Язык программирования C

Лекции и упражнения

5-е издание

SAMS

Стивен Прата

Primer Plus

Fifth Edition

Stephen Prata

SAMS

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

Язык программирования C

Лекции и упражнения

5-е издание

Стивен Прата

Москва • Санкт-Петербург • Киев
2013

ББК 32.973.26-018.2.75
П70
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского *Ю.И. Корниенко, Н.А. Мухина, В.Д. Цекича, С.А. Шестакова*
Под редакцией *Ю.Н. Артеменко*
По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Прата, Стивен.

П70 Язык программирования С. Лекции и упражнения, 5-е издание. : Пер. с англ. — М. : Издательский дом "Вильямс", 2013. — 960 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-0986-4 (рус.)

Книга известного специалиста и лектора в области компьютерных технологий посвящена последнему стандарту (C99) одного из наиболее распространенных языков программирования — C, который послужил основой для создания операционной системы Unix. Книгу отличает простой и доступный стиль изложения, изобилие примеров и множество рекомендаций по написанию высококачественных программ. Подробно рассматриваются такие вопросы, как представление данных в языке C, операции и операторы, управляющие структуры и функции. Немалое внимание уделяется обработке строк, вводу-выводу, работе с массивами и структурами и вопросам управления памятью. Исчерпывающие сведения о препроцессоре и стандартных библиотечных функциях дадут возможность эффективно создавать 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

ISBN 978-5-8459-0986-4 (рус.)
ISBN 0-672-32696-5 (англ.)

Оглавление

Содержание

Предисловие

Когда в 1984 году было написано первое издание этой книги, язык C не был широко известным языком программирования. С тех пор началось бурное развитие этого языка, и многие люди изучали C, пользуясь именно этой книгой. По самым приблизительным данным данную книгу в различных редакциях приобрело свыше 500 000 человек.

По мере того, как этот язык развивался от ранней версии Кернигана-Ритчи к стандарту ISO/ANSI 1990 года, вместе с ним совершенствовалась и эта книга, и в настоящее время вышло в свет ее пятое издание. Как и во всех более ранних версиях этой книги, моей целью остается поучительное, четко изложенное и полезное введение в язык C.

Подход и цели

Эта книга предназначена служить дружественным, удобным в использовании и пригодным для самостоятельного изучения справочным пособием. Чтобы соответствовать этому назначению в книге применяются следующие стратегии:

- Наряду с описанием основных особенностей языка C, излагаются основные принципы программирования; изначально не предполагается, что читатель является профессиональным программистом.
- Множество приведенных в книге коротких примеров, которые легко ввести с клавиатуры, служат одновременной иллюстрацией одного или двух понятий, ибо обучение путем выполнения представляет собой один из наиболее эффективных способов освоения новой информации.
- Рисунки и иллюстрации служат пояснениями к понятиям, которые трудно описать словами.
- Краткие описания основных свойств языка C помещаются в выделенные рамки с тем, чтобы на них было легче ссылаться и отыскивать.
- В конце каждой главы приводится список вопросов и упражнений, которые предназначены для того, чтобы помочь вам проверить и закрепить знания языка C.

Чтобы извлечь максимальную пользу, вы должны, насколько это возможно, играть активную роль при изучении материала данной книги. Не ограничивайтесь только чтением примеров, введите их и попытайтесь выполнить. Язык C является достаточно переносимым, но вы можете обнаружить различия между тем, как ведет себя та или иная программа в вашей системе и как она работает в системе автора. Не стесняйтесь экспериментировать, меняйте различные части программы, чтобы посмотреть, к каким эффектам это приведет. Вносите изменения в программу, чтобы она выполняла работу, немного отличающуюся от первоначального варианта. Время от времени игнорируйте случайные предупреждающие сообщения и посмотрите, что случится, если вы будете поступать вопреки рекомендациям в книге. Попытайтесь найти ответы на вопросы и выполнить упражнения. Чем больше вы сделаете сами, тем большему вы научитесь и больше запомните.

Я надеюсь, что вы найдете новейшее издание книги интересным и эффективным введением в язык программирования C.

Об авторе

Стивен Прата (Stephen Prata) преподает астрономию, физику и программирование в морском колледже города Кентфилд, штат Калифорния. Он получил диплом бакалавра в Калифорнийском технологическом институте и степень доктора философии в Калифорнийском университете (в Беркли). Его увлечение компьютерами началось с моделирования на компьютере звездных скоплений. Стивен является автором и соавтором более десятка книг, включая *C++ Primer Plus* и *Unix Primer Plus*.

Посвящение

С любовью Вики и Биллу Прата (Vicky and Bill Prata), которые на протяжении более 69 лет показывали окружающим, каким благом может быть супружество.

Стивен Прата

Благодарности

Я хотел бы поблагодарить Лоретту Ятс (Loretta Yates) из издательства Sams Publishing за вклад в реализацию этого проекта и Сонглин Кию (Songlin Qiu) из того же издательства за просмотр рукописи. Благодарю также Рона Личти (Ron Liechty) из компании Metrowerks и Грега Комо (Greg Comeau) из компании Comeau Computing за помощь в освоении новых свойств C99 и огромный вклад в службу работы с покупателями.

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

ГЛАВА 1

Предварительные сведения

В этой главе:

- Возможности и история создания языка C
- Действия, которые нужно выполнить для написания программ
- Немного о компиляторах и компоновщиках
- Стандарты языка C

Добро пожаловать в мир C — мощного языка программирования для профессионалов, который в не меньшей степени популярен и в среде любителей, и в среде программистов, пишущих программы для коммерческого применения. Эта глава подготовит вас к изучению и использованию этого мощного и широко распространенного языка, она ознакомит вас с различными операционными средами, в которых вы, скорее всего, будете наращивать ваши знания языка C. Прежде всего, мы ознакомимся с происхождением языка C и изучим некоторые его свойства, а также его сильные и слабые стороны. Затем мы изучим основы программирования и рассмотрим основные принципы программирования. В завершение мы обсудим, как выполнять программы на языке C в некоторых известных системах.

Как появился язык C?

Деннис Ритчи (Dennis Ritchie) из компании Bell Labs, создал язык C в 1972 году, когда он и Кен Томпсон (Ken Thompson) работали над созданием операционной системы Unix. Однако сам язык C не зародился просто так в голове Ритчи. Его предшественником был язык B, созданный Томпсоном, предшественником которого был ..., но это уже другая история. Наиболее важным является тот факт, что C задумывался как инструментальное средство для программистов-практиков, следовательно, его главной целью в этом случае было создание полезного языка программирования.

Большинство языков программирования создавались с целью быть полезными, но довольно-таки часто перед ними ставились другие цели. Например, главной целью языка Pascal было создание базиса для изучения основных принципов программирования. С другой стороны, язык BASIC создавался как язык программирования, приближенный к естественному английскому, чтобы облегчить задачу изучения языков

программирования студентам, не знакомых с компьютерами. Это достаточно важные цели, однако, они не всегда содействуют достижению прагматичности и повседневной пригодности. Тем не менее, разработка C как языка, предназначенного для программистов, сделала его одним из наиболее востребованных в настоящее время.

Почему язык C?

В течение трех последних десятилетий C стал одним из основных и наиболее широко распространенных языков программирования. Его популярность росла в связи с тем, что люди предпринимали попытки работать с ним, во время которых он показывал себя с лучшей стороны. За последнее десятилетия многие программисты перешли с языка на более претенциозный язык C++, но язык C сам по себе все еще остается важным языком, равно как и путем перехода к C++. По мере изучения C вы убедитесь, что он обладает многими достоинствами (рис 1.1). Некоторые из них мы отметим сейчас.

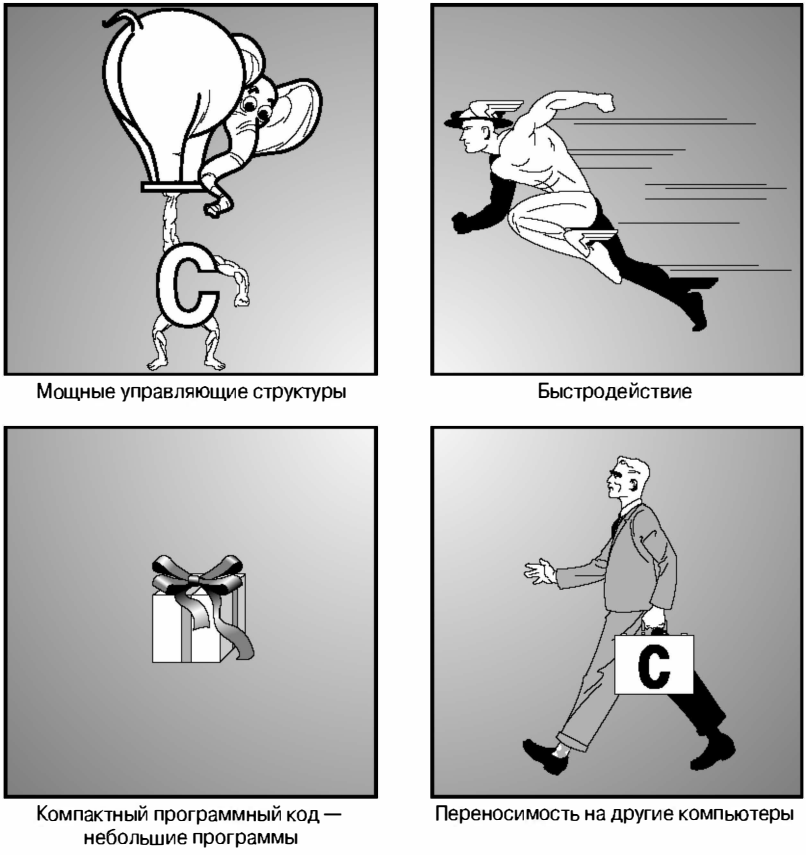


Рис. 1.1. *Достоинства языка C*

Конструктивные особенности

С представляет собой современный язык программирования, включающий управляющие средства, которые теория и практика вычислительных систем рассматривает как полезные и желательные. Его конструкция хорошо подходит для планирования сверху вниз, для структурного программирования и для модульного проектирования. Все это позволяет получать надежные и понятные программы.

Эффективность

С является эффективным языком программирования. Его конструкция продуктивно использует возможности компьютеров, на которых он установлен. Программы на С отличаются компактностью и быстротой исполнения. По сути дела С обладает некоторыми средствами точного управления, обычно характерными разве что для языка ассемблера. (*Язык ассемблера* — это мнемоническое представление множества инструкций, используемых конкретным центральным процессором; различные семейства центральных процессоров имеют различные языки ассемблера.) По желанию вы можете настроить ваши программы на максимальную скорость исполнения или на более эффективное использование памяти.

Переносимость

Язык С является переносимым языком, это означает, что программу, написанную на С для одной системы, можно выполнять на другой системе всего лишь с небольшими изменениями, причем иногда удается вообще обходиться без изменений. В тех случаях, когда изменения неизбежны, они ограничиваются простым редактированием нескольких записей в заголовочном файле, сопровождающем главную программу. Многие языки декларируются переносимыми, однако тот, кто преобразовывал программу на языке BASIC, предназначенном для персонального компьютера (ПК) компании IBM (IBM PC) в программу на языке BASIC для компьютера Apple (они являются близкими родственниками), либо предпринимал попытки выполнить в среде Unix программу на языке FORTRAN, которая предназначена для мэйнфрейма IBM, знает, что такой перенос — в лучшем случае очень трудоемкая операция. Язык С является лидером в смысле переносимости.

Компиляторы языка С (программы, преобразующие ваш код на С в инструкции, которые компьютер использует для внутренних целей) доступны примерно для 40 систем, от 8-разрядных микропроцессоров до суперкомпьютеров Cray. Однако следует отметить, что фрагменты программы, написанной специально для доступа к конкретным аппаратным устройствам, таким как монитор или специальные функции операционных систем, подобных Windows XP или OS X, обычно не принадлежат к числу переносимых.

Поскольку язык С тесно связан с Unix, операционные системы семейства Unix поставляются с компилятором С в виде части соответствующих пакетов. Инсталляция операционной системы Linux в общем случае также включает компилятор языка С. Несколько компиляторов языка С предназначены для персональных компьютеров, включая различные версии систем. Таким образом, используете ли вы домашний компьютер, профессиональную рабочую станцию или мэйнфрейм, у вас высокие шансы получить компилятор языка С для вашей конкретной системы.

Мощь и гибкость

С — это мощный и гибкий язык программирования (два наиболее предпочитаемых определения в литературе компьютерной тематики). Например, большая часть программных кодов мощной гибкой операционной системы Unix написана на С. Многие компиляторы и интерпретаторы других языков, таких как FORTRAN, Perl, Python, Pascal, LISP, Logo и BASIC, были реализованы на С. В результате, когда вы используете FORTRAN на Unix-машине, то в конечном итоге программа, написанная на С, выполняет работу по созданию окончательной исполняемой программы. Программы на С применялись для решения физических и технических задач, и даже для анимации специальных эффектов для множества фильмов, в числе которых "Gladiator" ("Гладиатор").

Ориентация на программистов

Язык С ориентирован на удовлетворение потребностей программистов. Он предоставляет вам доступ к оборудованию и позволяет манипулировать отдельными разрядами памяти. Он также предоставляет богатый выбор операций, которые позволяют вам кратко формулировать ваши задачи. Язык С обладает меньшей строгостью, чем, скажем, Pascal, в плане ограничения того, что вы сможете сделать. Такая гибкость является достоинством и одновременно означает опасность. Достоинство заключается в том, что решение многих задач, таких как преобразование форматов данных, в С намного проще, чем в других языках. Опасность состоит в том, что вы можете допускать такие ошибки, которые просто невозможны в других языках. Язык С предоставляет вам большую свободу действий, но при этом накладывает и большую ответственность.

