see a q<Enter>
I see a q<Enter>
```

Здесь <Enter> означает нажатие одноименной клавиши.

9. Оба следующих оператора читают и отбрасывают символы, начиная с конца строки и включая его. В чем отличие их поведения?

```
while (cin.get() != '\n')
   continue;
cin.ignore(80, '\n');
```

Упражнения по программированию

1. Напишите программу, которая подсчитывает количество символов вплоть до первого $ в строке, оставляя $ во входном потоке.
2. Напишите программу, которая копирует ваш клавиатурный ввод (вплоть до эмулируемого конца файла) в файл, чье имя передано в командной строке.
3. Напишите программу, копирующую один файл в другой. Имена файлов программа должна получать из командной строки. Если не удается открыть файл, должно выдаваться соответствующее сообщение.

4. Напишите программу, которая открывает два текстовых файла для ввода и один — для вывода. Программа должна соединять соответствующие строки входных файлов, используя в качестве разделителя пробел, и писать результаты в выходной файл. Например, предположим, что первый входной файл имеет следующее содержимое:

```
eggs kites donuts
balloons hammers
stones
```

   А второй файл — следующее:

```
zero lassitude
finance drama
```

   Результирующий файл должен выглядет так:

```
eggs kites donuts zero lassitude
balloons hammers finance drama
stones
```

5. Мэт и Пэт хотят пригласить своих друзей на вечеринку — почти так же, как они это делали в упражнении 8 из главы 16, за исключением того, что теперь им нужна программа, использующая файлы. Они просят вас написать программу, которая делает следующее:

   - Читает список друзей Мэта из текстового файла по имени `mat.dat`, который перечисляет по одному другу в строке. Имена сохраняются в контейнере и затем отображаются в отсортированном виде.
   - Читает список друзей Пэт из текстового файла по имени `pat.dat`, который перечисляет по одному другу в строке. Имена сохраняются в контейнере и затем отображаются в отсортированном виде.
   - Объединяет эти два списка, исключая дубликаты, и сохраняет результат в файле `matnpat.dat`, по одному другу в строке.

6. Рассмотрите определение класса, предложенное в упражнении 5 главы 14. Если вы еще не выполняли это упражнение, сделайте это сейчас. Затем сделайте следующее:

   Напишите программу, которая использует стандартный ввод-вывод C++ и файловый ввод-вывод в связи с данными типов `employee`, `manager`, `fink` и `highfink`, как определено в упражнении 5 главы 14. Программа должна быть в соответствии с главными строками листинга 17.17 в том, что должна позволять вносить новые данные в файл. При первом запуске программа должна запросить данные у пользователя, показать все введенные значения и сохранить информацию в файл. При последующих запусках она должна сначала прочитать и отобразить данные файла, дать возможность пользователю добавить новые данные и показать все данные снова. Единственное отличие должно быть в том, что данные должны быть представлены в виде массива указателей на тип `employee`. Таким образом, указатель может указывать на объект `employee` либо на объект любого типа из трех его наследников. Сохраняйте массив маленьким, чтобы облегчить его проверку программой; например, вы можете ограничить его размером в 10 элементов:

```
const int MAX = 10; // не более 10 объектов
...
employee * pc[MAX];
```

Для клавиатурного ввода программа должна использовать меню для того, чтобы предоставить пользователю выбор, какого типа объект нужно создать. Меню должно применять switch для использования операции new для создания объекта требуемого типа и присваивать его адрес указателю в массиве pc. Затем этот объект может вызвать виртуальную функцию setall() для запроса соответствующих данных от пользователя:

```
pc[i]->setall(); // вызывается функция, соответствующая типу объекта
```

Чтобы сохранить данные в файл, разработайте виртуальную функцию writeall():

```
for (i = 0; i < index; i++)
pc[i]->writeall(fout);// fout ofstream подключен к выходному файлу
```

На заметку!

В упражнении 6 используйте текстовый, а не бинарный ввод-вывод. (К сожалению, виртуальные объекты содержат указатели на таблицы или указатели на виртуальные функции, и write() копирует эту информацию в файл. Объект, наполняемый методом read() из файла, получает некорректные указатели на функции, что приводит к путанице в поведении виртуальных функций.) Используйте символы новой строки для отделения каждого поля данных от следующего — это облегчит идентификацию полей при вводе. Либо вы можете применить бинарный ввод-вывод, но не записывать объекты как единое целое. Вместо этого вы можете предусмотреть методы, которые применяют функции read() и write() для чтения и записи каждого поля каждого из классов индивидуально, а не в составе целого объекта. Таким образом, программа сможет записывать в файл только необходимые данные.

Сложность представляет восстановление данных из файла. Проблема состоит в том, как программа узнает, какого типа объект будет восстановлен следующим: employee, manager, fink либо highfink? Один из возможных подходов к решению этой проблемы заключается в следующем: при записи данных объекта в файл предварить его целым числом, идентифицирующим тип объекта, который последует. Затем, при вводе из файла, программа может читать это целое число и затем использовать switch для создания объекта соответствующего типа для приема данных:

```
enum classkind{Employee, Manager, Fink, Highfink}; // в заголовке класса
...
int classtype;
while((fin >> classtype).get(ch)){//перевод строки отделять целое от данных
    switch(classtype) {
        case Employee : pc[i] = new employee;
        : break;
```

Затем вы можете использовать указатель для вызова виртуальной функции getall() с целью чтения информации:

```
pc[i++]->getall();
```

7. Ниже представлена часть программы, которая читает клавиатурный ввод в вектор объектов string, сохраняет строковое содержимое (не объекты!) в файле, затем копирует содержимое файла обратно в вектор объектов string:

```
int main()
{
   using namespace std;
   vector<string> vostr;
   string temp;
   // получить строки
   cout << "Введите строки (или пустую строку для завершения):\n";
   while (getline(cin,temp) && temp[0] != '\0')
      vostr.push_back(temp);
   cout << "Ваш ввод.\n";
   for_each(vostr.begin(), vostr.end(), ShowStr);
   // сохранить в файле
   ofstream fout("strings.dat", ios_base::out | ios_base::binary);
   for_each(vostr.begin(), vostr.end(), Store(fout));
   fout.close();
   // восстановить содержимое файла
   vector<string> vistr;
   ifstream fin("strings.dat", ios_base::in | ios_base::binary);
   if (!fin.is_open())
   {
      cerr << "Не удалось открыть файл для ввода.\n";
      exit(EXIT_FAILURE);
   }
   GetStrs(fin, vistr);
   cout << "\nСтроки, прочитанные из файла:\n";
   for_each(vistr.begin(), vistr.end(), ShowStr);
   return 0;
}
```

Обратите внимание, что файл открывается в бинарном формате и замысел в том, чтобы ввод-вывод осуществлялся методами `read()` и `write()`. Остается сделать немного:

- Написать функцию `void ShowStr(const string &)`, которая отображает объект `string` с последующим символом перевода строки.
- Написать функтор `Store`, который пишет строку информации в файл. Конструктор `Store` должен специфицировать объект `ifstream`, а перегруженная функция `operator()(const string &)` должна указывать строку, подлежащую записи. План работ должен включать первоначальную запись размера строки, а затем — ее содержимого. Например, если `len` содержит размер строки, то вы должны использовать вот что:

  ```
  os.write((char *)&len, sizeof(std::size_t));  // сохранить длину
  os.write(s.data(), len);                      // сохранить символы
  ```

  Член `data()` возвращает указатель на массив, который содержит символы строки. Он подобен `c_str()`, за исключением того, что последний добавляет нулевой символ.
- Напишите функцию `GetStrs()`, которая восстанавливает информацию из файла. Она может использовать `read()` для получения размера строки и затем применять цикл для чтения такого количества символов из файла, добавляя их в изначально пустую временную строку. Поскольку данные объекта `string` — приватные, вы должны использовать метод класса для получения данных строки вместо того, чтобы читать их непосредственно.

ПРИЛОЖЕНИЕ А

Основания систем счисления

История сохранила свидетельства того, что в древних цивилизациях использовались многие системы представления чисел. Некоторые из них, как, например, система римских цифр, непригодны для использования в арифметических задачах. С другой стороны, система представления чисел, которую придумали древнеиндийские математики, претерпев некоторые изменения, была принята европейскими странами и известна как арабская система представления чисел; она используется для производства вычислений в самых разных сферах деятельности человека. Современные компьютерные системы представления чисел построены на концепции заполнителя и используют нуль, преимущества которого для записи чисел стали ясны еще древнеиндийским математикам. Однако они обобщают принципы представления чисел, используемые в других системах. Поэтому, несмотря на то, что для представления чисел мы обычно пользуемся десятичной системой, о чем будет сказано в следующем разделе, в вычислительной технике часто применяются числа, имеющие основание 8 (восьмеричная система), 16 (шестнадцатеричная система) и 2 (двоичная система).

Десятичные числа (основание 10)

Способ, который мы используем для записи чисел, основан на степени основания 10. Например, рассмотрим число 2468. 2 соответствует двум тысячам, 4 — четырем сотням, 6 — шести десяткам и 8 — восьми единицам:

2468 = 2×1000 + 4×100 + 6×10 + 8×1

Одну тысячу можно записать как 10×10×10, или 10 в третьей степени — 10^3. Используя такое обозначение, предыдущее соотношение можно записать следующим образом:

2468 = 2×103 + 4×102 + 6×101 + 8×100

Поскольку эта форма представления чисел основана на степени основания 10, мы называем ее представлением с основанием 10, или десятичным представлением. За основу можно выбрать любое другое число. Так, для записи целых чисел в языке C++ можно использовать восьмеричную (основание 8) и шестнадцатеричную (основание 16) форму. (Примечание: 10^0 равно 1, как и любое другое ненулевое число в нулевой степени.)

Восьмеричные целые числа (основание 8)

Восьмеричные числа основаны на степени с основанием 8, поэтому в восьмеричной системе для записи чисел используются цифры 0–7. Для обозначения восьмеричной системы в языке C++ служит префикс 0. Поэтому 0177 — восьмеричное значение.

Степень основания 8 можно использовать, чтобы получить эквивалентное значение в десятичной системе:

Восьмеричная система	Десятичная система
0177	$= 1\times8^2 + 7\times8^1 + 7\times8^0$
	$= 1\times64 + 7\times8 + 7\times1$
	= 127

Поскольку в Unix для представления значений часто используется именно восьмеричная система, эта форма записи предусмотрена и в языках C и C++.

Шестнадцатеричные числа (основание 16)

Шестнадцатеричные числа основаны на степени с основанием 16. Это означает, что 10 в шестнадцатеричной системе представляет значение 16 + 0, или 16. Чтобы представить значения от 9 до шестнадцатеричного 16, нужны дополнительные знаки. Для этого в стандартной записи шестнадцатеричной системы используются буквы от a до f. Как видно в табл. А.1, в языке C++ эти буквы могут быть записаны как в верхнем, так и в нижнем регистре.

Таблица А.1. Шестнадцатеричные знаки

Шестнадцатеричный знак	Десятичное значение
a или A	10
b или B	11
c или C	12
d или D	13
e или E	14
f или F	15

Для идентификации шестнадцатеричной записи в языке C++ применяется запись 0x или 0X. Поэтому 0x2B3 — шестнадцатеричное значение. Чтобы получить десятичный эквивалент числа 0x2B3, можно воспользоваться степенью основания 16:

Шестнадцатеричное	Десятичное
0×2B3	$= 2\times16^2 + 11\times16^1 + 3\times16^0$
	$= 2\times256 + 11\times16 + 3\times1$
	= 691

Шестнадцатеричная форма записи часто используется в документации по оборудованию для обозначения адресов памяти и номеров портов.

Двоичные числа (основание 2)

Независимо от того, используете ли вы десятичную, восьмеричную или шестнадцатеричную форму для записи целого числа, в памяти компьютера оно будет храниться в двоичной форме — в виде значения с основанием 2. В двоичной записи используются всего две цифры: 0 и 1.

Например, 10011011 – двоичное число. Учтите, что C++ не предусматривает возможности записи чисел в двоичной форме. Двоичные числа основаны на степени 2:

Двоичная запись	Десятичная запись
100110111	$= 1\times2^7 + 0\times2^6 + 0\times2^5 + 1\times2^4$
	$= 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
	= 128 + 0 + 0 + 16 + 8 + 0 + 2 + 1
	= 155

Двоичная форма записи очень удобна для памяти компьютера, в которой каждый индивидуальный элемент, называемый *битом*, может быть включен или выключен. Состояние "выключен" обозначается с помощью нуля, а состояние "включен" – с помощью единицы. Обычно память компьютера организована в виде элементов, называемых *байтами*, причем каждый байт равен 8 битам. Нумерация битов в байте соответствует связанной степени с основанием 2. Поэтому самый правый бит имеет номер 0, следующий бит – 1 и так далее. Например, на рис. А.1 показано 2-байтное целое число.

Рис. А.1. Двухбайтное целочисленное значение

Двоичная и шестнадцатеричная формы записи

Шестнадцатеричная форма записи часто применяется для упрощения представления двоичных данных, например, адресов памяти или целых чисел, содержащих установки битовых флагов. Дело в том, что каждый шестнадцатеричный знак соответствует 4-битному элементу. Эти соответствия представлены в табл. А.2.

Чтобы получить из шестнадцатеричного значения двоичное, достаточно заменить каждый шестнадцатеричный знак соответствующим двоичным эквивалентом. Например, шестнадцатеричное число 0xA4 соответствует двоичному 1010 0100. Подобным образом можно выполнить обратное преобразование из двоичной формы в шестнадцатеричную, преобразуя каждый 4-битный элемент в эквивалентный шестнадцатеричный знак. Например, шестнадцатеричным эквивалентом двоичного значения 1001 0101 является 0x95.

Таблица А.2. Шестнадцатеричные знаки и их двоичные эквиваленты

Шестнадцатеричные знаки	Двоичный эквивалент
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Пример из практики: прямой и обратный порядок следования байтов

Как это ни странно, две вычислительные платформы, в которых используется двоичное представление целых чисел, могут представлять по-разному одно и то же число. Например, компьютеры на базе процессоров Intel хранят байты с использованием архитектуры Little Endian (расположение байтов в обратном порядке), тогда как в компьютерах на базе процессоров Motorola, компьютерах MIPS на базе процессоров RISC и компьютерах DEC на базе процессоров Alpha реализована схема Big Endian (расположение байтов в прямом порядке). (Однако в последних двух системах могут быть реализованы обе схемы.)

Термины *Big Endian* и *Little Endian* можно расшифровать как "Big End In" и "Little End In". Они определяют порядок расположения байтов в машинном слове (которое обычно соответствует 2-байтному элементу) памяти. В компьютере на базе процессора Intel (Little Endian) первым сохраняется младший байт. Это означает, что шестнадцатеричное значение, например `0xABCD`, будет храниться в памяти как `0xCD 0xAB`. В компьютерах на базе процессоров Motorola (Big Endian) это же значение будет храниться в обратной последовательности, то есть `0xAB 0xCD`.

Впервые эти термины были упомянуты в книге *Путешествия Гулливера* Джонатана Свифта. Свифт высмеял абсурдность многих политических диспутов на примере двух враждующих политических групп лилипутов: "тупоконечников" (Big Endians), которые утверждали, что яйцо нужно разбивать с тупого конца, и "остроконечников" (Little Endians), которые, наоборот, утверждали, что яйцо нужно разбивать с острого конца.

Вы, как специалисты по программному обеспечению, должны хорошо понимать используемый порядок слов в искомой платформе. Среди всего прочего он влияет на интерпретацию данных, передаваемых по сети, а также на способ хранения данных в двоичных файлах. В предыдущем примере 2-байтный шаблон памяти `0xABCD` мог представлять десятичное значение 52651 в компьютере с поддержкой схемы Little Endian, и 43981 в компьютере с поддержкой схемы Big Endian.

ПРИЛОЖЕНИЕ Б

Зарезервированные слова языка C++

В языке C++ имеются специальные слова, зарезервированные для использования как самим языком программирования, так и его библиотеками. Зарезервированные слова нельзя применять в объявлениях в качестве идентификаторов. Зарезервированные слова делятся на три категории: служебные слова, альтернативные лексемы и зарезервированные имена библиотек C++.

Служебные слова C++

Служебными (или *ключевыми*) *словами* называются идентификаторы, формирующие словарь языка программирования. Они не могут использоваться в других целях, например, для именования переменных. В табл. Б.1 представлен перечень служебных слов языка C++. Служебные слова, выделенные полужирным, являются служебными словами в ANSI C99.

Таблица Б.1. Служебные слова C++

asm	**auto**	bool	**break**	**case**
catch	**char**	class	**const**	const_cast
continue	**default**	delete	**do**	**double**
dynamic_cast	**else**	**enum**	explicit	export
extern	false	**float**	**for**	friend
goto	**if**	**inline**	**int**	**long**
mutable	namespace	new	operator	private
protected	public	**register**	reinterpret_cast	**return**
short	**signed**	**sizeof**	**static**	static_cast
struct	**switch**	template	this	throw
true	try	**typedef**	typeid	typename
union	**unsigned**	using	virtual	**void**
volatile	wchar_t	**while**		

Альтернативные лексемы

Кроме служебных слов в языке C++ имеются алфавитные альтернативные варианты представления операций, называемые *альтернативными лексемами*. Они также являются зарезервированными. В табл. Б.2 приведен перечень алфавитных альтернативных лексем и соответствующих операций.

Таблица Б.2. Зарезервированные альтернативные лексемы C++ и их назначение

Лексема	Назначение
`and`	`&&`
`and_eq`	`&=`
`bitand`	`&`
`bitor`	`|`
`compl`	`~`
`not`	`!`
`not_e`	`!=`
`or`	`||`
`or_eq`	`|=`
`xor`	`^`
`xor_eq`	`^=`

Зарезервированные имена библиотек C++

Компилятор не разрешает использовать в качестве имен служебные слова и альтернативные лексемы. Существует еще один класс запрещенных имен, к использованию которых компилятор относится менее строго — *зарезервированные имена*. Они представляют собой имена, зарезервированные для использования библиотекой C++. Если одно из этих имен выбрать в качестве идентификатора, то предсказать результат будет невозможно. Другими словами, это может привести к возникновению ошибки компиляции, выдаче предупреждающего сообщения, неправильному ходу выполнения программы, а может быть и так, что выполнение программы не будет сопровождаться ошибками.

Язык программирования C++ резервирует имена макрокоманд, используемых в библиотечных заголовочных файлах. Если программа включает определенный заголовочный файл, то применять имена макрокоманд, определенные в этом заголовке (или в заголовках, включенных в этот заголовочный файл и так далее), для других целей нельзя. Например, если вы явно или неявно включите заголовочный файл `<climits>`, то использование `CHAR_BIT` в качестве идентификатора будет запрещено, поскольку это имя уже используется в этом заголовочном файле на макроуровне.

Язык C++ резервирует имена, которые начинаются с двойного подчеркивания или одного подчеркивания и следующей за ним буквой в верхнем регистре для любого использования, и имена, которые начинаются с одного подчеркивания для использования в качестве глобальной переменной.

Следовательно, нельзя создавать имена, такие как __gink или __Lynx, в любом случае, а имена, такие как _lynx, — в глобальном пространстве имен.

Язык программирования C++ резервирует имена, объявленные в библиотечных заголовочных файлах и имеющие внешнее связывание. Применительно к функциям, это касается сигнатуры (имени и списка параметров). Например, предположим, что имеется следующий код:

```
#include <cmath>
using namespace std;
```

В данном случае сигнатура функции tan(double) является зарезервированной. Это означает, что в вашей программе не может быть объявлена функция, которая имеет данный прототип:

```
int tan(double); // так делать нельзя
```

Эта запись совпадает не с прототипом библиотечной функции tan(), которая возвращает тип double, а с частью сигнатуры. А вот следующий прототип использовать можно:

```
char * tan(char *); // а так можно
```

В этой записи хоть и есть совпадение с идентификатором tan(), зато нет совпадения с сигнатурой.

ПРИЛОЖЕНИЕ В

Набор символов ASCII

Для хранения символов в компьютерах используются числовые коды. Наиболее распространенным кодом в США является код ASCII (American Standard Code for Information Interchange — Американский стандартный код для обмена информацией). Он является подмножеством (причем, очень малым подмножеством) кода Unicode. В языке C++ большинство символов можно представлять явным образом, заключая их в одинарные кавычки, например `'A'` для символа `A`. Кроме того, отдельный символ можно представлять посредством восьмеричного или шестнадцатеричного кода, ставя перед кодом обратную косую черту; например, коды `'\012'` и `'\0xa'` представляют один и тот же символ — перехода на новую строку (LF). Такие последовательности символов, называемые управляющими последовательностями, могут быть частью строки, как в `"Hello,\012my dear"`.

В табл. В.1 показан набор символов ASCII и соответствующие их представления в различных системах счисления. В этой таблице символ ^, используемый в качестве префикса, обозначает клавишу <Ctrl>.

Таблица В.1 Набор символов ASCII

Десятичная	Восьмеричная	Шестнадцатеричная	Двоичная	Символ	Имя ASCII
0	0	0	00000000	^@	NUL
1	01	0x1	00000001	^A	SOH
2	02	0x2	00000010	^B	STX
3	03	0x3	00000011	^C	ETX
4	04	0x4	00000100	^D	EOT
5	05	0x5	00000101	^E	ENQ
6	06	0x6	00000110	^F	ACK
7	07	0x7	00000111	^G	BEL
8	010	0x8	00001000	^H	BS
9	011	0x9	00001001	^I, Tab	HT
10	012	0xa	00001010	^J	LF
11	013	0xb	00001011	^K	VT
12	014	0xc	00001100	^L	FF
13	015	0xd	00001101	^M	CR
14	016	0xe	00001110	^N	SO
15	017	0xf	00001111	^O	SI
16	020	0x10	00010000	^P	DLE

Десятичная	Восьмеричная	Шестнадцатеричная	Двоичная	Символ	Имя ASCII
17	021	0x11	00010001	^Q	DC1
18	022	0x12	00010010	^R	DC2
19	023	0x13	00010011	^S	DC3
20	024	0x14	00010100	^T	DC4
21	025	0x15	00010101	^U	NAK
22	026	0x16	00010110	^V	SYN
23	027	0x17	00010111	^W	ETB
24	030	0x18	00011000	^X	CAN
25	031	0x19	00011001	^Y	EM
26	032	0x1a	00011010	^Z	SUB
27	033	0x1b	00011011	^[, Esc	ESC
28	034	0x1c	00011100	^\	FS
29	035	0x1d	00011101	^]	GS
30	036	0x1e	00011110	^^	RS
31	037	0x1f	00011111	^_	US
32	040	0x20	00100000	Пробел	SP
33	041	0x21	00100001	!	
34	042	0x22	00100010	“	
35	043	0x23	00100011	#	
36	044	0x24	00100100	$	
37	045	0x25	00100101	%	
38	046	0x26	00100110	&	
39	047	0x27	00100111	‘	
40	050	0x28	00101000	(	
41	051	0x29	00101001	)	
42	052	0x2a	00101010	*	
43	053	0x2b	00101011	+	
44	054	0x2c	00101100	‘	
45	055	0x2d	00101101	-	
46	056	0x2e	00101110	.	
47	057	0x2f	00101111	/	
48	060	0x30	00110000	0	
49	061	0x31	00110001	1	
50	062	0x32	00110010	2	
51	063	0x33	00110011	3	
52	064	0x34	00110100	4	
53	065	0x35	00110101	5	
54	066	0x36	00110110	6	

Десятичная	Восьмеричная	Шестнадцатеричная	Двоичная	Символ	Имя ASCII
55	067	0x37	00110111	7	
56	070	0x38	00111000	8	
57	071	0x39	00111001	9	
58	072	0x3a	00111010	:	
59	073	0x3b	00111011	;	
60	074	0x3c	00111100	<	
61	075	0x3d	00111101	=	
62	076	0x3e	00111110	>	
63	077	0x3f	00111111	?	
64	0100	0x40	01000000	@	
65	0101	0x41	01000001	A	
66	0102	0x42	01000010	B	
67	0103	0x43	01000011	C	
68	0104	0x44	01000100	D	
69	0105	0x45	01000101	E	
70	0106	0x46	01000110	F	
71	0107	0x47	01000111	G	
72	0110	0x48	01001000	H	
73	0111	0x49	01001001	I	
74	0112	0x4a	01001010	J	
75	0113	0x4b	01001011	K	
76	0114	0x4c	01001100	L	
77	0115	0x4d	01001101	M	
78	0116	0x4e	01001110	N	
79	0117	0x4f	01001111	O	
80	0120	0x50	01010000	P	
81	0121	0x51	01010001	Q	
82	0122	0x52	01010010	R	
83	0123	0x53	01010011	S	
84	0124	0x54	01010100	T	
85	0125	0x55	01010101	U	
86	0126	0x56	01010110	V	
87	0127	0x57	01010111	W	
88	0130	0x58	01011000	X	
89	0131	0x59	01011001	Y	
90	0132	0x5a	01011010	Z	
91	0133	0x5b	01011011	[	
92	0134	0x5c	01011100	\	

Десятичная	Восьмеричная	Шестнадцатеричная	Двоичная	Символ	Имя ASCII
93	0135	0x5d	01011101	]	
94	0136	0x5e	01011110	^	
95	0137	0x5f	01011111	_	
96	0140	0x60	01100000	‘	
97	0141	0x61	01100001	a	
98	0142	0x62	01100010	b	
99	0143	0x63	01100011	c	
100	0144	0x64	01100100	d	
101	0145	0x65	01100101	e	
102	0146	0x66	01100110	f	
103	0147	0x67	01100111	g	
104	0150	0x68	01101000	h	
105	0151	0x69	01101001	i	
106	0152	0x6a	01101010	j	
107	0153	0x6b	01101011	k	
108	0154	0x6c	01101100	l	
109	0155	0x6d	01101101	m	
110	0156	0x6e	01101110	n	
111	0157	0x6f	01101111	o	
112	0160	0x70	01110000	p	
113	0161	0x71	01110001	q	
114	0162	0x72	01110010	r	
115	0163	0x73	01110011	s	
116	0164	0x74	01110100	t	
117	0165	0x75	01110101	u	
118	0166	0x76	01110110	v	
119	0167	0x77	01110111	w	
120	0170	0x78	01111000	x	
121	0171	0x79	01111001	y	
122	0172	0x7a	01111010	z	
123	0173	0x7b	01111011	{	
124	0174	0x7c	01111100	|	
125	0175	0x7d	01111101	}	
126	0176	0x7e	01111110	~	
127	0177	0x7f	01111111	Del	

ПРИЛОЖЕНИЕ Г

Приоритеты операций

Приоритеты операций определяет порядок, в соответствии с которым они могут быть выполнены над значением. Операции в языке C++ разделены на 18 групп; все они представлены в табл. Г.1. Операции, образующие группу 1, имеют наивысший уровень приоритета выполнения; операции, представляющие группу 2, занимают следующий уровень приоритета и так далее. Если две операции применены к одному и тому же операнду (некоторое значение, над которым выполняется операция), то первой будет выполнена операция, имеющая более высокий уровень приоритета. Если две операции имеют одинаковый уровень приоритета, то для определения первоочередности выполнения C++ руководствуется правилами ассоциативности. Все операции в одной группе имеют одинаковый уровень приоритета и одинаковую ассоциативность, которая может определяться слева направо (в таблице — ассоциативность слева направо) или справа налево (в таблице — ассоциативность справа налево). Ассоциативность слева направо означает первоочередное выполнение левой операции, а ассоциативность справа налево означает первоочередное выполнение правой операции.

Некоторые символы, например `*` и `&`, могут использоваться для выполнения операций над несколькими операндами. В таких случаях одна форма будет *унарной* (один операнд), а другая — *бинарной* (два операнда). Чтобы определить, какая форма имеется в виду, компилятор обращается к контексту. В табл. Г.1 отмечены группы унарных и бинарных операций, когда в каждом из случаев применяется один и тот же символ.

Далее представлено несколько примеров приоритета и ассоциативности.

В следующем примере компилятору предстоит решить, какую операцию необходимо выполнить первой: прибавить `5` к `3` или умножить `5` на `6`:

```
3 + 5 * 6
```

Операция умножения (`*`) имеет более высокий приоритет, чем операция сложения (`+`), поэтому над операндом `5` сначала выполняется операция умножения, в результате чего выражение принимает вид `3 + 30`, или `33`.

В следующем примере компилятор должен решить, что необходимо выполнить в первую очередь: разделить `120` на `6` или `6` умножить на `5`:

```
120 / 6 * 5
```

Обе операции — умножение и деление — имеют одинаковый уровень приоритета, а ассоциативность в данном случае определяется слева направо. Следовательно, над операндом `6` в первую очередь будет выполнена операция слева, поэтому выражение примет вид `20 * 5`, или `100`.

В следующем примере компилятор должен решить, что нужно сделать: увеличить или уменьшить `str`:

```
char * str = "Whoa";
char ch = *str++;
```

Постфиксная операция ++ имеет более высокий уровень приоритета, чем унарная операция *. Таким образом, операция приращения применяется к str, а не к *str. Другими словами, после выполнения операции будет изменен не символ, на который указывает указатель, а указатель, в результате чего он будет переведен на следующий символ. Однако поскольку операция ++ является постфиксной, то инкрементирование указателя будет осуществлено после того, как переменной ch будет присвоено исходное значение *str. Таким образом, переменной ch присваивается символ W, а затем в str указатель переводится на символ h.

Далее показан похожий пример:

```
char * str = "Whoa";
char ch = *++str;
```

Префиксная операция ++ и унарная операция * имеют одинаковый уровень приоритета, а ассоциативность определяется справа налево. Таким образом, и в этом случае происходит приращение str, но не *srt. Поскольку операция ++ имеет префиксную форму, то сначала происходит инкремент str, а затем разыменовывается указатель. Поэтому str переводит указатель на символ h, после чего этот символ присваивается переменной ch.

Обратите внимание, что в табл. Г.1 при описании приоритета две операции, обозначаемые одним символом, делятся на *бинарные* и *унарные*, например, унарная адресная операция и бинарная операция AND.

Таблица Г.1. Приоритеты операций и ассоциативность

Операция	Ассоциативность	Назначение
		Первая группа приоритетов
::		Операция разрешения контекста
		Вторая группа приоритетов
(выражение)		Группирование
()	Слева направо	Вызов функции
()		Конструкция значения – то есть, *тип* (*выраж*)
[]		Список индексов массива
->		Косвенная принадлежность
		Прямая принадлежность
const_cast		Специализированное приведение типа
dynamic_cast		Специализированное приведение типа
reinterpret_cast		Специализированное приведение типа
static_cast		Специализированное приведение типа
typeid		Идентификация типа
++		Операция приращения (инкремент), постфиксная
--		Операция отрицательного приращения (декремент), постфиксная

Операция	Ассоциативность	Назначение
	Третья группа приоритетов (все унарные)	
`!`	Справа налево	Логическое отрицание
`~`		Битовое отрицание
`+`		Унарное сложение (знак плюс)
`-`		Унарное отрицание (знак минус)
`++`		Операция приращения (инкремента), префиксная
`--`		Операция отрицательного приращения (декремента), префиксная
`&`		Адрес
`*`		Разыменование (косвенное значение)
`()`		Приведение типа — то есть, (*тип*) *выраж*
`sizeof`		Размер в байтах
`new`		Динамическое выделение памяти
`new []`		Динамическое выделение массива
`delete`		Динамическое освобождение памяти
`delete []`		Динамическое освобождение массива
	Четвертая группа приоритетов	
`.*`	Слева направо	Разыменование члена
`->*`		Косвенное разыменование члена
	Пятая группа приоритетов (все бинарные)	
`*`	Слева направо	Умножение
`/`		Деление
`%`		Нахождение остатка целочисленного деления
	Шестая группа приоритетов (все бинарные)	
`+`	Слева направо	Сложение
`-`		Вычитание
	Седьмая группа приоритетов	
`<<`	Слева направо	Сдвиг влево
`>>`		Сдвиг вправо
	Восьмая группа приоритетов	
`<`	Слева направо	Меньше
`<=`		Меньше или равно
`>=`		Больше или равно
`>`		Больше

Операция	Ассоциативность	Назначение
	Девятая группа приоритетов	
`==`	Слева направо	Равно
`!=`		Не равно
	Десятая группа приоритетов (бинарные)	
`&`	Слева направо	Битовое И (AND)
	Одиннадцатая группа приоритетов	
`^`	Слева направо	Битовое исключающее ИЛИ (XOR)
	Двенадцатая группа приоритетов	
`|`	Слева направо	Битовое ИЛИ (OR)
	Тринадцатая группа приоритетов	
`&&`	Слева направо	Логическое И (AND)
	Четырнадцатая группа приоритетов	
`||`	Слева направо	Логическое ИЛИ (OR)
	Пятнадцатая группа приоритетов	
`:?`	Справа налево	Условная операция
	Шестнадцатая группа приоритетов	
`=`	Справа налево	Простое присваивание
`*=`		Умножение и присваивание
`/=`		Деление и присваивание
`%=`		Нахождение остатка и присваивание
`+=`		Сложение и присваивание
`-=`		Вычитание и присваивание
`&=`		Битовое И (AND) и присваивание
`^=`		Битовое исключающее ИЛИ (XOR) и присваивание
`|=`		Битовое ИЛИ (OR) и присваивание
`<<=`		Сдвиг влево и присваивание
`>>=`		Сдвиг вправо и присваивание
	Семнадцатая группа приоритетов	
`throw`	Слева направо	Генерация исключения
	Восемнадцатая группа приоритетов	
`,`	Слева направо	Комбинирование двух выражений в одном

ПРИЛОЖЕНИЕ Д

Другие операции

В целях экономии места, в основном тексте этой книги не были рассмотрены две группы операций. К первой группе относятся битовые операции, с помощью которых можно манипулировать индивидуальными битами в значении; эти операции перекочевали из языка С. Ко второй группе относятся двучленные операции разыменования; они были введены уже в C++. В этом приложении будут вкратце рассмотрены обе группы операций.

Битовые операции

Битовые операции выполняются над битами целочисленных значений. Например, операция сдвига влево смещает биты влево, а операция битового отрицания преобразовывает каждую единицу в нуль, и наоборот. В языке C++ всего насчитывается шесть битовых операций: <<, >>, ~, &, | и ^.

Операции сдвига

Операция сдвига влево имеет следующий синтаксис:

значение << *сдвиг*

Здесь *значение* — это целочисленное значение, к которому будет применена операция сдвига, а *сдвиг* — это количество битов сдвига. Например,

```
13 << 3
```

означает сдвиг всех битов в значении 13 на три позиции влево. При этом три бита слева выходят за пределы значения и отбрасываются, а новообразованные позиции справа заполняются нулями (рис. Д.1).

Рис. Д.1. Операция сдвига влево

Поскольку значение в каждой позиции бита представляет удвоенное значение бита, находящегося справа (см. приложение А), то смещение на одну позицию влево эквивалентно умножению на 2. Точно так же смещение на две позиции эквивалентно умножению на 2^2, а смещение на n позиций эквивалентно умножению на 2^n. Таким образом, результатом операции `13 << 3` равно 13×2^3, или 104.

Операция сдвига влево похожа на такую же операцию в языке ассемблера. Однако в языке ассемблера эта операция приводит к изменению содержимого регистра, а в языке C++ образуется новое значение без изменения существующих значений. Рассмотрим, например, следующий фрагмент кода:

```
int x = 20;
int y = x << 3;
```

Этот код не изменяет значение `x`. Выражение `x << 3` использует значение `x` для получения нового значения, подобно тому, как в выражении `x + 3` образуется новое значение без изменения значения переменной `x`.

Чтобы в результате выполнения операции сдвига влево изменить значение переменной, необходимо использовать присваивание. Для этого можно использовать обычное присваивание или операцию <<=, которая объединяет сдвиг битов и присваивание:

```
x = x << 4; // обычное присваивание
y <<= 2;    // сдвиг и присваивание
```

Операция сдвига вправо (>>), исходя из своего названия, осуществляет сдвиг битов вправо. Она имеет следующий синтаксис:

```
значение >> сдвиг
```

Здесь, *значение* — это целочисленное значение, которое будет сдвинуто, а *сдвиг* — это количество битов сдвига. Например,

```
17 >> 2
```

означает сдвиг всех битов в значении `17` на две позиции вправо. Для беззнаковых целых чисел новообразованные позиции слева заполняются нулями, а биты, выходящие за границы значения, отбрасываются. Для целых чисел со знаком новообразованные позиции могут быть заполнены нулями или значением исходного крайнего левого бита. Выбор варианта зависит от реализации C++. (На рис. Д.2 показан пример заполнения нулями.)

Рис. Д.2. Операция сдвига вправо

Сдвиг на одну позицию вправо эквивалентен целочисленному делению на 2. В общем случае, сдвиг на *n* позиций вправо эквивалентен целочисленному делению на 2^n.

В языке C++ определена также операция сдвига вправо с присваиванием, которую можно использовать для замены значения переменной на сдвинутое значение:

```
int q = 43;
q >>= 2; // замена 43 на 43 >> 2, или 10
```

В некоторых системах использование операций сдвига вправо и влево позволяет ускорить целочисленное умножение и деление на 2 по сравнению с операцией деления, однако в связи с тем, что компилятор будет работать лучше с оптимизированным кодом, эти различия будут едва заметными.

Логические битовые операции

Логические битовые операции аналогичны обычным логическим операциям, за исключением того, что они выполняются над значением по его отдельным битам, а не над всем значением в целом. Например, рассмотрим обычную операцию отрицания (`!`) и операцию битового отрицания (или нахождение дополнительного кода числа — `~`). Операция `!` преобразует значение `true` (ненулевое) в `false`, и значение `false` в `true`. Операция `~` преобразует каждый индивидуальный бит на противоположный (`1` в `0` и `0` в `1`). Например, рассмотрим значение `3`, имеющее тип `unsigned char`:

```
unsigned char x = 3;
```

Выражение `!x` имеет значение `0`. Чтобы определить значение `~x`, его необходимо записать в двоичной форме: `00000011`. Затем потребуется преобразовать каждый `0` в `1` и наоборот. В результате получится значение `11111100`, которому в десятичной системе будет соответствовать значение `252`. (На рис. Д.3 показан пример 16-битного эквивалента.) Новое значение называется *дополнением* исходного значения.

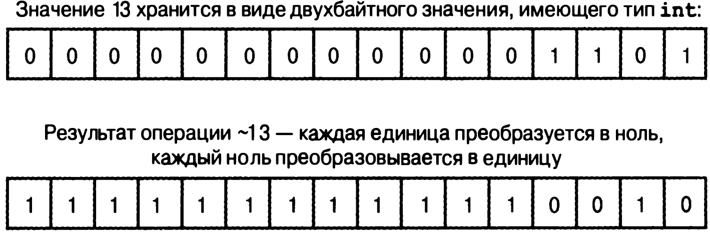


Рис. Д.3. Операция логического отрицания

Битовая операция ИЛИ (`|`) комбинирует два целочисленных значения с целью получения нового целочисленного значения. Каждому биту в новом значении присваивается `1`, если одному или другому либо обоим соответствующим битам в исходных значениях присвоена единица. Если соответствующим битам присвоен `0`, то искомому биту присваивается ноль (рис. Д.4).

В табл. Д.1 приведен порядок комбинирования битов при выполнении операции `|`.

Рис. Д.4. Битовая операция ИЛИ

Таблица Д.1. Результат выполнения операции b1 | b2

Значения, присвоенные битам	b1 = 0	b1 = 1
b2 = 0	0	1
b2 = 1	1	1

Операция != комбинирует битовую операцию ИЛИ и присваивание:

```
a != b; // переменной a присваивается результат a | b
```

Битовая операция исключающего ИЛИ (^) комбинирует два целочисленных значения с целью получения нового целочисленного значения. Каждому биту в новом значении присваивается 1, если одному или другому либо обоим соответствующим битам в исходных значениях присвоена единица. Если обоим соответствующим битам присвоен ноль или единица, то искомому биту будет присвоен ноль (рис. Д.5).

В табл. Д.2 приведен порядок комбинирования битов при выполнении операции ^.

Таблица Д.2. Результат выполнения операции b1 ^ b2

Значения, присвоенные битам	b1 = 0	b1 = 1
b2 = 0	0	1
b2 = 1	1	0

Рис. Д.5. Битовая операция исключающего ИЛИ

Операция ^= комбинирует битовую операцию исключающего ИЛИ и присваивание:

```
a ^= b; // присваивает переменной a результат операции a ^ b
```

Битовая операция И (&) комбинирует два целочисленных значения с целью получения нового целочисленного значения. Каждому биту в новом значении присваивается 1, если только каждому соответствующему биту в исходных значениях присвоена единица. Если каждому соответствующему биту присвоен ноль, то искомому биту будет присвоен ноль (рис. Д.6).

Рис. Д.6. Битовая операция И

В табл. Д.3 приведен порядок комбинирования битов при выполнении операции &.

Таблица Д.3. Результат выполнения операции b1 & b2

Значения, присвоенные битам	b1 = 0	b1 = 1
b2 = 0	0	0
b2 = 1	0	1

Операция &= комбинирует битовую операцию И с присваиванием:

```
a & b; // присваивает переменной a результат операции a & b
```

Альтернативные варианты представления битовых операций

Язык C++ предлагает альтернативные варианты представления некоторых битовых операций, как показано в табл. Д.4. Они предназначены для тех случаев, когда набор символов не позволяет использовать традиционные представления битовых операций.

Таблица Д.4. Представления битовых операций

Стандартное представление	Альтернативное представление
&	bitand
&=	and_eq
|	bitor
|=	or_eq
~	compl
^	xor
^=	xor_eq

Альтернативные варианты позволяют записывать операции вроде следующих:

```
b = compl a bitand b;   // то же, что и b = ~a & b;
c = a xor b;            // то же, что и c = a ^ c;
```

Примеры использования битовых операций

Нередко при осуществлении управления аппаратными устройствами возникает необходимость в том, чтобы включить или выключить биты или проверить их состояние. Эти действия можно выполнять с помощью битовых операций. Сейчас мы поговорим о них вкратце.

В следующих примерах `lottabits` представляет основное значение, а `bit` — значение, соответствующее определенному биту. Биты пронумерованы справа налево, начиная с нулевого, поэтому значение, соответствующее позиции *n* бита, равно 2^n. Например, целое число, в котором только третьему биту присвоена единица, имеет значение 2^3, или 8. Вообще, каждый индивидуальный бит соответствует степени 2, о чем речь шла в приложении А. Поэтому мы будем использовать термин *бит* для обозначения степени 2; это будет соответствовать определенному биту, которому присвоено значение 1, а все остальные биты будут нулевыми.

Включение бита

Следующие две операции включают бит в `lottabits`, который соответствует биту, представленному значением `bit`:

```
lottabits = lottabits | bit;
lottabits |= bit;
```

Каждая операция присваивает соответствующему биту единицу, вне зависимости от предыдущего значения бита. Это объясняется тем, что операция ИЛИ для 1 и 0 или 1 дает 1. Все остальные биты в `lottabits` остаются неизмененными. Это объясняется тем, что операция ИЛИ для 0 и 0 дает 0, а для 0 и 1 дает 1.

Переключение бита

Следующие операции переключают бит в `lottabits`, который соответствует биту, представленному значением `bit`. Другими словами, они включают бит, если он выключен и выключают бит, если он включен:

```
lottabits = lottabits ^ bit;
lottabits ^= bit;
```

Операция исключающего ИЛИ для 1 и 0 дает 1, включая выключенный бит; операция исключающего ИЛИ для 1 и 1 дает 0, выключая включенный бит. Все остальные биты в `lottabits` остаются неизмененными. Это объясняется тем, что исключающее ИЛИ для 0 и 0 дает 0, а для 0 и 1 дает 1.

Выключение бита

Следующий оператор выключает бит в `lottabits`, который соответствует биту, представленному значением `bit`:

```
lottabits = lottabits & ~bit;
```

Этот оператор выключает бит независимо от его предыдущего состояния. Во-первых, операция `~bit` дает целое число, в котором каждому его биту будет присвоено

значение 1 *кроме* того бита, которому изначально было присвоено значение 1; этот бит будет хранить нулевое значение. Операция И для 0 и любого бита дает 0, выключая, таким образом, этот бит. Все остальные биты в lottabits остаются неизмененными. Это объясняется тем, что операция И для 1 и любого бита дает то же значение, которое хранилось в этом бите.

Далее показана более короткая запись этого же оператора:

```
lottabits &= ~bit;
```

Проверка значения, хранящегося в бите

Предположим, что вам необходимо проверить, имеет ли бит, соответствующий bit, значение 1 в lottabits. Следующая проверка вряд ли будет работать:

```
if (lottabits == bit) // ничего хорошего
```

Дело в том, что даже если соответствующий бит в lottabits хранит 1, то другие биты также могут иметь 1. Вышеприведенное равенство справедливо *только* тогда, когда соответствующий бит имеет значение 1. Проблема связана с использованием операции И для lottabits и bit. В результате ее выполнения будет получено значение 0 во всех остальных битах, поскольку И для 0 и любого значения дает 0. Неизмененным останется только тот бит, который будет соответствовать значению бита, поскольку операция И для 1 и любого значения дает это же значение. Таким образом, подходящим вариантом будет следующий:

```
if (lottabits & bit == bit) // проверка бита
```

Обычно программисты упрощают эту запись до такого вида:

```
if (lottabits & bit) // проверка бита
```

Поскольку в bit один бит имеет значение 1, а остальные биты имеют нули, то результатом операции lottabits & bit будет либо 0 (что равносильно false), либо bit, что, будучи ненулевым значением, соответствует true.

Операции разыменования членов

Язык C++ позволяет определять указатели на члены класса. Эти указатели включают специальные обозначения для их объявления и разыменования. Чтобы посмотреть, что включает указатель, для начала рассмотрим простой класс:

```
class Example
{
private:
   int feet;
   int inches;
public:
   Example();
   Example(int ft);
   ~Example();
   void show_in() const;
   void show_ft() const;
   void use_ptr() const;
};
```

Рассмотрим член inches этого класса. Без определенного объекта inches представляет собой метку. Другими словами, класс определяет inches как идентификатор члена, однако вам нужен объект, прежде чем вы на самом деле сможете выделить участок памяти:

```
Example ob; // теперь ob.inches существует
```

Таким образом, вы определяете реальную ячейку памяти, используя идентификатор inches вместе с определенным объектом. (В функции-члене можно опустить имя объекта, однако впоследствии объект будет воспринят как объект, на который указывает указатель.)

Указатель на член для идентификатора inches можно определить следующим образом:

```
int Example::*pt = &Example::inches;
```

Этот указатель немного отличается от обычного указателя. Обычный указатель указывает на определенную ячейку памяти. А указатель pt не указывает на определенную ячейку памяти, поскольку в объявлении не идентифицирован определенный объект. Наоборот, указатель pt идентифицирует местоположение члена inches в объекте Example. Подобно идентификатору inches, идентификатор pt предназначен для использования вместе с идентификатором объекта. По существу, выражение *pt играет роль идентификатора inches. Таким образом, идентификатор объекта можно использовать для того, чтобы определить, к какому объекту производится доступ, а указатель pt — для того, чтобы определить член inches этого объекта. Например, метод класса может использовать следующий код:

```
int Example::*pt = &Example::inches;
Example ob1;
Example ob2;
Example *pq = new Example;
cout << ob1.*pt << endl;     // отображает член inches ob1
cout << ob2.*pt << endl;     // отображает член inches ob2
cout << po->pt << endl;      // отображает член inches *po
```

Здесь, .* и -> называют *операциями разыменования членов.* Когда у вас имеется определенный объект, например ob1, то ob1.*pi идентифицирует член inches объекта ob1. Подобным образом, pq->*pt идентифицирует член inches объекта, на который указывает pq.

В предыдущем примере при изменении объекта изменялся используемый член inches. Однако можно изменить и сам указатель pt. Поскольку feet имеет тот же тип, что и inches, можно переопределить указатель pt, чтобы он указывал на член feet, а не на член inches; впоследствии ob1.*pt будет указывать на член feet объекта ob1:

```
pt = &Example::feet;         // переопределение pt
cout << ob1.*pt << endl;     // отображает член feet объекта ob1
```

По сути, комбинация *pt замещает имя члена и может использоваться для идентификации различных имен членов (такого же типа).

Указатели на члены можно использовать и для идентификации функций-членов. Синтаксис этой операции несколько запутанный. Вспомните, что объявление указателя на функцию void() обычного типа без аргументов выглядит следующим образом:

```
void (*pf)();                // pf указывает на функцию
```

Объявление указателя на функцию-член необходимо для того, чтобы показать, что функция принадлежит определенному классу. Далее представлен пример объявления указателя на метод класса Example:

```
void (Example::*pf)() const; // pf указывает на функцию-член Example
```

Этот пример показывает, что pf может применяться точно так же, как и метод Example. Обратите внимание, что элемент Example::*pf должен быть заключен в скобки. Для этого указателя можно присвоить адрес определенной функции-члена:

```
pf = &Example::show_inches;
```

Заметьте, что в отличие от присваивания указателя на обычную функцию здесь вы можете и должны использовать адресную операцию. Выполнив присваивание, можно будет использовать объект для вызова функции-члена:

```
Example ob3(20);
(ob3.*pf)();  // вызывает show_inches() с использованием объекта ob3
```

Всю конструкцию ob3.*pf необходимо заключить в скобки, чтобы идентифицировать выражение, которое представляет имя функции.

Поскольку show_feet() имеет ту же форму прототипа, что и show_inches(), то pf можно использовать также и для доступа к методу show_feet():

```
pf = &Example::show_feet;
(ob3.*pf)();  // применяет show_feet() к объекту ob3
```

В определении класса, представленного в листинге Д.1, имеется метод use_ptr(), который применяет указатели на члены для доступа к элементам данных и функциям-членам класса Example.

Листинг Д.1. memb_pt.cpp

```
// memb_pt.cpp -- разыменование указателей на члены классов
#include <iostream>
using namespace std;

class Example
{
private:
   int feet;
   int inches;

public:
   Example();
   Example(int ft);
   ~Example();
   void show_in() const;
   void show_ft() const;
   void use_ptr() const;
};

Example::Example()
{
   feet = 0;
   inches = 0;
}
```

```
Example::Example(int ft)
{
   feet = ft;
   inches = 12 * feet;
}

Example::~Example()
{
}

void Example::show_in() const
{
   cout << inches << " дюймов\n";
}

void Example::show_ft() const
{
   cout << feet << " футов\n";
}

void Example::use_ptr() const
{
   Example yard(3);
   int Example::*pt;
   pt = &Example::inches;
   cout << "Установка pt в &Example::inches:\n";
   cout << "this->pt: " << this->*pt << endl;
   cout << "yard.*pt: " << yard.*pt << endl;
   pt = &Example::feet;
   cout << "Установка pt в &Example::feet:\n";
   cout << "this->pt: " << this->*pt << endl;
   cout << "yard.*pt: " << yard.*pt << endl;
   void (Example::*pf)() const;
   pf = &Example::show_in;
   cout << "Установка pf в &Example::show_in:\n";
   cout << "Использование (this->*pf)(): ";
   (this->*pf)();
   cout << "Использование (yard.*pf)(): ";
   (yard.*pf)();
}

int main()
{
   Example car(15);
   Example van(20);
   Example garage;

   cout << "Вывод car.use_ptr():\n";
   car.use_ptr();
   cout << "\nВывод van.use_ptr():\n";
   van.use_ptr();

   return 0;
}
```

Ниже показан пример выполнения программы из листинга Д.1:

```
Вывод car.use_ptr():
Установка pt в &Example::inches:
this->pt: 180
yard.*pt: 36
Установка pt в &Example::feet:
this->pt: 15
yard.*pt: 3
Установка pf в &Example::show_in:
Использование (this->*pf)(): 180 дюймов
Использование (yard.*pf)(): 36 дюймов

Вывод van.use_ptr():
Установка pt в &Example::inches:
this->pt: 240
yard.*pt: 36
Установка pt в &Example::feet:
this->pt: 20
yard.*pt: 3
Установка pf в &Example::show_in:
Использование (this->*pf)(): 240 дюймов
Использование (yard.*pf)(): 36 дюймов
```

В этом примере указателю присваиваются значения во время компиляции. В более сложном классе вы можете использовать указатели на элементы данных и методы, для которых точный член, связанный с указателем, определяется во время выполнения.

ПРИЛОЖЕНИЕ Е

Шаблонный класс string

Большая часть материала из этого приложения посвящена техническим вопросам. Однако если вы желаете просто узнать о возможностях шаблонного класса string, можете сконцентрироваться на описаниях различных методов string.

Класс string основан на шаблонном определении:

```
template<class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
class basic_string {...};
```

Здесь charT представляет тип, который хранится в строке. Параметр traits представляет класс, определяющий необходимые свойства, которыми должен обладать тип для представления строки. Например, он должен иметь метод length(), который возвращает длину строки, представленную в виде массива, имеющего тип charT. Окончание такого массива обозначается значением charT(0), которое является обобщенной формой нулевого символа. (Выражение charT(0) обозначает приведение 0 к типу charT. Вместо него могло быть просто значение 0, как для типа char, или, в общем случае, вместо него мог быть объект, созданный конструктором charT.) Класс также включает методы сравнения значений и так далее. Параметр Allocator представляет класс для управления распределением памяти под строку. Шаблон allocator<charT>, предлагаемый по умолчанию, использует операции new и delete стандартными способами.

Существуют две предварительно определенные специализации:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

Эти специализации, в свою очередь, используют следующие специализации:

```
char_traits<char>
allocator<char>
char_traits<wchar_t>
allocator<wchar_t>
```

Вы можете создать класс srting для некоторого типа, отличного от char или wchar_t, определяя класс traits и используя шаблон basic_string.

Тринадцать типов и константа

Шаблон basic_string определяет несколько типов, которые могут применяться в определениях методов:

```
typedef traits                                  traits_type;
typedef typename traits::char_type              value_type;
typedef Allocator                               allocator_type;
typedef typename Allocator::size_type           size_type;
typedef typename Allocator::difference_type     difference_type;
typedef typename Allocator::reference           reference;
typedef typename Allocator::const_reference     const_reference;
typedef typename Allocator::pointer             pointer;
typedef typename Allocator::const_pointer       const_pointer;
```

Обратите внимание, что `traits` является шаблонным параметром, который соответствует некоторому определенному типу, например, `char_traits<char>`; `traits_type` становится `typedef` для данного специфического типа. Обозначение

```
typedef typename traits::char_type              value_type;
```

означает, что `char_type` представляет собой имя типа, определенного в классе, представляемом `traits`. Служебное слово `typename` используется для того, чтобы сообщить компилятору, что выражение `traits::char_type` представляет собой тип. Для специализации `string`, например, `value_type` имеет тип `char`.

`size_type` используется подобно `size_of`, за исключением того, что она возвращает размер строки в виде сохраняемого типа. Для специализации `string` это может быть тип `char`, в случае которого `size_type` будет эквивалентен `size_of`. Этот тип является беззнаковым.

`difference_type` служит для определения расстояния между двумя элементами строки, которое выражается в единицах, соответствующих размеру одного элемента. Обычно это знаковая версия типа на основе `size_type`.

Для специализации `char` тип `pointer` является типом `char*`, а `reference` — типом `char&`. Однако если создать специализацию для спроектированного типа, то эти типы (`pointer` и `reference`) могут относиться к классу, имеющему те же свойства, что и базовые указатели и ссылки.

Чтобы алгоритмы стандартной библиотеки шаблонов (Standard Template Library — STL) можно было использовать в строках, шаблон определяет некоторые типы итераторов:

```
typedef (models random access iterator)         iterator;
typedef (models random access iterator)         const_iterator;
typedef std::reverse_iterator<iterator>         reverse_iterator;
typedef std::reverse_iterator<const_iterator>  const_reverse_iterator;
```

Шаблон определяет также и статическую константу:

```
static const size_type npos = -1;
```

Поскольку `size_type` является беззнаковым типом, то присваивание значения `-1` на самом деле будет соответствовать присваиванию `npos` наибольшего возможного беззнакового значения. Это значение соответствует значению, которое больше самого большого индекса массива.

Информация о данных, конструкторы и вспомогательные элементы

Конструкторы можно описать по результатам их работы. Поскольку приватные части класса могут не зависеть от реализации, то эти результаты следует описывать в виде информации, доступной как часть общедоступного интерфейса. В табл. Е.1 приводятся некоторые методы, чьи возвращаемые значения могут использоваться для описания результатов работы конструкторов и других методов. Обратите внимание, что большинство терминологии взято из STL.

Таблица Е.1. Некоторые методы данных шаблонного класса string

Метод	Результат
begin()	Итератор, указывающий на первый символ в строке (также доступный в версии const, которая возвращает итератор const).
end()	Итератор, указывающий на элемент, следующий за последним элементом (также доступный в версии const).
rbegin()	Обратный итератор, указывающий на элемент, следующий за последним элементом (также доступный в версии const).
rend()	Обратный итератор, указывающий на первый символ (также доступный в версии const).
size()	Количество элементов в строке, равное расстоянию от begin() до end().
length()	То же, что и size().
capacity()	Размещенное количество элементов в строке. Может быть больше, чем действительное количество символов. Значение capacity()-size() представляет количество символов, которые могут быть присоединены к строке до того, как возникнет необходимость в размещении большего количества памяти.
max_size()	Максимально допустимый размер строки.
data()	Указатель типа const charT*, который указывает на первый элемент массива, чьи первые элементы size() равны соответствующим элементам в строке, управляемой *this. Указатель не должен считаться действительным после того, как был модифицирован сам объект string.
c_str()	Указатель типа const charT*, который указывает на первый элемент массива, чьи первые size() элементов равны соответствующим элементам в строке, управляемой *this, и чей следующий элемент является символом charT(0) (маркер окончания строки) для типа charT. Указатель не должен считаться действительным после того, как был модифицирован сам объект string.
get_allocator()	Копия объекта allocator, который используется для распределения памяти для объекта string.

Необходимо понимать различие между методами begin(), rend(), data() и c_str(). Все они связаны с первым символом в строке, но разными способами. Методы begin() и rend() возвращают итераторы, которые представляют собой обобщенную форму указателя, о чем было сказано в главе 16. В частности, метод begin() возвращает модель однонаправленного итератора, а rend() возвращает

копию обратного итератора. Оба метода относятся к действительной строке, управляемой объектом string. (Поскольку класс string использует динамическое распределение памяти, то действительное содержимое строки не должно находиться внутри объекта, поэтому мы используем термин *управление* для описания взаимосвязи между объектом и строкой.) Вы можете использовать методы, которые возвращают итераторы посредством алгоритмов STL на основе итераторов. Например, функцию reverse() библиотеки STL можно применять для того, чтобы содержимое строки было представлено в обратном порядке:

```
string word;
cin >> word;
reverse(word.begin(), word.end());
```

С другой стороны, методы data() и c_str() возвращают обычные указатели. Более того, возвращаемые указатели указывают на первый элемент *массива*, который содержит символы строки. Этот массив может быть, но не обязательно, копией исходной строки, управляемой объектом string. (Внутреннее представление объекта string может быть определено в виде массива, но это не обязательно.) Поскольку возвращаемые указатели могут указывать на исходные данные, то они имеют тип const, поэтому не могут применяться для изменения данных. Кроме этого, указатели могут измениться после изменения строки, а, значит, они могут указывать на исходные данные. Различие между методами data() и c_str() заключается в том, что массив, на который указывает c_str(), завершается пустым символом (или его эквивалентом), в то время как data() просто гарантирует наличие действительных символов строки. Таким образом, метод c_str() может использоваться, например, в качестве аргумента для функции, которая должна получить строку в стиле языка С:

```
string file("tofu.man");
ofstream outFile(file.c_str());
```

Подобным же образом методы data() и size() могут использоваться вместе с функцией, которая должна получить указатель на элемент массива и значение, которое представляет количество элементов для обработки:

```
string vampire("Do not stake me, oh my darling!");
int vlad = byte_check(vampire.data(), vampire.size());
```

В реализации C++ может быть выбран вариант представления строки объекта string в виде динамически размещаемой строки в стиле языка С для реализации прямого итератора в виде указателя char *. В этом случае в реализации может быть выбран вариант, при котором методы begin(), data() и c_str() будут возвращать один и тот же указатель. Однако более предпочтительным и простым вариантом является возврат ссылок на три различных объекта данных.

Ниже показаны шесть конструкторов и один деструктор для шаблонного класса basic_string:

```
explicit basic_string(const Allocator& a = Allocator());

basic_string(const charT* s, const Allocator& a = Allocator());

basic_string(const basic_string& str, size_type pos = 0,
             size_type n = npos, const Allocator& a = Allocator());

basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
```

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());

template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
        const Allocator& a = Allocator());

~basic_string();
```

Обратите внимание, что каждый из шести конструкторов имеет аргумент вида:

```
const Allocator& a = Allocator()
```

Вспомните, что Allocator — это имя шаблонного параметра для класса allocator, который предназначен для управления памятью. Allocator() — это конструктор для этого класса, предлагаемый по умолчанию. Таким образом, по умолчанию конструкторы используют версию объекта allocator, предлагаемую по умолчанию, однако они дают возможность применения другой версии объекта allocator. В следующих разделах речь пойдет о каждом конструкторе отдельно.