Наряду с этим, большинство реализаций языка С сопровождаются обширными библиотеками полезных функций на С. Эти функции способны удовлетворить многие из потребностей, с которыми сталкивается программист.

Недостатки

Язык С не лишен недостатков. Так же как в случае людей, недостатки и достоинства являются противоположными сторонами одного и того же свойства. Например, как мы уже упоминали, свобода выражений в языке С также требует дополнительной ответственности. В частности, использование в С указателей (чтобы знать, что такое указатели, вы должны заглянуть в последующие главы данной книги) означает, что возникают условия для появления программных ошибок, которые очень трудно отследить. Один из известных людей перефразировал данный комментарий следующим образом: ценой свободы является постоянная бдительность. Выразительность языка С в сочетании с богатством его операций позволяет писать такие программные коды, которые трудно понять. Мы отнюдь не настаиваем на том, чтобы вы писали малопонятные коды, но такая возможность существует. В конце концов, для какого другого языка устраивается ежегодный конкурс на самый парадоксальный программный код? В языке С много достоинств, но, и, несомненно, еще больше недостатков. Однако вместо того, чтобы погружаться в это дело глубже, перейдем к новой теме.

Откуда пошел язык C?

В начале восьмидесятых годов прошлого столетия C уже был доминирующим языком программирования в среде миникомпьютеров, функционировавших под управлением операционной систем Unix. С тех пор он распространился на персональные компьютеры (микрокомпьютеры) и мэйнфреймы (большие вычислительные машины). Обратите внимание на рис. 1.2. Многие компании по разработке и поставке программного обеспечения предпочитают использовать именно язык C при создании программ для текстовых процессоров, крупномасштабных электронных таблиц, компиляторов и других программных продуктов. Эти компании убедились в том, что с помощью C можно создавать компактные и эффективные программы. Что важнее, они знают, что эти в программы легко вносить изменения и легко адаптировать к новым моделям компьютеров.

Все, что хорошо для компаний и для ветеранов языка C, хорошо также и для других пользователей. Все больше и больше пользователей компьютеров обращаются к языку C, чтобы воспользоваться его преимуществами. Чтобы программировать на языке C, вовсе не надо быть компьютерным профессионалом.

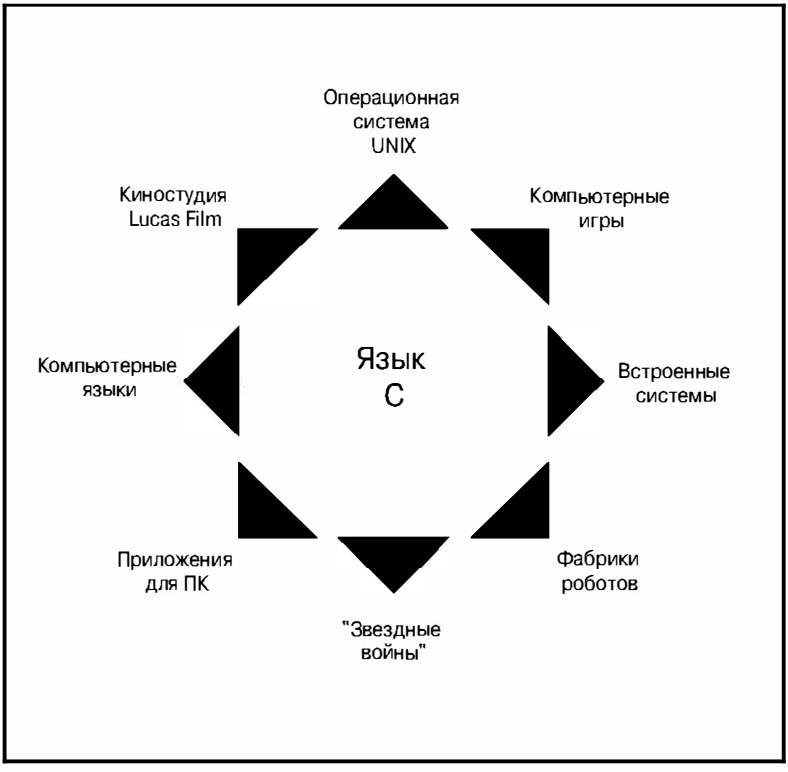


Рис. 1.2. *Где используется язык C*

В девяностых годах прошлого столетия многие компании, изготавливающие и поставляющие программное обеспечение, стали переходить на язык C++ при реализации крупных программных проектов. Язык C++ добавляет к C инструментальные средства объектно-ориентированного программирования. (*Объектно-ориентированное программирование* (*object-oriented programming*) представляет собой философию, которая пытается формировать язык таким образом, чтобы он соответствовал задаче, в отличие от формулирования задачи таким образом, чтобы она соответствовала языку программирования.) C++ в первом приближении можно рассматривать как надмножество языка C в том смысле, что программа на C также является или почти является программой на C++. Изучая язык C, вы фактически изучаете некоторую часть языка C++. Несмотря на популярность более новых языков, таких как, например, C++ и Java, C сохраняет лидирующее положение по способности решать задачи из области разработки программного обеспечения, неизменно завоевывая наивысший балл в рейтинге богатства возможностей языков программирования. В частности, C неизменно используется для программирования встроенных систем. Иначе говоря, он все чаще применяется для программирования обычных микропроцессоров, встроенных в автомобили, камеры, DVD-проигрыватели и в другие современные устройства, используемые в быту. Наряду с этим C посягает на долговременное господство языка FORTRAN в области научного программирования. И, наконец, как язык, создававшийся для разработки операционных систем, он играет ключевую роль в создании операционной системы Linux. Таким образом, и в первой декаде двадцать первого века C удерживает за собой роль сильного языка.

Короче говоря, C является одним из наиболее важных языков программирования и надолго останется таковым. Если вы хотите заниматься разработкой программ, то вопрос, можете ли вы работать в языке C, вы непременно должны ответить утвердительно.

Как работают компьютеры

Прежде чем вы приступите к изучению программирования в языке C, вы, очевидно, должны получить хотя бы самое приблизительное представление о том, как работает компьютер. Эти знания помогут вам понять, какая связь между написанием программы на C и что на самом деле происходит, когда вы исполняете эту программу.

Современные компьютеры состоят из нескольких компонентов. *Центральное процессорное устройство* (ЦП) выполняет основную вычислительную работу. *Память с произвольным доступом*, или оперативное запоминающее устройство (ОЗУ), представляет собой рабочую область, в которой содержатся программы и файлы. Постоянная память, обычно жесткий диск, запоминает эти программы и файлы и хранит их, даже когда компьютер выключен. Периферийные устройства различного назначения, такие как клавиатура, мышь и монитор, обеспечивают обмен данными между вами и компьютером. ЦП обрабатывает ваши программы, поэтому рассмотрим более подробно его роль.

Функции ЦП достаточно просты. Он извлекает команду из памяти и выполняет ее. Затем он извлекает следующую команду и выполняет ее, и так далее. (Центральный процессор с тактовой частотой 1 ГГц выполняет порядка одного миллиарда таких операций в секунду, так что ЦП ведет монотонную жизнь в бешеном темпе.) ЦП имеет

собственную рабочую область, состоящую из нескольких *регистров*, каждый из них может запоминать некоторое число. Один регистр содержит адрес следующей команды в памяти, а ЦП использует эту информацию для извлечения следующей команды. После получения следующей команды ЦП запоминает ее в другом регистре и обновляет первый регистр адресом очередной команды. Центральный процессор выполняет ограниченный набор команд (получивший название *набора инструкций*). Наряду с этим, эти команды достаточно специфичны, многие из них требуют от ЦП переместить число из одного места в другое, например, из ячейки памяти в тот или иной регистр.

Здесь следует отметить два интересных факта. Во-первых, все, что хранится в компьютере, хранится в виде чисел. Числа сохраняются как числа. Символы, такие как буквы алфавита, которые вы используете в текстовых документах, сохраняются как числа, при этом каждый символ имеет свой числовой код. Команды, которые компьютер загружает в свои регистры, сохраняются как числа, каждая команда из системы команд имеет свой числовой код. Во-вторых, компьютерная программа в конечном итоге должна быть выражена в этом числовом коде, или, другими словами, с помощью *машинного языка*.

Одним из последствий такого принципа работы компьютера является то, что если вы хотите, чтобы компьютер выполнил какую-то работу, вы должны ввести конкретный список инструкций (программу), в котором подробно расписано, что и как нужно сделать. Вы должны создать программу на языке, который понятен непосредственно компьютеру (на машинном языке). Это кропотливая и утомительная работа, требующая большой точности. Такая простая операция, как сложение двух чисел, должна быть разбита на несколько действий, примерно следующим образом:

1. Скопировать число из ячейки памяти 2000 в регистр 1.
2. Скопировать число из ячейки памяти 2004 в регистр 2.
3. Сложить содержимое регистра 2 с содержимым регистра 1 и оставить результат сложения в регистре 1.
4. Скопировать содержимое регистра 1 в ячейку памяти 2008.

Каждую из этих инструкций вы должны представить в числовом коде! Если написание программ в таком стиле вам нравится, вынужден огорчить вас, сообщив, что золотой век программирования в машинных кодах давно ушел в прошлое. Однако если вы все-таки предпочитаете что-нибудь более интересное, добро пожаловать в языки программирования высокого уровня.

Языки программирования высокого уровня и компиляторы

Языки программирования высокого уровня, такие как C, существенно упрощают вашу жизнь как программиста несколькими способами. Во-первых, вы не должны представлять команды в числовом коде. Во-вторых, команды, которые вы используете, намного ближе к тому, как вы думаете о задаче, нежели к тому, как она представлена в рамках детализированного подхода, который использует компьютер. Вместо того чтобы обременять себя мыслями о том, какие действия конкретный ЦП должен предпринять, чтобы решить конкретную задачу, вы можете выразить свои пожелания на

более абстрактном уровне. Чтобы сложить два числа, вы можете, например, написать следующую конструкцию:

```
total = mine + yours;
```

Увидев код, подобный этому, вы сразу же догадываетесь, что он делает, в то же время, просматривая эквивалентный код на машинном языке, содержащий несколько команд, выраженных в числовой форме, вы не сразу поймете, о чем идет речь. К сожалению, для компьютера это верно с точностью до наоборот, для него команда на языке высокого уровня — непонятная бессмыслица. Именно в этот момент на передний план выступают компиляторы. Компилятор — это программа, которая переводит программу, представленную на языке высокого уровня, в детальный набор команд на машинном языке, понимаемых компьютером. На вашу долю приходится творческое мышление в командах языка высокого уровня, а всю трудоемкую детализацию компилятор берет на себя.

Подход с использованием компилятора дает еще одно преимущество. В общем случае с каждой моделью компьютера связан собственный уникальный машинный язык. Следовательно, программа, написанная на машинном языке, скажем, для ЦП Intel Pentium ничего не говорит процессору Motorola PowerPC. В то же время вы можете приспособить компилятор для конкретного машинного языка. По этой причине, располагая нужным компилятором или набором компиляторов, вы можете преобразовать одну и ту же программу на языке высокого уровня в различные программы на разных машинных языках. Вы решаете задачу программирования только один раз, после чего вы предоставляете возможность вашим компиляторам транслировать ее решение на множество различных машинных языков.

Короче говоря, языки высокого уровня, такие как язык C, Java и Pascal, описывают действия в более абстрактной форме и привязаны к конкретному ЦП или к конкретной системе команд. К тому же, языки высокого уровня значительно легче изучать, на нем намного проще программировать, чем на машинных языках.

Использование языка C: семь этапов

Язык C, как уже говорилось, является транслируемым языком. Если вы привыкли работать с транслируемым языком, например, с языком Pascal или FORTRAN, вам известны основные действия, выполняемые для сборки программы, написанной на C. Тем не менее, если вы имели дело с интерпретируемым языком, например, BASIC, или графическим интерфейсно-ориентированным языком, таким как, например, Visual Basic, или если у вас вообще нет программистского опыта, вы должны ознакомиться с особенностями компиляции. Мы вскоре рассмотрим этот процесс, и вы сможете убедиться сами, что он достаточно прост и практичен. Во-первых, чтобы дать вам общее предоставление о программировании, разобьем процедуру написания программы на языке C на семь этапов (рис. 1.3). Обратите внимание на то обстоятельство, что это идеализация. На практике, особенно в случае крупных проектов, вы должны перемещаться назад и вперед, используя то, чему вы научились на более позднем этапе, для уточнения результатов, полученных на более ранней стадии.

Рис. 1.3. *Семь этапов программирования*

Этап 1: определение целей программы

Вполне естественно, вы должны начинать с четкого видения того, что, по вашему мнению, программа должна делать. В переводе на информацию, которая нужна вашей программе, обдумайте вычисления и манипуляции, которые программа должна выполнить, а также информацию, которую она должна возвратить. На этом уровне планирования вы должны мыслить в общих терминах, а не в терминах некоторого конкретного компьютерного языка.

Этап 2: проектирование программы

После того, как станет ясна концептуальная картина того, что ваша программа должна сделать, вы должны решить, как она должна это сделать. Каким должен быть пользовательский интерфейс? Как должна быть организована эта программа? Каковыми будут целевые пользователи? Сколько времени потребуется для завершения разработки программы?

Вы также должны решить, как представлять данные в программе и, возможно, во вспомогательных файлах, а также какие методы следует использовать для обработки данных. На начальном этапе изучения программирования в C ответы на эти вопросы не вызовут у вас затруднений, но если вы окажетесь в более сложной ситуации, то поймете, что эти решения потребуют от вас учета множества обстоятельств. Правильный выбор способа представления информации может существенно облегчить разработку программы и обработку данных.

Подчеркнем еще раз, вы должны мыслить общими категориями и не думать о конкретном коде, однако некоторые из ваших решений могут быть основаны на общих характеристиках языка. Например, программист, работающий на C, имеет гораздо больше вариантов представления данных, чем, скажем, программист, имеющий дело с языком Pascal.

Этап 3: написание кода

Теперь, когда вы создали проект вашей программы, вы можете приступать к ее реализации, для чего необходимо написать программный код. Иначе говоря, вы переводите проект программы на язык C. Именно на этой стадии потребуются все ваши знания языка C. Вы можете набросать решения на бумаге, но в конечном итоге вы должны будете ввести составленный вами код в компьютер. Механика этого процесса зависит от среды программирования, в которой вы работаете. Вскоре мы ознакомим вас с характеристиками некоторых операционных сред. В общем случае вы используете текстовый редактор для построения так называемого файла исходного кода. Этот файл содержит интерпретацию проекта вашей программы на языке C. В листинге 1.1 показан пример исходного кода на C.

Листинг 1.1. Пример исходного кода на языке C

```
#include <stdio.h>
int main(void)
{
   int dogs;

   printf("Сколько у вас собак?\n");
   scanf("%d", &собак);
   printf("Следовательно, у вас %d собак(а, и)!\n", dogs);

   return 0;
}
```

К числу работ, которые вы должны выполнить на этом этапе, относится документирование ваших действий. Простейшим способом документирования является комментарий, которым снабжается программный код на C, и в который вы помещаете необходимые пояснения. В главе 2 подробно описано, как следует употреблять комментарии в вашем программном коде.

Этап 4: компиляция

Следующим этапом разработки является компиляция исходного кода. И в этом случае детали зависят от среды программирования, поэтому мы вскоре рассмотрим некоторые из распространенных сред. А пока мы рассмотрим концептуальное представление того, что происходит на этом этапе.

Напомним, что компилятор представляет собой программу, в обязанности которой входит преобразование исходного кода в исполняемый код. *Исполняемый код* — это собственный язык машины, или *машинный язык* вашего компьютера. Этот язык из подробных команд, представленных в числовом коде. Как вы уже прочли выше, разные компьютеры имеют разные машинные языки, а компилятор языка C транслирует код C в конкретный машинный язык.

Компиляторы языка С вставляют также коды из библиотек программ на С в окончательный вариант программы; упомянутые библиотеки содержат комплект стандартных программ, например, `printf()` и `scanf()`, дабы вы, при необходимости, могли ими воспользоваться. (Если говорить точнее, то библиотечные программы в вашу программу включает инструмент, получивший название *компоновщика*, или *редактора связей*, тем не менее, в большинстве систем его запускает компилятор.) В конечном итоге получается исполняемый файл, который понимает компьютер, и который можно запускать на выполнение.

Компилятор проверяет также, не содержит ли ошибок ваша программа на С. Когда компилятор находит ошибки, он уведомляет об их наличии и не создает исполняемый файл. Понимание "жалоб" компилятора — это еще одна обязанность, которую вам придется освоить.

Этап 5: запуск программы на выполнение

Как правило, исполняемый файл представляет собой программу, которую вы можете запускать на выполнение. Чтобы запустить программу, во многих известных средах, включая консоли MS-DOS, Unix, Linux, необходимо ввести с клавиатуры имя исполняемого файла. Другие среды, такие как система VMS на миникомпьютерах VAX, могут потребовать ввода команды запуска или использования какого-либо другого механизма. Среды *IDE* (*Integrated development environments — интегрированная среда разработки*), подобные тем, что поставляются для Windows и Macintosh, позволяют редактировать и выполнять программы на С внутри среды, выбирая соответствующие пункты меню или нажимая специальные клавиши. Полученная программа может быть запущена на выполнение непосредственно из операционной системы путем одиночного или двойного щелчка на имени файла или на соответствующей пиктограмме.

Этап 6: тестирование и отладка программы

Тот факт, что ваша программа работает — хороший знак, тем не менее, есть вероятность, что она работает неправильно. Отсюда следует, что вы должны убедиться, что ваша программа делает именно то, что и должна. Достаточно часто в своих программах вы будете обнаруживать ошибки. Отладка — это процесс обнаружения и исправления программных ошибок. Допущение ошибок является естественной составляющей процесса обучения. Они, по-видимому, присущи программированию, так что когда вы сочетаете программирование с обучением, лучше быть готовым к частым напоминаниям об ошибках. По мере того, как вы становитесь все более квалифицированным и проницательным программистом, ваши ошибки также становятся все более разрушительными и трудно обнаруживаемыми.

У вас есть много возможностей совершить ошибку. Вы можете совершить принципиальную ошибку в проекте программы. Вы можете некорректно реализовать хорошую идею. Вы можете упустить из виду недопустимые входные данные, которые исказят вашу программу. Вы можете неправильно использовать конструкции самого языка С. Вы можете допускать ошибки при наборе кода с клавиатуры. Вы можете неправильно расставить скобки и так далее. Самостоятельно дополните этот печальный список примерами из своей практики.

К счастью, ситуация небезнадежна, хотя могут наступить такие моменты, когда вам покажется, что это именно так. Компилятор отслеживает многие виды ошибок, кроме того, можно предпринять определенные усилия, дабы помочь самому себе в поиске ошибок, которые не отловил компилятор. По мере изучения данной книги вы найдете в ней множество советов по практической отладке программ.

Этап 7: сопровождение и модификация программы

Когда вы создаете программу для себя или для кого-нибудь еще, то, скорее всего, предполагаете, что она будет использоваться достаточно часто. Если это так, возможно, появятся причины для внесения в нее изменений. Может быть, существует какой-то незначительный дефект, который проявляется при вводе имени, начинающегося с букв "Zz", либо возникает желание улучшить что-либо в программе. Вы можете добавить в нее новую функциональную возможность. Вы можете адаптировать программу для выполнения в различных компьютерных системах. Решение задач подобного рода существенно упрощается, если вы четко документируете программу и следуете проверенным на практике рекомендациям.

Комментарий

Программирование обычно не является таким последовательным процессом, каковым является описанный выше процесс. Время от времени вам приходится перемещаться туда и обратно по этапам. Например, когда вы пишете программный код, вы можете прийти к заключению, что намеченный ранее план неосуществим. Вы можете увидеть лучший способ решения задачи. Возможно, после анализа выполнения программы возникнет желание изменить проектное решение. Документирование совершаемых действий поможет перемещаться по этапам туда и обратно.

Многие из изучающих программирование пренебрегают этапами 1 и 2 (определение целей и построение проекта программы) и переходят непосредственно к этапу 3 (написание программы). Первые написанные вами программы будут достаточно простыми, чтобы вы могли "прокрутить" весь процесс разработки в голове. Если вы допустите ошибку, ее легко найти. По мере того как ваши программы становятся все крупнее и сложнее, мысленное представление программы начинает подводить, а на выявление ошибок уходит все больше времени. В конечном итоге те, кто пренебрегает стадией планирования, обречены на бесполезную потерю времени, на долгие часы замешательства и путаницы, к тому же получая уродливые, плохо функционирующие и трудные для понимания программы. Чем крупнее и сложнее задача, тем больше времени приходится затрачивать на планирование ее решения.

Вывод, который следует из всего вышесказанного, заключается в том, что вы должны выработать в себе привычку составлять планы, прежде чем приступать к написанию собственно кода. Воспользуйтесь старой, доброй и вполне оправданной технологией "карандаша и бумаги", чтобы сформулировать цели вашей программы и набросать эскиз ее проекта. Если вы это сделаете, то в конечном итоге получите большую экономию времени и останетесь довольны результатами.

Механика программирования

Действия, которые вы должны выполнить, чтобы получить программу, зависит от среды вашего компьютера. Поскольку C — переносимый язык, с ним можно работать в различных средах, включая операционные системы Unix, Linux, MS-DOS (вы не ошиблись, некоторые все еще пользуются этой операционной системой), Windows и Macintosh. В этой книге не хватит места, чтобы рассмотреть все эти операционные среды, в частности, в силу того, что отдельные программные продукты развиваются, умирают и заменяются другими.

Однако, прежде всего, рассмотрим некоторые аспекты, которыми обладают все среды языка C, в том числе и указанные выше. Вообще говоря, вам вовсе не нужно знать, по каким правилам выполняется C-программа, но это полезные знания. Это поможет понять, почему для создания программы на C необходимо пройти через определенные этапы.

Когда вы пишете программу на языке C, вы сохраняете то, что написано, в текстовом файле, который называется файлом исходного текста. Большинство C-систем, в том числе и упомянутые выше, требуют, чтобы имя файла заканчивалось на `.c` (например, `wordcount.c` или `budget.c`). Часть имени, находящаяся перед точкой, называется *базовым именем*, а часть, следующая за точкой, — *расширением*. Следовательно, `budget` — это базовое имя, а `c` — расширение. Сочетание `budget.c` образует имя файла. Это имя должно также удовлетворять требованиям конкретной операционной системы компьютера. Например, MS-DOS представляет собой операционную систему для персональных компьютеров производства компании IBM и совместимых с ними. Она требует, чтобы базовое имя содержало не более восьми символов, и в силу этого обстоятельства упомянутое выше имя файла `wordcount.c` не будет допустимым именем файла в DOS. Некоторые системы Unix ограничивают совокупную длину имени файла 14-ю символами, включая расширение; другие системы Unix допускают длинные имена порядка 255 символов. Операционные системы Linux, Windows и Macintosh также разрешают использование длинных имен.

Итак, дабы иметь что-то конкретное, на что можно было бы ссылаться, рассмотрим файл с именем `concrete.c`, содержащий исходный код на C, представленный в листинге 1.2.

Листинг 1.2. Программа `concrete.c`

```
#include <stdio.h>
int main(void)
{
   printf("Бетон содержит песок и цемент.\n");
   return 0;
}
```

Пусть вас пока не беспокоят детали содержимого файла исходного кода, представленного в листинге 1.2, мы вернемся к ним в главе 2.

Файлы объектного кода, исполняемые файлы и библиотеки

Базовая стратегия программирования на C заключается в том, чтобы использовать программы, которые преобразуют ваш исходный код в исполняемый файл, содержащий готовый к выполнению программный код на машинном языке. Эта работа выполняется в два этапа: компиляция и компоновка. Компилятор преобразует ваш исходный код в промежуточный код, а компоновщик комбинирует этот код с другим кодом, в результате получается исполняемый файл. В C используется такой двухэтапный подход для модульной организации программ. Вы можете компилировать индивидуальные модули по отдельности, а затем с помощью компоновщика объединить скомпилированные модули. Таким образом, если потребуется изменить какой-то один модуль, не нужно будет повторно компилировать остальные модули. Кроме того, компоновщик связывает вашу программу с заранее откомпилированным библиотечным кодом.

Существует несколько вариантов форматов промежуточных файлов. Наиболее предпочтительным является вариант, выбранный для реализаций, описанных в данной книге, который предусматривает преобразование исходного кода в код на машинном языке, после чего результат помещается в *файл объектного кода*, или, сокращенно, в *объектный файл*. (При этом предполагается, что ваш исходный код хранится в одном файле.) И хотя объектный файл содержит коды в машинном языке, он еще не готов к выполнению. Объектный файл содержит трансляцию вашего исходного кода, но это незаконченная программа.

Первый элемент, которого не хватает в файле объектного кода — это *код запуска*, представляющий собой код, который действует в качестве интерфейса между вашей программой и операционной системой. Например, вы можете запускать программу на одинаковых персональных компьютерах, один из которых функционирует под управлением DOS, а другой — под управлением Linux. В обоих случаях оборудование одно и то же, так что используется один и тот же объектный код, в то же время нужны разные коды запуска для DOS и для Linux, поскольку эти системы поддерживают программы по-разному.

Вторым отсутствующим элементом являются коды библиотечных программ. Практически все C-программы используют стандартные программы (именуемые *функциями*), которые являются частью библиотеки C. Например, `concrete.c` использует функцию `printf()`. Объектный файл не содержит код этой функции, он просто содержит команду, требующую использования функции `printf()`. Фактический код хранится в файле, получившем название *библиотеки*. Библиотечный файл содержит объектные коды для множества функций.

Роль компоновщика заключается в том, чтобы собрать воедино эти три элемента — ваш объектный код, стандартный код запуска и библиотечный код — с последующим запоминанием в отдельном файле, который называется исполняемым. Что касается библиотечного кода, то компоновщик извлекает только код, необходимый для функций, вызываемых из библиотеки, как показано на рис. 1.4.

Короче, как объектный, так и исполняемый файлы содержат команды на машинном языке. В то же время, объектный файл содержит только результат трансляции ваших программных кодов, в то время как исполняемый файл — также и машинные коды использованных вами стандартных библиотечных программ и код инициализации.

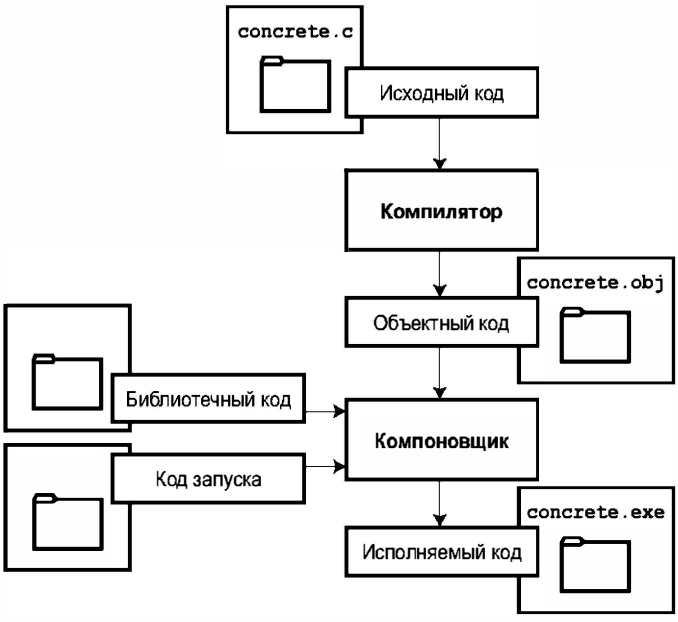


Рис. 1.4. *Компилятор и компоновщик*

В некоторых системах вы должны запускать транслятор и компоновщик отдельно. В других системах компилятор запускает компоновщик автоматически, так что вам остается только выдать команду на компиляцию. Теперь рассмотрим несколько конкретных систем.