Конструкторы по умолчанию

Прототип для конструктора по умолчанию имеет вид:

```
explicit basic_string(const Allocator& a = Allocator());
```

Обычно можно принимать аргумент для класса allocator, предлагаемый по умолчанию, и использовать конструктор для создания пустых строк:

```
string bean;
wstring theory;
```

После вызова конструктора устанавливаются следующие отношения:

- Метод data() возвращает непустой указатель, к которому может быть добавлено значение 0.
- Метод size() возвращает 0.
- Возвращаемое значение для capacity() не определено.

Предположим, что вы присваиваете значение, возвращаемое методом data(), указателю str. В этом случае первое условие означает, что str + 0 является действительным.

Конструкторы, использующие массивы

Конструкторы, использующие массивы, позволяют инициализировать объект string на основе строки в стиле языка C; в общем случае, они позволяют инициализировать специализацию charT на основе массива значений charT:

```
basic_string(const charT* s, const Allocator& a = Allocator());
```

Чтобы определить, сколько необходимо скопировать символов, конструктор применяет метод traits::length() к массиву, на который указывает s. (Указатель s не должен быть пустым указателем.) Например, оператор

```
string toast("Here's looking at you, kid.");
```

инициализирует объект toast, используя указанную строку символов. Метод traits::length() для типа char использует пустой символ, чтобы определить количество символов, которые необходимо скопировать.

После вызова конструктора устанавливаются следующие отношения:

- Метод data() возвращает указатель на первый элемент копии массива s.
- Метод size() возвращает значение, равное traits::length().
- Метод capacity() возвращает значение, как минимум такое же большое, как и size().

Конструкторы, использующие часть массива

Конструкторы, использующие часть массива, позволяют инициализировать объект string на основе части строки в стиле языка C; в общем случае, они позволяют инициализировать специализацию charT на основе части массива значений charT:

```
basic_string(const charT* s, size_type n,
      const Allocator& a = Allocator());
```

Данный конструктор копирует в создаваемый объект общее количество символов n из массива, на который указывает s. Обратите внимание, что копирование будет продолжаться до тех пор, пока указатель s будет иметь меньшее количество символов, чем n. Если n превышает длину s, то конструктор интерпретирует содержимое за строкой так, как будто там содержатся данные, имеющие тип charT.

Для этого конструктора необходимо, чтобы указатель s не был пустым, и чтобы выполнялось условие n < npos. (Вспомните, что npos является статической константой класса, равной максимально возможному количеству элементов в строке.) Если n будет равно npos, конструктор возбуждает исключение out_of_range. (Поскольку n имеет тип size_type, а npos является максимальным значением size_type, n не может быть больше чем npos.) В противном случае после вызова конструктора будут установлены следующие отношения:

- Метод data() возвращает указатель на первый элемент копии массива s.
- Метод size() возвращает n.
- Метод capacity() возвращает значение, как минимум такое же большое, как и size().

Конструкторы копирования

Конструктор копирования предлагает несколько аргументов со значениями по умолчанию:

```
basic_string(const basic_string& str, size_type pos = 0,
      size_type n = npos, const Allocator& a = Allocator());
```

При вызове конструктора копирования только с одним аргументом basic_string будет инициализирован новый объект для аргумента string:

```
string mel("I'm ok!");
string ida(mel);
```

Здесь ida получает копию строки, управляемой mel.

Необязательный второй аргумент pos определяет позицию в исходной строке, начиная с которой будет произведено копирование:

```
string att("Telephone home.");
string et(att, 4);
```

Номера позиций начинаются с 0, поэтому позиция 4 соответствует символу p. Таким образом, et присваивается строка "Telephone home.".

Необязательный третий аргумент n определяет максимальное количество символов, которые будут скопированы. Таким образом,

```
string att("Telephone home.");
string pt(att, 4, 5);
```

присваивает pt строку "Telephone". Однако этот конструктор не выходит за пределы исходной строки; например,

```
string pt(att, 4, 200)
```

останавливается после того, как будет скопирована точка. Таким образом, конструктор на самом деле копирует количество символов, которое равно наименьшему из n и str.size() - pos.

Для этого конструктора необходимо, чтобы pos <= str.size(), то есть, чтобы исходная позиция, с которой начнется копирование, находилась в пределах исходной строки; если это условие нарушается, конструктор возбуждает исключение out_of_range. В противном случае, если copy_len будет представлять наименьше значение из n и str.size() - pos, то после вызова конструктора будут установлены следующие отношения:

- Метод data() возвращает указатель на копию элементов copy_len, скопированных из строки str, начиная с позиции pos в str.
- Метод size() возвращает copy_len.
- Метод capacity() возвращает значение, как минимум такое же большое, как и size().

Конструкторы, использующие n копий символа

Конструктор, использующий n копий символа, создает объект string, состоящий из n последовательных символов, каждый из которых имеет значение c:

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

Для конструктора необходимо, чтобы выполнялось условие n < npos. Если n будет равно npos, конструктор возбудит исключение out_of_range. В противном случае после вызова конструктора будут установлены следующие отношения:

- Метод data() возвращает указатель на первый элемент строки, состоящей из n элементов, каждый из которых имеет символ c.
- Метод size() возвращает n.
- Метод capacity() возвращает значение, как минимум такое же большое, как и size().

Конструкторы, использующие диапазон

Эти конструкторы используют диапазон, определяемый итератором в стиле библиотеки STL:

```
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
     const Allocator& a = Allocator());
```

Итератор begin указывает на элемент в исходной строке, с которого начнется копирование, а end указывает на последнюю позицию, которая будет скопирована.

Эту форму можно использовать для массивов, строк или контейнеров библиотеки STL:

```
char cole[40] = "Old King Cole was a merry old soul.";
string title(cole + 4, cole + 8);
vector<char> input;
char ch;
while (cin.get(ch) && ch != '\n')
   input.push_back(ch);
string str_input(input.begin(), input.end());
```

Во время первого использования InputIterator оценивается как тип const char *. Во второй раз InputIterator оценивается как тип vector<char>::iterator.

После вызова конструктора устанавливаются следующие отношения:

- Метод data() возвращает указатель на первый элемент строки, сформированной посредством копирования элементов из диапазона (begin, end).
- Метод size() возвращает расстояние между begin и end. (Расстояние измеряется в единицах, равных размеру типа данных, полученному при разыменовании итератора.)
- Метод capacity() возвращает значение, как минимум такое же большое, как и size().

Различные операции с памятью

Работа некоторых методов связана с памятью, а именно с очисткой содержимого памяти, изменением размеров строки, подгонкой вместимости строки. В табл. Е.2 перечислены методы, работа которых связана с памятью.

Таблица Е.2. Некоторые методы, работа которых связана с памятью

Метод	Результат выполнения
void resize(size_type n)	Возбуждает исключение out_of_range, если n > pos. В противном случае изменяет размер строки до n, отбрасывая конец строки, если n < size(), и заполняя строку символами charT(0), если n > size().
void resize(size_type n, charT c)	Возбуждает исключение out_of_range, если n > npos. В противном случае изменяет размер строки до n, отбрасывая конец строки, если n < size(), и заполняя строку символами charT(0), если n > size().

Окончание табл. Е.2

Метод	Результат выполнения
`void reserve(size_type res_arg = 0)`	Устанавливает вместимость `capacity()` больше или равной `res_arg`. Поскольку при этом происходит повторное размещение строки, то предыдущие ссылки, итераторы и указатели на строку аннулируются.
`void clear()`	Удаляет все символы из строки.
`bool empty() const`	Возвращает `true`, если `size() == 0`.

Доступ к строке

Существуют четыре варианта доступа к индивидуальным символам, два из которых используют операцию `[]`, а два других — метод `at()`:

```
reference operator[](size_type pos);
const_reference operator[](size_type pos) const;
reference at(size_type n);
const_reference at(size_type n) const;
```

`operator[]()` позволяет обращаться к индивидуальному элементу строки, используя нотацию массива; этот вариант можно использовать для получения или изменения значения. `operator[]()` можно применять вместе с объектами `const`; этот вариант можно использовать только для получения значения:

```
string word("tack");
cout << word[0];       // отображает t
word[3] = 't';         // заменяет k на t
const ward("garlic");
cout << ward[2];       // отображает r
```

Методы `at()` предлагают похожую схему доступа, за исключением того, что в записи аргумента функции предусматривается индекс:

```
string word("tack");
cout << word.at(0);    // отображает t
```

Различие между ними, помимо различия в синтаксисе, заключается в том, что методы `at()` предусматривают проверку границ и возбуждают исключение `out_of_range`, если `pos >= size()`. Обратите внимание, что `pos` имеет тип `size_type`, который является беззнаковым типом; таким образом, для `pos` не допускаются отрицательные значения. Методы `operator[]()` не выполняют проверку границ, поэтому их поведение будет неопределенным, если `pos >= size()`, кроме тех случаев, когда версия `const` возвращает эквивалент нулевого символа, если `pos == size()`.

Таким образом, у вас имеется возможность выбора между безопасной работой (использование `at()` и проверки исключений) и скоростью выполнения (используя запись массива).

Существует также функция, возвращающая новую строку, которая является подстрокой исходной строки:

```
basic_string substr(size_type pos = 0, size_type n = npos) const;
```

Она возвращает строку, являющуюся копией строки, начиная с позиции pos и включая n символов, или до конца строки, смотря, что наступит раньше. Например, в следующем фрагменте кода pet присваивается подстрока "donkey":

```
string message("Maybe the donkey will learn to sing.");
string pet(message.substr(10, 6));
```

Основные варианты присваивания

Существует три перегруженных метода присваивания:

```
basic_string& operator=(const basic_string& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
```

Первый из них присваивает объект string другому объекту, второй присваивает строку в стиле языка С объекту string, а третий присваивает один символ объекту string. Таким образом, допустимыми являются следующие действия:

```
string name("George Wash");
string pres, veep, source;
pres = name;
veep = "Road Runner";
source = 'X';
```

Поиск строки

Класс string предлагает шесть функций поиска, каждая из которых имеет четыре прототипа. В следующих разделах мы вкратце рассмотрим эти функции.

Семейство find()

Методы find() имеют следующие прототипы:

```
size_type find (const basic_string& str, size_type pos = 0) const;
size_type find (const charT* s, size_type pos = 0) const;
size_type find (const charT* s, size_type pos, size_type n) const;
size_type find (charT c, size_type pos = 0) const;
```

Первый член возвращает начальную позицию, в которой впервые обнаруживается подстрока str в вызываемом объекте, при этом поиск начинается с позиции pos. Если подстрока не будет обнаружена, метод возвращает npos.

Ниже показан код для поиска позиции подстроки "hat" в строке longer:

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find(shorter); // присваивает loc1 позицию 1
size_type loc2 = longer.find(shorter, loc1 + 1); // присваивает loc2
                                                  // позицию 16
```

Поскольку второй поиск начинается с позиции 2 (буква a в слове That), то позиция, в которой впервые будет обнаружена подстрока hat, находится в конце строки. Чтобы не допустить сбоя, используется значение string::npos:

```
if (loc1 == string::npos)
   cout << "Not found\n";
```

Второй метод выполняет то же самое, за исключением того, что в качестве подстроки он использует массив символов, а не объект string:

```
size_type loc3 = longer.find("is"); //присваивает loc3 позицию 5
```

Третий метод выполняет то же самое, что и второй, за исключением того, что он применяет только первые n символов строки s. Результат будет таким же, как и при использовании конструктора basic_string(const charT* s, size_type n) и применении результирующего объекта в качестве аргумента string для первой формы find(). Например, следующая строка осуществляет поиск подстроки "fun":

```
size_type loc4 = longer.find("funds", 3); //присваивает loc4 позицию 10
```

Четвертый метод выполняет то же самое, что и первый, за исключением того, что в качестве подстроки в нем используется один символ, а не объект string:

```
size_type loc5 = longer.find('a'); //присваивает loc5 позицию 2
```

Семейство rfind()

Методы rfind() имеют следующие прототипы:

```
size_type rfind(const basic_string& str, size_type pos = npos) const;
size_type rfind(const charT* s, size_type pos = npos) const;
size_type rfind(const charT* s, size_type pos, size_type n) const;
size_type rfind(charT c, size_type pos = npos) const;
```

Эти методы работают аналогично методам find(), за исключением того, что в них осуществляется поиск позиции pos, в которой или перед которой последний раз обнаруживается строка или символ. Если подстрока не будет найдена, метод возвращает npos.

Далее показан фрагмент кода, в котором производится поиск подстроки "hat" в строке longer, начиная с ее конца:

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.rfind(shorter); // присваивает loc1 позицию 16
size_type loc2 = longer.rfind(shorter, loc1 - 1); // присваивает loc2
                                                  // позицию 1
```

Семейство find_first_of()

Методы find_first_of() имеют следующие прототипы:

```
size_type find_first_of(const basic_string& str, size_type pos = 0) const;
size_type find_first_of(const charT* s, size_type pos, size_type n) const;
size_type find_first_of(const charT* s, size_type pos = 0) const;
size_type find_first_of(charT c, size_type pos = 0) const;
```

Эти методы работают подобно соответствующим методам find(), за исключением того, что вместо поиска полностью совпадающей подстроки они ищут одиночный символ, который первым совпадает с одним из символов подстроки.

```
string longer("That is a funny hat.");
string shorter("fluke");
size_type loc1 = longer.find_first_of(shorter); //присваивает loc1 позицию 10
size_type loc2 = longer.find_first_of("fat");//присваивает loc2 позицию 2
```

Первой из пяти букв в слове fluke в строке longer обнаруживается буква f в слове funny. Первой из трех букв в слове fat в строке longer обнаруживается буква a в слове That.

Семейство find_last_of()

Методы find_last_of() имеют следующие прототипы:

```
size_type find_last_of (const basic_string& str, size_type pos = npos) const;
size_type find_last_of (const charT* s, size_type pos, size_type n) const;
size_type find_last_of (const charT* s, size_type pos = npos) const;
size_type find_last_of (charT c, size_type pos = npos) const;
```

Эти методы работают подобно соответствующим методам rfind(), за исключением того, что вместо поиска полностью совпадающей подстроки они ищут одиночный символ, который последним совпадает с одним из символов подстроки.

Ниже показан фрагмент кода для поиска позиции, в которой в строке longer последний раз обнаруживается любой из символов в словах "hat" и "any":

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find_last_of(shorter);//присваивает loc1 позицию 18
size_type loc2 = longer.find_last_of("any");//присваивает loc2 позицию 17
```

Последней из трех букв в слове hat в строке longer обнаруживается буква t в слове hat. Последней из трех букв в слове any в строке longer обнаруживается буква a в слове hat.

Семейство find_first_not_of()

Методы find_first_not_of() имеют следующие прототипы:

```
size_type find_first_not_of(const basic_string& str, size_type pos = 0) const;
size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
size_type find_first_not_of(charT c, size_type pos = 0) const;
```

Эти методы работают подобно соответствующим методам find_first_of(), за исключением того, что они производят поиск позиции, в которой впервые обнаруживается любой символ, не совпадающий ни с одним из символов в подстроке.

Ниже показан фрагмент кода для поиска позиции, в которой в строке longer впервые обнаруживается символ, не совпадающий с символами в словах "This" и "Thatch":

```
string longer("That is a funny hat.");
string shorter("This");
size_type loc1 = longer.find_first_not_of(shorter); // присваивает loc1
                                                     // позицию 2
size_type loc2 = longer.find_first_not_of("Thatch");// присваивает loc2
                                                     // позицию 4
```

Буква a в слове That является первой буквой в строке longer, которая не совпадает ни с одной буквой в слове This. Первый пробел в строке longer является первым символом, который не совпадает ни с одним символом в слове Thatch.

Семейство find_last_not_of()

Методы find_last_not_of() имеют следующие прототипы:

```
size_type find_last_not_of (const basic_string& str,
      size_type pos = npos) const;
size_type find_last_not_of (const charT* s, size_type pos, size_type n) const;
size_type find_last_not_of (const charT* s, size_type pos = npos) const;
size_type find_last_not_of (charT c, size_type pos = npos) const;
```

Эти методы работают подобно соответствующим методам find_last_of(), за исключением того, что они производят поиск позиции, в которой последний раз обнаруживается любой символ, не совпадающий ни с одним из символов в подстроке.

Ниже показан фрагмент кода для поиска позиций, в которых в строке longer последние два раза обнаруживается любой символ, не совпадающий ни с одним из символов в слове "That":

```
string longer("That is a funny hat.");
string shorter("That.");
size_type loc1 = longer.find_last_not_of(shorter); // присваивает loc1
                                                   // позицию 15
size_type loc2 = longer.find_last_not_of(shorter, 10);// присваивает loc2
                                                       // позицию 10
```

Последний пробел в строке longer является последним символом, которого нет в строке shorter. Буква f в строке longer является последним символом, которого нет в строке shorter вплоть до позиции 10.

Методы и функции сравнения

Класс string предлагает методы и функции, позволяющие проводить сравнение двух строк. Для начала рассмотрим прототипы методов:

```
int compare(const basic_string& str) const;
int compare(size_type pos1, size_type n1, const basic_string& str) const;
int compare(size_type pos1, size_type n1, const basic_string& str,
      size_type pos2, size_type n2) const;
int compare(const charT* s) const;
int compare(size_type pos1, size_type n1, const charT* s,
      size_type n2 = npos) const;
```

В этих методах используется метод traits::compare(), который определяется для конкретного символьного типа, применяемого в строке. Первый метод возвращает значение меньше нуля, если первая строка предшествует второй строке в соответствии с порядком, заданным в traits::compare(). Он возвращает 0, если две строки являются одинаковыми, и значение больше нуля, если за первой строкой будет следовать вторая. Если две строки идентичны вплоть до конца самой короткой из этих двух строк, то короткая строка будет предшествовать длинной.

В следующем примере сопоставляются строки s1 с s3 и s1 с s2:

```
string s1("bellflower");
string s2("bell");
string s3("cat");
int a13 = s1.compare(s3); // a13 < 0
int a12 = s1.compare(s2); // a12 > 0
```

Второй метод подобен первому, за исключением того, что сравнение производится по n1 символов, начиная с позиции pos1 в первой строке.

В следующем примере сопоставляются первые четыре символа в строке s1 с s2:

```
string s1("bellflower");
string s2("bell");
int a2 = s1.compare(0, 4, s2); // a2 = 0
```

Третий метод подобен первому, за исключением того, что сравнение производится по n1 символов, начиная с позиции pos1 в первой строке, и n2 символов, начиная с позиции pos2 во второй строке. Например, в следующем фрагменте кода сопоставляются out в stout с out в about:

```
string st1("stout boar");
string st2("mad about ewe");
int a3 = st1.compare(2, 3, st2, 6, 3); // a3 = 0
```

Четвертый метод подобен первому, за исключением того, что в нем для второй строки применяется массив символов, а не объект string.

Пятый метод подобен третьему, за исключением того, что в нем для второй строки используется массив символов, а не объект string.

Функции сравнения, не являющиеся членами, представляют собой перегруженные операции отношения:

```
operator==()
operator<()
operator<=()
operator>()
operator>=()
operator!=()
```

Каждая операция перегружается, чтобы произвести сравнение объекта string с объектом string, объекта string с массивом строк и массива строк с объектом string. Эти операции определяются на основе метода compare(), поэтому с точки зрения обозначения применять их гораздо удобнее.

Модификаторы строк

Класс string предлагает несколько методов модифицирования строк. Большинство из них имеет множество перегруженных версий, поэтому они могут использоваться для объектов string, массивов строк, индивидуальных символов и диапазонов итераторов.

Методы присоединения и добавления

Чтобы присоединить одну строку к другой можно использовать перегруженную операцию += или метод append(). Каждый из них возбуждает исключение length_error, если результат будет больше максимального размера строки. С помо-

щью операции += можно присоединить объект string, массив строк или индивидуальный символ к другой строке:

```
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
```

С помощью методов append() можно присоединять объект string, массив строк или индивидуальный символ к другой строке. Кроме этого, с их помощью можно присоединить часть объекта string, определив начальную позицию и количество присоединяемых символов, либо же определяя присоединяемый диапазон. Можно присоединить часть строки, определив необходимое количество символов. Вариант присоединения символа позволяет определить, сколько будет скопировано экземпляров этого символа. Ниже показаны прототипы различных методов append():

```
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos, size_type n);
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
basic_string& append(const charT* s);
basic_string& append(const charT* s, size_type n);
basic_string& append(size_type n, charT c); // присоединяет n копий c
```

Далее приводится несколько примеров:

```
string test("The");
test.append("ory");      // строка test представляет "Theory"
test.append(3,'!');      // строка test представляет "Theory!!!"
```

Функция operator+() перегружается, чтобы разрешить конкатенацию строк. Перегруженные функции не изменяют строку; наоборот, они создают новую строку, которая состоит из одной строки, присоединенной к другой строке. Функции добавления не являются функциями-членами и позволяют добавлять объект string к объекту string, массив строк к объекту string и объект string к символу. Ниже показаны некоторые примеры:

```
string st1("red");
string st2("rain");
string st3 = st1 + "uce"; // строка st3 представляет "reduce"
string st4 = 't' + st2;   // строка st4 представляет "train"
string st5 = st1 + st2;   // строка st5 представляет "redrain"
```

Дополнительные методы присваивания

Кроме базовой операции присваивания, класс string предлагает методы assign(), посредством которых можно присваивать объекту string всю строку целиком, часть строки или последовательность одинаковых символов. Ниже представлены прототипы различных методов assign():

```
basic_string& assign(const basic_string&);
basic_string& assign(const basic_string& str, size_type pos, size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
```

```
basic_string& assign(size_type n, charT c); // присваивает n копий c

template<class InputIterator>
  basic_string& assign(InputIterator first, InputIterator last);
```

Далее показаны некоторые примеры:

```
string test;
string stuff("set tubs clones ducks");
test.assign(stuff, 1, 5);    // строка test представляет "et tu"
test.assign(6, '#");         // строка test представляет "######"
```

Методы вставки

С помощью методов вставки insert() можно вставить объект string, массив строк, символ или несколько символов в объект string. Эти методы подобны методам append(), за исключением того, что они принимают дополнительный аргумент, указывающий на позицию, в которую будет произведена вставка нового материала. Этот аргумент может быть представлен позицией или итератором. Материал вставляется перед точкой вставки. Некоторые методы возвращают ссылку на результирующую строку. Если pos1 будет находиться за пределами искомой строки или если pos2 будет находиться за пределами вставляемой строки, метод возбуждает исключение out_of_range. Если размер результирующей строки окажется больше максимального, метод возбуждает исключение length_error. Ниже показаны прототипы различных методов insert():

```
basic_string& insert(size_type pos1, const basic_string& str);

basic_string& insert(size_type pos1, const basic_string& str,
      size_type pos2, size_type n);

basic_string& insert(size_type pos, const charT* s, size_type n);

basic_string& insert(size_type pos, const charT* s);

basic_string& insert(size_type pos, size_type n, charT c);

iterator insert(iterator p, charT c = charT());

void insert(iterator p, size_type n, charT c);

template<class InputIterator>
      void insert(iterator p, InputIterator first, InputIterator last);
```

Например, следующий фрагмент кода вставляет строку "former" перед буквой b в строке "The banker":

```
string st3("The banker.");
st3.insert(4, "former ");
```

Следующий фрагмент кода вставляет строку " waltzed" (не включая символ !, который будет девятым символом) перед самой точкой в конце строки "The former banker.":

```
st3.insert(st3.size() - 1, " waltzed!", 8);
```

Методы удаления

Методы удаления erase() удаляют символы из строки. Ниже представлены их прототипы:

```
basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

Первая форма перемещает символ из позиции pos на n символов далее или в конец строки, смотря, что наступит раньше. Вторая форма удаляет одиночный символ, на который ссылается позиция итератора, и возвращает итератор следующему элементу или, если элементов больше нет, возвращает end(). Третья форма удаляет символы в диапазоне [first, last), то есть, включая first и не включая last. Метод возвращает итератор элементу, который следует за последним удаленным символом.

Методы замены

Различные методы replace() распознают часть строки, которая должна быть заменена, и распознают замену. Заменяемую часть можно распознать по начальной позиции и счетчику символов или через диапазон итератора. В качестве замены может выступать объект string, массив строк или определенный символ, дублированный несколько раз. Объекты string и массивы впоследствии можно модифицировать, указывая определенную часть, используя позицию и счетчик, просто счетчик или диапазон итератора. Ниже представлены прототипы различных методов replace():

```
basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
basic_string& replace(size_type pos1, size_type n1,
      const basic_string& str, size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1,
      const charT* s, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);
basic_string& replace(iterator i1, iterator i2, const basic_string& str);
basic_string& replace(iterator i1, iterator i2, const charT* s, size_type n);
basic_string& replace(iterator i1, iterator i2, const charT* s);
basic_string& replace(iterator i1, iterator i2, size_type n, charT c);
template<class InputIterator>
      basic_string& replace(iterator i1, iterator i2, InputIterator j1,
      InputIterator j2);
```

А вот пример:

```
string test("Take a right turn at Main Street.");
test.replace(7,5,"left"); // заменяет слово right словом left
```

Для поиска позиций, используемых в replace, можно применить find():

```
string s1 = "old";
string s2 = "mature";
string s3 = "The old man and the sea";
string::size_type pos = s3.find(s1);
if (pos != string::npos)
   s3.replace(pos, s1.size(), s2);
```

В этом примере слово old будет заменено словом mature.

Другие методы модифицирования: copy() и swap()

Метод copy() копирует объект string или его часть в целевой массив строк:

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

В этом случае s указывает на искомый массив, n указывает на количество копируемых символов, а pos указывает на позицию в объекте string, с которой начнется копирование. Будут скопированы n символов или символы вплоть до последнего в объекте string, смотря, что наступит раньше. Функция возвращает количество скопированных символов. Метод не присоединяет пустой символ, и программист сам должен определить, может ли массив уместить скопированные символы.

Внимание!

Метод copy() не присоединяет пустой символ и не проверяет, может ли искомый массив уместить скопированные символы.