Операционная система Unix

Поскольку популярность языка C началась с систем на базе Unix, мы начнем именно с этой операционной системы.

Редактирование в системе Unix

Язык C в системе Unix не имеет собственного редактора. В этом случае используется один из редакторов Unix общего назначения, например, emacs, jove, vi или текстовый редактор системы X Window System.

Вам достаточно добросовестно выполнить две процедуры: правильно ввести программу с клавиатуры и выбрать имя для файла, в котором будет храниться эта программа. Как уже говорилось выше, это имя должно оканчиваться на `.c`. Обратите внимание, что система Unix различает буквы верхнего и нижнего регистров. Поэтому `budget.c`, `BUDGET.c` и `Budget.c` — три различных допустимых имени исходных файлов, в то же время `BUDGET.C` таковым не является, так как расширение `.C` представлено в верхнем, а не нижнем регистре. С помощью редактора vi мы ввели приведенную ниже программу и сохранили ее в файле `inform.c`.

```
#include <stdio.h>
int main(void)
{
   printf(".c завершает имя файла с программой на C.\n");
   return 0;
}
```

Этот текст представляет собой исходный код, а `inform.c` — исходный файл. Здесь важно отметить, что исходный файл — это начало процесса, но не его конец.

Компиляция в системе Unix

Наша программа, хотя совершенная во всех других отношениях, все же непонятна компьютеру. Компьютер не понимает таких выражений, как `#include` и `printf`. (На этой стадии вы, скорее всего, тоже, однако у вас есть надежда вскоре узнать, что это такое, в то время как у компьютера нет никаких шансов.) Как уже было отмечено выше, мы нуждаемся в помощи компилятора при трансляции программного кода (исходного кода) в код компьютера (машинный код). Результатом этих усилий будет исполняемый файл, который содержит все машинные коды, которые нужны компьютеру, чтобы выполнить работу.

Компилятор языка C в операционной системе Unix называется *cc*. Чтобы скомпилировать программу `inform.c`, вы должны ввести с клавиатуры следующую команду:

```
cc inform.c
```

Через несколько секунд вновь отобразится подсказка системы Unix, уведомляющая о том, что дело сделано. Вы можете получить предупреждающее сообщение или сообщение об ошибке, если программа написана неправильно, однако предположим, что все прошло удачно. (Если компилятор жалуется, что не понимает слова `void`, это означает, что ваша система не имеет компилятора ANSI C. Более подробно о стандартах мы поговорим далее. На данный момент имеет смысл просто удалить слово `void`.) Если воспользоваться командой `ls` для получения списка файлов, обнаружится файл с именем `a.out` (рис. 1.5). Это исполняемый файл, содержащий оттранслированную (или скомпилированную) программу. Чтобы запустить его, достаточно ввести с клавиатуры команду

```
a.out
```

в ответ выдается следующее сообщение:

```
.c завершает имя файла с программой на C.
```

Если вы хотите иметь исполняемый файл (`a.out`), вы должны его переименовать. В противном случае данный файл будет заменяться новым файлом `a.out` всякий раз, когда вы компилируете какую-либо программу.

А что можно сказать об объектном коде? Компилятор создает файл объектного кода, имеющий то же базовое имя, что и исходный файл, но с расширением `.o`. В нашем примере файл объектного кода получает имя `inform.o`, но вы его не найдете, поскольку компоновщик удалит его, как только построение исполняемой программы будет завершено. Однако если первоначальная программа использует более одного исходного файла, файлы объектного кода будут сохранены. Когда далее в этой книге мы будем рассматривать программы с множеством исходных файлов, вы убедитесь, что это была неплохая идея.

Операционная система Linux

Linux представляет собой широко распространенную Unix-подобную операционную систему с открытым исходным кодом, которая работает на различных платформах, включая IBM и Macintosh. Подготовка С-программы в среде Linux мало в чем отличается от подготовки в среде системы Unix, за исключением того, что вам придется воспользоваться общедоступным и бесплатным компилятором языка С с именем *gcc*, предоставляемым GNU. Команда компиляции имеет следующий вид:

```
gcc inform.c
```

Обратите внимание на то, что инсталляция gcc производится по желанию пользователя во время установки системы Linux, таким образом, вам (или кому-то другому) придется устанавливать компилятор gcc, если он не был до этого инсталлирован. Как правило, при инсталляции создается и псевдоним cc, указывающий на компилятор gcc, так что в командной строке можно использовать cc вместо gcc. Более подробная информация о gcc, включая сведения о новых версиях, доступна по адресу:

```
http://www.gnu.org/software/gcc/gcc.html
```

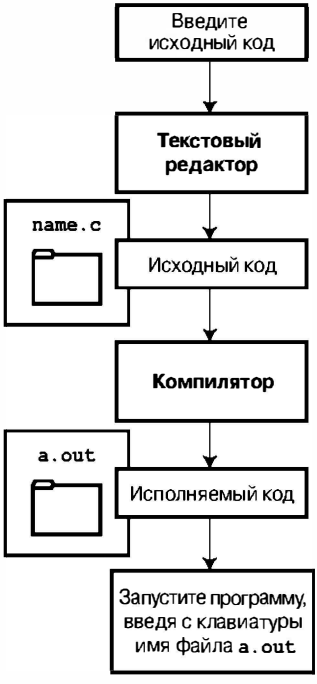


Рис. 1.5. *Подготовка программы на языке С в среде операционной системы Unix*

Интегрированная среда разработки (Windows)

Компилятор языка С не является частью стандартного пакета операционной системы Windows, так что, возможно, у вас появится необходимость получить и установить этот компилятор. Всего лишь несколько поставщиков, в числе которых такие компании, как Microsoft, Borland, Metrowerks и Digital Mars, могут предложить среды *IDE* (интегрированная среда разработки) для Windows. (В настоящее время большинство из них представляют собой комбинированные компиляторы языков С и C++.) Все они имеют в своем составе быстродействующие интегрированные среды, позволяющие компилировать С-программы. Ключевая особенность заключается в том, что каждая из этих сред имеет встроенный редактор, которым можно пользоваться для написания программ на С. Каждая IDE-среда предлагает систему меню, которые позволяют именовать и сохранять файлы исходных кодов, а также компилировать и запускать на выполнение программы, не покидая для этого среду. Каждая IDE-среда возвращает вас обратно в редактор, если компилятор обнаруживает какие-либо ошибки, при этом указываются строки программы, содержащие ошибки.

Среды IDE для Windows поначалу могут показаться устрашающими в силу того, что предлагают целый набор *целей*, то есть операционных сред, в которых программа будет использоваться. Например, они могут предложить следующий выбор: 16-разрядная программа для Windows, 32-разрядная программа для Windows, файл библиотеки

DLL (Dynamic-Link Library – динамически подключаемая библиотека) и так далее. Многие из целей предусматривают использование графического интерфейса Windows. Чтобы осуществить эти (а также и другие) выборы, обычно создается *проект*, куда добавляются имена исходных файлов, которые должны использоваться. Конкретные действия зависят от используемого программного продукта. Как правило, вы сначала используете меню File (Файл) или Project (Проект) для создания проекта. При этом важно выбрать правильную форму проекта. Примеры, приводимые в этой книге, носят общий характер и служат иллюстрацией выполнения программы в среде командной строки. Различные среды IDE для Windows предлагают один или несколько вариантов, удовлетворяющих этому не слишком требовательному условию. Например, Microsoft Visual C 7.1 предлагает вариант Win32 Console Application. Для Metrowerks CodeWarrior 9.0 выбирайте сначала Win32 C Stationery, а затем C Console App или WinSIOUX C App (последний вариант поддерживает качественный пользовательский интерфейс). Что касается других систем, то желательно найти вариант, использующий такие термины, как DOS EXE, Console или Character Mode executable. В этих режимах ваша исполняемая программа будет выполняться в консольном окне. После создания проекта нужного типа воспользуйтесь меню IDE, чтобы открыть новый файл с исходным кодом. В большинстве программных продуктов это делается через меню File. Возможно, для добавления исходного файла в проект потребуется выполнить дополнительные действия.

Поскольку среды IDE для Windows обычно рассчитаны на работу с языками C и C++, вы должны указать, что хотите выбрать C. В некоторых программных продуктах, например, Metrowerks CodeWarrior, для этого используется тип проекта. В других продуктах, таких как Microsoft Visual C++, для этого служит расширение файла `.c`. В то же время большая часть программ на C работают и как программы на языке C++.

Вы можете столкнуться еще с одной проблемой: окно, в котором отображается процесс выполнения, исчезает с экрана сразу после того, как программа завершается. Если это имеет место, можете заставить программу остановиться до тех пор, пока не будет нажата клавиша <Enter>. Чтобы сделать это, поместите следующую строку в конец программы непосредственно перед оперетором `return`:

```
getchar();
```

Эта строка считывает нажатие клавиши, поэтому программа будет ожидать нажатия клавиши <Enter>.

Иногда, в зависимости от того, как функционирует программа, она может ожидать нажатие любой клавиши. В этом случае следует воспользоваться функцией `getchar()` дважды:

```
getchar();
getchar();
```

Например, если последнее, что сделала программа, было приглашение ввести свой вес, вы набираете на клавиатуре свои килограммы, а затем нажимаете клавишу <Enter>, чтобы ввести эти данные. Программа считывает значение вашего веса, первая функция `getchar()` прочитает нажатие клавиши <Enter>, а вторая `getchar()` заставит программу остановиться до тех пор, пока снова не будет нажата <Enter>. Если сейчас вы еще не можете оценить всех преимуществ этого приема, вы непременно сделаете это после того, как более подробно изучите ввод данных в C.

И хотя различные среды IDE в принципиальных вопросах мало чем отличаются друг от друга, детали меняются от одного программного продукта к другому, а в рамках линейки одного программного продукта — от версии к версии. Вам придется немного поэкспериментировать, чтобы изучить, как работает ваш компилятор. Кроме того, возможно, придется обратиться за советами к справочникам или поработать с онлайновым руководством.

Компиляторы DOS для персональных компьютеров IBM PC

По мнению многих, работать в DOS на ПК сегодня не модно, тем не менее, это один из возможных вариантов для тех, кто владеет компьютерами с ограниченными ресурсами и ограниченным бюджетом, а также для тех, кто предпочитает работать с более простой операционной системой, без красочных эффектов оконной среды. Многие среды IDE для Windows предоставляют инструментальные средства командной строки, обеспечивая возможность программировать в среде командной строки DOS. Компилятор Comeau C/C++, который доступен во многих операционных системах, в том числе и в некоторых вариантах Unix и Linux, имеет версию командной строки DOS. Кроме того, существуют компиляторы языка C, используемые в бесплатных и платных программных средствах, которые работают под DOS. Например, в проекте GNU существует версия компилятора `gcc` для DOS.

Файлы исходных кодов должны быть текстовыми файлами, но не документами текстового процессора. (Документы текстового процессора содержат дополнительную информацию о шрифтах и форматировании.) Вы должны использовать текстовый редактор, например, Windows Notepad или программу EDIT, которая поставляется вместе с некоторыми версиями DOS. Можно пользоваться и текстовым процессором, если с помощью пункта меню Save As (Сохранить как) сохранять файл как текстовый. Файл должен иметь расширение `.c`. Некоторые текстовые процессоры автоматически добавляют расширение `.txt` к именам текстовых файлов. Если это произойдет с вашим файлом, вам придется поменять его имя, заменив `txt` на `c`. Компиляторы языка C для ПК обычно, но не всегда, создают промежуточный объектный файл с расширением `.obj`. В отличие от компиляторов C в Unix, компиляторы C обычно не удаляют этих файлов после того, как закончат работу. Существуют компиляторы, которые генерируют файлы на языке ассемблера с расширением `.asm` или используют свой собственный формат.

Некоторые компиляторы автоматически запускают компоновщик по окончании компиляции; другие же могут потребовать, чтобы вы запускали компоновщик вручную. Компоновка завершается созданием исполняемого файла, при этом к первоначальному базовому имени файла с исходным кодом добавляется расширение `.EXE`.

Например, компиляция и компоновка файла исходного кода с именем `concrete.c` порождают файл с именем `concrete.exe`. Некоторые компиляторы предлагают возможность построения исполняемого файла с именем `concrete.com`. В любом случае вы можете запустить программу на выполнение, введя с клавиатуры базовое имя файла в командной строке:

```
C:>concrete
```

Работа с языком C в системах Macintosh

Наиболее широко известным компилятором языков C/C++ в системах Macintosh является компилятор Metrowerks CodeWarrior. (Версии CodeWarrior в системах Windows и Macintosh имеют очень похожие интерфейсы.) Он предоставляет среду IDE, в основу которой положена концепция проектов, подобная той, какую вы найдете в компиляторе для Windows. Начните с выбора пункта New Project (Новый проект) в меню File. Вам будет предоставлена возможность выбора типа проекта. Для недавних версий компилятора CodeWarrior используйте вариант Std C Console. (В различных версиях CodeWarrior имеются разные пути для этого выбора.) Вам, возможно, придется выбирать между версией 68KB (для процессоров типа Motorola 680x0), версией PPC (для процессоров PowerPC) и версией Carbon (для OS X).

В новом проекте присутствует небольшой файл исходного кода как часть начального проекта. Можете откомпилировать и выполнить эту программу, чтобы убедиться в корректности установки системы.

Языковые стандарты

В настоящее время доступно множество реализаций языка C. В идеальном случае, когда вы пишете программу на C, она должна работать одинаково на любой реализации при условии, что в ней не используется код, специфичный для конкретной машины. Чтобы добиться этого на практике, различные реализации должны соответствовать общепризнанному стандарту.

Сначала для языка C не было никакого стандарта. С другой стороны, общепризнанным стандартом служило первое издание книги Брайана Кернигана и Денниса Ритчи *Язык программирования C* (в настоящее время доступно второе издание этой книги, выпущенное издательским домом "Вильямс" в 2006 году); этот стандарт получил обозначение *K&R C* или *classic C* (классический C). Приложение Б настоящей книги можно рассматривать как руководство по реализациям языка C. Создатели компиляторов, например, утверждают, что предлагают полную реализацию K&R. Однако, хотя в упомянутом приложении дано определение языка C, в нем не описана стандартная библиотека C. Язык C зависит от своей библиотеки в большей степени, нежели другие языки, поэтому возникает необходимость также и в разработке стандарта на библиотеку. В условиях отсутствия какого-либо официального стандарта, библиотека, поставляемая вместе с реализацией C для Unix, становится стандартом де-факто.

Первый стандарт ANSI/ISO C

По мере того как язык C развивался и получал все более широкое применение в различных системах, сообщество пользователей языка C ощутило острую потребность во всеобъемлющем, современном и строгом стандарте. Чтобы удовлетворить эту потребность институт ANSI (American National Standards Institute – Национальный институт стандартизации США) образовал в 1983 году специальный комитет (X3J11), целью которого была разработка нового стандарта, который формально был принят в 1989 году. Этот новый стандарт (ANSI C) определяет как сам язык, так и стандартную библиотеку C. Организация ISO (International Organization for Standardization – Международная организация по стандартизации) приняла стандарт языка C (ISO C) в 1990 году.

ISO C и ANSI C по сути дела являются одним и тем же стандартом. Окончательную версию стандарта ANSI/ISO часто называют *C89* (именно в этом году институт ANSI утвердил этот стандарт) или *C90* (поскольку в этом году данный стандарт был утвержден ISO). Поскольку версия ANSI появилась первой, часто используется термин *ANSI C*.

Комитет X3J11 выдвинул несколько руководящих принципов. Возможно, самым интересным был принцип, гласящий: "сохраняйте дух языка C". Комитет перечислил следующие идеи, которые выступают в качестве выражений этого духа:

- Доверять программисту.
- Не мешать программисту делать то, что он считает необходимым.
- Не увеличивать язык и сохранять его простоту.
- Предусматривать только один способ выполнения операции.
- Делать операцию быстродействующей, даже если при этом не гарантируется переносимость.

Согласно последнему пункту, комитет имел в виду, что реализация должна определить конкретную операцию через действия, которые проявляют себя наилучшим образом на целевом компьютере, а не пытаться любой ценой навязать абстрактное универсальное определение. Вы будете сталкиваться с примерами этой философии по мере изучения языка.

Стандарт C99

В 1994 году начались работы по пересмотру этого стандарта, в результате чего появился стандарт C99. Объединенный комитет ANSI/ISO, известный в те времена как комитет *C9X*, подтвердил базовые принципы стандарта C90, в том числе принцип малых размеров и простоты языка C. Цель, озвученная комитетом, состояла в том, чтобы не добавлять в язык новые свойства за исключением тех, которые необходимы для достижения новых целей, поставленных перед языком. Одной из этих целей была поддержка интернационализации, например, разработка способов работы с наборами интернациональных символов. Второй целью была "кодификация существующих методов устранения очевидных дефектов". Таким образом, при необходимости переноса C на 64-разрядные процессоры комитет положил в основу дополнений к стандарту опыт тех, кто решал эту задачу в реальных условиях. Третьей целью было повышение пригодности языка C для выполнения критических вычислений в рамках научных и технических проектов.

Три указанных выше момента — интернационализация, исправление дефектов и повышение вычислительной полезности — были основными причинами, которые обусловили внесение изменений. Остальные планы, предусматривавшие изменения, были консервативными по своей природе, например, минимизация несоответствий стандарту C90 и языку C++ и сохранение концептуальной простоты языка. В формулировке документа, принятого комитетом, сказано: "... комитет будет удовлетворен, если C++ станет *большим* и амбициозным языком".

В результате изменения стандарта C99 позволяют сохранить естественную суть языка C, а сам язык C остается экономным, чистым и эффективным. В этой книге рассматриваются многие изменения, внесенные в C99.

Поскольку в настоящее время в большинстве компиляторов не реализованы все изменения стандарта C99, может случиться, что вы не найдете некоторых из них в своей системе. Либо вы, возможно, обнаружите, что некоторые свойства C99 станут доступными только тогда, когда вы измените настройки компилятора.

На заметку!

В этой книге используется термин *ISO/ANSI C*, обозначающий свойства, общие для обоих стандартов, и *C99* для ссылки на новые свойства. Иногда будут встречаться ссылки на стандарт *C90* (например, при обсуждении, сопровождающем первое появление того или иного свойства в языке C).

Как организована эта книга

Существует множество способов организации информации. Один из наиболее простых подходов заключается в том, что сначала представляется все, что касается темы A, затем все, что имеет отношение к теме B, и так далее. Такой подход существенно облегчает ссылки, поскольку вы можете найти всю информацию, касающуюся данной темы, в одном месте. В то же время это не самый лучший вариант при изучении предмета. Например, если вы начали изучать английский язык с запоминания всех существительных, ваши возможности выражать мысли будут жестко ограничены. Разумеется, вы можете указывать на объект и выкрикивать его название, но в то же время вас будут значительно лучше понимать окружающие, если вы выучите несколько существительных, глаголов, прилагательных и прочего, а также несколько правил, показывающих, как эти элементы языка соотносятся друг с другом.

Чтобы сбалансировать подачу материала, в данной книге используется спиралевидный подход, который состоит в том, что в начальных главах начинается изучение сразу нескольких тем с последующим возвратом к более полному их обсуждению в последующих главах. Например, понятие функции играет важную роль в понимании всего языка C. Таким образом, несколько начальных глав содержат краткие обсуждения функций, поэтому, когда вы будете читать полное описание функций в главе 9, вам будет значительно легче осваивать тонкости применения функций. Аналогично, в начальных главах дается упрощенное предварительное описание строк и циклов, так что вы сможете пользоваться этими полезными инструментальными средствами еще до того, как вы изучите их во всех подробностях.

Соглашения, принятые в этой книге

Теперь мы готовы приступить к изучению самого языка C. В этом разделе рассматриваются некоторые соглашения, используемые для представления материала книги.

Шрифты и начертание

Для текстов программ, входных и выходных данных используется моноширинный шрифт, который приблизительно напоминает то, что вы можете увидеть на экране или в листингах выходных данных. Ниже показан пример:

```
#include <stdio.h>
int main(void)
```

```
{
  printf("Бетон содержит песок и цемент.\n");
  return 0;
}
```

Тот же моноширинный шрифт применяется для представления терминов, связанных с кодом, например, `main()`, и имен файлов, таких как `stdio.h`. В книге также используется моноширинный шрифт с курсивным начертанием для обозначения заполнителей, которые должны заменяться конкретными значениями. Примером может служить модель объявления:

имя_типа имя_переменной;

В данном случае вы можете, например, заменить *имя_типа* на `int`, а *имя_переменной* — на `zebra_count`.

Выходные данные программы

Выходные данные компьютера представляются в том же самом формате, а входные данные пользователя выделяются полужирным начертанием. Иллюстрацией могут служить следующие выходные данные:

```
Пожалуйста, введите название книги.
Нажмите [enter] в начале строки для останова.
```

```
My Life as a Budgie
Теперь введите имя автора.
```

Mack Zackles

Строки, представленные моноширинным шрифтом, являются выходными данными программы, а строка, выделенная полужирным начертанием — это данные, введенные пользователем.

Существует множество способов обмена данными между вами и компьютером. Тем не менее, мы будем полагать, что вы вводите команды с клавиатуры, а ответ компьютера вы читаете с экрана.

Специальные клавиши

Как правило, вы отсылаете строку команд, нажимая клавишу, обозначенную как <Enter>, <c/r>, <Return> или похожим образом. В тексте мы ссылаемся на нее как на клавишу <Enter>. Обычно, в данной книге считается само собой разумеющимся, если вы нажимаете клавишу <Enter> в конце каждой строки ввода. Тем не менее, чтобы заострить ваше внимание на некоторых моментах, рассмотрим несколько примеров, в которых клавиша <Enter> упоминается напрямую (для ее представления используется обозначение `[enter]`). Квадратные скобки означают, что вы нажимаете одну клавишу, а не вводите с клавиатуры слово *enter*. Мы также пользуемся управляющими символами, например, <Ctrl+D>. Таким способом обозначается нажатие клавиши <D> при удержании в нажатом состоянии клавиши <Ctrl> (или, возможно, <Control>).

Системы, использованные при подготовке данной книги

Некоторые аспекты языка C, такие как объем памяти, отводимый для хранения числа, зависят от системы. Когда речь идет о примерах, и мы ссылаемся на "нашу систему", мы имеем в виду ПК с процессором Pentium, функционирующий под управлением операционной системы Windows XP Professional и использующий компилятор Metrowerks CodeWarrior Development Studio 9.2, Microsoft Visual C++ 7.1 (версия, которая поставляется вместе с Microsoft Visual Studio .NET 2003) или gcc 3.3.3. На момент написания данной книги поддержка стандарта C99 была неполной, и ни один из этих компиляторов не поддерживал возможности C99. Тем не менее, эти компиляторы отвечали большинству требований нового стандарта. Большая часть примеров была отлажена с помощью Metrowerks CodeWarrior Development Studio 9.2, установленного на машине Macintosh G4.

Время от времени в данной книге делаются ссылки на выполнение программ в системе, функционирующей под управлением Unix. Таковой является версия Berkeley's BSD 4.3 системы Unix, функционирующая на компьютере VAX 11/750. Кроме того, несколько программ прошли отладку на ПК Pentium, работающем под управлением Linux и использующем компиляторы gcc 3.3.1 и Comeau 4.3.3. Коды демонстрационных программ, описанных в данной книге, вы можете найти на Web-сайте издательства Sams по адресу `www.samspublishing.com` и на Web-сайте издательского дома "Вильямс" по адресу `www.williamspublishing.com`. Введите номер ISBN книги (без дефисов) в окне поиска и активизируйте поиск. Перейдите на страницу этой книги и загрузите код.

Требования к системе

У вас должен быть установлен компилятор языка C либо вы должны иметь доступ к такому компилятору. C работает на огромном множестве различных компьютерных систем, так что перед вами большой выбор. Убедитесь в том, что вы используете компилятор языка C, предназначенный для вашей конкретной системы. Некоторые из примеров в этой книге требуют поддержки нового стандарта C99, однако большинство примеров будут работать с компилятором, поддерживающим стандарт C90. Если компилятор, который вы используете, был разработан до появления стандартов ANSI/ISO, вам, возможно, придется достаточно часто вносить поправки, что должно побудить вас к обновлению компилятора.

Большинство поставщиков компиляторов делают скидки для студентов и преподавателей, и если вы попадаете в эту категорию клиентов, внимательно изучите Web-сайты поставщиков.

Специальные элементы

В данной книге используются несколько специальных элементов, которые позволяют подчеркнуть важность того или иного вопроса. Ниже показан их внешний вид и описано то, как они используются.

Врезка

Врезка содержит более глубокий анализ или дополнительную информацию, которая позволяет подробнее осветить тему.

Совет

Советы содержат краткие полезные рекомендации, касающиеся разрешения конкретных ситуаций в программировании.

Внимание!

Предупреждения о потенциальных ловушках.

На заметку!

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

Резюме

C — это мощный и компактный язык программирования. Его широкое распространение объясняется тем, что он предлагает полезные инструментальные средства и обеспечивает эффективное управление оборудованием, а также тем, что программы на этом языке легче переносятся с одной системы на другую.

C принадлежит к числу транслируемых языков. Компиляторы и компоновщики (редакторы связей) языка C — это программы, которые переводят исходные коды на языке C в исполняемые коды.

Программирование на языке C может требовать значительных усилий, может оказаться обременительным и приносить одни лишь разочарования, но в то же время оно может стать увлекательным и захватывающим занятием и приносить удовлетворение. Мы надеемся, что язык C станет для вас источником творческого удовлетворения, каковым он стал для нас.

Вопросы для самоконтроля

Ответы на эти вопросы находятся в приложении A.

1. Что означает *переносимость* в контексте программирования?
2. Объясните, какое различие существует между файлом исходного кода, файлом объектного кода и исполняемым файлом.
3. Что собой представляют семь основных этапов программирования?
4. Что делает компилятор?
5. Что делает компоновщик?

Упражнения по программированию

Мы не предполагаем, что вы уже готовы писать программный код на C, поэтому данное упражнение концентрируется на начальных этапах процесса программирования.

1. Вы только что были приняты на работу в компанию MacroMuscle, Inc. (Программное обеспечение для крупных организаций). Компания выходит на европейский рынок и желает иметь в своем распоряжении программу, которая переводит дюймы в сантиметры (1 дюйм = 2,54 см). Компания хочет, чтобы программа выдавала пользователю приглашение на ввод значения в дюймах. Ваша задача заключается в том, чтобы определить цели программы и разработать проект программы (этапы 1 и 2 процесса программирования).

ГЛАВА 2

Введение в язык C

В этой главе:

- Операция: =
- Функции: `main()`, `printf()`
- Написание простой программы на языке C
- Создание целочисленных переменных, присваивание им значений и отображение этих значений на экране
- Символ новой строки
- Включение комментариев в программы, создание программ, содержащих более одной функции, поиск программных ошибок
- Что такое ключевые слова

Как выглядит программа на языке C? Пролистав эту книгу, вы найдете множество примеров. Возможно, вы сочтете, что программа на C выглядит несколько странно, будучи усыпанной такими символами, как {, `cp->tort` и `*ptr++`. Однако по мере того, как вы будете углубляться в содержимое книги, они, а также другие характерные для C символы перестанут казаться странными, они станут для вас более привычными, вам даже станет трудно обходиться без них! Эту главу мы начнем с того, что рассмотрим простую демонстрационную программу и объясним, что она делает. В то же время мы уделим основное внимание некоторым базовым свойствам языка C.