Метод swap() осуществляет обмен содержимого двух объектов string с помощью алгоритма с константным временем:

```
void swap(basic_string<charT,traits,Allocator>&);
```

Вывод и ввод

Для отображения объектов string класс string перегружает операцию <<. Она возвращает ссылку на объект istream, поэтому можно осуществлять конкатенацию вывода:

```
string claim("The string class has many features.");
cout << claim << endl;
```

Класс string перегружает операцию >>, поэтому можно считывать ввод в строку:

```
string who;
cin >> who;
```

Ввод прекращается, если будет достигнут конец файла, если будет прочитано максимальное количество символов, которые может уместить строка, или если будет достигнут пробельный символ. (Определение пробельного символа зависит от набора символов и типа, который представляет charT.)

Существуют две функции getline(). Первая имеет следующий прототип:

```
template<class charT, class traits, class Allocator>
   basic_istream<charT,traits>& getline(basic_istream<charT,traits>& is,
   basic_string<charT,traits,Allocator>& str, charT delim);
```

Она считывает символы из потока ввода is в строку str, пока не будет достигнут символ-ограничитель delim, пока не будет достигнут максимальный размер строки или пока не будет достигнут конец файла. Символ delim читается (то есть удаляется из потока ввода), но не сохраняется. Во втором варианте отсутствует третий аргумент и в нем используется символ новой строки (или его обобщенная форма), а не delim:

```
string str1, str2;
getline(cin, str1);      // чтение до конца строки
getline(cin, str2, '.'); // чтение до символа точки
```

ПРИЛОЖЕНИЕ Ж

Методы и функции библиотеки STL

Стандартная библиотека шаблонов (Standard Template Library — STL) содержит эффективные реализации наиболее распространенных алгоритмов. Она представляет их в виде общих функций, которые могут использоваться с любым контейнером, удовлетворяющим требованиям к определенному алгоритму, а также в виде методов, которые могут применяться в реализациях определенных классов контейнеров. Это приложение предполагает, что вы уже имеете некоторое представление о библиотеке STL. Для начала вы должны ознакомиться с материалом главы 16. В этом приложении предполагается, что вам знакомо понятие, например, итераторов и конструкторов.

Члены, общие для всех контейнеров

Все контейнеры определяют типы, перечисленные в табл. Ж.1. В этой таблице X — это тип контейнера, например, vector<int>, а T — это тип, хранящийся в контейнере, вроде int. Примеры, следующие за таблицей, помогают понять назначение контейнеров.

Таблица Ж.1. Типы, определенные для всех контейнеров

Тип	Значение
X::value_type	T, тип элемента
X::reference	T &
X::const_reference	const T &
X::iterator	Тип итератора, указывающего на T; его поведение подобно типу T *
X::const_iterator	Тип итератора, указывающий на const T; его поведение подобно типу const T *
X::difference_type	Знаковый целочисленный тип, используемый для представления расстояний от одного итератора до другого (например, различие между двумя указателями)
X::size_type	Беззнаковый целочисленный тип size_type может представлять размеры объектов данных, количество элементов и списки индексов массивов

Служебное слово typedef в определении класса используется для определения этих членов. Эти типы можно применять для объявления подходящих переменных. Например, в следующем фрагменте кода используется громоздкий способ замены в

векторе объектов string первого обнаруженного слова "bonus" словом "bogus", чтобы продемонстрировать возможность использования типов членов для объявления переменных:

```
vector<string> input;
string temp;
while (cin >> temp && temp != "quit")
   input.push_back(temp);
vector<string>::iterator want=
   find(input.begin(), input.end(), string("bonus"));
if (want != input.end())
{
   vector<string>::reference r = *want;
   r = "bogus";
}
```

В этом коде r становится ссылкой на элемент в input, на который указывает want. Аналогично, продолжая предыдущий пример, можно записать следующий код:

```
vector<string>::value_type s1 = input[0]; // s1 имеет тип string
vector<string>::reference s2 = input[1];  // s2 имеет тип string &
```

В результате создается s1 в качестве нового объекта string, представляющего собой копию input[0], и s2 в качестве ссылки на input[1]. Этот пример можно упростить, если, конечно, вы уже знаете, что шаблон основан на типе string:

```
string s1 = input[0];   // s1 имеет тип string
string & s2 = input[1]; // s2 имеет тип string &
```

Более сложные типы из табл. Ж.1 можно использовать в более специфичном коде, в котором тип контейнера и элемент являются обобщенными. Предположим, например, что вам необходима функция min(), которая принимает ссылку на контейнер в качестве своего аргумента и возвращает наименьший элемент в контейнере. Здесь предполагается, что операция < определена для типа значения, применяемого для реализации шаблона, и что вы не будете использовать алгоритм min_element() библиотеки STL, который использует интерфейс итератора. Поскольку в качестве аргумента может выступать vector<int>, list<string> или deque<double>, то для представления контейнера вы используете шаблон с шаблонным параметром, например, Bag. (Другими словами, Bag является шаблонным типом, который можно реализовать как vector<int>, list<string> или как некоторый другой тип контейнера.) Таким образом, типом аргумента для функции является const Bag & b. А что можно сказать о возвращаемом типе? Это должен быть тип значения для контейнера, то есть Bag::value_type. Однако, на данный момент, Bag является просто шаблонным параметром и компилятор не имеет возможности узнать о том, что член value_type на самом деле является типом. С помощью служебного слова typename можно показать, что членом класса является typedef:

```
vector<string>::value_type st;    // vector<string> - определенный класс
typename Bag::value_type m;       // Bag — пока еще не определенный тип
```

Здесь в первом определении компилятор получает доступ к шаблонному определению vector, в котором говорится, что value_type — это typedef. Во втором определении служебное слово typename гарантирует, что каким бы ни был Bag, комбинация Bag::value_type является именем типа. Эти рассуждения приводят к следующему определению:

```
template<typename Bag>
typename Bag::value_type min(const Bag & b)
{
   typename Bag::const_iterator it;
   typename Bag::value_type m = *b.begin();
   for (it = b.begin(); it != b.end(); ++it)
      if (*it < m)
         m = *it;
      return m;
}
```

Эту шаблонную функцию впоследствии можно было бы применять следующим образом:

```
vector<int> temperatures;
// ввод значений температуры в вектор
int coldest = min(temperatures);
```

Параметр `temperatures` может привести к тому, что `Bag` будет интерпретироваться как `vector<int>`, а `typename Bag::value_type` — как `vector<int>::value_type`, который, в свою очередь, имеет тип `int`.

Все контейнеры также содержат функции-члены или операции, перечисленные в табл. Ж.2. В этой таблице `X` — это тип контейнера, например, `vector<int>`, а `T` — это тип, хранящийся в контейнере, подобный `int`. Также `a` и `b` являются значениями типа `X`.

Таблица Ж.2. Методы, определенные для всех контейнеров

Метод/Операция	Описание
`begin()`	Возвращает итератор на первый элемент.
`End()`	Возвращает итератор на элемент, следующий за последним элементом контейнера.
`rbegin()`	Возвращает обратный итератор на элемент, следующий за последним элементом контейнера.
`rend()`	Возвращает обратный итератор на первый элемент.
`size()`	Возвращает количество элементов.
`maxsize()`	Возвращает размер наибольший возможный контейнер.
`empty()`	Возвращает `true`, если контейнер пуст.
`swap()`	Обмен содержимого двух контейнеров .
`==`	Возвращает `true`, если размер двух контейнеров одинаковый, и они имеют одинаковую упорядоченность элементов.
`!=`	`a != b` возвращает `!(a == b)`.
`<`	`a < b` возвращает `true`, если `a` лексикографически предшествует `b`.
`>`	`a > b` возвращает `b < a`.
`<=`	`a <= b` возвращает `!(a > b)`.
`>=`	`a >= b` возвращает `!(a < b)`.

Операция > для контейнера предполагает, что операция > определяется для типа значения. Лексикографическое сравнение представляет собой обобщенный вариант алфавитной сортировки. В нем сопоставляются элементы двух контейнеров с целью нахождения элемента в одном контейнере, не равного соответствующему элементу в другом контейнере. В этом случае считается, что контейнеры имеют тот же порядок, что и у несоответствующих пар элементов. Например, если в двух контейнерах первые десять элементов идентичны, а одиннадцатый в первом контейнере меньше, чем одиннадцатый элемент во втором контейнере, то считается, что первый контейнер предшествует второму. Если два контейнера имеют разную длину, и элементы в более коротком контейнере будут такими же, как и соответствующие элементы во втором контейнере, то контейнер с меньшей длиной будет предшествовать контейнеру с большей длиной.

Дополнительные члены для векторов, списков и двусторонних очередей

Векторы, списки и двусторонние очереди — все они представляют собой последовательности, и все имеют методы, перечисленные в табл. Ж.3. В этой таблице X — это тип контейнера, например, vector<int>, T — это тип, хранящийся в контейнере, вроде int, a — значение типа X, t — значение типа X::value_type, i и j — входные итераторы, q2 и p — итераторы, q и q1 — разыменовываемые итераторы (к ним можно применять операцию *),и n — это целое число, имеющее тип X::size_type.

Таблица Ж.3. Методы, определенные для векторов, списков и двусторонних очередей

Метод	Описание
a.insert(p,t)	Вставляет копию t перед p; возвращает итератор, указывающий на вставленную копию t. Значением по умолчанию для t является T(), то есть значение, используемое для типа T при отсутствии явной инициализации.
a.insert(p,n,t)	Вставляет n копий t перед p; не возвращает значение.
a.insert(p,I,j)	Вставляет копии элементов в диапазоне [i, j) перед p; не возвращает значение.
a.resize(n,t)	Если выполняется условие n > a.size(), то вставляет n - a.size() копий t перед a.end(); t имеет значение по умолчанию T(), то есть значение, используемое для типа T при отсутствии явной инициализации. Если выполняется условие n < a.size(), то элементы, следующие за n-м элементом, удаляются.
a.assign(I,j)	Заменяет текущее содержимое копиями элементов в диапазоне [i, j).
a.assign(n,t)	Заменяет текущее содержимое n копиями t. Значением по умолчанию для t является T(), то есть значение, используемое для типа T при отсутствии явной инициализации.
a.erase(q)	Удаляет элемент, на который указывает q; возвращает итератор на элемент, который следует за q.

Окончание табл. Ж.3

Метод	Описание
a.erase(q1,q2)	Удаляет элементы в диапазоне [q1, q2); возвращает итератор, указывающий на элемент, на который изначально указывал q2.
a.clear()	Выполняет то же самое, что и erase(a.begin(), a.end()).
a.front()	Возвращает *a.begin() (первый элемент).
a.back()	Возвращает *--a.end() (последний элемент).
a.push_back(t)	Вставляет t перед a.end().
a.pop_back()	Удаляет последний элемент.

В табл. Ж.4 описаны методы, общие для двух из трех классов vector, list и deque.

Таблица Ж.4. Методы, определенные для некоторых последовательностей

Метод	Описание	Контейнер
a.push_front(t)	Вставляет копию t перед первым элементом.	list, deque
a.pop_front()	Удаляет первый элемент.	list, deque
a[n]	Возвращает *(a.begin() + n).	vector, deque
a.at(n)	Возвращает *(a.begin() + n); генерирует исключение out_of_range, если n > a.size().	vector, deque

Шаблон vector дополнительно имеет методы, перечисленные в табл. Ж.5. Здесь a соответствует контейнеру vector, а n является целым числом, имеющим тип X::size_type.

Таблица Ж.5. Дополнительные методы для векторов

Метод	Описание
a.capacity()	Возвращает общее количество элементов, которые может вмещать вектор, не требуя повторного размещения.
a.reserve(n)	Сигнализирует объекту a о том, что необходима память как минимум для n элементов. После вызова метода вместимость будет измеряться как минимум n элементами. Повторное размещение происходит в тех случаях, когда n больше текущей вместимости. Если n больше чем a.max_size(), метод генерирует исключение length_error.

Шаблон list дополнительно имеет методы, представленные в табл. Ж.6. В ней a и b — это контейнеры list, T — это тип, хранящийся в списке, например, int, t — значение, имеющее тип T, i и j — входные итераторы, q2 и p — итераторы, q и q1 — разыменовываемые итераторы, n — это целое число, имеющее тип X::size_type. В таблице используется стандартное обозначение STL [i, j), которое обозначает диапазон от i до j, включая i и не включая j.

Таблица Ж.6. Дополнительные методы для списков

Метод	Описание
`a.splice(p,b)`	Перемещает содержимое списка `b` в список `a`, вставляя его перед `p`.
`a.splice(p,b,i)`	Перемещает элемент в список `b`, на который указывает `i`, перед позицией `p` в списке `a`.
`a.splice(p,b,i,j)`	Перемещает элементы в диапазоне [`i`, `j`) списка `b` в список `a` перед позицией `p`.
`a.remove(const T& t)`	Удаляет все элементы из списка `a`, которые имеют значение `t`.
`a.remove_if(Predicate pred)`	При условии, что `i` является итератором, указывающим на список `a`, удаляет все значения, для которых `pred(*i)` равно `true`. (`Predicate` — это булевская функция или функциональный объект, рассмотренный в главе 15.)
`a.unique()`	Удаляет все элементы, кроме первого, из каждой группы последовательных равных элементов.
`a.unique(BinaryPredicate bin_pred)`	Удаляет все элементы, кроме первого, из каждой группы последовательных элементов, для которых `bin_pred(*i, *(i - 1))` равно `true`. (`BinaryPredicate` — это булевская функция или функциональный объект, рассмотренный в главе 15.)
`a.merge(b)`	Объединяет содержимое списка `b` и содержимое списка `a`, используя операцию <, определенную для типа значения. Если элемент в списке `a` эквивалентен элементу в списке `b`, то элемент списка `a` ставится первым. После объединения список `b` оказывается пустым.
`a.merge(b, Compare comp)`	Объединяет содержимое списка `b` и содержимое списка `a`, используя функцию `comp` или функциональный объект. Если элемент в списке `a` эквивалентен элементу в списке `b`, то элемент списка `a` ставится первым. После объединения список `b` оказывается пустым.
`a.sort()`	Сортирует список `a` с использованием операции <.
`a.sort(Compare comp)`	Сортирует список `a`, используя функцию `comp` или функциональный объект.
`a.reverse()`	Изменяет порядок элементов в списке `a` на противоположный.

Дополнительные члены для множеств и таблиц

Ассоциативные контейнеры, моделями которых являются таблицы и множества, имеют шаблонные параметры `Key` и `Compare`, которые показывают, соответственно, тип ключа, используемого для упорядочения содержимого, и функциональный объект, называемый *объектом сравнения*, который применяется для сравнения значений

ключа. Контейнеры set и multiset хранят ключи как значения, поэтому ключ имеет тот же тип, что и значение. В контейнерах map и multimap хранимые значения одного типа (шаблонный параметр T) связаны с типом ключа (шаблонный параметр Key), а типом значения является pair<const Key, T>. Ассоциативные контейнеры имеют дополнительные члены для описания этих особенностей, как показано в табл. Ж.7.

Таблица Ж.7. Типы, определенные для ассоциативных контейнеров

Тип	Значение
X::key_type	Key, тип ключа.
X::key_compare	Compare, которое имеет значение по умолчанию less<key_type>.
X::value_compare	Тип бинарного предиката, такой же, как и key_compare для set и multuset; он определяет порядок значений pair<const Key, T> в контейнере map или multimap.
X::mapped_type	T, тип ассоциативных данных (только map и multimap).

Ассоциативные контейнеры предлагают методы, перечисленные в табл. Ж.8. В общем случае для объекта сравнения не обязательно, чтобы значения с одним и тем же ключом были идентичными друг другу; термин *эквивалентные ключи* означает, что два значения, которые могут или не могут быть идентичными, имеют один и то же ключ. В этой таблице X — это класс контейнера, a — объект, имеющий тип X. Если X использует несколько ключей (то есть является multiset или multimap), то a_eq представляет собой объект, имеющий тип X. Как обычно, i и j — входные итераторы, ссылающиеся на элементы value_type, [i, j) — допустимый диапазон, p и q2 — итераторы по a, q и q1 — разыменовываемые итераторы по a, [q1, q2) — допустимый диапазон, t — это значение X::value_type (это может быть пара значений), а k — значение X::key_type.

Таблица Ж.8. Методы, определенные для множеств, мультимножеств, карт и мультикарт

Метод	Описание
a.key_comp()	Возвращает объект сравнения, используемый при создании a.
a.value_comp()	Возвращает объект value_compare_type.
a_uniq.insert(t)	Вставляет значение t в контейнер a, если и только если a еще не содержит значения с эквивалентным ключом. Метод возвращает значение, имеющее тип pair<iterator,bool>. Компонент bool имеет значение true, если вставка была осуществлена, в противном случае — false. Компонент итератора указывает на элемент, чей ключ эквивалентен ключу t.
a_eq.insert(t)	Вставляет t и возвращает итератор, указывающий на эту позицию.
a.insert(p,t)	Вставляет t, используя p в качестве подсказки, где функция insert() должна начинать поиск. Если a является контейнером с уникальными ключами, то вставка будет произведена только в том случае, если a не будет содержать элемент с эквивалентным ключом; в противном случае будет произведена вставка. Вне зависимости от того, производится вставка или нет, метод возвращает итератор, указывающий на позицию с эквивалентным ключом.

Окончание табл. Ж.8

Метод	Описание
`a.insert(i,j)`	Вставляет элемент из диапазона [`i`, `j`) в `a`.
`a.erase(k)`	Удаляет все элементы из `a`, чьи ключи эквивалентны `k`, и возвращает количество удаленных элементов.
`a.erase(q)`	Удаляет элемент, на который указывает `q`.
`a.erase(q1,q2)`	Удаляет элементы из диапазона [`q1`, `q2`).
`a.clear()`	Выполняет то же самое, что и `erase(a.begin(), a.end())`.
`a.find(k)`	Возвращает итератор, указывающий на элемент, чей ключ эквивалентен `k`; возвращает `a.end()`, если таковой элемент не будет найден.
`a.count(k)`	Возвращает количество элементов, имеющих ключи, эквивалентные `k`.
`a.lower_bound(k)`	Возвращает итератор на первый элемент, ключ которого не меньше `k`.
`a.upper_bound(k)`	Возвращает итератор на первый элемент, ключ которого больше `k`.
`a.equal_range(k)`	Возвращает пару, чей первый член — `a.lower_bound(k)`, а второй член — `a.upper_bound(k)`.
`a.operator[](k)`	Возвращает ссылку на значение, связанное с ключом `k` (только для контейнеров `map`).

Функции библиотеки STL

Библиотека алгоритмов STL, которая поддерживается заголовочными файлами `algorithm` и `numeric`, предлагает большое количество шаблонных функций, не являющихся членами, на основе итераторов. Как было сказано в главе 16, имена шаблонных параметров подбираются таким образом, чтобы показать, что должны выполнять определенные параметры. Например, `ForwardIterator` применяется, чтобы показать, что параметр должен, как минимум, моделировать требования однонаправленного итератора, а `Predicate` используется, чтобы показать параметр, который должен быть функциональным объектом с одним аргументом и возвращаемым значением `bool`. В стандартной версии C++ алгоритмы разделены на четыре группы: операции, не изменяющие последовательности, операции, видоизменяющие последовательности, операции сортировки и связанные с ней операции, а также операции с числами. Термин *последовательная операция* говорит о том, что функция принимает пару итераторов в качестве аргументов, чтобы определить диапазон, или последовательность, в которой будут производиться действия. Термин *видоизменяющий* означает, что функция может видоизменять контейнер.

Операции, не изменяющие последовательности

В табл. Ж.9 перечислены операции, не изменяющие последовательности. Аргументы в этой таблице не показаны, а перегруженные функции представлены только один раз. За таблицей следует более подробное их описание, включая прототипы. Таким образом, вы можете просмотреть таблицу, чтобы узнать, что выполняет определенная функция, и затем ознакомиться с детальной информацией о ней.

Таблица Ж.9. Операции, не изменяющие последовательности

Функция	Описание
`for_each()`	Применяет не изменяющий функциональный объект к каждому элементу диапазона.
`find()`	Находит первое появление значения в диапазоне.
`find_if()`	Находит первое значение, удовлетворяющее предикатному критерию проверки в диапазоне.
`find_end()`	Находит последнее появление подпоследовательности, значения которой совпадают со значениями второй последовательности. Совпадение может определяться равенством или бинарным предикатом.
`find_first_of()`	Находит первое появление любого элемента второй последовательности, который совпадает со значением в первой последовательности. Совпадение может определяться равенством или бинарным предикатом.
`adjacent_find()`	Находит первый элемент, который совпадает с соседним элементом. Совпадение может определяться равенством или бинарным предикатом.
`count()`	Возвращает число, соответствующее тому, сколько раз данное значение было обнаружено в диапазоне.
`count_if()`	Возвращает число, соответствующее тому, сколько раз данное значение было обнаружено в диапазоне, при этом совпадение определяется бинарным предикатом.
`mismatch()`	Находит первый элемент в одном диапазоне, который не совпадает с соответствующим элементом во втором диапазоне, и возвращает итераторы по каждому из них. Совпадение определяться равенством или бинарным предикатом.
`equal()`	Возвращает `true`, если каждый элемент в одном диапазоне совпадает с соответствующим элементом во втором диапазоне. Совпадение может определяться равенством или бинарным предикатом.
`search()`	Находит первое появление подпоследовательности, значения которой совпадают со значениями второй последовательности. Совпадение может определяться равенством или бинарным предикатом.
`search_n()`	Находит первую подпоследовательность `n` элементов, которые совпадают с данным диапазоном. Совпадение может определяться равенством или бинарным предикатом.

Теперь давайте рассмотрим более подробно операции, не изменяющие последовательности. Для каждой функции показан прототип (прототипы) и приводится краткое пояснение. Пары итераторов указывают на диапазоны, с выбранным именем шаблонного параметра, указывающего на тип итератора. Как обычно, диапазон определяется в виде `[first, last)`, включая `first` и не включая `last`. Некоторые функции принимают два диапазона, которые не могут находиться в одном и том же контейнере. Например, `equal()` можно применять для сравнения списка с вектором. Функции, передаваемые в виде аргументов, являются функциональными объектами, которые могут быть указателями (примером которых являются имена функций) или объектами, для которых определена операция `()`. Как и в главе 16, предикат представляет собой булевскую функцию с одним аргументом, а бинарным предикатом

является булевская функция с двумя аргументами. (Функции не обязательно должны иметь тип `bool`, поскольку они возвращают значение `0`, соответствующее значению `false`, и ненулевое значение, соответствующее значению `true`.)

for_each()

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

Функция `for_each()` применяет функциональный объект `f` к каждому элементу в диапазоне [`first`, `last`). Она также возвращает `f`.

find()

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

Функция `find()` возвращает итератор на первый элемент в диапазоне [`first`, `last`), который имеет значение `value`; если элемент не будет найден, она возвращает `last`.

find_if()

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
```

Функция `find_if()` возвращает итератор `it` на первый элемент в диапазоне [`first`, `last`), для которого результат вызова функционального объекта `pred(*i)` равен `true`; если элемент не будет найден, функция возвращает `last`.

find_end()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2,
   class BinaryPredicate>
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);
```

Функция `find_end()` возвращает итератор `it` на последний элемент в диапазоне [`first1`, `last1`), который отмечает начало подпоследовательности, совпадающей с содержимым диапазона [`first2`, `last2`). В первой версии при сопоставлении элементов используется операция `==`, определенная для конкретного типа значения. Во второй версии для сопоставления элементов применяется бинарный предикатный функциональный объект `pred`. То есть элементы, на которые указывает `it1` и `it2`, совпадают, если результат `pred(*it1, *it2)` равен `true`. Обе версии возвращают `last1`, если элемент не был найден.

find_first_of()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of(
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2);
```

```
template<class ForwardIterator1, class ForwardIterator2,
   class BinaryPredicate>
ForwardIterator1 find_first_of(
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2,
   BinaryPredicate pred);
```

Функция `find_first()` возвращает итератор `it` на первый элемент в диапазоне [`first1`, `last1`), который совпадает с любым элементом в диапазоне [`first2`, `last2`). В первой версии для сопоставления элементов используется операция == для типа значения. Во второй версии для сопоставления элементов применяется бинарный предикатный функциональный объект `pred`. То есть элементы, на которые указывает `it1` и `it2`, совпадают, если результат `pred(*it1, *it2)` равен `true`. Обе версии возвращают `last1`, если элемент не был найден.

adjacent_find()

```
template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
   BinaryPredicate pred);
```

Функция `adjacent_find()` возвращает итератор `it` на первый элемент в диапазоне [`first1`, `last1`), при условии, что элемент совпадает со следующим элементом. Функция возвращает `last`, если такая пара элементов не найдена. В первой версии для сопоставления элементов используется операция == для типа значения. Во второй версии для сопоставления элементов применяется бинарный предикатный функциональный объект `pred`. То есть элементы, на которые указывает `it1` и `it2`, совпадают, если результат `pred(*it1, *it2)` равен `true`.

count()

```
template<class InputIterator, class T>
iterator_traits<InputIterator>::difference_type count(
   InputIterator first, InputIterator last, const T& value);
```

Функция `count()` возвращает количество элементов в диапазоне [`first`, `last`), которые совпадают со значением `value`. Для сопоставления элементов используется операция == для типа значения. Возвращаемым типом является целочисленный тип, способный уместить максимальное количество элементов, которые может содержать контейнер.

count_if()

```
template<class InputIterator, class Predicate>
iterator_traits<InputIterator>::difference_type count_if(
   InputIterator first, InputIterator last, Predicate pred);
```

Функция `count_if()` возвращает количество элементов в диапазоне [`first`, `last`), для которых функциональный объект `pred` возвращает значение `true` при передаче элемента в качестве аргумента.

mismatch()

```
template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
   InputIterator1 last1, InputIterator2 first2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
   InputIterator1 last1, InputIterator2 first2, BinaryPredicate pred);
```

Каждая из функций mismatch() находит первый элемент в диапазоне [first1, last1), который не совпадает с соответствующим элементом в диапазоне, начинающемся с first2, и возвращает пару, содержащую итераторы на два несоответствующих элемента. Если не будет найдено ни одного несовпадения, возвращаемым значением будет pair<last1, first2 + (last1 — first1)>. В первой версии для сопоставления элементов используется операция ==, определенная для конкретного типа значения. Во второй версии для сопоставления элементов применяется бинарный предикатный функциональный объект pred. То есть элементы, на которые указывают it1 и it2, не совпадают, если результат pred(*it1, *it2) равен false.

equal()

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, BinaryPredicate pred);
```

Функция equal() возвращает значение true, если каждый элемент в диапазоне [first1, last1) совпадает с соответствующим элементом в последовательности, начинающейся с first2; в противном случае функция возвращает false. В первой версии для сопоставления элементов используется операция == для типа значения. Во второй версии для сопоставления элементов применяется бинарный предикатный функциональный объект pred. То есть элементы, на которые указывает it1 и it2, совпадают, если результат pred(*it1, *it2) равен true.

search()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2,
   class BinaryPredicate>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
  ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);
```

Функция search() находит первое появление в диапазоне [first1, last1), которое совпадает с соответствующей последовательностью, найденной в диапазоне [first2, last2). Функция возвращает last1, если такая последовательность не найдена. В первой версии для сопоставления элементов используется операция == для типа значения. Во второй версии для сопоставления элементов применяется бинарный предикатный функциональный объект pred. То есть элементы, на которые указывает it1 и it2, совпадают, если результат pred(*it1, *it2) равен true.

search_n()

```
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
    Size count, const T& value);

template<class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
    Size count, const T& value, BinaryPredicate pred);
```

Функция search()_n находит первое появление в диапазоне [firts1, last1), которое совпадает с последовательностью, состоящей из count последовательных появлений значения value. Функция возвращает last1, если такая последовательность не найдена. В первой версии для сопоставления элементов используется операция == для типа значения. Во второй версии для сопоставления элементов применяется бинарный предикатный функциональный объект pred. То есть элементы, на которые указывает it1 и it2, совпадают, если результат pred(*it1, *it2) равен true.

Операции, видоизменяющие последовательности

В табл. Ж.10 перечислены операции, видоизменяющие последовательности. Аргументы в этой таблице не показаны, а перегруженные функции показаны только один раз. За таблицей следует более подробное их описание, включая прототипы. Таким образом, вы можете просмотреть таблицу, чтобы узнать, что выполняет определенная функция, после чего обратиться к детальной информации о ней

Таблица Ж.10. Операции, видоизменяющие последовательности

Функция	Описание
copy()	Копирует элементы из диапазона в позицию, определяемую итератором.
copy_backward()	Копирует элементы из диапазона в позицию, определяемую итератором. Копирование начинается с конца диапазона и продолжается в обратном порядке.
swap()	Обмен значений, которые хранятся в позициях, определяемых ссылками.
swap_ranges()	Обмен соответствующих значений в двух диапазонах.
iter_swap()	Обмен двух значений, хранящихся в позициях, определяемых итераторами.
transform()	Применяет функциональный объект к каждому элементу в диапазоне (или к каждой паре элементов в паре диапазонов) и копирует возвращаемое значение в соответствующую позицию в другом диапазоне.
replace()	Заменяет каждое появление значения в диапазоне другим значением.
replace_if()	Заменяет каждое появление значения в диапазоне другим значением, если результат предикатного функционального объекта, примененного к исходному значению, равен true.
replace_copy()	Копирует один диапазон в другой и заменяет каждое появление определенного значения другим значением.
replace_copy_if()	Копирует один диапазон в другой и заменяет каждое значение, для которого результат предикатного функционального объекта дает true, указанным значением.
fill()	Заполняет диапазон указанным значением.

Окончание табл. Ж.10

Функция	Описание
`fill_n()`	Присваивает указанное значение n последовательным элементам.
`generate()`	Присваивает каждому значению в диапазоне возвращаемое значение генератора, представляющего собой функциональный объект, не принимающий аргументов.
`generate_n()`	Присваивает первым `n` значениям в диапазоне возвращаемое значение генератора, представляющего собой функциональный объект, не принимающий аргументов.
`remove()`	Удаляет все появления указанного значения из диапазона и возвращает для результирующего диапазона итератор на элемент, следующий за последним элементом.
`remove_if()`	Удаляет все появления значений, для которых предикатный объект возвращает `true` из диапазона, и возвращает для результирующего диапазона итератор на элемент, следующий за последним элементом.
`remove_copy()`	Копирует элементы из одного диапазона в другой, опуская элементы, равные определенному значению.
`remove_copy_if()`	Копирует элементы из одного диапазона в другой, опуская элементы, для которых предикатный функциональный объект возвращает `true`.
`unique()`	Сокращает каждую последовательность из двух или более эквивалентных элементов в диапазоне до одного элемента.
`unique_copy()`	Копирует элементы из одного диапазона в другой, сокращая каждую последовательность из двух или более эквивалентных элементов в диапазоне до одного элемента.
`reverse()`	Изменяет порядок элементов в диапазоне на противоположный.
`reverse_copy()`	Копирует диапазон в обратном порядке в другой диапазон.
`rotate()`	Интерпретирует диапазон как циклическое упорядочение и циклически сдвигает элементы влево.
`rotate_copy()`	Копирует один диапазон в другой в циклическом порядке.
`random_shuffle()`	Случайным образом перераспределяет элементы в диапазоне.
`partition()`	Помещает все элементы, удовлетворяющие предикатному функциональному объекту, перед всеми элементами, не удовлетворяющими этой функции.
`stable_partition()`	Помещает все элементы, удовлетворяющие предикатному функциональному объекту, перед элементами, не удовлетворяющими этой функции. Сохраняет относительный порядок значений в каждой группе.

Теперь давайте рассмотрим эти операции более подробно. Для каждой функции показан прототип (прототипы) и приводится краткое пояснение. Пары итераторов указывают на диапазоны, с выбранным именем шаблонного параметра, указывающего на тип итератора. Как обычно, диапазон определяется в виде `[first, last)`, включая `first` и не включая `last`. Функции, передаваемые в виде аргументов, являются функциональными объектами, могущие быть указателями или объектами, для которых определена операция `()`. Как и в главе 16, предикатом является булевская

функция с одним аргументом, а бинарным предикатом — булевская функция с двумя аргументами. (Функции не обязательно должны иметь тип bool, поскольку они возвращают значение 0, соответствующее значению false, и ненулевое значение, соответствующее значению true.) Так же, как и в главе 16, унарной функциональным объектом является функция, принимающая один аргумент, а бинарным функциональным объектом — функция, принимающая два аргумента.

copy()

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
   OutputIterator result);
```

Функция copy() копирует элементы из диапазона [first, last) в диапазон [result, result + (last - first)). Функция возвращает результат result + (last - first), то есть итератор, указывающий на позицию, следующую за последней позицией, в которую был скопирован элемент. Функция требует, чтобы результат result не находился в диапазоне [first, last), то есть чтобы искомый диапазон не перекрывал исходный.

copy_backward()

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
   BidirectionalIterator1 last, BidirectionalIterator2 result);
```

Функция copy_backward() копирует элементы из диапазона [first, last) в диапазон [result -(result - result), result). Копирование начинается с элемента в позиции last - 1, который копируется в позицию result - 1, и продолжается в обратном порядке до first. Функция возвращает result - (last - first), то есть итератор, указывающий на позицию, следующую за последней позицией, в которую был скопирован элемент. Функция требует, чтобы result не находился в диапазоне [first, last). Однако поскольку копирование осуществляется в обратном порядке, возможно перекрытие исходного и искомого диапазонов.

swap()

```
template<class T> void swap(T& a, T& b);
```

Функция swap() осуществляет обмен значений, которые хранятся в двух позициях, определяемых ссылками.

swap_ranges()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1,
   ForwardIterator1 last1, ForwardIterator2 first2);
```

Функция swap_ranges() осуществляет обмен значений из диапазона [first1, last1) соответствующими значениями из диапазона, начинающегося с first2. Два диапазона не должны перекрывать друг друга.

```
template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

Функция iter_swap() выполняет обмен значений, хранящихся в двух позициях, определяемых итераторами.

transform()

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
   OutputIterator result, UnaryOperation op);

template<class InputIterator1, class InputIterator2,
   class OutputIterator, class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, OutputIterator result, BinaryOperation binary_op);
```

Первая версия функции transform() применяет унарный функциональный объект op к каждому элементу в диапазоне [first, last) и присваивает возвращаемое значение соответствующему элементу в диапазоне, начиная с result. Таким образом, result присваивается результат op(*first) и так далее. Для искомого диапазона функция возвращает result + (last - first), то есть значение, следующее за последним значением.

Вторая версия функции transform() применяет бинарный функциональный объект op к каждому элементу в диапазоне [first1, last1) и к каждому элементу в диапазоне [first2, last2), и присваивает возвращаемое значение соответствующему элементу в диапазоне, начиная с result. Таким образом, result присваивается результат op(*first1, *first2) и так далее. Для искомого диапазона функция возвращает result + (last - first), то есть значение, следующее за последним значением.

replace()

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
   const T& old_value, const T& new_value);
```

Функция replace() заменяет каждое появление значения old_value в диапазоне [first, last) значением new_value.

replace_if()

```
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
   Predicate pred, const T& new_value);
```

Функция replace_if() заменяет каждое значение old в диапазоне [first, last), для которого результат pred(old) равен true, значением new_value.

replace_copy()

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
   OutputIterator result,const T& old_ value, const T& new_ value);
```

Функция replace_copy() копирует элементы из диапазона [first, last) в диапазон, начинающийся с result, и заменяя каждое значение old_value значением

new_value. Для искомого диапазона функция возвращает результат result + (last - first), то есть значение, следующее за последним значением.

replace_copy_if()

```
template<class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
   OutputIterator result, Predicate pred, const T& new_ value);
```

Функция replace_copy_if() копирует элементы из диапазона [first, last) в диапазон, начинающийся с result, заменяя каждое значение old, для которого результат pred(old) равен true, значением new_value. Для искомого диапазона функция возвращает result + (last - first), то есть значение, следующее за последним значением.

fill()

```
template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

Функция fill() присваивает каждому элементу в диапазоне [first, last) значение value.

fill_n()

```
template<class OutputIterator, class Size, class T>
void fill_n(OutputIterator first, Size n, const T& value);
```

Функция fill_n() присваивает n первым элементам, начиная с позиции first, значение value.

generate()

```
template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

Функция generate() присваивает каждому элементу в диапазоне [first, last) значение gen(), где gen — функциональный объект генератора, то есть функция, не принимающая аргументов. Например, gen может быть указателем на функцию rand().

generate_n()

```
template<class OutputIterator, class Size, class Generator>
void generate_n(OutputIterator first, Size n, Generator gen);
```

Функция generate_n() присваивает первым n элементам в диапазоне, начинающемся с first, значение gen(), где gen — функциональный объект генератора, то есть функция, не принимающая аргументов. Например, gen может быть указателем на функцию rand().

remove()

```
template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first,
   ForwardIterator last, const T& value);
```

Функция remove() удаляет все появления значения value в диапазоне [first, last) и возвращает для результирующего диапазона итератор на элемент, следующий за последним элементом. Эта функция является устойчивой, то есть порядок не удаляемых элементов остается неизмененным.

На заметку!

Поскольку различные функции remove() и unique() не являются функциями-членами и, также, поскольку они не ограничены применением только к контейнерам библиотеки STL, они не могут переустанавливать размер контейнера. Наоборот, они возвращают итератор, который указывает на новую позицию, следующую за последней позицией. Обычно удаленные элементы просто смещаются в конец контейнера. Однако для контейнеров библиотеки STL можно использовать возвращаемый итератор и один из методов erase() для сброса end().

remove_if()

```
template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first,
   ForwardIterator last, Predicate pred);
```

Функция remove_if() удаляет все появления значений val, для которых результат pred(val) равен true, из диапазона [first, last) и возвращает для результирующего диапазона итератор, указывающий на элемент, следующий за последним элементом. Функция является устойчивой, то есть порядок не удаляемых элементов остается неизмененным.

remove_copy()

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
   OutputIterator result, const T& value);
```

Функция remove_copy() копирует значения из диапазона [first, last) в диапазон, начинающийся с result, пропуская при копировании экземпляры value. Функция возвращает для результирующего диапазона итератор, указывающий на элемент, следующий за последним элементом. Функция является устойчивой, то есть порядок не удаляемых элементов остается неизмененным.

remove_copy_if()

```
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
   OutputIterator result, Predicate pred);
```

Функция remove_copy_if() копирует значения из диапазона [first, last) в диапазон, начинающийся с result, пропуская экземпляры val, для которых результат pred(val) при копировании равен true. Функция возвращает для результирующего диапазона итератор, указывающий на элемент, следующий за последним элементом. Функция является устойчивой, то есть порядок не удаляемых элементов остается неизмененным.

unique()

```
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);
```

```
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
   BinaryPredicate pred);
```

Функция `unique()` сокращает каждую последовательность из двух или более эквивалентных элементов в диапазоне `[first, last)` до одного элемента, и для нового диапазона возвращает итератор, указывающий на элемент, следующий за последним элементом. В первой версии для сопоставления элементов используется операция == для типа значения. Во второй версии для сопоставления элементов применяется бинарный предикатный функциональный объект `pred`. То есть элементы, на которые указывает `it1` и `it2`, совпадают, если результат `pred(*it1, *it2)` равен `true`.

unique_copy()

```
template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
   OutputIterator result);
```

```
template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
   OutputIterator result, BinaryPredicate pred);
```

Функция `unique_copy()` копирует элементы из диапазона `[first, last)` в диапазон, начинающийся с `result`, сокращая каждую последовательность из двух или более идентичных элементов до одного элемента. Для нового диапазона функция возвращает итератор, указывающий на элемент, следующий за последним элементом. В первой версии для сопоставления элементов используется операция ==, определенная для типа значения. Во второй версии для сопоставления элементов применяется бинарный предикатный функциональный объект `pred`. То есть элементы, на которые указывает `it1` и `it2`, совпадают, если результат `pred(*it1, *it2)` равен `true`.

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

Функция `reverse()` изменяет порядок элементов на противоположный в диапазоне `[first, last)`, вызывая `swap(first, last - 1)`, и так далее.

reverse_copy()

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
   BidirectionalIterator last, OutputIterator result);
```

Функция `reverse_copy()` копирует элементы из диапазона `[first, last)` в диапазон, начинающийся с `result`, в обратном порядке. Два диапазона не должны перекрывать друг друга.

rotate()

```
template<class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
   ForwardIterator last);
```

Функция `rotate()` циклически сдвигает элементы влево в диапазоне `[first, last)`. Элемент в позиции `middle` перемещается в позицию `first`, а элемент в позиции `middle + 1` перемещается в позицию `first + 1` и так далее. Элементы, предше-

ствующие middle, циклически проходят через конец контейнера, поэтому элемент в позиции first будет следовать за элементом last - 1.

rotate_copy()

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first,
   ForwardIterator middle, ForwardIterator last, OutputIterator result);
```

Функция rotate_copy() копирует элементы из диапазона [first, last) в диапазон, начинающийся с result, используя циклически сдвинутую последовательность, описанную для функции rotate().

replace()

```
template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
```

Эта версия функции random_shuffle() перемешивает элементы в диапазоне [first, last). Распределение является однородным, то есть каждая возможная перетасовка исходного порядка является равновозможной.

random_shuffle()

```
template<class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first,
   RandomAccessIterator last, RandomNumberGenerator& random);
```

Эта версия функции random_shuffle() перемешивает элементы в диапазоне [first, last). Распределение определяется посредством функционального объекта random. Для n элементов выражение random(n) должно вернуть значение из диапазона [0, n).

partition()

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
   BidirectionalIterator last, Predicate pred);
```

Функция partition() помещает каждый элемент, чье значение val является таким, что результат pred(val) равен true, перед всеми элементами, не удовлетворяющими этому условию. Функция возвращает итератор на позицию, следующую за последней позицией, содержащей значение, для которого результат предикатного функционального объекта был равен true.

stable_partition()

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(BidirectionalIterator first,
   BidirectionalIterator last, Predicate pred);
```

Функция stable_partition() помещает каждый элемент, чье значение val является таким, что результат pred(val) равен true, перед всеми элементами, которые не удовлетворяют этому условию. Эта функция сохраняет относительную упорядоченность внутри каждой из двух групп. Функция возвращает итератор на позицию, следующую за последней позицией, содержащей значение, для которого результат предикатного функционального объекта был равен true.

Операции сортировки и связанные с ними операции

В табл. Ж.11 перечислены операции сортировки и связанные с ними операции. Аргументы в этой таблице не показаны, а перегруженные функции показаны только один раз. Каждая функция имеет версию, в которой используется операция < для упорядочения элементов, и версию, в которой для упорядочения элементов применяется функциональный объект сравнения. За таблицей следует более подробное их описание, включая прототипы. Таким образом, вы можете просмотреть таблицу, чтобы узнать, что выполняет определенная функция, а затем обратиться к детальной информации.

Таблица Ж.11. Операции сортировки и связанные с ними операции

Функция	Описание
`sort()`	Сортирует диапазон.
`stable_sort()`	Сортирует диапазон, сохраняя относительную упорядоченность эквивалентных элементов.
`partial_sort()`	Частично сортирует диапазон, при этом для первых `n` элементов производится полная сортировка.
`partial_sort_copy()`	Копирует частично отсортированный диапазон в другой диапазон.
`nth_element()`	Для представленного итератора на диапазон находит элемент, который мог быть здесь, если бы диапазон был отсортирован, и помещает сюда этот элемент.
`lower_bound()`	Для представленного значения находит первую позицию в отсортированном диапазоне, перед которым может быть вставлено значение, сохраняя прежнюю упорядоченность.
`upper_bound()`	Для представленного значения находит последнюю позицию в отсортированном диапазоне, перед которым может быть вставлено значение, сохраняя прежнюю упорядоченность.
`equal_range()`	Для представленного значения находит самый большой поддиапазон отсортированного диапазона, такой, в котором до любого элемента, не нарушая упорядоченность, может быть вставлено значение.
`binary_search()`	Возвращает `true`, если отсортированный диапазон содержит значение, эквивалентное данному значению; в противном случае возвращает `false`.
`merge()`	Объединяет два отсортированных диапазона в третий диапазон.
`inplace_merge()`	Объединяет два последовательно отсортированных диапазона, не создавая третий диапазон.
`includes()`	Возвращает `true`, если каждый элемент из одной последовательности можно найти в другой последовательности.
`set_union()`	Создает объединение двух последовательностей, то есть последовательность, содержащую все элементы, представленные в каждой из последовательностей.

Окончание табл. Ж.11

Функция	Описание
`set_intersection()`	Создает пересечение последовательностей, то есть последовательность, содержащую только те элементы, которые могут быть найдены в обеих последовательностях.
`set_difference()`	Создает разность двух последовательностей, то есть последовательность, содержащую только те элементы, которые могут быть найдены в первой последовательности, но не во второй.
`set_symmetric_difference()`	Создает последовательность, содержащую элементы, которые могут быть найдены в одной из двух последовательностей, но не в каждой из них.
`make_heap`	Преобразовывает диапазон в частично упорядоченное полное бинарное дерево.
`push_heap()`	Добавляет диапазон в частично упорядоченное полное бинарное дерево.
`pop_heap()`	Удаляет наибольший элемент в частично упорядоченном полном бинарном дереве.
`sort_heap()`	Сортирует частично упорядоченное полное бинарное дерево.
`min()`	Возвращает наименьшее из двух значений.
`max()`	Возвращает наибольшее из двух значений.
`min_element()`	Находит первое появление наименьшего значения в диапазоне.
`max_element()`	Находит появление наибольшего значения в диапазоне.
`lexicographic_compare()`	Выполняет лексикографическое сравнение двух последовательностей, возвращая `true`, если первая последовательность лексикографически меньше второй; в противном случае возвращает `false`.
`next_permutation()`	Генерирует следующую перестановку в последовательности.
`previous_permutation()`	Генерирует предыдущую перестановку в последовательности.

Функции, представленные в этом разделе, определяют порядок двух элементов с помощью операции <, определенной для элементов, или посредством объекта сравнения, определяемого шаблонным типом `Compare`. Если `comp` является объектом `Compare`, то `comp(a,b)` является обобщенной формой `a < b` и возвращает `true`, если `a` предшествует `b` в схеме упорядочения. Если результат `a < b` равен `false`, и `b < a` также равен `false`, то `a` и `b` эквивалентны друг другу. Объект сравнения должен обеспечивать как минимум *строгое квазиупорядочение*. Это упорядочение определяется следующими положениями:

- Выражение `comp(a,a)` должно иметь результат `false`, обобщая тот факт, что значение не может быть меньше, чем является на самом деле. (Это требование касательно строгости.)
- Если результат `comp(a,b)` равен `true`, и результат `comp(b,c)` также равен `true`, то результат `comp(a,c)` равен `true` (то есть сравнение является транзитивным отношением).

- Если элемент a эквивалентен b, а элемент b эквивалентен c, то элемент a эквивалентен c (то есть эквивалентность является транзитивным отношением).

Если операцию < выполнять над целыми числами, то под эквивалентностью будет подразумеваться равенство, однако для общих случаев это не всегда так. Например, вы можете определить структуру с несколькими членами, описывающими почтовые адреса, и определить объект сравнения comp, который будет упорядочивать структуры по почтовому индексу. Тогда любые два адреса, имеющие одинаковые почтовые индексы, могут быть эквивалентными, но не равными друг другу.

Теперь давайте рассмотрим более детально операции сортировки и связанные с ними операции. Для каждой функции показан прототип (прототипы) и приводится краткое пояснение. Этот раздел состоит из нескольких подразделов. Как и ранее, пары итераторов указывают на диапазоны, с выбранным именем шаблонного параметра, указывающего на тип итератора. Как обычно, диапазон определяется в виде [first, last), включая first и не включая last. Функции, передаваемые в виде аргументов, являются функциональными объектами, которые могут быть указателями или объектами, для которых определена операция (). Как и в главе 16, предикат представляет собой булевскую функцию с одним аргументом, а бинарный предикат — булевскую функция с двумя аргументами. (Функции не обязательно должны иметь тип bool, поскольку они возвращают значение 0, соответствующее значению false, и ненулевое значение, соответствующее значению true.) Кроме этого, как упоминалось в главе 16, унарным функциональным объектом является функция, принимающая один аргумент, а бинарным функциональным объектом — функция, принимающая пару аргументов.

Сортировка

Для начала рассмотрим алгоритмы сортировки.

sort()

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,Compare comp);
```

Функция sort() выполняет сортировку по возрастанию элементов в диапазоне [first, last), используя для сравнения операцию <, определенную для типа значения. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

stable_sort()

```
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first,
   RandomAccessIterator last, Compare comp);
```

Функция stable_sort() выполняет сортировку элементов в диапазоне [first, last), сохраняя относительную упорядоченность эквивалентных элементов. В пер-

вом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

partial_sort()

```
template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator
middle, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator
middle, RandomAccessIterator last, Compare comp);
```

Функция partial_sort() выполняет частичную сортировку элементов в диапазоне [first, last). Первые элементы middle - first отсортированного диапазона помещаются в диапазон [first, middle), а остальные элементы остаются несортированными. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

partial_sort_copy()

```
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(InputIterator first,
InputIterator last, RandomAccessIterator result_first,
RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator partial_sort_copy(InputIterator first,
   InputIterator last, RandomAccessIterator result_first,
   RandomAccessIterator result_last, Compare comp);
```

Функция partial_sort_copy() копирует первые n элементов отсортированного диапазона [first, last) в диапазон [result_first, result_first + n). Значение n меньше, чем last - first и result_last - result_first. Функция возвращает result_first + n. В первом варианте используется операция <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

nth_element()

```
template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
   RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
   RandomAccessIterator last, Compare comp);
```

Функция nth_element() находит элемент в диапазоне [first, last), могущий быть расположенным в позиции nth, в которой был отсортирован диапазон, и помещает его в позицию nth. В первом варианте используется операция <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

Бинарный поиск

Алгоритмы этой группы предполагают, что диапазон является отсортированным. Для них необходим только однонаправленный итератор, однако они являются самыми эффективными алгоритмами для случайных итераторов.

lower_bound()

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first,
   ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first,
   ForwardIterator last, const T& value, Compare comp);
```

Функция `lower_bound()` находит первую позицию в отсортированном диапазоне `[first, last)`, перед которой значение `value` может быть вставлено без нарушения упорядочения. Функция возвращает итератор, указывающий на эту позицию. В первом варианте используется операция <, а во втором варианте для определения порядка сортировки применяется объект сравнения `comp`.

upper_bound()

```
template<class ForwardIterator, class T>
ForwardIterator upper_bound(ForwardIterator first,
   ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator first,
   ForwardIterator last, const T& value, Compare comp);
```

Функция `upper_bound()` находит последнюю позицию в отсортированном диапазоне `[first, last)`, перед которой значение `value` может быть вставлено без нарушения упорядочения. В первом варианте используется операция <, а во втором варианте для определения порядка сортировки применяется объект сравнения `comp`.

equal_range()

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first,
   ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first,
   ForwardIterator last, const T& value, Compare comp);
```

Функция `equal_range()` находит наибольший поддиапазон `[it1, it2)` в отсортированном диапазоне `[first, last)`, в котором значение `value` может быть вставлено перед любым итератором в этом диапазоне без нарушения упорядоченности. Функция возвращает пару `pair`, состоящую из `it1` и `it2`. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения `comp`.

binary_search()

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first,
   ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first,
   ForwardIterator last, const T& value, Compare comp);
```

Функция binary_search() возвращает true, если в отсортированном диапазоне [first, last) будет найдено значение, эквивалентное значению value; в противном случае функция возвращает false. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

На заметку!

Вспомните, что если для сортировки используется операция <, то значения a и b будут эквивалентны друг другу, если результат сравнения a < b и b < a будет равен false. Для обычных чисел под эквивалентностью подразумевается равенство, однако для структур, отсортированных на основе только одного члена, это не так. Таким образом, может существовать несколько позиций, в которые может быть вставлено новое значение с сохранением упорядоченности данных. Аналогично, если объект сравнения comp используется для формирования упорядоченности, то под эквивалентностью подразумевается, что результат comp(a,b) и comp(b,a) равен false. (Это обобщенная форма утверждения, что a и b эквивалентны друг другу, если a не меньше b, и b не меньше a.)

Слияние

Функции слияния работают с отсортированными диапазонами.

merge()

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2, OutputIterator result);

template<class InputIterator1, class InputIterator2,
   class OutputIterator, class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2, OutputIterator result,
   Compare comp);
```

Функция merge() выполняет слияние элементов из отсортированного диапазона [first1, last1) и из отсортированного диапазона [first2, last2), помещая результат в диапазон, начинающийся с result. Искомый диапазон не должен перекрывать ни один из диапазонов, вовлеченных в слияние. Если в обоих диапазонах будут найдены эквивалентные элементы, то элементы из первого диапазона будут предшествовать элементам второго диапазона. Для результирующего слияния возвращаемым значением является итератор на элемент, следующий за последним элементом. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

inplace_merge()

```
template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
   BidirectionalIterator middle, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
   BidirectionalIterator middle, BidirectionalIterator last, Compare comp);
```

Функция inplace_merge() осуществляет слияние двух последовательно отсортированных диапазонов — [first, middle) и [middle, last) — в одну отсортированную последовательность, хранящуюся в диапазоне [first, last). Элементы из первого диапазона будут предшествовать эквивалентным элементам из второго диапазона. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

Работа с множествами

Операции над множествами могут выполняться над любыми отсортированными последовательностями, включая set и multiset. Для контейнеров, содержащих несколько экземпляров значения, например multiset, определения обобщаются. Объединение двух мультимножеств содержит большее количество появлений каждого элемента, а пересечение — меньшее количество появлений каждого элемента. Предположим, например, что мультимножество A содержит семь строк "apple", а мультимножество B — четыре такие строки. Объединение A и B будет содержать семь экземпляров строки "apple", а пересечение — четыре упомянутых экземпляра.

includes()

```
template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2, Compare comp);
```

Функция includes() возвращает значение true, если каждый элемент из диапазона [first2, last2) будет также найден в диапазоне [first1, last1); в противном случае функция возвращает значение false. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

set_union()

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2, OutputIterator result);

template<class InputIterator1, class InputIterator2,
   class OutputIterator, class Compare>
OutputIterator set_union(InputIterator1 first1,
   InputIterator1 last1, InputIterator2 first2,
   InputIterator2 last2, OutputIterator result, Compare comp);
```

Функция set_union() формирует множество, которое является объединением диапазонов [first1, last1) и [first2, last2), и копирует результат в позицию, на которую указывает result. Результирующий диапазон не должен перекрывать ни один из исходных диапазонов. Для результирующего диапазона функция возвращает итератор на элемент, следующий за последним элементом. Объединение представляет собой множество, состоящее из всех элементов, которые можно найти либо в одном из множеств, либо в обоих. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

set_intersection()

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1,
   InputIterator1 last1, InputIterator2 first2,
   InputIterator2 last2, OutputIterator result);

template<class InputIterator1, class InputIterator2,
   class OutputIterator, class Compare>
OutputIterator set_intersection(InputIterator1 first1,
   InputIterator1 last1, InputIterator2 first2,
   InputIterator2 last2, OutputIterator result, Compare comp);
```

Функция set_intersection() формирует множество, которое представляет собой пересечение диапазонов [first1, last1) и [first2, last2), и копирует результат в позицию, на которую указывает result. Результирующий диапазон не должен перекрывать ни один из первоначальных диапазонов. Для сформированного диапазона функция возвращает итератор на элемент, следующий за последним элементом. Пересечение представляет собой множество, содержащее элементы, общие для обоих множеств. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

set_difference()

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_difference(InputIterator1 first1,
   InputIterator1 last1, InputIterator2 first2,
   InputIterator2 last2, OutputIterator result);

template<class InputIterator1, class InputIterator2,
   class OutputIterator, class Compare>
OutputIterator set_difference(InputIterator1 first1,
   InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
   OutputIterator result, Compare comp);
```

Функция set_difference() формирует множество, представляющее собой разность между диапазонами [first1, last1) и [first2, last2), и копирует результат в позицию, на которую указывает result. Результирующий диапазон не должен перекрывать ни один из исходных диапазонов. Для сформированного диапазона функция возвращает итератор на элемент, следующий за последним элементом. Разность представляет собой множество, содержащее элементы, которые были найдены в первом множестве и не найдены во втором. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

set_symmetric_difference()

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_symmetric_difference( InputIterator1 first1,
   InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
   OutputIterator result);

template<class InputIterator1, class InputIterator2,
   class OutputIterator, class Compare>
OutputIterator set_symmetric_difference(InputIterator1 first1,
   InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
   OutputIterator result, Compare comp);
```

Функция set_symmetric_difference() формирует последовательность, которая представляет собой симметричную разность между диапазонами [first1, last1) и [first2, last2), и копирует результат в позицию, на которую указывает result. Результирующий диапазон не должен перекрывать ни один из исходных диапазонов. Для сформированного диапазона функция возвращает итератор на элемент, следующий за последним элементом. Симметричная разность представляет собой множество, содержащее элементы, которые были найдены в первом множестве и не найдены во втором, и которые были найдены во втором множестве и не найдены в первом. Это то же самое, что и разность между объединением и пересечением. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

Работа с частично упорядоченными полными бинарными деревьями

Частично упорядоченное полное бинарное дерево является общей формой представления данных, при которой первый элемент является наибольшим элементом. Всякий раз, когда удаляется первый элемент или добавляется любой другой, может возникнуть необходимость в перегруппировке частично упорядоченного полного бинарного дерева с целью сохранения его свойства. Частично упорядоченное полное бинарное дерево создается таким образом, чтобы обеспечить эффективное выполнение этих двух операций.

make_heap()

```
template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
   Compare comp);
```

Функция make_heap() создает частично упорядоченное полное бинарное дерево, определяемое диапазоном [first, last). В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

push_heap()

```
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);
```

Функция push_heap() **предполагает, что диапазон** [first, last - 1) **является действительным частично упорядоченным полным бинарным деревом, и добавляет значение в позицию** last - 1 **(то есть пропускает конец дерева, которое предполагается действительным) частично упорядоченного полного бинарного дерева, в результате чего дерево** [first, last) **становится действительным. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения** comp.

pop_heap()

```
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);
```

Функция pop_heap() **предполагает, что диапазон** [first, last) **является действительным частично упорядоченным полным бинарным деревом. Она осуществляет обмен значениями в позициях** last - 1 **и** first, **в результате чего диапазон** [first, last - 1) **становится действительным частично упорядоченным полным бинарным деревом. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения** comp.

sort_heap()

```
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);
```

Функция sort_heap() **предполагает, что диапазон** [first, last) **является частично упорядоченным полным бинарным деревом, и выполняет его сортировку. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения** comp.