Простой пример программы на языке C

Рассмотрим простой пример программы на языке C. Эта программа, представленная в листинге 2.1, служит для того, чтобы заострить ваше внимание на некоторых особенностях программирования на C. Прежде чем вы прочтете построчное пояснение к программе, ознакомьтесь сначала с листингом 2.1 и попробуйте выяснить без помощи комментариев, что делает эта программа.

Листинг 2.1. Программа `first.c`

```
#include <stdio.h>
int main(void)                  /* простая программа */
{
   int num;                     /* определить переменную с именем num */
   num = 1;                     /* присвоить значение переменной num */
```

```
    printf("Я простой ");       /* использовать функцию printf() */
    printf("компьютер.\n");
    printf("Моей любимой цифрой является %d, так как она первая.\n",num);
    return 0;
}
```

Если вы думаете, что программа что-то отобразит на экране, то вы не ошиблись! Что конкретно будет отображено на экране, может быть не очевидно, поэтому выполните программу и ознакомьтесь с ее результатами. Прежде всего, воспользуйтесь услугами вашего любимого редактора (или "любимым" редактором вашего компилятора), чтобы создать файл, содержащий текст листинга 2.1. Назначьте этому файлу имя, оканчивающееся на `.c` и удовлетворяющее требованиям, предъявляемым к именам файлов в вашей локальной системе. Например, вы можете присвоить программе имя `first.c`. Теперь откомпилируйте и выполните программу. (См. главу 1, в которой содержатся общие сведения по этому процессу.) Если все пойдет хорошо, выходные данные программы примут следующий вид:

```
Я простой компьютер.
Моей любимой цифрой является 1, так как она первая.
```

В конечном итоге этот результат не является неожиданным, однако что собой представляют символы `\n` и `%d` в программе? Кроме того, некоторые строки программы выглядят довольно странно. Самое время для пояснений.

Пояснение к программе

Давайте совершим два прохода по исходному коду программы. Первый проход ("Проход 1: краткий обзор") освещает значение каждой строки и поможет вам получить общее представление о том, что происходит. На втором проходе ("Проход 2: детали программы") исследуются конкретные результаты и подробности, дабы вы могли глубже понять особенности программы.

На рис. 2.1 обобщены все части программы на C; он содержит больше элементов, чем использует наша первая программа.

Проход 1: краткий обзор

В этом разделе представлена каждая строка приведенной выше программы, за которой следует ее краткое описание; в следующем разделе (посвященном проходу 2) тема, поднятая в этом разделе, рассматривается более полно.

```
#include <stdio.h>          ←включить другой файл
```

Эта строка требует от компилятора включить информацию, хранящуюся в файле `stdio.h`, который является стандартной частью всех пакетов компилятора языка C; этот файл обеспечивает поддержку ввода с клавиатуры и отображения вывода.

```
int main(void)              ←имя функции
```

Программа на языке C состоит из одной или большего числа *функций* — базовых модулей программ на C. Рассматриваемая программа состоит из одной функции с именем `main`.

Круглые скобки показывают, что `main()` есть имя функции, `int` указывает на то, что функция `main()` возвращает некоторое целое число, а `void` говорит о том, что функция `main()` не принимает никаких аргументов. Это детали, которые мы будем рассматривать далее. А сейчас просто примем `int` и `void` как часть способа определения функции `main()` языка C, соответствующего стандарту ISO/ANSI. (Если в вашем распоряжении имеется компилятор языка C, разработанный до появления стандарта ISO/ANSI, опустите `void`; чтобы в дальнейшем избежать несоответствий, вам потребуется более современное программное обеспечение.)

Рис. 2.1. *Анатомия программы на языке C*

```
/* простая программа */  ←комментарий
```

Символы `/*` и `*/` заключают в себе комментарии, то есть примечания, которые помогают понять смысл программы. Они предназначены исключительно для читателя и компилятором игнорируются.

```
{                        ←начало тела функции
```

Эта открывающаяся фигурная скобка обозначает начало оператора, составляющего функцию. Определение функции заканчивается закрывающей фигурной скобкой (`}`).

```
int num;                 ←оператор объявления
```

Этот оператор объявляет переменную с именем `num` и уведомляет, что переменная `num` имеет тип `int` (целое число).

```
num = 1;                 ←оператор присваивания
```

Оператор `num = 1;` присваивает значение `1` переменной с именем `num`.

```
printf("Я простой ");    ←оператор вызова функции
```

Первый оператор, использующий функцию `printf()`, отображает на экране фразу `Я простой`, оставляя курсор в той же строке. Используемая здесь функция `printf()` является частью стандартной библиотеки C. Она носит название *функции*, а использование функции в программе называется *вызовом функции*.

```
printf("компьютер.\n");  ←еще один оператор вызова функции
```

Следующий вызов функции `printf()` приписывает слово `компьютер` в конец напечатанной предыдущей фразы. `\n` — это код, указывающий компьютеру начать новую строку, то есть переместить курсор в начало следующей строки.

```
printf("Моей любимой цифрой является %d, так как она первая.\n",num);
```

Последнее использование функции `printf()` приводит к печати значения переменной `num` (которое равно 1), вставленной во фразу, заключенную в кавычки. Код `%d` указывает компьютеру, где и в какой форме печатать значение `num`.

```
return 0;                ←оператор возврата
```

Функция C может предоставить, или *возвратить*, число в объект, который ее вызвал. Пока что рассматривайте эту строку как одно из требований стандарта ISO/ANSI C, регламентирующих использование функции `main()`.

```
}                        ←конец программы
```

Как мы и обещали, программа оканчивается закрывающей фигурной скобкой.

Проход 2: детали программы

Теперь, когда вы вкратце ознакомились с листингом 2.1, рассмотрим представленную в нем программу подробнее. Мы снова будем рассматривать отдельные строки программы, но на этот раз используем каждую строку кода в качестве отправной точки для более глубокого исследования деталей, обозначаемых соответствующими кодами, и как основу для того, чтобы выработать более общий взгляд на особенности программирования на C.

Директивы #include и заголовочные файлы

```
#include <stdio.h>
```

Именно с этой строки начинается программа. Результат выполнения строки `#include <stdio.h>` такой же, как если бы вы ввели с клавиатуры содержимое файла `stdio.h` в ваш файл в той точке, в которой появляется строка `#include`. По сути дела, это операция вырезания и вставки. Директива `include` (включить файлы) представляет собой удобный способ совместного использования информации, который применяется во многих программах.

Оператор `#include` представляет собой пример *директивы препроцессора* в С. В общем случае компиляторы языка С выполняют некоторую подготовительную работу над исходным кодом перед компиляцией; это называется предварительной обработкой.

Файл `stdio.h` поставляется как часть всех пакетов компиляторов С. Он содержит информацию о функциях ввода и вывода, таких как `printf()`, и предназначен для использования компилятором. Его имя происходит от "*standard input/output header*" (заголовочный файл стандартного ввода-вывода). Разработчики языка С называют совокупность информации, которая помещается в верхней части файла заголовком (*header*), а реализации С обычно поставляются с множеством заголовочных файлов.

По большей части заголовочные файлы содержат информацию, используемую компилятором для создания конечных исполняемых программ. Например, они могут определять константы или указывать имена функций и способы их использования. Однако фактический программный код функции представляет собой библиотечный файл предварительно откомпилированного кода, а не заголовочный файл. В обязанности компоновщика, который является компонентом компилятора, входит поиск необходимого библиотечного кода. Короче говоря, заголовочный файл оказывает помощь в правильной компоновке вашей программы.

Стандарт ISO/ANSI С сформулировал требования, какие заголовочные файлы должны быть предоставлены. Для одних программ необходимо включать файл `stdio.h`, для других программ он не нужен. Документация конкретной реализации языка С должна включать описание функций из библиотеки программ на С. Такие описания функций показывают, какие заголовочные файлы нужны. Например, описание функции `printf()` требует использования файла `stdio.h`. Пропуск заголовочного файла может вообще не повлиять на выполнение некоторой конкретной программы, однако не нужно на это надеяться. Каждый раз, когда в этой книге используются библиотечные функции, вы будете применять включаемые файлы, определенные стандартом ISO/ANSI для этих функций.

Почему ввод и вывод не являются встроенными

Возможно, вас удивит, почему такая важная информация, как ввод и вывод, не включаются в программу автоматически. Один из ответов состоит то, что не все программы используют пакет ввода-вывода, а один из принципов языка С запрещает перегружать программу ненужными функциями. Этот принцип экономного использования ресурсов делает язык С особо удобным для написания встроенных программ, например, программного кода для процессора, управляющего автоматизированной подачей топлива.

Между прочим, строка с директивой #include вообще не является оператором языка С! Символ # в первой строке означает, что эта строка должна обрабатываться препроцессором до передачи ее компилятору. Далее вы столкнетесь с различными примерами команд препроцессора, а в главе 16 эта тема будет рассматриваться более подробно.

Функция main()

```
int main(void)
```

В этой строке программы объявляется функция с именем main. Действительно, main — более чем простое имя, однако это был единственно возможный выбор. Программа на языке С (с некоторыми исключениями, на которых мы сейчас не будем останавливаться) всегда начинается с выполнения функции main(). Вы всегда можете выбрать имя для любой другой функции, однако, чтобы начать выполнение программы, в ней обязательно должна присутствовать функция main(). А для чего нужны скобки? Они идентифицируют main() как функцию. Вскоре мы приступим к изучению функций. Сейчас просто следует знать, что функции представляют собой базовые модули программы на языке С. int — это возвращаемый тип функции main(). Это значит, что тип значения, которое может вернуть функция main(), является целочисленным. Вернуть куда? Да в операционную систему (этот вопрос мы рассмотрим в главе 6).

В круглых скобках, которые следуют за именем функции, обычно находится информация, передаваемая функциям. В условиях рассматриваемого простого примера ничего не передается, поэтому в скобках содержится слово void. (В главе 11 вводится еще один формат, позволяющий передавать информацию в функцию main() из операционной системы.)

Если просмотреть старые программные коды на С, довольно-таки часто можно встретить программы, которые начинаются со следующего формата:

```
main()
```

Стандарт С90 неохотно смирился с этой формой, а стандарт С99 вовсе ее не признает. Следовательно, даже если имеющийся у вас компилятор позволяет вам сделать это, у вас ничего не получится.

Вы можете также столкнуться со следующей формой:

```
void main()
```

Некоторые компиляторы допускают такую форму, но ни один стандарт не упоминает их даже в качестве возможного варианта. В силу этого обстоятельства компиляторы не должны воспринимать эту форму, и некоторые из них не воспринимают. Таким образом, ориентируйтесь на стандартную форму, и вы не будете иметь проблем при переносе с одного компилятора на другой.

Комментарии

```
/* простая программа */
```

Части программы, заключенные в символы /* */, представляют собой комментарии. Комментарии существенно облегчают понимание вашей программы всеми, кто ее изучает (в том числе и вам). Одно из полезных свойств комментариев в языке С заключается в том, что они могут быть помещены в любом месте программы, включая ту

же строку, где находится код, для пояснения которого они предназначаются. Более пространный комментарий может располагаться в собственной строке и даже занимать несколько строк. Все, что находится между открывающей (/*) и закрывающей (*/) последовательностями, компилятором игнорируется. Ниже представлены примеры правильных и неправильных форм комментариев:

```
/* Это комментарий на C. */
/* Этот комментарий расположен
   в двух строках. */
/*
  Вы также можете сделать это.
*/
/* Такой комментарий недопустим ввиду отсутствия маркера окончания.
```

Стандарт C99 позволяет использовать еще один стиль комментариев, а именно — тот, который предлагают языки C++ и Java. Новый стиль предполагает применение символов // для представления комментария, умещающегося в одной строке:

```
// Данный комментарий умещается в одной строке.
int rigue;      // Комментарий можно также поместить сюда.
```

Поскольку конец строки означает конец комментария, этот стиль требует маркера комментария только в начале.

Новая форма комментариев решает потенциальную проблему, характерную для старой формы комментария. Предположим, что имеется следующий программный код:

```
/*
Я надеюсь, что этот вариант работает.
*/
x = 100;
y = 200;
/* Теперь попробуем сделать что-нибудь еще. */
```

Предположим, что вы решили удалить четвертую строку, но случайно стерли также и третью строку (маркер */). В результате получаем такой код:

```
/*
Я надеюсь, что этот вариант работает.
y = 200;
/* Теперь попробуем сделать что-нибудь еще. */
```

Теперь компилятор соединяет в пару маркер /* из первой строки и маркер */ в четвертой строке, объединяя все четыре строки в один комментарий, включая строку, которая, по предположению, была частью программного кода. Поскольку форма // охватывает не более одной строки, это не приводит к проблеме “исчезновения кода”.

Некоторые компьютеры не поддерживают эту возможность, предусмотренную стандартом C99, другие могут потребовать изменить параметры компилятора, чтобы стали доступными функции, предусмотренные стандартом C99.

Мы считаем, что излишняя принципиальность может оказаться обременительной, поэтому в книге используются обе формы комментариев.

Скобки, тело функции и блоки

```
{
...
}
```

В листинге 2.1 фигурные скобки определяют границы функции `main()`. В общем случае все функции языка C используют фигурные скобки для обозначения начала и, соответственно, конца тела функции. Их наличие обязательно, так что не забывайте их проставить в нужных местах. Для этих целей можно применять только фигурные скобки (`{}`), но не круглые (`()`) или квадратные (`[]`).