Поиск максимального и минимального значений

Функции минимума и максимума возвращают минимальное и максимальное значения пар значений и последовательностей значений.

min()

```
template<class T> const T& min(const T& a, const T& b);

template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);
```

Функция min() **возвращает меньшее из двух значений. Если два значения эквивалентны друг другу, возвращается первое значение. В первом варианте используется**

<, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

max()

```
template<class T> const T& max(const T& a, const T& b);

template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);
```

Функция max() возвращает большее из двух значений. Если два значения эквивалентны друг другу, возвращается первое значение. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

min_element()

```
template<class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first,
   ForwardIterator last, Compare comp);
```

Функция min_element() возвращает первый итератор it в диапазоне [first, last), такой, что ни один элемент из диапазона не будет меньше *it. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

max_element()

```
template<class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first,
   ForwardIterator last, Compare comp);
```

Функция max_element() возвращает первый итератор it в диапазоне [first, last), такой, что ни один элемент в диапазоне не будет больше *it. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

lexicographical_compare()

```
template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(InputIterator1 first1,
   InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2, Compare comp);
```

Функция lexicographical_compare() возвращает значение true, если последовательность элементов в диапазоне [first1, last1) лексикографически меньше, чем последовательность элементов в диапазоне [first2, last2); в противном случае функция возвращает false. При лексикографическом сравнении сравнивается пер-

вый элемент одной последовательности с первым элементом другой последовательности, то есть, сравниваются `*first1` с `*first2`. Если `*first1` меньше, чем `*first2`, то функция возвращает `true`. Если `*first2` меньше `*first1`, функция возвращает `false`. Если они эквивалентны, сравнивается следующий элемент в каждой последовательности. Этот процесс продолжается до тех пор, пока не будет найдено два неэквивалентных соответствующих элемента, или пока не будет достигнут конец последовательности. Если две последовательности эквивалентны друг другу вплоть до завершения одной из последовательностей, то более короткая последовательность будет меньше. Если две последовательности эквивалентны и имеют одинаковую длину, функция возвращает `false`. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения `comp`. Лексикографическое сравнение является обобщенной формой алфавитного сравнения.

Работа с перестановками

Перестановкой последовательности называется изменение порядка элементов. Например, последовательность, состоящая из трех элементов, имеет шесть возможных вариантов упорядочения, поскольку в качестве первого элемента могут быть выбраны три элемента. Выбор определенного элемента для первой позиции оставляет возможность выбора двух элементов для второй позиции и одного элемента для третьей позиции. Например, шесть вариантов перестановки цифр 1, 2 и 3 выглядят следующим образом:

```
123 132 213 232 312 321
```

В общем случае последовательность, состоящая из n элементов, имеет $n \times (n-1) \times \ldots \times 1$, или $n!$, возможных вариантов перестановок.

Функции перестановки подразумевают, что последовательность всех возможных перестановок может быть изменена в лексикографическом порядке, как было показано в предыдущем примере шести вариантов перестановок. В общем случае это означает, что существует определенная перестановка, предшествующая и следующая за каждой перестановкой. Например, 213 непосредственно предшествует 232, а 312 непосредственно следует за ней. Однако первая перестановка (123 в нашем примере) не имеет предшествующей, а последняя перестановка (321 в нашем примере) не имеет последующей.

next_permutation()

```
template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
   BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first,
   BidirectionalIterator last, Compare comp);
```

Функция `next_permutation()` преобразует последовательность в диапазоне `[first, last)` в следующий вариант перестановки в лексикографическом порядке. Если следующий вариант перестановки существует, то функция возвращает `true`. Если такой вариант не существует (то есть диапазон содержит последнюю перестановку в лексикографическом порядке), то функция возвращает `false` и преобразует диапазон в первый вариант перестановки в лексикографическом порядке. В первом

варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

prev_permutation()

```
template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,

   BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
   BidirectionalIterator last, Compare comp);
```

Функция previous_permutation() преобразует последовательность в диапазоне [first, last) в предыдущий вариант перестановки в лексикографическом порядке. Если предыдущий вариант перестановки существует, то функция возвращает true. Если такой вариант не существует (то есть диапазон содержит первую перестановку в лексикографическом порядке), функция возвращает false и преобразует диапазон в последний вариант перестановки в лексикографическом порядке. В первом варианте используется <, а во втором варианте для определения порядка сортировки применяется объект сравнения comp.

Числовые операции

В табл. Ж.12 перечислены числовые операции, которые описаны в заголовочном файле numeric. Аргументы в этой таблице не показаны, а перегруженные функции показаны только один раз. Каждая функция имеет версию, в которой используется < для упорядочения элементов, и версию, в которой для упорядочения элементов пименяется функциональный объект сравнения. За таблицей следует более подробное их описание, включая прототипы. Таким образом, вы можете просмотреть таблицу, чтобы узнать, что выполняет определенная функция, и затем обратиться к детальной информации о ней.

Таблица Ж.12. Числовые операции

Функция	Описание
accumulate()	Вычисляет совокупную сумму по значениям из диапазона.
inner_product()	Вычисляет скалярное произведение двух диапазонов.
partial_sum()	Копирует частичные суммы, вычисленные из одного диапазона, во второй диапазон.
adjacent_difference()	Копирует смежные разности, вычисленные из элементов одного диапазона, в другой диапазон.

Теперь давайте рассмотрим более подробно каждую из этих операций. Для каждой функции представлен прототип (прототипы) и дано краткое пояснение.

accumulate()

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);
```

```
template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
   BinaryOperation binary_op);
```

Функция `accumulate()` присваивает `acc` значение `init`; затем она выполняет операцию `acc = acc + *i` (первая версия) или `acc = binary_op(acc, *i)` (вторая версия) для каждого итератора `i` в диапазоне `[first, last)` по порядку. Далее функция возвращает результирующее значение `acc`.

inner_product()

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T,
   class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, T init, BinaryOperation1 binary_op1,
   BinaryOperation2 binary_op2);
```

Функция `inner_product()` присваивает `acc` значение `init`; затем она выполняет операцию `acc = *i * *j` (первая версия) или `acc = binary_op(*i, *j)` (вторая версия) для каждого итератора `i` в диапазоне `[first1, last1)` по порядку, и каждого соответствующего итератора `j` в диапазоне `[first2, first2 + (last1 - first1))`. Другими словами, она вычисляет значение на основе первых элементов из каждой последовательности, затем на основе вторых элементов в каждой последовательности, и так далее до тех пор, пока не будет достигнут конец первой последовательности. (Следовательно, вторая последовательность как минимум должна иметь такую же длину, как и первая.) В завершение функция возвращает результирующее значение `acc`.

partial_sum()

```
template <class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator first, InputIterator last,
   OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(InputIterator first, InputIterator last,
   OutputIterator result, BinaryOperation binary_op);
```

Функция `partial_sum()` присваивает `*first` результату `*result` или `*first + *(first + 1)` результату `(result + 1)` (первая версия), либо присваивает `binary_op(first, *(first + 1))` результату `*(result + 1)` (вторая версия) и так далее. Другими словами, n-й элемент последовательности, начинающейся с `result`, содержит сумму (или эквивалент `binary_op`) первых `n` элементов последовательности, начинающейся с `first`. Функция возвращает итератор на элемент, следующий за последним элементом. Алгоритм допускает равенство `result` и `first`, то есть результат можно копировать поверх исходной последовательности, если в этом возникнет необходимость.

adjacent_difference()

```
template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator first,
   InputIterator last, OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(InputIterator first,
   InputIterator last, OutputIterator result, BinaryOperation binary_op);
```

Функция `adjacent_difference()` присваивает `*first` позиции `result(result = *first)`. Последующим позициям в искомом диапазоне присваиваются разности (или эквивалент `binary_op`) смежных позиций в исходном диапазоне. Другими словами, следующей позиции в искомом диапазоне `(result + 1)` присваивается `*(first + 1) - *first` (первая версия) или `binary_op(*(first + 1), *first)` (вторая версия) и так далее. Функция возвращает итератор на элемент, следующий за последним элементом. Алгоритм допускает равенство `result` и `first`, то есть результат можно копировать поверх исходной последовательности, если в этом возникнет необходимость.

ПРИЛОЖЕНИЕ 3

Рекомендуемая литература и ресурсы в Internet

Программированию на языке C++ посвящено множество хороших книг и ресурсов в Internet. Ниже предложен список литературы, который следует рассматривать скорее как репрезентативный, а не как полный. Помимо перечисленных, существует еще большое количество хороших книг и сайтов. Но, тем не менее, этот список охватывает весьма широкий диапазон литературы.

Рекомендуемая литература

- Booch, Grady. *Object-Oriented Analysis and Design*, Second Edition. Reading, MA: Addison-Wesley, 1993.

  В этой книге изложены принципы объектно-ориентированного программирования (ООП), рассматриваются методы ООП и приводятся некоторые примеры приложений. Все примеры выполнены на языке C++.

- Booch, Grady, Jim Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1998.

  Книга, написанная создателями языка UML (Unified Modeling Language), описывает ядро UML и предлагает множество примеров его использования.

- Cline, Marshall, Greg Lomow, and Mike Girou. *C++ FAQs*, Second Edition. Reading, MA: Addison-Wesley, 1999.

  Книга содержит большое количество ответов на часто задаваемые вопросы по C++.

- Jacobson, Ivar. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley, 1994.

  В этой книге собраны удачные руководства и методы (разработка объектно-ориентированного программного обеспечения — Object-Oriented Software Engineering [OOSE]) создания крупномасштабных систем программного обеспечения.

- Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference*. Reading, MA: Addison-Wesley, 1999.

  В книге описывается библиотека Standard Template Library (STL) и функциональные особенности библиотеки C++, например, поддержка комплексных чисел и потоков ввода-вывода.

- Lee, Richard C and William M. Tepfenhart. *UML and C++*, Second Edition. Upper Saddle River, New Jersey: Prentice Hall, 2001.

  Самоучитель UML, включающий обзор основ C++.

- Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, Second Edition. Reading, MA: Addison-Wesley, 1998.

  Книга предназначена для программистов, уже знакомых с C++, и предлагает 50 правил и указаний. Часть из них посвящена техническим вопросам. Например, в ней объясняется, когда необходимо определять конструкторы копирования и операции присваивания. Другая часть посвящена общим вопросам, например, отношениям *is-a* и *has-a*.

- Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Reading, MA: Addison-Wesley, 2001.

  Руководство по выбору контейнеров и алгоритмов; рассматриваются и другие аспекты использования библиотеки STL.

- Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Reading, MA: Addison-Wesley, 1996.

  Книга продолжает традицию *Effective C++*, объясняя некоторые менее понятные аспекты языка программирования, и демонстрирует примеры реализации различных задач, например, проектирование интеллектуальных указателей. В ней собран опыт программистов C++ за последние несколько лет.

- Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, Bill Lorensen, and William Lorenson. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

  Эта книга знакомит и исследует технику объектного моделирования (Object Modeling Technique — OMT), способ разделения задач на удобные для решения объекты.

- Rumbaugh, James, Ivar Jacobson, and Grady Booch. *Unified Modeling Reference Manual*. Reading, MA: Addison-Wesley, 1998.

  Книга, написанная создателями UML, предлагает полное описание UML в формате справочного руководства.

- Stroustrup, Bjarne. *The C++ Programming Language*, Third Edition. Reading, MA: Addison-Wesley, 1997.

  Страуструп — создатель C++, поэтому в этой книге вы найдете авторитетный текст. Если вы знакомы с C++, то вы должны были заметить, что эта книга наиболее часто упоминается в различных источниках. В ней не только описывается язык программирования, но и предлагается множество примеров его использования и рассматривается методология ООП. По мере совершенствования языка выходили новые издания этой книги, а это издание включает описание элементов стандартной библиотеки, например, STL и строки.

- Stroustrup, Bjarne. *The Design and Evolution of C++*. Reading, MA: Addison-Wesley, 1994.

  Если вас интересуют вопросы эволюции языка программирования C++, обратитесь к этой книге.

- Vandevoorde, David and Nocolai M. Jpsittos. *C++ Templates: The Complete Guide*. Reading, MA: Addison-Wesley, 2003.

  О шаблонах можно сказать очень много, о чем свидетельствует этот детальный справочник.

Ресурсы в Internet

- Стандарт языка C++ ISO/ANSI C++ Standard (ISO/IEC 14882:2003) является технически переработанной версией стандарта 1998 года (14882:1998); его можно найти как в ресурсах Национального института стандартизации США (American National Standards Institute — ANSI), так и в Международной организации по стандартизации (International Organization for Standardization — ISO).

  ANSI предлагает печатную копию, которая стоит $281, и загружаемую электронную версию в формате PDF (только для индивидуального пользователя) по цене $18. Обе версии доступны на сайте

  `http://webstore.ansi.org`

  На сайте ISO можно найти документ в виде загружаемого файла в формате PDF, который стоит 352 швейцарских франка; можно также приобрести версию на компакт-диске за такую же сумму:

  `www.iso.org`

- Сайт C++ FAQ Lite посвящен часто задаваемым вопросам (на английском, китайском, французском, русском и португальском языках), и является упрощенной версией книги Клина (Cline) и других. На данный момент его можно найти по адресу

  `www.parashift.com/c++-faq-lite`

- Модерируемая дискуссионная групап, посвященная вопросам по C++:

  `comp.lang.c++.moderated`

- *C/C++ Users Journal*

  Ежемесячный журнал пользователей C/C++ предназначен главным образом для профессиональных программистов. На посвященном ему Web-сайте (`www.cuj.com`) можно отыскать немало полезных ресурсов.

ПРИЛОЖЕНИЕ И

Переход к стандарту ANSI/ISO C++

Предположим, что у вас оказалась программа, написанная на языке С или более ранней версии C++ (а, может, это просто ваша привычка), и вы хотите преобразовать ее код в код стандартной версии C++. В этом приложении вы найдете указания по преобразованию. Часть из них связана с переходом из С в C++, а другая часть связана с переходом из более ранних версий C++ в стандартную версию C++.

Используйте альтернативные варианты для некоторых директив препроцессора

Препроцессор С/C++ предлагает множество директив. В общем случае, в C++ принято использовать директивы, предназначенные для управления процессом компиляции, и не применять директивы в качестве замены кода. Например, директива `#include` является необходимым компонентом для управления файлами программ. Другие директивы, например, `#ifndef` и `#endif`, позволяют управлять процессом компиляции определенных блоков программы. Директива `#pragma` позволяет управлять параметрами определенного компилятора. Все эти директивы являются полезными, а в некоторых случаях просто необходимыми инструментальными средствами. Однако следует быть осторожным при использовании директивы `#define`.

Используйте `const` вместо `#define` для определения констант

Символические константы облегчают чтение и поддержку кода программы. Имена констант указывают на их назначение, и если вам нужно будет изменить значение, то для этого достаточно изменить его один раз в определении, а затем выполнить повторную компиляцию. В языке С для создания символических имен констант применяется директива препроцессора:

```
#define MAX_LENGTH 100
```

Препроцессор заменяет текст в вашем исходном коде, подставляя `100` вместо `MAX_LENGTH` до компиляции.

В языке C++ для этой цели используется модификатор `const` в объявлении переменной:

```
const int MAX_LENGTH = 100;
```

В результате `MAX_LENGTH` будет интерпретироваться как константа только для чтения, имеющая тип `int`.

Использование модификатора const обладает рядом преимуществ. Во-первых, в объявлении явным образом именуется тип. При работе с #define необходимо использовать различные суффиксы для чисел, чтобы указать типы, отличные от char, int или double; например, необходимо использовать 100L, чтобы обозначить тип long, и 3.14F, чтобы обозначить тип float. Более важным является то, что const можно без труда использовать и для составных типов, как показано в следующем примере:

```
const int base_vals[5] = {1000, 2000, 3500, 6000, 10000};
const string ans[3] = {"yes", "no", "maybe"};
```

И, наконец, идентификаторы const подчиняются тем же правилам обзора данных, что и переменные. Таким образом, можно создать константы с глобальной областью видимости, имена которых будут соответствовать области видимости пространства имен, и блокировать область видимости. Если, например, определить константу в определенной функции, то можно будет не беспокоиться о том, что может возникнуть конфликт определения с глобальной константой, используемой где-нибудь в другом участке программы. Например, рассмотрим следующее:

```
#define n 5
const int dz = 12;
...
void fizzle()
{
   int n;
   int dz;
   ...
}
```

Препроцессор заменит

```
int n;
```

на

```
int 5;
```

и вызовет ошибку компилятора. Однако переменная dz, определенная в fizzle(), будет локальной. Также, при необходимости, fizzle() может использовать операцию разрешения контекста (::) и для обращения к константе использовать конструкцию ::dz.

Служебное слово в языке C++ const позаимствовано из языка C, однако в версии C++ оно является более полезным. Например, в версии C++ имеется внутреннее связывание внешних значений const, отличное от внешнего связывания, предлагаемого по умолчанию, применяемого для переменных и const в языке C. Это означает, что каждый файл в программе, использующей const, должен сам определять const. Хотя для этого и понадобится вводить дополнительный код, однако на самом деле внутреннее связывание облегчает жизнь. Посредством внутреннего связывания вы можете помещать определения const в заголовочном файле, который используется различными файлами в данном проекте. Для внешнего связывания такая схема вызовет ошибку компилятора, а для внутреннего связывания — нет. Также, поскольку const необходимо определять в том файле, который ее использует (она должна находиться в заголовочном файле, который используется этим файлом), вы можете применять значения const в виде аргументов размера массива:

```
const int MAX_LENGTH = 100;
...
double loads[MAX_LENGTH];
for (int i = 0; i < MAX_LENGTH; i++)
   loads[i] = 50;
```

В языке С такая схема работать не будет, поскольку определение объявления для MAX_LENGTH может находиться в отдельном файле и будет недоступным при компиляции этого определенного файла. По правде говоря, следует упомянуть, что в языке С для создания констант с внутренним связыванием можно использовать модификатор static. В языке С++, в котором static предлагается по умолчанию, об этом можно не вспоминать.

Между прочим, пересмотренный стандарт C Standard (C99) позволяет применять const в качестве спецификации размера массива, однако массив интерпретируется как новая форма массива, называемая *переменным массивом*, который не является частью стандартной версии С++.

Единственная роль директивы #define по-прежнему остается полезной — в качестве стандартной идиомы, используемой для управления компиляцией заголовочного файла:

```
// blooper.h
#ifndef _BLOOPER_H_
#define _BLOOPER_H_
// code goes here
#endif
```

Однако для обычных символических констант следует всегда использовать const вместо #define. Еще один хороший альтернативный вариант, который необходимо применять тогда, когда у вас имеется совокупность связанных целочисленных констант, заключается в использовании enum:

```
enum {LEVEL1 = 1, LEVEL2 = 2, LEVEL3 = 4, LEVEL4 = 8};
```

Используйте inline вместо #define для определения коротких функций

Традиционный способ создания близкой к эквивалентной подставляемой функции в языке С предусматривает использование макроопределения #define:

```
#define Cube(X) X*X*X
```

В результате препроцессор выполнит следующую замену, в которой X будет заменена соответствующим аргументом для Cube():

```
y = Cube(x);         // заменяет y = x*x*x;
y = Cube(x + z++);   // заменяет x + z++*x + z++*x + z++;
```

Поскольку препроцессор заменяет текст, а не передает аргументы, использование этих макроопределений может привести к неожиданным и неверным результатам. Такие ошибки можно свести к минимуму, если в макроопределении применить несколько пар круглых скобок, дабы определить правильный порядок выполнения операций:

```
#define Cube(X) ((X)*(X)*(X))
```

Однако даже такая форма не имеет отношения к использованию значений вроде z++.

Применение служебного слова inline для обозначения подставляемых функций в языке С++ является более зависимым, поскольку при этом происходит передача аргументов. Более того, в качестве подставляемых функций в С++ могут использоваться обычные функции или методы класса:

```
class dormant
{
private:
   int period;
   ...
public:
   int Period() const { return period; } // автоматически подставляемая
   ...
};
```

Единственная положительная особенность макроопределения #define состоит в том, что в нем не определяется тип, поэтому его можно использовать для любого типа, для которого имеет смысл данная операция. В языке С++ можно создавать подставляемые шаблоны, чтобы получить функции, не зависящие от типа, и при этом сохранять передачу аргументов.

Короче говоря, вместо макроопределений #define языка С следует использовать подставляемые функции языка С++.

Используйте прототипы функций

На самом деле, выбора здесь нет: применять прототипы в языке С не обязательно, но в С++ использовать их необходимо. Обратите внимание, что функция, определяемая перед своим первым использованием, как, например, встроенная функция, является ее же прототипом.

Использовать const в прототипах функций и заголовках следует тогда, когда это необходимо. В частности, следует использовать const с параметрами указателя и параметрами ссылки, представляющими данные, которые не должны изменяться. Это не только позволит компилятору перехватывать ошибки, влекущие за собой изменение данных, но и обобщит функцию. Другими словами, функция с указателем const или ссылкой может обрабатывать как данные const, так и другие данные, а функция, которая не может использовать const с указателем или ссылкой, может обрабатывать только другие данные.

Используйте приведение типов

Одной из неприятных особенностей языка С является его недисциплинированное приведение типов, что весьма раздражало Страуструпа. Действительно, приведение типов часто применяется в программах, однако стандартная схема приведения типов недостаточно ограничена. Например, рассмотрим следующий фрагмент кода:

```
struct Doof
{
   double feeb;
   double steeb;
```

```
    char sgif[10];
};
Doof leam;
short * ps = (short *) & leam;      // старый синтаксис
int * pi = int * (&leam);           // новый синтаксис
```

Указатель одного типа приводится к указателю совершенно другого типа, и от этого в языке C никак не избавиться.

В определенном отношении эта ситуация подобна ситуации с оператором goto. Проблема заключалась в том, что этот оператор был настолько гибким, что его применение приводило к запутыванию кода. Решение заключалось в том, чтобы для обработки обычных задач, в которых необходимо использовать goto, предложить более ограниченные, структурированные версии этого оператора. В результате были разработаны такие элементы языка программирования, как циклы for и while и операторы if else. Стандартная версия C++ предлагает похожее решение проблемы недисциплинированного приведения типов, а именно — ограниченное приведение типов для обработки самых обычных ситуаций, в которых требуется выполнять приведение типов. Ниже перечислены операции приведения типов, которые были рассмотрены в главе 15:

```
dynamic_cast
static_cast
const_cast
reinterpret_cast
```

Таким образом, если приведение типа необходимо применить к указателю, то тогда следует использовать по возможности одну из этих операций. В результате будет выполнено гарантированно корректное приведение типа.

Знакомьтесь с функциональными особенностями C++

Если вы применяли malloc() и free(), то вместо них следует использовать new и delete. Если для обработки ошибок вы использовали setjmp() и longjmp(), то вместо них следует обращаться к try, throw и catch. Старайтесь использовать тип bool для значений, представляющих true и false.

Используйте новую организацию заголовков

В стандартной версии C++ определены новые имена для заголовочных файлов, о чем говорилось в главе 2. Если вы применяли заголовочные файлы в старом стиле, вам следует перейти к использованию имен в новом стиле. Это не просто "косметическое" изменение, наоборот — новые версии часто обладают новыми особенностями. Например, заголовочный файл ostream обеспечивает поддержку ввода и вывода для расширенных таблиц символов. Он также предлагает новые манипуляторы, такие как boolalpha и fixed (см. главу 17). Эти манипуляторы обеспечивают более простой интерфейс по сравнению с функциями setf() или iomanip для настройки многих параметров форматирования. Если вы используете setf(), то при определе-

нии констант вместо ios следует указывать ios_base; то есть необходимо применять ios_base::fixed вместо ios::fixed. Кроме того, новые заголовочные файлы включают пространства имен.

Используйте пространства имен

Пространства имен помогают организовывать идентификаторы, используемые в программе, таким образом, чтобы избежать возникновения конфликтных ситуаций с использованием имен. Поскольку стандартная библиотека, реализованная в новой организации заголовочного файла, помещает имена в пространство имен std, то для использования этих заголовочных файлов необходимо указывать пространства имен.

Для простоты в примерах из этой книги обычно применяется директива using, чтобы сделать доступными все имена из пространства имен std:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std; // директива using
```

Однако массовое экспортирование всех имен в пространство имен, независимо от того, есть в этом необходимость или нет, противоречит целям пространств имен.

Предпочтительнее помещать директиву using в пределах функции, в результате чего имена будут доступны только в ее пределах.

Рекомендуется прибегнуть к еще лучшему варианту: чтобы сделать доступными те имена, которые необходимы программе, нужно использовать либо объявление using, либо операцию разрешения контекста (::). Например, следующий фрагмент кода:

```
#include <iostream>
using std::cin; // объявление using
using std::cout;
using std::endl;
```

делает доступными cin, cout и endl для остальной части файла. Однако операция разрешения контекста делает имя доступным только в том выражении, в котором применяется эта операцию:

```
cout << std::fixed << x << endl; //использование операции разрешения контекста
```

Несмотря на то что этот процесс может показаться изнурительным, вы сможете помещать общие объявления using в заголовочный файл:

```
// mynames -- заголовочный файл
using std::cin; // объявление using
using std::cout;
using std::endl;
```

Продолжая этот процесс далее, вы сможете помещать объявления using в пространства имен:

```
// mynames -- заголовочный файл
#include <iostream>
namespace io
{
```

```
    using std::cin;
    using std::cout;
    using std::endl;
}
namespace formats
{
    using std::fixed;
....using std::scientific;
    using std:boolalpha;
}
```

Затем в программу можно будет включить этот файл и использовать те пространства имен, которые ей необходимы:

```
#include "mynames"
using namespace io;
```

Используйте шаблон autoptr

Каждая операция `new` должна сопровождаться соответствующей операцией `delete`. Если работа функции, в которой используется операция `new`, прерывается из-за возникшего исключения, это может привести к возникновению проблем. Как было сказано в главе 15, в случае использования объекта `autoptr` для отслеживания объекта, созданного `new`, операция `delete` активизируется автоматически.

Используйте класс string

Традиционная строка в стиле языка C не принадлежит к реальному типу. Вы можете хранить строку в массиве символов и инициализировать массив символов по строке. Однако с помощью операции присваивания нельзя присвоить строку массиву символов; наоборот, необходимо применять `strcpy()` или `strncpy()`. С помощью условных операций невозможно произвести сравнение строк в стиле языка C; наоборот, необходимо использовать `strcmp()`. (Если, например, использовать операцию >, то это не будет синтаксической ошибкой; напротив, программа будет сопоставлять адреса строки, а не ее содержимое.)

С другой стороны, используя класс `string` (см. главу 16), можно представлять строки посредством объектов. Для работы со строками определены операции присваивания, условные операции и операция сложения (для конкатенации). Более того, класс `string` позволяет осуществлять автоматическое управление памятью, поэтому обычно можно не беспокоиться о том, что пользователь введет строку, которая либо переполнит массив, либо будет усечена еще до сохранения.

Класс `string` предлагает множество удобных методов. Например, можно присоединять один объект `string` к другому, а также присоединять строку в стиле языка C или даже значение `char` к объекту `string`. Для функций, требующих аргумент строки в стиле C, можно вызывать метод `c_str()` для возврата подходящего указателя на тип `char`.

Класс `string` предлагает не только надежный комплекс методов обработки задач, связанных со строками, вроде механизма поиска подстроки, но и функциональные возможности, совместимые со стандартной библиотекой шаблонов Standard Template Library — STL), поэтому для объектов `string` можно использовать алгоритмы STL.

Используйте библиотеку STL

Библиотека STL (см. главу 16 и приложение Ж) предлагает готовые решения многих программных задач, поэтому вам обязательно следует ее использовать. Например, вместо объявления массива объектов double или string, вы можете создать объект vector<double> или объект vector<string>. Преимущества этого подобны преимуществам применения объектов string вместо строк в стиле С. Операция присваивания определена, поэтому вы можете использовать ее для присваивания одного объекта vector другому. Объект vector можно передавать по ссылке, а функция, получающая такой объект, может с помощью метода size() определять количество элементов в объекте vector. Возможность управлять памятью позволяет автоматически изменять размеры при использовании метода pushback() для добавления элементов в объект vector. И, естественно, библиотека предлагает некоторые полезные методы класса и обобщенные алгоритмы.

Если вам необходим список, двусторонняя очередь, стек, обычная очередь, последовательность или карта, вам следует использовать библиотеку STL, которая предлагает полезные шаблоны контейнеров. Библиотека алгоритмов разработана таким образом, что вы можете без труда копировать содержимое вектора в список или сравнивать содержимое последовательности с вектором. Библиотека STL является своеобразным инструментальным средством, предлагающим базовые элементы, которые при необходимости можно включать в сборку.

Обширная библиотека алгоритмов разрабатывалась очень тщательно, поэтому вы можете достичь прекрасных результатов при относительно небольших усилиях в программировании. Концепция итератора, задействованная при реализации алгоритмов, означает, что алгоритмы можно применять не только к контейнерам STL, но и, например, к традиционным массивам тоже.

ПРИЛОЖЕНИЕ К

Ответы на вопросы для самоконтроля

Ответы на вопросы для самоконтроля из главы 2

1. Они называются функциями.
2. Содержимое файла `ostream` будет заменено этой директивой до окончательной компиляции.
3. Определения, созданные в пространстве имен `std`, будут доступны программе.
4. `cout << "Hello, world\n";`

   или

   `cout << "Hello, world" << endl;`
5. `int cheeses;`
6. `cheeses = 32;`
7. `cin >> cheeses;`
8. `cout << "Мы видели " << cheeses << " разновидности сыра\n";`
9. Функция `froop()` вызывается с одним аргументом, который будет иметь тип `double`; функция будет возвращать значение, имеющее тип `int`. Например, ее можно использовать следующим образом:

   ```
   int gval = froop(3.14159);
   ```

   Функция `rattle()` не имеет возвращаемого значения и ожидает аргумент `int`. Например, ее можно вызвать так:

   ```
   rattle(37);
   ```

   Функция `prune()` возвращает `int` и ожидает использования без аргумента. Например, ее можно использовать следующим образом:

   ```
   int residue = prune();
   ```
10. Служебное слово `return` не нужно использовать в функции, если она возвращает тип `void`. Однако его можно применять, если вы не предоставляете возвращаемого значения:

    ```
    return;
    ```

Ответы на вопросы для самоконтроля из главы 3

1. Имея возможность выбора нескольких целочисленных типов, можно подобрать такой тип, который будет наиболее точно соответствовать решению данной задачи. Например, вы могли бы использовать short для экономии памяти или long для гарантированного объема памяти или определить, что данный тип позволяет увеличить скорость вычислений.

2.

```
short rbis = 80;          // или short int rbis = 80;
unsigned int q = 42110;   // или unsigned q = 42110;
unsigned long ants = 3000000000;
```

   Примечание: int не в состоянии хранить значение 3000000000.

3. В C++ нельзя автоматически защититься от превышения целочисленных пределов; чтобы узнать об ограничениях, можно воспользоваться заголовочным файлом climits.

4. Константа 33L имеет тип long, а константа 33 — тип int.

5. Эти операторы не эквивалентны друг другу, хотя в некоторых системах результат их выполнения будет одинаковым. Более того, первый оператор присваивает букву A переменной grade только в той системе, в которой используется код ASCII, а второй оператор работает для других кодировок. Кроме этого, 65 является константой int, а 'A' — константой char.

6. Вот четыре способа:

```
char c = 88;
cout << c << endl;          // тип char выводит в виде символа

cout.put(char(88));         // put() выводит char в виде символа

cout << char(88) << endl;   // новый стиль приведения типа значения
                            // к типу char

cout << (char)88 << endl;   // старый стиль приведения типа значения
                            // к типу char
```

7. Ответ зависит от того, сколько байтов содержат эти типы. Если тип long имеет 4 байта, то потерь не будет. Это объясняется тем, что наибольшим значением типа long является примерно 2 миллиарда, что составляет 10 цифр. Поскольку double предлагает как минимум 13 значащих цифр, округление проводить не нужно.

8.

   а. `8 * 9 + 2 is 72 + 2 is 74`

   б. `6 * 3 / 4 is 18 / 4 is 4`

   в. `3 / 4 * 6 is 0 * 6 is 0`

   г. `6.0 * 3 / 4 is 18.0 / 4 is 4.5`

   д. `15 % 4 is 3`

9. Для этой задачи подойдет один из следующих вариантов:

```
int pos = (int) x1 + (int) x2;
int pos = int(x1) + int(x2);
```

Ответы на вопросы для самоконтроля из главы 4

1.
   а. `char actors[30];`
   б. `short betsie[100];`
   в. `float chuck[13];`
   г. `long double dipsea[64];`

2. `int oddly[5] = {1, 3, 5, 7, 9};`

3. `int even = oddly[0] + oddly[4];`

4. `cout << ideas[1] << "\n"; // or << endl;`

5. `char lunch[13] = "cheeseburger"; // количество символов + 1`

   или

   `char lunch[] = "cheeseburger";//пусть компилятор сам считает элементы`

6.

```
struct fish {
   char kind[20];
   int weight;
   float length;
};
```

7.

```
fish petes =
{
   "trout",
   13,
   12.25
};
```

8. `enum Response {No, Yes, Maybe};`

9.

```
double * pd = &ted;
cout << *pd << "\n";
```

10.

```
float * pf = treacle; // или = &treacle[0]
cout << pf[0] << " " << pf[9] << "\n";
// или использовать *pf и *(pf + 9)
```

11.

```
unsigned int size;
cout << "Введите положительное целое число: ";
cin >> size;
int * dyn = new int [size];
```

12. Да, этот код правильный. Выражение "Home of the jolly bytes" является строковой константой; следовательно, она оценивается как адрес начала строки. Объект cout интерпретирует адрес char как приглашение на вывод строки, однако приведение типа (int *) преобразует адрес к типу указателя на тип int, который впоследствии выводится в виде адреса. Короче говоря, оператор выводит адрес строки.

13.

```
struct fish
{
   char kind[20];
   int weight;
   float length;
};
fish * pole = new fish;
cout << "Введите сорт рыбы: ";
cin >> pole->kind;
```

14. В результате использования cin >> address программа будет пропускать пробельные символы, пока не будет найден символ, отличный от пробельного. Затем будут считываться символы, пока снова не будет найден пробельный. Таким образом, будет пропущена новая строка, следующая за вводом чисел, что избавляет от этой проблемы. С другой стороны, будет считываться только одно слово, а не вся строку.

Ответы на вопросы для самоконтроля из главы 5

1. Цикл с проверкой на входе оценивает проверочное выражение до входа в тело цикла. Если условие изначально равно false, то содержимое тела цикла никогда не выполнится. Цикл с проверкой на выходе оценивает проверочное выражение после обработки содержимого тела цикла. Таким образом, тело цикла выполняется один раз, даже если проверочное выражение изначально равно false. Циклы for и while являются циклами с проверкой на входе, а цикл do while — циклом с проверкой на выходе.

2. Будет напечатано следующее:

```
01234
```

Обратите внимание, что cout << endl; не является частью тела цикла (поскольку отсутствуют фигурные скобки).

3. Будет напечатано следующее:

```
0369
12
```

4. Будет напечатано следующее:

```
6
8
```

5. Будет напечатано следующее:

```
k = 8
```

6. Проще всего использовать операцию *=:

```
for (int num = 1; num <= 64; num *= 2)
cout << num << " ";
```

7. Операторы заключают в двойные кавычки для того, чтобы сформировать один составной оператор, или блок.

8. Да, первый оператор является правильным. Выражение `1,024` состоит из двух выражений, `1` и `024`, разделенных операцией запятой. Значение является значением выражения справа. Это `024`, которое является восьмеричным эквивалентом десятичного `20`, поэтому в объявлении переменной `x` присваивается значение `20`. Второй оператор также правилен. Однако в соответствии с приоритетом выполнения операций выражение будет оценено следующим образом:

```
(y = 1), 024;
```

То есть выражение слева присваивает переменной `y` значение `1`, а значение всего выражения, которое не используется, равно `024`, или `20`.

9. `cin >> ch` формирует пропуски пробелов, новых строк и табуляции. Две другие конструкции формируют чтение этих символов.

Ответы на вопросы для самоконтроля из главы 6

1. Обе версии кода дают одинаковые ответы, однако вариант `if else` более эффективен. Посмотрим, что произойдет, например, если `ch` является пробелом. Версия 1: после инкрементирования `spaces` выполняется проверка, является ли символ символом новой строки. Это непроизводительные затраты времени, поскольку код уже установил, что `ch` является пробелом и, таким образом, не может быть новой строкой. Версия 2: в этой же ситуации проверка на предмет новой строки пропускается.

2. `++ch` и `ch + 1` имеют одно и то же числовое значение. Однако `++ch` имеет тип `char` и организует вывод в виде символа, в то время как `ch + 1`, поскольку добавляет `char` к типу `int`, является типом `int` и организует вывод в виде числа.

3. Поскольку программа использует `ch = '$'` вместо `ch == '$'`, комбинированный ввод и вывод выглядит следующим образом:

```
Hi!
H$i$!$
$Send $10 or $20 now!
S$e$n$d$ $ct1 = 9, ct2 = 9
```

Каждый символ преобразуется в `$` до вывода на экран во второй раз. Также значение выражения `ch == $` является кодом символа `$`, следовательно, ненулевым, следовательно, `true`; таким образом, каждый раз происходит инкрементирование `ct2`.

4.

а. `weight >= 115 && weight < 125`

б. `ch == 'q' || ch == 'Q'`

в. `x % 2 == 0 && x != 26`

г. `x % 2 == 0 && !(x % 26 == 0)`

д. `donation >= 1000 && donation <= 2000 || guest == 1`

е. `(ch >= 'a' && ch <= 'z') ||(ch >= 'A' && ch <= 'Z')`

5. Не обязательно. Например, если `x` равно `10`, то `!x` равно `0`, а `!!x` равно `1`. Однако, если `x` является переменной `bool`, то `!!x` является `x`.

6. `(x < 0)? -x    x`

   или

   `(x >= 0)? x : -x;`

7.

```
switch (ch)
{
   case 'A':  a_grade++;
              break;
   case 'B':  b_grade++;
              break;
   case 'C':  c_grade++;
              break;
   case 'D':  d_grade++;
              break;
   default:   f_grade++;
              break;
}
```

8. Если вы используете целочисленные метки, и пользователь вводит не целое число, например **q**, то программа "зависнет", потому что целочисленный ввод не может быть обработан как символ. Однако если применять целочисленные метки, и пользователь введет целое число, например **5**, то символьный ввод будет интерпретировать `5` как символ. Затем часть `default` оператора `switch` может предположить, что введен другой символ.

9. Вот одна из версий кода:

```
int line = 0;
char ch;
while (cin.get(ch) && ch != 'Q')
{
   if (ch == '\n')
      line++;
}
```

Ответы на вопросы для самоконтроля из главы 7

1. Эти три шага таковы: определение функции, предоставление прототипа и вызов функции.

2.

   а. `void igor(void);     // или void igor()`

   б. `float tofu(int n);  // или float tofu(int);`

   в. `double mpg(double miles, double gallons);`

г. `long summation(long harray[], int size);`

д. `double doctor(const char * str);`

е. `void ofcourse(boss dude);`

ж. `char * plot(map *pmap);`

3.

```
void set_array(int arr[], int size, int value)
{
   for (int i = 0; i < size; i++)
      arr[i] = value;
}
```

4.

```
void set_array(int * begin, int * end, int value)
{
   for (int * pt = begin; pt != end; pt++)
      pt* = value;
}
```

5.

```
double biggest (const double foot[], int size)
{
   double max;
   if (size < 1)
   {
      cout << "Неверный размер массива: " << size << endl;
      cout << "Возврат значения, равного 0\n";
      return 0;
   }
   else  // эта часть не является необходимой, поскольку return
         // завершает выполнение программы
   {
      max = foot[0];
      for (int i = 1; i < size; i++)
         if (foot[i] > max)
            max = foot[i];
      return max;
   }
}
```

6. Вы используете классификатор const вместе с указателями для защиты от изменения указываемых исходных данных. Когда программа передает базовый тип, например, int или double, она передает его по значению, поэтому функция работает с копией. Следовательно, исходные данные уже являются защищенными.

7. Строку можно сохранить в массиве char, ее можно представить строковой константой, заключив в двойные кавычки, а еще можно представить посредством указателя, указывающего на первый символ строки.

8.

```
int replace(char * str, char c1, char c2)
{
   int count = 0;
   while (*str) // пока не достигнут конец строки
   {
```

```
        if (*str == c1)
        {
           *str = c2;
           count++;
        }
        str++; // переход к следующему символу
     }
     return count;
  }
```

9. Поскольку C++ интерпретирует "pizza" как адрес ее первого элемента, выполнение операции * даст значение этого первого элемента, который представлен буквой p. Так как C++ интерпретирует строку "taco" как адрес ее первого элемента, то "taco"[2] будет трактоваться как значение элемента, расположенного во второй позиции справа (буква c). Другими словами, строковая константа действует точно так же, как и имя массива.

10. Чтобы передать структуру по значению, вы просто передаете имя структуры glitz. Чтобы передать ее адрес, вы используете адресную операцию &glitz. Передача по значению автоматически защищает исходные данные, однако отнимает время и расходует память. Передача по адресу экономит время и память, но не защищает исходные данные, если только вы не будете использовать для параметра функции модификатор const. Кроме того, передача по значению означает, что вы можете применять обычное обозначение члена структуры, а передача указателя означает необходимость использования косвенной операции членства.

11. `int judge (int (*pf)(const char *));`

Ответы на вопросы для самоконтроля из главы 8

1. В качестве встроенных функций удобно использовать короткие нерекурсивные функции, которые могут занимать одну строку кода.

2. а. `void song(char * name, int times = 1);`

   б. Ни одна из них. Только прототипы содержат информацию о значениях по умолчанию.

   в. Да, при условии, что вы сохраните значение по умолчанию для times:

   `void song(char * name = "O, My Papa", int times = 1);`

3. Для вывода кавычки можно использовать либо строку "\"", либо символ '"'. Следующие функции демонстрируют оба способа:

```
#include <iostream.h>
void iquote(int n)
{
   cout << "\"" << n << "\"";
}
void iquote(double x)
{
   cout << '"' << x << '"';
}
```

```
void iquote(const char * str)
{
    cout << "\"" << str << "\"";
}
```

4.

а. Эта функция не может изменять члены структуры, поэтому используйте классификатор const:

```
void show_box(const box & container)
{
 cout << "Made by " << container.maker << endl;
 cout << "Height = " << container.height << endl;
 cout << "Width = " << container.width << endl;
 cout << "Length = " << container.length << endl;
 cout << "Volume = " << container.volume << endl;
}
```

б.

```
void set_volume(box & crate)
{
 crate.volume = crate.height * crate.width * crate.length;
}
```

5.

а. Это можно сделать с помощью значения по умолчанию для второго аргумента:

```
double mass(double d, double v = 1.0);
```

Это можно сделать также с помощью перегрузки функции:

```
double mass(double d, double v);
double mass(double d);
```

б. Значение по умолчанию нельзя использовать для повторения значения, потому что вам необходимо предусмотреть значения по умолчанию справа налево. Вы можете воспользоваться перегрузкой:

```
void repeat(int times, const char * str);
void repeat(const char * str);
```

в. Можно использовать перегрузку функции:

```
int average(int a, int b);
double average(double x, double y);
```

г. Этого сделать нельзя, поскольку оба варианта будут иметь одну и ту же сигнатуру.

6.

```
template<class T>
T max(T t1, T t2) // или T max(const T & t1, const T & t2)
{
    return t1 > t2? t1 : t2;
}
```

7.

```
template<> box max(box b1, box b2)
{
    return b1.volume > b2.volume? b1 : b2;
}
```

Ответы на вопросы для самоконтроля из главы 9

1.

а. homer автоматически становится автоматической переменной.

б. secret необходимо определять как внешнюю переменную в одном файле, а в другом файле ее необходимо объявлять с применением extern.

в. topsecret можно определить как статическую переменную с внешним связыванием, добавляя перед внешним определением служебное слово static. Либо ее можно определить в неименованном пространстве имен.

г. beencalled следует определить как локальную статическую переменную, добавляя перед объявлением в функции служебное слово static.

2. Объявление using делает доступным одиночное имя из пространства имен и имеет область видимости, соответствующую объявленной области, в которой встречается объявление using. Директива using делает доступными все имена в пространстве имен. Если вы применяете директиву using, то это равносильно тому, что вы объявили имена в самой маленькой объявленной области, содержащей как объявление using, так и само пространство имен.

3.

```
#include <iostream>
int main()
{
    double x;
    std::cout << "Введите значение: ";
    while (! (std::cin >> x) )
    {
        std::cout << "Недопустимый ввод. Пожалуйста, введите число: ";
        std::cin.clear();
        while (std::cin.get() != '\n')
            continue;
    }
    std::cout << "Значение = " << x << std::endl;
    return 0;
}
```

4. Переписанный код:

```
#include <iostream>
int main()
{
    using std::cin;
    using std::cout;
    using std::endl;
    double x;
    cout << "Введите значение: ";
    while (! (cin >> x) )
    {
        cout << "Недопустимый ввод. Пожалуйста, введите число: ";
        cin.clear();
```

```
        while (cin.get() != '\n')
            continue;
    }
    cout << "Значение = " << x << endl;
    return 0;
}
```

5. В каждом файле у вас могут существовать отдельные определения статической функции. Либо же каждый файл может определять соответствующую функцию `average()` в неименованном пространстве имен.

6.
```
10
4
0
Other: 10, 1
another(): 10, -4
```

7.
```
1
4, 1, 2
2
2
4, 1, 2
2
```

Ответы на вопросы для самоконтроля из главы 10

1. Класс представляет собой определение типа, определяемого пользователем. Объявление класса описывает способ хранения данных и методы (функции-члены класса), которые могут использоваться для доступа к этим данных и манипулирования этими данными.

2. Класс представляет операции, которые можно выполнять над объектом класса посредством общедоступного интерфейса методов класса; это абстракция. Класс может использовать приватную видимость (по умолчанию) для данных-членов, а это означает, что доступ к данным может быть осуществлен только через функции-членов; это сокрытие данных. Детальная часть реализации, например, представление данных и код метода, сокрыты; это инкапсуляция.

3. Класс определяет тип, включая информацию о том, как он может быть использован. Объект представляет собой переменную или другой объект данных, который, например, создается с помощью операции `new`; объект создается и используется в соответствии с определением класса. Между классом и объектом существует такая же связь, как и между стандартным типом и переменной этого типа.

4. Если вы создаете несколько объектов данного класса, то каждый объект будет обладать пространством памяти для хранения своего собственного набора данных. Однако все объекты используют один набор функций-членов. (Обычно методы являются общедоступными, а данные-члены — приватными, однако это вопрос политики, а не требований к классу.) Примечание: в программе исполь-

зуется cin.get(*char *, int) вместо cin >> для чтения имен, поскольку cin.get() считывает всю строку сразу, а не только одно слово (см. главу 4).

5. В этом примере используются массивы char для хранения символьных данных, однако вы можете применять объекты класса string.

```
// #include <cstring>
// class definition
class BankAccount
{
private:
 char name[40]; // или std::string name;
 char acctnum[25]; или or std::string acctnum;
 double balance;
public:
 BankAccount(const char * client, const char * num, double bal = 0.0);
 // или BankAccount(const std::string & client,
 // const std::string & num, double bal = 0.0);
 void show(void) const;
 void deposit(double cash);
 void withdraw(double cash);
};
```

6. Вызов конструктора класса производится тогда, когда вы создаете объект этого класса или когда вы явным образом обращаетесь к конструктору. Деструктор класса вызывается после завершения работы с объектом.

7. Ниже показаны два возможных решения (учтите, что вы должны включить cstring или string.h, чтобы использовать strncpy(), или включить string для использования класса string):

```
{
   strncpy(name, client, 39);
   name[39] = '\0';
   strncpy(acctnum, num, 24);
   acctnum[24] = '\0';
   balance = bal;
}
```

или

```
BankAccount::BankAccount(const std::string & client,
const std::string & num, double bal)
{
   name = client;
   acctnum = num;
   balance = bal;
}
```

Имейте в виду, что аргументы по умолчанию присутствуют в прототипе, а не в определении функции.

8. Конструктор по умолчанию либо не имеет аргументов, либо имеет значения по умолчанию для всех аргументов. Работая с конструктором по умолчанию, вы можете объявлять объекты без их инициализации, даже если вы уже определили инициализирующий конструктор. Он также позволяет объявлять массивы.

9.

```
// stock3.h
#ifndef STOCK3_H_
#define STOCK3_H_
class Stock
{
private:
    std::string company;
    int shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    Stock(); // конструктор по умолчанию
    Stock(const std::string & co, int n, double pr);
    ~Stock() {} // ничего не делающий деструктор
    void buy(int num, double price);
    void sell(int num, double price);
    void update(double price);
    void show() const;
    const Stock & topval(const Stock & s) const;
    int numshares() const { return shares; }
    double shareval() const { return share_val; }
    double totalval() const { return total_val; }
    string co_name() const { return company; }
};
```

10. Указатель `this` — это указатель, доступный методам класса. Он указывает на объект, который был использован для вызова метода. Таким образом, `this` является адресом объекта, а `*this` представляет сам объект.

Ответы на вопросы для самоконтроля из главы 11

1. Ниже показан прототип для файла определения класса и определение функции для файла методов:

```
// прототип
Stonewt operator*(double mult);
// определение — позволяет конструктору выполнять свои функции
Stonewt Stonewt::operator*(double mult)
{
    return Stonewt(mult * pounds);
}
```

2. Функция-член является частью определения класса и вызывается определенным объектом. Функция-член может обращаться к членам вызывающего объекта явным образом, не используя операцию принадлежности. Дружественная функция не является частью класса, поэтому она называется функцией прямого вызова. Она не может обращаться к членам класса явным образом, поэтому она должна использовать операцию принадлежности, примененную к объекту, который передается в виде аргумента. Сравните, например, ответы на первый и четвертый вопросы.

3. Чтобы обращаться к приватным членам, она должна быть дружественной, и не должна быть таковой для доступа к общедоступным членам.
4. Ниже показан прототип для файла определения класса и определение функции для файла методов:

```
// прототип
friend Stonewt operator*(double mult, const Stonewt & s);
// определение — позволяет конструктору выполнять свои функции
Stonewt operator*(double mult, const Stonewt & s)
{
   return Stonewt(mult * s.pounds);
}
```

5. Следующие пять операций не могут быть перегружены:

```
sizeof
.
.*
::
? :
```

6. Эти операции необходимо определять с использованием функций-членов.
7. Ниже показан возможный вариант прототипа и определения:

```
// прототип и подставляемое определение
operator double () {return mag;}
```

Обратите, однако, внимание, что лучше всего использовать метод `magval()`, чем определять эту функцию преобразования.

Ответы на вопросы для самоконтроля из главы 12

1. а. Синтаксис подходящий, однако, этот конструктор оставляет неинициализированным указатель `str`. Конструктор должен либо присвоить указателю `NULL`, либо использовать операцию `new[]` для инициализации указателя.

   б. Этот конструктор не создает новую строку, а просто копирует адрес старой строки. Следует использовать `new[]` и `strcpy()`.

   в. Он копирует строку, не выделяя память для ее хранения. Чтобы выделить соответствующий объем памяти, необходимо использовать `new char[len + 1]`.
2. Во-первых, когда объект этого типа прекращает работу, данные, на которые указывает член-указатель объекта, остаются в памяти, занимая пространство и оставаясь недоступными, поскольку указатель был утрачен. Эту ситуацию можно исправить, если использовать деструктор класса, который освободит память, выделенную операцией `new` в функциях конструкторов. Во-вторых, после того как деструктор освободит эту память, он не станет освобождать ее еще раз, если программа будет инициализировать один такой объект другим объектом. Это объясняется тем, что при инициализации одного объекта другим по умолчанию копируются значения указателя, и не копируются данные, на которые указывает указатель, в результате чего появляются два указателя на одни и те же данные. Выход заключается в том, чтобы определить конструктор

копирования класса, благодаря которому при инициализации копировались бы данные, представляющие копию данных, на которые указывает указатель. В-третьих, при присваивании одного объекта другому может произойти аналогичная ситуация, когда два указателя будут указывать на одни и те же данные. Выход заключается в перегрузке операции присваивания, в результате чего будут копироваться данные, а не указатели.

3. C++ автоматически предлагает следующие функции-члены:
   - Конструктор по умолчанию, если вы не определяете конструкторы.
   - Конструктор копирования, если вы его не определяете.
   - Операцию присваивания, если вы ее не определяете.
   - Деструктор по умолчанию, если вы его не определяете.
   - Операцию взятия адреса, если вы ее не определяете.

   Конструктор по умолчанию ничего не делает, однако позволяет объявлять массивы и неинициализированные объекты. Конструктор копирования и операция присваивания, предлагаемые по умолчанию, используют почленное присваивание. Деструктор по умолчанию ничего не делает. Неявная операция взятия адреса возвращает адрес вызывающего объекта (то есть значение указателя `this`).

4. Член `personality` необходимо объявлять либо как массив символов, либо как указатель на тип `char`. Или вы можете создать на его основе объект `String` или объект `string`. Объявление не может сделать методы общедоступными. В коде имеются также незначительные ошибки. Ниже показан возможный вариант решения, в котором изменения (а не удаленные части кода) выделены полужирным:

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private: // необязательная часть
    char personality[40]; // определяет размер массива
    int talents;
public: // обязательная часть
// методы
    nifty();
    nifty(const char * s);
    friend ostream & operator<<(ostream & os, const nifty & n);
}; // обратите внимание на завершающую точку с запятой
nifty::nifty()
{
    personality[0] = '\0';
    talents = 0;
}
nifty::nifty(const char * s)
{
    strcpy(personality, s);
    talents = 0;
}
ostream & operator<<(ostream & os, const nifty & n)
{
    os << n.personality << '\n';
```

```
    os << n.talent << '\n';
    return os;
}
```

А вот еще один возможный вариант решения:

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private: // необязательная часть
    char * personality; // создает указатель
    int talents;
public: // обязательная часть
    // методы
    nifty();
    nifty(const char * s);
    nifty(const nifty & n);
    ~nifty() { delete [] personality; }
    nifty & operator=(const nifty & n) const;
    friend ostream & operator<<(ostream & os, const nifty & n);
}; // обратите внимание на завершающую точку с запятой
nifty::nifty()
{
    personality = NULL;
    talents = 0;
}
nifty::nifty(const char * s)
{
    personality = new char [strlen(s) + 1];
    strcpy(personality, s);
    talents = 0;
}
ostream & operator<<(ostream & os, const nifty & n)
{
    os << n.personality << '\n';
    os << n.talent << '\n';
    return os;
}
```

5. а.

```
Golfer nancy;                      // конструктор по умолчанию
Golfer lulu("Little Lulu");        // Golfer(const char * name, int g)
Golfer roy("Roy Hobbs", 12);       // Golfer(const char * name, int g)
Golfer * par = new Golfer;         // конструктор по умолчанию
Golfer next = lulu; // Golfer(const Golfer &g)
Golfer hazard = "Weed Thwacker";  // Golfer(const char * name, int g)
*par = nancy;                // операция присваивания по умолчанию
nancy = "Nancy Putter";   // Golfer(const char * name, int g), then
// операция присваивания по умолчанию
```

Обратите внимание, что некоторые компиляторы дополнительно вызывают операцию присваивания по умолчанию для операторов # 5 и #6.

б. Класс должен определять операцию присваивания, при которой будут копироваться данные, а не адреса.

Ответы на вопросы для самоконтроля из главы 13

1. Общедоступные члены базового класса становятся общедоступными членами производного класса. Защищенные члены базового класса становятся защищенными членами производного класса. Приватные члены базового класса наследуются, но к ним не может быть осуществлен непосредственный доступ. В ответе на вопрос 2 рассматриваются исключения из этих общих правил.

2. Методы конструктора не наследуются, деструктор не наследуется, операция присваивания не наследуется, и дружественные функции не наследуются.

3. Если бы возвращаемым типом был void, вы могли бы использовать одну операцию присваивания, а не последовательность операций:

   ```
   baseDMA magazine("Pandering to Glitz", 1);
   baseDMA gift1, gift2, gift3;
   gift1 = magazine;          // нормально
   gift 2 = gift3 = gift1;    // больше не является допустимым
   ```

   Если метод возвращает объект, а не ссылку, то выполнение метода может быть немного замедлено, поскольку оператор возврата может включать копирование объекта.

4. Конструкторы вызываются в порядке порождения, и первым вызывается самый первый конструктор. Деструкторы вызываются в обратном порядке.

5. Да, для каждого класса требуются его собственные конструкторы. Если производный класс не добавляет новых членов, то конструктор может иметь пустое тело, но обязательно должен существовать.

6. Вызывается только метод производного класса. Он заменяет определение базового класса. Метод базового класса вызывается только тогда, когда производный класс не переопределяет метод или когда вы используете операцию разрешения контекста. Однако в действительности вы должны объявлять виртуальной любую функцию, которая будет переопределена.

7. Производный класс должен определять операцию присваивания, если конструкторы производного класса используют операцию new или new [] для инициализации указателей, которые являются членами данного класса. В общем случае производный класс должен определять операцию присваивания, если присваивание, предлагаемое по умолчанию, не подходит для членов производного класса.

8. Да, вы можете присвоить адрес объекта производного класса указателю на базовый класс. Вы можете присвоить адрес объекта базового класса указателю на производный класс (нисходящее преобразование) только путем явного приведения типа; использование такого указателя не всегда безопасно.

9. Да, вы можете присвоить объект производного класса объекту базового класса. Однако любые члены данных, которые являются новыми по отношению к производному типу, не передаются базовому типу. Программа использует операцию присваивания базового класса. Присваивание в обратном порядке (по отноше-

нию к производному) возможно, только если производный класс определяет операцию преобразования, которая является конструктором, имеющим ссылку на базовый тип в виде своего единственного аргумента, или если он определяет операцию присваивания с параметром базового класса.

10. Она может сделать это, поскольку в C++ разрешается ссылаться на базовый тип для ссылки на любой тип, который является производным от данного базового типа.

11. Передача объекта по значению активизирует конструктор копирования. Так как формальный аргумент является объектом базового класса, то вызывается конструктор копирования базового класса. Конструктор копирования получает в качестве своего аргумента ссылку на базовый класс, и эта ссылка может указывать на производный объект, передаваемый в качестве аргумента. Смысл состоит в том, что таким образом создается новый объект базового класса, члены которого соответствуют части базового класса производного объекта.

12. Передача объекта по ссылке вместо передачи по значению позволяет функции воспользоваться виртуальными функциями. Кроме того, передача объекта по ссылке вместо передачи по значению может расходовать меньше памяти и занимать меньше времени, особенно для больших объектов. Главное преимущество передачи по значению заключается в том, что при этом защищаются исходные данные, однако то же самое можно сделать и с помощью передачи ссылки как типа `const`.

13. Если `head()` является обычным методом, то `ph->head()` активизирует `Corporation::head()`. Если `head()` является виртуальной функцией, то `ph->head()` активизирует `PublicCorporation::head()`.

14. Во-первых, ситуация не соответствует модели *is-a*, поэтому общедоступное наследование не подходит. Во-вторых, определение `area()` в `House` скрывает вариант `Kitchen area()`, поскольку оба метода имеют разные сигнатуры.

Ответы на вопросы для самоконтроля из главы 14

1.

`class Bear`	`class PolarBear`	Общедоступный; полярный медведь (polar bear) является разновидностью медведя (bear).
`class Kitchen`	`class Home`	Приватный; в доме (home) имеется кухня (kitchen).
`class Person`	`class Programmer`	Общедоступный; программист (programmer) — это человек (person).
`class Person`	`class HorseAndJockey`	Приватный; в связке лошадь-жокей (horse-jockey) присутствует человек (person).
`class Person,` `class Automobile`	`class Driver`	`Person` является общедоступным классом, поскольку водитель (driver) — это человек (person); `Automobile` является приватным, поскольку у водителя есть автомобиль (automobile).

2.