Фигурные скобки также могут использоваться для организации операторов в блоки или модули в рамках функции. Если вам приходилось работать с языками Pascal, ADA, Modula-2 или Algol, вы заметите, что фигурные скобки подобны операторам `begin` и `end` в упомянутых языках.

Объявления

```
int num;
```

Эта строка программы называется *оператором объявления*. Оператор объявления относится к числу наиболее важных возможностей языка C. В рассматриваемом примере объявлены два объекта. Во-первых, где-то в функции у вас имеется *переменная* с именем `num`. Во-вторых, `int` объявляет `num` как целое число, то есть число без десятичной точки, или без дробной части. (`int` представляет собой пример *типа данных*.) Компилятор использует эту информацию для того, чтобы выделить в памяти подходящее пространство для переменной `num`. Точка с запятой в конце строки показывает, что данная строка является *оператором*, или командой, языка C. Точка с запятой является частью этого оператора, а не просто разделителем между операторами, как, например, в языке Pascal.

Слово `int` представляет собой *ключевое слово* языка C, обозначающее один из основных типов C. Ключевые слова — это слова, используемые для построения языковых конструкций, и вы не можете употреблять их в других целях. Например, вы не можете использовать `int` в качестве имени функции или переменной. Однако эти ограничения, налагаемые на применение ключевых слов, не выходят за рамки конкретных языков, и вы вполне можете дать имя "int" своему домашнему питомцу. (Правда, местные обычаи или законы могут запрещать подобное.)

Слово `num` в данном примере является *идентификатором*, то есть именем, которое вы выбираете для переменной, функции или какого-то другого логического объекта. Таким образом, объявление соединяет конкретный идентификатор с конкретной ячейкой в памяти компьютера и при этом устанавливает тип информации или тип данных, которые будут храниться в этой ячейке.

В языке C *все* переменные должны быть объявлены *до того*, как они будут использованы. Это значит, что вы должны составить списки всех переменных, которые вы используете в программе, и указать, к какому типу данных принадлежит каждая переменная. Объявление переменных считается хорошим тоном в программирования, а в языке C оно обязательно.

Традиционно, язык C требует, чтобы переменные были объявлены в начале того или иного блока, при этом объявлениям не могут предшествовать никакие другие типы операторов. Таким образом, тело функции `main()` может иметь следующий вид:

```
int main()      // традиционные правила
{
   int doors;
   int dogs;
   doors = 5;
   dogs = 3;
   // другие операторы
}
```

Стандарт C99, обобщая опыт работы с языком C++, теперь позволяет помещать объявления в любом месте блока. Тем не менее, вы все еще должны объявлять переменную, прежде чем ее использовать в первый раз. Следовательно, если ваш компилятор поддерживает эту возможность, ваш программный код может принять следующий вид:

```
int main()      // правила стандарта C99
{
   // несколько операторов
   int doors;
   doors = 5;  // первое использование переменной doors
   // дальнейшие операторы
   int dogs;
   dogs = 3;   // первое использование переменной dogs
   // остальные операторы
}
```

В целях большей совместимости с более ранними системами эта книга будет следовать первоначальным соглашениям. (Некоторые из современных компиляторов поддерживают возможности, определенные стандартом C99, только если вы задействуете их.)

В этом месте у вас, по-видимому, возникнут три вопроса. Первый: что такое тип данных? Второй: какие у вас имеются варианты выбора конкретного имени? Третий: а почему, собственно говоря, нужно объявлять переменные? Рассмотрим ответы на некоторые вопросы.

Типы данных

Язык C использует несколько видов (или типов) данных: например, целые числа, символы и числа с плавающей запятой. Объявление той или иной переменной как имеющей целочисленный или символьный тип позволяет компьютеру должным образом хранить, осуществлять выборку и интерпретировать данные. Множество возможных типов данных вы изучите в следующей главе.

Выбор имен

Для каждой переменной вы должны выбрать имена, имеющие содержательный смысл (например, `chip_count` вместо `x3`, если ваша программа подсчитывает количество микросхем). Если имени недостаточно, воспользуйтесь комментарием, чтобы объяснить, что представляет данная переменная. Документирование программы в таком стиле считается в программировании хорошим тоном.

Количество символов, которые вы можете использовать для именования переменной, зависит от конкретной реализации. Стандарт C99 разрешает использовать до 63 символов, если речь не идет о внешних идентификаторах (см. главу 12), в которых распознается только 31 символ. По сравнению с требованиями стандарта C90, который разрешал использовать, соответственно, 31 и шесть символов, это заметный прогресс. Более ранние версии компиляторов часто позволяли с этой целью применять максимум восемь символов. По сути дела, вы можете использовать большее число символов, чем указанный максимум, однако компилятор просто проигнорирует избыточные символы. В силу этого обстоятельства, в системах с восьмисимвольным ограничением имена `shakespeare` и `shakespencil` рассматриваются как одно и то же имя, поскольку первые восемь символов обоих имен совпадают. (Если вы хотите поупражняться с именем, состоящим из 63 символов, составьте его сами.)

В вашем распоряжении имеются буквы нижнего и верхнего регистров, цифры и знак подчеркивания (_). Первым символом должна быть буква или знак подчеркивания. Несколько примеров допустимых и недопустимых имен представлены в табл. 2.1.

Таблица 2.1. Допустимые и недопустимые имена

Допустимое имя	*Недопустимое имя*
`wiggles`	`$Z]**`
`cat2`	`2cat`
`Hot_Tub`	`Hot-Tub`
`taxRate`	`tax rate`
`_kcab`	`don't`

Операционная система и библиотека C часто используют идентификаторы с одним или двумя символами подчеркивания, например `_kcab`, так что лучше избегать употреблять эти символы. Стандартные идентификаторы, начинающиеся с одного или двух символов подчеркивания, такие как библиотечные идентификаторы, являются *зарезервированными*. Это означает, что хотя их использование не является синтаксической ошибкой, оно, тем не менее, приводит к конфликту имен. Имена в C *чувствительны к регистру* в том смысле, что буквы верхнего регистра рассматриваются как отличные от соответствующих букв нижнего регистра. По этой причине идентификатор `stars` отличается от `Stars` и `STARS`.

Для упрощения интернационального использования языка C стандарт C99 обеспечивает возможность работы с расширенным набором символов с помощью механизма *UCN* (Universal Character Names — имена в универсальных символах). За справками обращайтесь в приложение Б.

Четыре причины объявления переменных

Некоторые более ранние языки программирования, как, например, первоначальные варианты языков FORTRAN и BASIC, позволяют использовать переменные без их объявления. Почему нельзя применять этот упрощенный подход в C? На то существует целый ряд причин, часть из которых приведена ниже:

- Размещение объявлений всех переменных в одном месте упрощает читателю понимание того, для чего предназначена данная программа. Это особенно важ-

но в тех случаях, когда в вашей программе используются смысловые имена (например, taxrate вместо, скажем, r). Если для раскрытия смысла имени не достаточно, воспользуйтесь комментарием, поясняющим, что представляет собой конкретная переменная. Документирование программы в подобном стиле считается хорошим тоном в программировании.

- Мысли о том, какую переменную следует объявить, заставляют вас провести некоторое планирование, прежде чем погружаться в процесс написания кода. Какая информация нужна для того, чтобы начать писать программу? Какие выходные данные должна выдавать программа? Как наилучшим образом представить данные?
- Объявление переменных позволяет избежать одной из наиболее тонких и трудных для обнаружения программных ошибок, а именно — неправильно написанного имени переменной. Например, предположим, что во время использования одного из языков, в котором отсутствуют объявления, вы употребили следующий оператор:

  ```
  RADIUS1 = 20.4;
  ```

  и в каком-то другом месте программы вы неправильно ввели с клавиатуры оператор

  ```
  CIRCUM = 6.28 * RADIUSl;
  ```

  Вы не заметили, как вместо цифры 1 вы проставили букву l. Язык создает новую переменную с именем RADIUSl и использует случайное значение (ноль или просто "мусор"). Переменная CIRCUM получит неправильное значение, и вы потратите кучу времени, чтобы выяснить почему. В С это не может случиться (если, разумеется, у вас хватит ума объявить два таких похожих друг на друга имени), поскольку компилятор выдаст сообщение об ошибке, когда необъявленная переменная RADIUSl появится в программе.
- Ваша программа на С не скомпилируется, если вы не объявите переменные. Если перечисленные выше аргументы не возымеют действия, вас, возможно, убедит следующая серьезная мысль. Предположим, что вам необходимо объявить переменные, тогда где это сделать? Как уже говорилось выше, язык С до появления стандарта С99 требовал, чтобы все объявления выполнялись в начале блока. Один из доводов в пользу этого метода состоит в том, что группирование объявлений в каком-либо одном месте облегчает понимание того, *что* делает программа. Разумеется, существуют аргументы и в пользу распределения объявлений по всей программе, что разрешает делать стандарт С99. Идея заключается в том, чтобы объявлять переменную в моменты, когда вы будете готовы присвоить ей конкретное значение. В таких ситуациях вы едва ли забудете присвоить ей соответствующее значение. На практике, однако, многие компиляторы еще не поддерживают упомянутое нововведение стандарта С99.

Присваивание

```
num = 1;
```

В следующей строке программы представлен оператор присваивания — одна из основных операций языка С.

В рассматриваемом примере эта операция требует "присвоить значение 1 переменной num". Предшествующая ему строка `int num;` резервирует в памяти компьютера пространство для хранения переменной num, а строка с оператором присваивания записывает значение в эту ячейку. Позже вы можете присвоить переменной num другое значение, если возникнет такая необходимость; вот почему num называется переменной. Обратите внимание, что оператор присваивания назначает конкретное значение справа налево. Кроме того, этот оператор также оканчивается точкой с запятой, как показано на рис. 2.2.

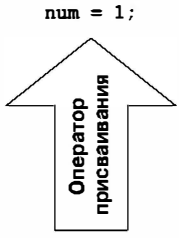


Рис. 2.2. *Оператор присваивания является одной из базовых операций языка C*

Функция `printf()`

```
printf("Я простой ");
printf("компьютер.\n");
printf("Моей любимой цифрой является %d, так как она первая.\n",num);
```

Во всех этих строках используется стандартная функция C с именем `printf()`. Круглые скобки показывают, что `printf` является именем функции. Материал, заключенный в круглые скобки, представляет собой информацию, передаваемую из функции `main()` в функцию `printf()`.

Например, первая строка передает функции `printf()` фразу `Я простой`. Такая информация называется *аргументом* или, более точно, фактическим аргументом функции (рис. 2.3). Что делает функция `printf()` с этим аргументом? Она просматривает все, что заключено в двойные кавычки, и выводит этот текст на экран.

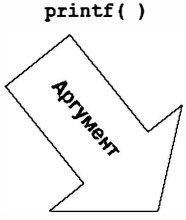


Рис. 2.3. *Функция `printf()` с аргументом*

Первая строка с функцией `printf()` являет собою пример того, как *вызвать* или как *обратиться* к функции в C. Для этого достаточно назвать имя функции и поместить нужный аргумент (аргументы) в круглые скобки. Когда дойдет дело до этой строки, управление передается указанной функции (`printf()` в рассматриваемом случае). Как только функция выполнит свою задачу, управление возвращается в исходную, то есть в *вызывающую* функцию, в данном примере — `main()`.

Чем отличается следующая строка `printf()`? В этом случае в кавычках присутствуют символы `\n`, однако они не выводятся на печать! Почему так происходит? Символы `\n` означают начало новой строки. Комбинация `\n` (вводится как два символа) представляет один символ, получивший название символа новой строки. Для функции `printf()` она означает "начать новую строку с крайней левой позиции".

Другими словами, печать символа новой строки выполняет ту же операцию, что и нажатие клавиши <Enter> на стандартной клавиатуре. Почему не используется клавиша <Enter> при печати аргументов функции `printf()`? Да потому, что это будет воспринято как непосредственная команда вашему редактору, но не как команда, которую необходимо сохранить в вашем исходном коде. Другими словами, когда вы нажмете клавишу <Enter>, редактор покинет строку, с которой вы работаете, и начнет новую строку. В то же время символ новой строки определяет, какой вид примут выходные данные программы при отображении на экране.

Символ новой строки представляет собой пример управляющей последовательности. Управляющая последовательность используется для представления символов, которые трудно или просто невозможно ввести с клавиатуры. Примерами таких последовательностей могут служить символы `\t` для представления нажатия клавиши <Tab> и `\b` — для <Backspace>. В любом случае управляющая последовательность начинается с косой черты с левым уклоном ("слеша") `\`. Мы вернемся к этой теме в главе 3.

Таким образом, все это объясняет, почему три оператора `printf()` вывели только две, а не три строки: первый оператор не содержит символа новой строки, зато второй и третий — содержат.

Завершающая строка `printf()` привносит еще одну странность: что будет напечатано вместо комбинации символов `%d`? Вспомните, что выходные данные этой строки выглядят следующим образом:

```
Моей любимой цифрой является 1, так как она первая.
```

Вот именно! При печати этой строки вместо группы символов `%d` появилась цифра `1`, а `1` — это значение переменной `num`. Комбинация `%d` представляет собой заполнитель, который показывает, где должно быть распечатано значение переменной `num`. Эта строка подобна следующему оператору BASIC:

```
PRINT "Моей любимой цифрой является "; num; ", так как она первая."
```

Версия C делает немного больше, чем это. Символ `%` уведомляет программу, что в этом месте будет напечатана переменная, а `d` указывает на то, что переменная должна быть напечатана как десятичное целое число. Функция `printf()` предлагает на выбор несколько вариантов, включая и шестнадцатеричные целые числа, и числа с плавающей запятой. Действительно, `f` в `printf()` является напоминанием о том, что это форматирующая функция печати. Каждый тип данных имеет собственный спецификатор; по мере того, как в данной книге будут вводиться все новые типы, будут также вводиться и соответствующие спецификаторы.