```
Gloam::Gloam(int g, const char * s) : glip(g), fb(s) { }
Gloam::Gloam(int g, const Frabjous & fr) : glip(g), fb(fr) { }
// примечание: в первых строках использовался конструктор
копирования Frabjous по умолчанию
void Gloam::tell()
{
   fb.tell();
   cout << glip << endl;
}
```

3.

```
Gloam::Gloam(int g, const char * s)
      : glip(g), Frabjous(s) { }
Gloam::Gloam(int g, const Frabjous & fr)
      : glip(g), Frabjous(fr) { }
// примечание: в первых строках использовался конструктор
// копирования Frabjous по умолчанию
void Gloam::tell()
{
   Frabjous::tell();
   cout << glip << endl;
}
```

4.

```
class Stack<Worker *>
{
private:
   enum {MAX = 10};       // специфичная для класса константа
   Worker * items[MAX];  // хранит элементы стека
   int top;               // индекс вершины стека
public:
   Stack();
   Boolean isempty();
   Boolean isfull();
   Boolean push(const Worker * & item); // добавляет элемент в стек
   Boolean pop(Worker * & item);//выталкивает элемент с вершины стека
};
```

5.

```
ArrayTP<string> sa;
StackTP< ArrayTP<double> > stck_arr_db;
ArrayTP< StackTP<Worker *> > arr_stk_wpr;
```

В листинге 14.18 генерируется четыре шаблона: `ArrayTP<int, 10>`, `ArrayTP<double, 10>`, `ArrayTP<int,5>` **и** `Array<ArrayTP<int,5>, 10>`.

6. **Если два пути наследования класса имеют общего предка, то класс будет иметь две копии членов этого предка. Проблема можно решить, если класс предка сделать виртуальным базовым классом по отношению к его непосредственным наследникам.**

Ответы на вопросы для самоконтроля из главы 15

1.

   а. Объявление дружественного класса должно выглядеть следующим образом:

   ```
   friend class clasp;
   ```

   б. Для этого необходимо опережающее объявление, с тем чтобы компилятор мог интерпретировать `void snip(muff &)`:

   ```
   class muff; // опережающее объявление
   class cuff {
   public:
   void snip(muff &) { ... }
     ...
   };
   class muff {
     friend void cuff::snip(muff &);
     ...
   };
   ```

   в. Во-первых, объявление класса `cuff` должно предшествовать классу `muff`, с тем чтобы компилятор мог понять термин `cuff::snip()`. Во-вторых, компилятору необходимо опережающее объявление `muff`, чтобы он мог понять `snip(muff &)`:

   ```
   class muff; // опережающее объявление
   class cuff {
   public:
     void snip(muff &) { ... }
     ...
   };
   class muff {
     friend void cuff::snip(muff &);
     ...
   };
   ```

2. Нет. Чтобы класс `A` имел дружественный класс, являющийся функцией-членом класса `B`, объявление `B` должно предшествовать объявлению `A`. Опережающего объявления не будет достаточно, поскольку оно не может показать классу `A`, что `B` является классом, однако оно не будет показывать имена членов класса. Аналогично, если `B` имеет друга, который является функцией-членом `A`, то итоговое объявление `A` должно предшествовать объявлению `B`. Оба эти требования исключают друг друга.

3. Доступ к классу возможен только через его общедоступный интерфейс; это означает, что единственное, что вы можете сделать с объектом `Sauce`, это вызвать конструктор для его создания. Другие члены (`soy` и `sugar`) являются приватными по умолчанию.

4. Предположим, что функция `f1()` вызывает функцию `f2()`. Оператор возврата в `f2()` возобновляет выполнение программы в следующем операторе после вызова функции `f2()` в функции `f1()`. Оператор `throw` возвращает программу через существующую последовательность вызовов функций до блока `try`, кото-

рый прямо или косвенно будет содержать вызов функции f2(). Он может находиться в f1() или в функции, вызывающей функцию f2(), и так далее. Отсюда выполнение передается следующему совпавшему блоку catch, а не первому оператору после вызова функции.

5. Блоки catch необходимо упорядочить от наиболее глубокого в цепочке наследования до наименее глубокого.

6. В примере # 1 условие if будет равно true, если pg будет указывать на объект Superb или на объект любого класса, происходящего от Superb. В частности, оно также будет равно true, если pg будет указывать на объект Magnificient. В примере # 2 условие if будет равно true только для объекта Superb, а не для объектов, происходящих от Superb.

7. Операция dynamic_cast позволяет выполнять только восходящее преобразование типов в иерархии класса, а операция static_cast — и восходящее, и нисходящее преобразование типов. Операция static_cast также позволяет выполнять преобразование перечислимых типов в целочисленные и наоборот, а также преобразование между различными числовыми типами.

Ответы на вопросы для самоконтроля из главы 16

1.

```
#include <string>
using namespace std;
class RQ1
{
private:
   string st; // объект string
public:
   RQ1() : st("") {}
   RQ1(const char * s) : st(s) {}
   ~RQ1() {};
   // дополнительный код
};
```

Явный конструктор копирования, деструктор и операция присваивания больше не нужны, поскольку объект string самостоятельно управляет памятью.

2. Можно присвоить один объект string другому. Объект string самостоятельно управляет памятью, поэтому обычно не нужно заботиться о превышении строкой объема памяти, которым обладает содержащий ее объект.

3.

```
#include <string>
#include <cctype>
using namespace std;
void ToUpper(string & str)
{
   for (int i = 0; i < str.size(); i++)
      str[i] = toupper(str[i]);
}
```

4.

```
auto_ptr<int> pia= new int[20];   // неправильно, необходимо
                                  // использовать new, а не new[]
auto_ptr<string>(new string);     // неправильно, отсутствует имя
                                  // указателя
int rigue = 7;
auto_ptr<int>(&rigue);            // неправильно, память не выделена
                                  // с помощью операции new
auto_ptr dbl (new double);        // неправильно, пропущен <double>
```

5. Принцип LIFO в стеке означает, что вам, возможно, придется удалить множество клюшек для гольфа, прежде чем будет найдена необходимая.

6. Последовательность будет содержать только одну копию каждого значения, поэтому, скажем, пять счетов 5 будут храниться как одно число 5.

7. Итераторы позволят использовать объекты с интерфейсом наподобие указателей для перемещения по данным, организованным не в виде массива (например, данные в двусвязном списке).

8. Функции библиотеки STL могут использоваться с обычными указателями на порядковые массивы и с итераторами на классы контейнеров STL; этим доказывается их универсальность.

9. Можно присвоить один объект `vector` другому. Объект `vector` управляет своей собственной памятью, поэтому вы можете вставлять элементы в вектор, а он будет автоматически изменять свои размеры. С помощью метода `at()` можно инициировать автоматическую проверку границ.

10. Двум функциям `sort()` и функции `random_shuffle()` необходим итератор произвольного доступа, а объект `list` имеет двунаправленный итератор. Для сортировки можно использовать функции-члены `sort()` шаблонного класса списка (см. приложение Ж) вместо функций общего назначения, однако функции-члена, эквивалентной `random_shuffle()`, не существует. Тем не менее, вы можете копировать список в вектор, перетасовать вектор, и скопировать результат обратно в список.

Ответы на вопросы для самоконтроля из главы 17

1. Файл `iostream` определяет классы, константы и манипуляторы, которые используются для управления процессами ввода и вывода. Эти объекты управляют потоками и буферами, применяемыми для обработки ввода-вывода. Этот файл также создает стандартные объекты (`cin`, `cout`, `cerr` и `clog` и их эквиваленты для широких символов), которые используются для обработки стандартных потоков ввода и вывода, связанных с каждой программой.

2. При вводе с клавиатуры генерируется последовательность символов. При вводе **121** генерируется три символа, и каждый из них представляется однобайтным двоичным кодом. Если значение необходимо сохранить как `int`, то эти три символа должны быть преобразованы в одно двоичное представление числа 121.

3. По умолчанию стандартный вывод и стандартные ошибки отправляют вывод на стандартное устройство вывода, которым обычно является монитор. Однако если ваша операционная система перенаправит вывод в файл, то стандартный вывод будет связан с файлом, а не с экраном, а стандартные ошибки — с экраном.

4. Класс `ostream` определяет версию функции `operator<<()` для каждого базового типа C++. Компилятор интерпретирует выражение вроде

   `cout << spot`

   следующим образом:

   `cout.operator<<(spot)`

   Затем он может сопоставить этот вызов метода с прототипом функции, имеющей такой же тип аргумента.

5. Можно связать методы вывода, возвращающие тип `ostream &`. В результате метод будет вызываться посредством объекта для возврата этого объекта. Возвращенный объект впоследствии может активизировать следующий метод в последовательности.

6.

```
//rq17-6.cpp
#include <iostream>
#include <iomanip>
int main()
{
   using namespace std;
   cout << "Введите целое число: ";
   int n;
   cin >> n;
   cout << setw(15) << "основание 10" << setw(15)
        << "основание 16" << setw(15) << "основание 8" << "\n";
   cout.setf(ios::showbase); // или cout << showbase;
   cout << setw(15) << n << hex << setw(15) << n
        << oct << setw(15) << n << "\n";
   return 0;
}
```

7.

```
//rq17-7.cpp
#include <iostream>
#include <iomanip>
int main()
{
   using namespace std;
   char name[20];
   float hourly;
   float hours;
   cout << "Введите ваше имя: ";
   cin.get(name, 20).get();
   cout << "Ваш почасовой заработок: ";
   cin >> hourly;
   cout << "Количество отработанных часов: ";
   cin >> hours;
   cout.setf(ios::showpoint);
   cout.setf(ios::fixed, ios::floatfield);
```

```
    cout.setf(ios::right, ios::adjustfield);
    // или cout << showpoint << fixed << right;
    cout << "Первый формат:\n";
    cout << setw(30) << name << ": $" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";
    cout << "Второй формат:\n";
    cout.setf(ios::left, ios::adjustfield);
    cout << setw(30) << name << ": $" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";
    return 0;
}
```

8. Результат выполнения выглядит следующим образом:

   ```
   ct1 = 5; ct2 = 9
   ```

   Первая часть программы игнорирует пробелы и символы новой строки, а вторая часть — нет. Обратите внимание, что вторая часть программы начинает чтение с символа новой строки, следующим за первым `q`, и воспринимает символ новой строки как часть строки.

9. `ignore()` будет порождать сбойные ситуации, если строка ввода превысит 80 символов. В этом случае будут пропущены только первые 80 символов.

Предметный указатель

A

ADT (abstract data type), 519
ASCII (American Standard Code for Information Interchange), 1065

C

Символы ASCII, 1065
Спецификатор const, 445

F

FIFO, 635

I

Integrated Development Environment (IDE), 44

L

LIFO (last-in, first-out), 432; 449; 635

N

NBTS (null-byte-terminated string), 880

R

RTTI (Run Time Type Identification), 863

S

STL (Standard Template Library), 899; 955; 1103

U

Unified Modeling Language — UML, 1049

А

Адаптер, 920
Адрес функции, 355
Алгоритм, 36
Аргумент, 57; 76; 314
  действительный, 314
  определяемый по умолчанию, 391
  передача с использованием стека, 433
  ссылочный, 376
  формальный, 314
Ассоциативность, 124

Б

Байт, 95
Библиотека, 40
  стандартная C, 42
  символьных функций cctype, 275
  шаблонов,
    STL, 899; 1103
Бинарные файлы, 1033
Бит, 95
Блок, 221
  catch, 834
  try, 834
  перехвата, *См.* Обработчик исключений, 834
Буфер, 973
  сброс, 974; 984
Бьерн Страуструп, 39

В

Ввод
  строковый, 1014
Ввод-вывод, 290
  простой файловый, 290
  текстовый, 290
Вектор, 551; 899
Виртуальные базовые классы, 759
Включение, *См.* Композиция, 731
Возврат кода ошибки, 832
Время
  выполнения, 173
  компиляции, 173
Вызов функции, 76
Выражение, 122
  логическое, 266

Г

Генератор, 943

Д

Данные, 36
Декорирование имен, 399
Деннис Ритчи, 35
Деструктор, 498
  класса, 493
Динамическая идентификация типов, 863
Динамическая память, 588
Динамическое распределение памяти, 449; 705
Динамическое связывание, 688
Директива
  include, 121
  using, 61; 85; 457
Директива препроцессора
  #define, 424
  #endif, 424
  #ifndef, 424
Доминирование, 770

З

Защищенное наследование, 751

И

Иерархическое представление, 731
Имя
  зарезервированное, 1062
Индекс, 139
Инициализация, 79; 98
Инкапсуляция, 481
Интегрированная среда разработки, 44
Интеллектуальный указатель, 897; 898
Интерфейс, 479
Исключение, 834
  неперехваченное, 858
  непредвиденное, 858
Исходный код, 44
Итератор, 899; 910
  входной, 915
  выходной, 916
  двунаправленный, 917
  иерархия итераторов, 917
  однонаправленный, 916
  произвольного доступа, 917

К

Квалификатор const, 115
Класс, 37; 63; 73; 475; 478
  Allocator, 894
  auto_ptr, 894
  Brass, 684
  Customer, 646
  exception, 849
  ios_base, 987; 996
  ostream, 979; 987
  Queue, 635
  Stock, 482
  String, 153; 606; 879; 952; 1085
  traits, 894
  valarray, 732; 959
  Vector, 551; 899; 959
  абстрактный базовый, 698
  базовый, 662
  виртуальный
    базовый, 759
  вложенный, 823
  наследование, 662
  область видимости класса, 517
  порождение класса, 664
  производный, 662
  статический член класса, 589
  шаблон, 772
  шаблонный, 797
    vector, 142
Классификатор, 445
Клиент-серверная модель, 492
Ключевое слово, 83
  auto, 445
  extern, 445; 447
  register, 445
  static, 445
  volatile, 445
Код
  исходный, 44
  обобщенный, 39
Комментарий, 58
Компилятор, 36; 46
  CC, 46
Композиция, 731
Компоновщик, 47
Конкатенация
  вывода, 981
  строк с помощью cout, 73

Константа
char, 109
перечислитель, 169
с плавающей точкой, 121
целочисленная, 103
Конструктор, 506; 666; 880
класса, 493
string, 881
по умолчанию, 496
копирования, 597
по умолчанию, 597
Контейнер, 899; 926
ассоциативный, 936
обратимый, 930
Координаты
полярные, 345
прямоугольные, 344
Копирование
глубокое, 602
поверхностное, 600
почленное, 600
Куча, 198

Л

Лексема, 66
альтернативная, 1062
Литерал
строковый, 338
Логическое выражение, 266

М

Манипулятор, 64; 987
Массив, 102; 138
двумерный, 252
динамический, 182
имя массива, 190
нотация массивов, 190
объектов, 513
структур, 165
Метод
erase(), 902
insert(), 903
виртуальный, 693
шаблонный, 945
Механизм динамической идентификация типов, 863
Множественное наследование, 743; 753
с совместно используемым предком, 754

Н

Набор символов ASCII, 1065
Наследование, 387; 673; 705
защищенное, 751
классов, 662
множественное, 743; 753; 771
с совместно используемым предком, 754
полиморфное общедоступное, 675
приватное, 742; 745
с виртуальными базовыми классами, 759
Нулевой символ, 142

О

Область видимости, 428; 825
класса, 517
Обобщенное программирование, 879; 910
Обработчик исключений, 834
Объединение, 167
анонимное, 168
Объект, 37; 63; 480
Объектно-ориентированное программирование, 476; 481
Объекты функций, 899
Объявление
using, 457; 752; 753
опережающее, 819
Операнд, 122
Оператор
break, 284
continue, 284
if, 259
if else, 261
switch, 279
возврата, 56
объявления, 67
присваивания, 67
составной, 221
Операции приведения типов, 872
Операционная система, 35
Операция
?:, 277
const_cast, 873
delete, 181
dynamic_cast, 864
new, 178; 852
с адресацией, 450

reinterpret_cast, 875
static_cast, 875
ассоциативность, 1070
бинарная, 562; 1069
битовая, 1073
вставки, 979
декремента (–), 217
запятой, 223
ИЛИ (||), 266; 1076
И (&&), 267; 1077
инкремента (++), 217
логического отрицания, 1075
НЕ (!), 272
последовательная, 1110
приоритет, 1069
присваивания, 70
разрешения контекста (::), 438
разыменования, 174; 1079
сдвига, 1073
унарная, 562; 1069
Опережающее объявление, 819
Отношение
has-a, 731; 742
is-a, 673

П

Память
динамическая, 449; 588
Параметр, *См.* Аргумент, 76
-выражение, 785
действительный, 314
нетипизированный, 785
передача параметров по ссылке, 372
формальный, 314
Перегрузка
операций, 63; 530
операции <<, 544; 633
функций, 151
Передача параметров по ссылке, 342; 372
Переменная, 92
автоматическая, 315
булевская, 114
внешняя, 436
временная, 377
глобальная, 437
имя переменной, 94
инициализация, 438
локальная, 315
объявление, 69; 438
определение, 438
регистровая, 433
ссылочная, 368
статическая, 439; 443
Переносимость, 40
Перестановка, 953; 1134
Перечисление, 169
диапазон, 171
Подсчет ссылок, 898
Последовательность, 928
Почленное присваивание, 164
Предикат, 943
бинарный, 943
Препроцессор, 60
Приватное наследование, 742; 745
Приведение типов, 131
Приоритет, 124
операций, 1069
Пробел
обобщенный, 66
Программа
переносимая, 40
Программирование
восходящее, 38
обобщенное, 39; 400; 879; 910
сверху вниз, 330
снизу вверх, 330
структурное, 37
Проектирование, 37
нисходящее, 37
сверху вниз, 37
Произвольный доступ, 1039
Пространство имен, 61
глобальное, 456
конфликты, 454
область видимости, 455
область объявления, 454
Прототип функции, 312
Псевдонимы типов, 239

Р

Разрядность, 94
Расширения исходного кода, 45
Редактор связей, 47
Рекурсия, 352

С

Связывание, 182; 428
  динамическое, 182; 189
  статическое, 182; 189
  языковое, 448
Сигнальный символ, 242
Сигнатура функции, 395
Символическая константа, 99
Символ новой строки, 65
Система счисления, 1057
  восьмеричная, 1057
  двоичная, 1058
  десятичная, 1057
  шестнадцатеричная, 1058
Сокрытие данных, 481; 492
Сортировка
  полным упорядочением, 908
  строгим квазиупорядочением, 908
Состояние потока, 1005
Спецификатор, 445
  const, 377
Список аргументов, 57
Ссылка, 368
Статический контроль типов, 314
Статическое связывание, 688
Стек, 432
  раскручивание стека, 841
  указателей, 778; 780
Строка, 142
  работа со строками, 886
  сравнение строк с использованием
    класса string, 232
Строковый ввод, 1014
Строковый литерал, 338
Структура, 159
  внешнее объявление, 162
  динамическая, 195
  локальное объявление, 162
  массивы структур, 165
  почленное присваивание, 164
  член структуры, 160
Структурное программирование, 37
Субобъект, 742

Т

Таблица виртуальных функций, 691
Терминатор, 56
Тип данных, 94; 477
  bool, 122; 114
  char, 106; 477
  double, 122; 477
  int, 95; 477
  float, 122
  long, 95
  long double, 122
  short, 95
  абстрактный, 519
  автоматическое преобразование, 567
  арифметический, 122
  без знака, 100
  неявное преобразование, 570
  приведение типов, 131
  приведение типов в классах, 567
  составной, 137
  с плавающей точкой, 122
Точность, 991; 996

У

Указатель, 172; 174; 178; 200
  null, 180
  this, 507; 509
  арифметика указателей, 189
  инициализация, 175
  интеллектуальный, *См.* класс auto_ptr, 897
  на функцию, 355
  нотация указателей, 190
  объявление указателей, 175; 188
  разыменование указателей, 188
  сложение указателей, 187
Универсальное символьное имя, 112

Ф

Файл, 1021
  iomanip, 1000
  iostream, 59; 975
  бинарный, 1033
  включаемый, 60
  заголовочный, 60
Файловый ввод-вывод, 1021
Флаг, 993
  установка, 993
Форматирование
  внутреннее, 1046

Функтор, 942
адаптируемый, 948
предопределенный, 946
Функция, 37; 75; 108
abort(), 831
accumulate(), 1135
adjacent_difference(), 1137
adjacent_find(), 1113
binary_search(), 1128
cin.get(), 1014
copy(), 920; 1117
copy_backward(), 1117
count(), 1113
count_if(), 1113
cout.put(), 108
equal(), 1114
equal_range(), 1127
fill(), 1119
fill_n(), 1119
find(), 1112
find_end(), 1112
find_first_of(), 1112
find_if(), 1112
for_each(), 906; 1112
generate(), 1119
generate_n(), 1119
get(), 149
getline(), 148; 158; 884
includes(), 1129
inner_product(), 1136
inplace_merge(), 1129
length(), 887
lexicographical_compare(), 1133
lower_bound(), 1127
main(), 55
make_heap(), 1131
max(), 1133
max_element(), 1133
merge(), 1128
main(), 481; 1132
min_element(), 1133
mismatch(), 1114
next_permutation(), 1134
nth_element(), 1126
partial_sort(), 1126
partial_sort_copy(), 1126
partial_sum(), 1136
partition(), 1122
peek(), 1017
pop_heap(), 1132
prev_permutation(), 1135
push_back(), 930
push_front(), 930
push_heap(), 1131
putback(), 1018
random_shuffle(), 1122
remodel(), 896
remove(), 1119
remove_copy(), 1120
remove_copy_if(), 1120
remove_if(), 1120
replace(), 1118; 1122
replace_copy(), 1118
replace_copy_if(), 1119
replace_if(), 1118
reverse_copy(), 1121
rotate(), 1121
rotate_copy(), 1122
search(), 1114
search_n(), 1115
set_difference(), 1130
set_intersection(), 1130
set_symmetric_difference(), 1131
set_union(), 1129
setf(), 996
sort(), 906; 1125
sort_heap(), 1132
sqrt(), 78
stable_partition(), 1122
stable_sort(), 1125
stricmp(), 893
strlen(), 157
swap(), 1117
swap_ranges(), 1117
tolower(), 956
transform(), 951
unique(), 1120
unique_copy(), 1121
upper_bound(), 1127
адрес функции, 355
библиотечная, 78
бинарная, 943
виртуальная, 691
возвращаемое значение, 57; 76
встроенная, 365; 487
вызываемая, 76

вызывающая, 76
глобальная, 906
заголовок, 56; 81
неформатированного ввода, 1010
объекты функций, 899
описание, 56
определение, 56; 308
определяемая пользователем, 80
перегрузка, 245; 411
преобразования, 573
прототип, 312
работающая с диапазоном массива, 331
сигнатура, 395
список аргументов, 57
с множеством аргументов, 417
тело, 56
тип возвращаемого значения, 57
унарная, 943
шаблон, 400
форматированного ввода, 1002
явная специализация, 405
Функция-член
дружественная, 818

Х

Хранилище, 198
автоматическое, 198
динамическое, 198
статическое, 198

Ц

Целочисленная константа, 103
Целочисленный тип, 94
int, 95
long, 95
short, 95
Цикл
do while, 239
for, 205
while, 233
задержки, 237

Ч

Черный ящик, 85
Числовые операции, 1135
Число с плавающей точкой, 116
Члены-шаблоны, 792
Член состояния, 559
Чувствительность к регистру, 53

Ш

Шаблонный метод, 945
Шаблон
vector, 899
класса, 772
как параметр, 795
неявная реализация, 790
рекурсивное использование, 786
специализация, 789
частичная специализация, 791
члены-шаблоны, 792
явная реализация, 790
явная специализация, 790
функции, 400
перегруженный, 403
создание экземпляра шаблона, 409
специализация, 409

Э

Экземпляр, 480

Я

Языковое связывание, 448
Язык программирования
процедурный, 36

Научно-популярное издание

Стивен Прата

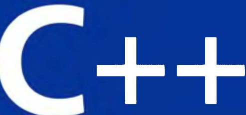
Язык программирования C++ Лекции и упражнения, 5-е издание

Верстка *Т.Н. Артеменко*
Художественный редактор *В.Г. Павлютин*

Издательский дом "Вильямс"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 29.09.2006. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 95,46. Уч.-изд. л. 65,08.
Тираж 3000 экз. Заказ № 2878.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

Язык программирования 

Лекции и упражнения

5-е издание

Эта книга заслужила репутацию тщательно выверенного, хорошо организованного и наиболее полного учебного пособия по языку C++ как для начинающих программистов, так и для профессиональных разработчиков, что подтверждено продажами в США более 100 тысяч экземпляров предыдущих изданий. Она стала своего рода классикой индустрии программирования, обучая основополагающим принципам написания и структуризации кода, а также нисходящему проектированию программного обеспечения. Кроме того, в книге подробно рассматриваются концепции классов, наследования и обработки исключений, а также новейшие тенденции в области объектно-ориентированного программирования. Всемирно известный автор и лектор Стивен Прата предложил одно из наиболее удачных введений в язык C++, отличающееся точностью, полнотой и наглядностью. Фундаментальные концепции программирования тесно переплетаются с их воплощением в языке C++. Множество коротких и вместе с тем поучительных примеров иллюстрируют не более одной или двух концепций за раз, что упрощает их усвоение и поощряет читателя немедленно применять их на практике. Представленные в конце каждой главы вопросы для самоконтроля и упражнения по программированию акцентируют внимание на наиболее критической информации, помогая читателю надежно закрепить полученные знания. Написанная в дружественной манере и простая в чтении, эта книга станет полезной как студентам, изучающим программирование, так и разработчикам, использующим другие языки, которые желают быстро ознакомиться с фундаментальными основами C++.

- Обсуждение базового языка C с включенными средствами C++
- Концептуальное руководство по использованию новых средств C++
- Простые в понимании примеры, иллюстрирующие каждый новый языковой аспект
- Сотни реальных примеров программ
- Множество врезок, акцентирующих внимание на наиболее тонких моментах, связанных с C++
- Вопросы для самоконтроля и упражнения по программированию, помогающие закрепить полученные знания
- Обобщенный язык C++, не привязанный к какой-либо аппаратной платформе, операционной системе и компилятору
- Соответствие стандарту ISO/ANSI, включая шаблоны, стандартную библиотеку шаблонов, класс string, исключения, RTTI и пространства имен

Стивен Прата — профессор физики и астрономии в морском колледже города Кентфилд, штат Калифорния, где, помимо физики и астрономии, преподает компьютерное программирование и дискретную математику. Он получил степень доктора философии в Калифорнийском университете в Беркли. Его увлечение компьютерами началось с моделирования на компьютере звездных скоплений. Профессор Прата является членом Американского астрономического общества, и в прошлом был участником программы стипендий Фулбрайта. Стивен Прата — автор множества бестселлеров, среди которых предыдущие издания этой книги, а также пятого издания книги *C Primer Plus* (*Язык программирования C. Лекции и упражнения, 5-е издание. ИД "Вильямс", 2006 г.*) и *Unix Primer Plus*.

НА WEB-САЙТЕ

Исходные коды всех примеров, рассмотренных в книге, можно загрузить с Web-сайта издательства по адресу: http://www.williamspublishing.com.

Категория: программирование
Предмет рассмотрения: язык C++
Уровень: для начинающих пользователей

ISBN 5-8459-1127-3

06213

9 785845 911278

ВИЛЬЯМС
Издательский дом "Вильямс"
www.williamspublishing.com

SAMS www.samspublishing.com