Оператор возврата

```
return 0;
```

Оператор возврата является завершающим оператором рассматриваемой программы. `int` в конструкции `int main(void)` означает, что функция `main()` возвращает целое значение. Стандарт языка C требует, чтобы поведение функции `main()` было именно таким. Функции C, возвращающие значения, делают это с помощью оператора возврата, состоящего из ключевого слова `return`, за которым следует возвращаемое значение и точка с запятой. Если вы не укажете в функции `main()` оператор возврата, большинство компиляторов уведомит вас о его отсутствии, но при этом компиляция программы прерываться не будет.

На этом этапе вы можете считать оператор возврата в функции `main()` чем-то необходимым для обеспечения логической завершенности, однако в некоторых операционных системах, в том числе DOS и Unix, он имеет практическое применение. В главе 11 эта тема рассматривается более подробно.

Структура простой программы

Теперь, после ознакомления с конкретным примером, вы готовы к изучению нескольких общих правил написания программ на языке C. *Программа* состоит из совокупности одной или нескольких функций, одна из которых обязательно должна быть названа `main()`. Описание *функции* состоит из заголовка и тела функции. *Заголовок* содержит операторы препроцессора, такие как `#include`, а также имя функции. Вы можете распознать имя функции по круглым скобкам, внутри которых может быть пусто. *Тело* функции заключено в фигурные скобки ({}) и состоит из некоторой последовательности операторов, при этом каждый оператор завершается точкой с запятой (рис. 2.4).

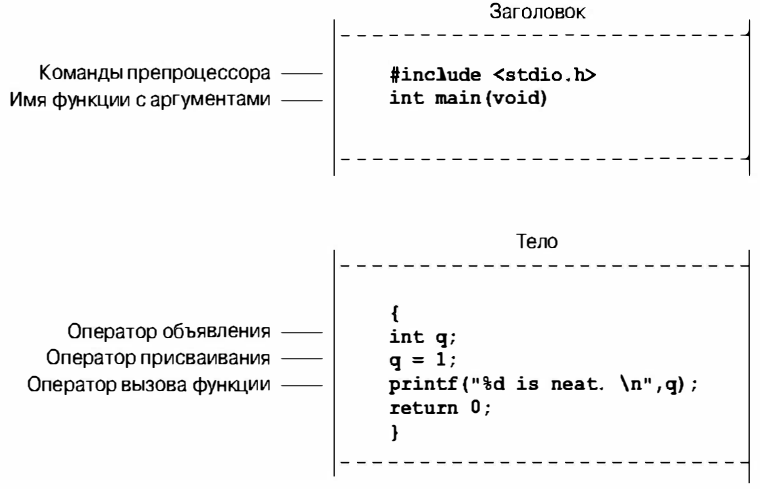


Рис. 2.4. *У функции имеется заголовок и тело*

В примере, приведенном в данной главе, использовался *оператор объявления*, объявляющий имя и тип переменной. В нем также присутствовал *оператор присваивания*, задавая значение переменной. Кроме того, в нем использовались три оператора печати, каждый из которых вызывал функцию `printf()`. Эти операторы печати представляют собой примеры *операторов вызова функции*. И, наконец, `main()` завершается оператором возврата. Короче говоря, простая стандартная программа на C должна быть представлена в следующем формате:

```
#include <stdio.h>
int main(void)
{
  операторы
  return 0;
}
```

Советы касательно удобства чтения программы

Написание удобочитаемых программ является хорошим тоном в программировании. Удобочитаемую программу легче понять, ее проще корректировать и модифицировать. Процесс придания программе удобочитаемого вида также помогает проверить вашу собственную концепцию того, что делает программа.

Вы уже ознакомились с двумя методами повышения удобочитаемой программы: выбор смысловых имен переменных и использование комментариев. Обратите внимание на то, что оба эти метода дополняют друг друга. Если вы присвоите переменной имя `width`, тогда не нужен комментарий, который говорит, что эта переменная представляет ширину, в то же время переменная с именем `video_routine_4` ("видеопрограмма 4") требует пояснения, для чего она предназначена. Существует также метод, использующий пустые строки для отделения одного концептуального раздела функции от другого. Например, в простой демонстрационной программе присутствует пустая строка, отделяющая раздел объявлений от раздела действий. В языке С пустые строки не обязательны, в то же время они повышают удобочитаемость программы.

Четвертый метод использует одну строку на каждый оператор. И снова это вопрос соглашений, повышающих удобочитаемость программ, но отнюдь не требование языка С. В С принят так называемый формат свободной формы. Вы можете размещать несколько операторов в одной строке или расположить один оператор в нескольких строках. Приведенный ниже программный код вполне допустим, но очень неудобен:

```
int main( void ) { int four; four
=
4
;
printf(
       "%d\n",
four); return 0;}
```

Точка с запятой уведомляет компилятор, где заканчивается один оператор и начинается другой, в то же время программная логика становится более понятной, если вы будете соблюдать соглашение, действующее в рамках примера, рассматриваемого в настоящей главе (рис. 2.5).

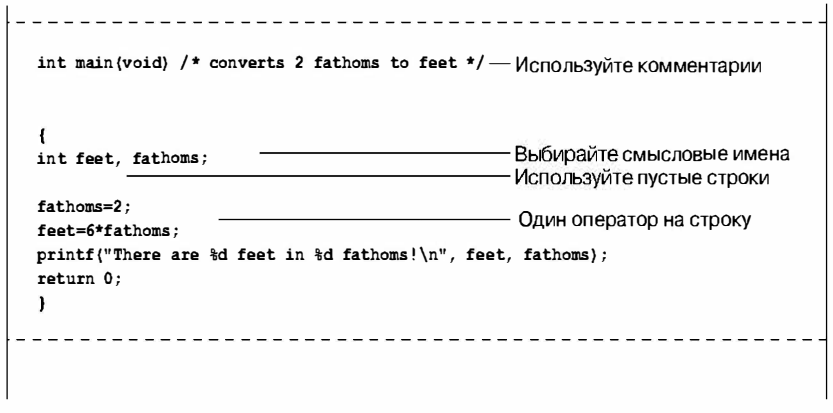


Рис. 2.5. *Приведение программы к удобочитаемому виду*

Еще один шаг в использовании языка C

Первая демонстрационная программа была совсем простой, и следующий пример, представленный в листинге 2.2, ничуть не труднее.

Листинг 2.2. Программа `fathm_ft.c`

```
// fathm_ft.c – преобразует 2 морских сажени в футы
#include <stdio.h>
int main(void)
{
   int feet, fathoms;

   fathoms = 2;
   feet = 6 * fathoms;
   printf("В %d морских саженях содержится %d футов!\n", fathoms, feet);
   printf("Да, я сказал %d футов!\n", 6 * fathoms);

   return 0;
}
```

Что нового в этой программе? Код предоставляет описание программы, объявляет несколько переменных, выполняет операции умножения и распечатывает значения двух переменных. Рассмотрим все это более подробно.

Документирование

Во-первых, программа начинается с комментария (используется новый стиль), в котором указано имя файла программы и назначение программы. Этот вид документирования не требует много времени, зато он принесет большую помощь позднее, когда вы будете просматривать сразу несколько файлов или распечатывать их.

Множественные объявления

Обратите внимание, что программа объявляет две переменных, а не использует оператор объявления для каждой переменной. Чтобы сделать это, в операторе объявления отделяйте переменные друг от друга (`feet` и `fathoms`) запятыми. То есть:

```
int feet, fathoms;
```

и

```
int feet;
int fathoms;
```

эквивалентны.

Умножение

В-третьих, данная программа выполняет умножение. Она использует огромную вычислительную мощь компьютерной системы с тем, чтобы умножить 2 на 6. В C, как и во многих других языках программирования, * представляет собой символ умножения.

Поэтому оператор

```
feet = 6 * fathoms;
```

означает "получите значение переменной fathoms, умножьте его на 6 и присвойте результат вычисления переменной feet".

Распечатка нескольких значений

И, наконец, эта программа оригинальным образом использует функцию printf(). Если вы откомпилируете и выполните пример, выходные результаты будут иметь примерно следующий вид:

```
В 2 морских саженях содержится 12 футов!
Да, я сказал 12 футов!
```

На этот раз программа делает две подстановки при первом вызове функции printf(). Первая комбинация символов %d в кавычках была заменена на значение первой переменной (fathoms) в списке, который следует за сегментом в кавычках, а вторая такая комбинация была заменена значением второй переменной (feet) из этого списка. Обратите внимание на то, что список переменных, предназначенных для печати, находится в хвостовой части этого оператора, непосредственно за частью, заключенной в кавычки. Отметим также, что каждый элемент списка отделен от остальных запятой.

Второе использование функции printf() показывает, что распечатываемое значение не обязательно должно быть переменной, оно вполне может быть выражением наподобие 6 * fathoms, которое приводится к соответствующему типу.

Эта программа достаточно проста, однако она может служить ядром программы перевода морских саженей в футы. Все что для этого нужно — это присваивание дополнительных значений в интерактивном режиме; далее в этой главе мы покажем, как это сделать.

Множество функций

До сих пор в программах использовались стандартные функции printf(). В листинге 2.3 демонстрируется возможность внедрения в программу наряду с функцией main() вашей собственной функции.

Листинг 2.3. Программа two_func.c

```
/* two_func.c — программа, использующая две функции в одном файле */
#include <stdio.h>
void butler(void);   /* прототип функции в стандарте ISO/ANSI C */
int main(void)
{
   printf("Я вызываю дворецкого.\n");
   butler();
   printf("Да. Принесите мне чай и записываемые компакт-диски.\n");
   return 0;
}
```

```
void butler(void)   /* начало определения функции */
{
   printf("Вы звонили, сэр?\n");
}
```

Выходные данные будут иметь следующий вид:

```
Я вызываю дворецкого.
Вы звонили, сэр?
Да. Принесите мне чай и записываемые компакт-диски.
```

Функция `butler()` трижды появляется в рассматриваемой программе. В первый раз она появляется в виде прототипа, передающего компилятору информацию о функциях, которые будут использованы в данной программе. Во второй раз она появляется в `main()` в форме вызова функции. И, наконец, в данной программе представлено определение функции, которое является исходным кодом самой функции. Рассмотрим по очереди каждое из этих трех появлений.

Прототипы были введены стандартом C90, поэтому более ранние компиляторы их не распознают. (Вскоре будет дано объяснение, что делать, когда приходится работать с такими компиляторами.) Прототип — это форма объявления, которая уведомляет компилятор о том, что вы используете конкретную функцию. Он также определяет свойства этой функции. Например, первое ключевое слово `void` в прототипе функции `butler()` указывает на то, что `butler()` не имеет возвращаемого значения. (В общем случае функция может возвратить значение в вызывающую функцию для последующего его использования, но `butler()` этого не делает.) Второе `void`, то, что в указано в функции `butler(void)`, означает, что функция `butler()` не принимает аргументов. Поэтому, когда достигается место в `main()`, в котором вызывается `butler()`, выполняется проверка корректности использования функции `butler()`. Обратите внимание на тот факт, что ключевое слово `void` употребляется в смысле “пусто”, а не в смысле “неправильно”.

Ранние версии C поддерживали более ограниченную форму объявления функции, в которой вы могли определить только возвращаемый тип, опуская при этом описания аргументов:

```
void butler();
```

В более ранних программных кодах на C использовались объявления функций, подобные представленным выше, но не прототипы функций. Стандарты C90 и C99 распознают ранние версии, но в то же время указывают на то, что их употребление постепенно сворачивается, и советуют их больше не применять. Если вы имеете дело с какой-то старой программой, возможно, потребуется привести объявления старого типа к прототипам. В последующих главах настоящей книги мы вернемся к изучению прототипов, объявлениям функций и возвращаемым значениям.

Далее, вы вызываете функцию `butler()` внутри `main()`, просто указав ее имя, включая круглые скобки. Когда функция `butler()` завершит свою работу, программа переходит к выполнению следующего оператора в `main()`.

И, наконец, функция `butler()` объявлена точно так же, как и функция `main()`, с заголовком и телом, заключенным в фигурные скобки. Заголовок повторяет информацию, заданную прототипом: `butler()` не принимает никаких аргументов и не возвращает значения. Для компиляторов ранних версий опустите второе `void`.

Еще один момент, на который следует обратить внимание, заключается в том, что место вызова функции butler() в функции main() не является местом определения butler() в файле, место вызова указывает, когда функция butler() должна быть выполнена. В этой программе вы могли бы, например, поместить определение butler() над определением функции main(), и программа выполнялась так же, то есть функция butler() будет выполнена между двумя обращениями к printf() в функции main(). Напомним, что все программы на C начинаются с выполнения функции main(), при этом не имеет значения, в каком месте программного файла эта функция находится. В то же время на практике, как правило, в списке функций программы на C первой идет main(), поскольку она обычно играет роль базового каркаса программы.

Стандарт C рекомендует, чтобы вы предусматривали в программе прототипы для всех функций, которые используете. Стандартные файлы include берут на себя эту обязанность в отношении стандартных библиотечных функций. Например, в соответствии со стандартом языка C, файл stdio.h содержит прототип функции printf(). Заключительный пример в главе 6 демонстрирует способ расширения прототипирования на функции без ключевого слова void, а в главе 9 эти функции рассматриваются более подробно.

Предварительные сведения об отладке

Теперь, когда вы научились писать простые программы на C, у вас появился "шанс" допускать простые ошибки. Программная ошибка в английском языке обозначается словом "*bug*" ("дефект"), а выявление и исправление ошибок называется *отладкой* ("debugging").

В листинге 2.4 представлена программа, в которой допущено несколько программных ошибок. Проверьте, сколько программных ошибок вы сможете обнаружить.

Листинг 2.4. Программа nogood.c

```
/* nogood.c — программа с ошибками */
#include <stdio.h>
int main(void)
(
   int n, int n2, int n3;
/* В этой программе допущено несколько ошибок
   n = 5;
   n2 = n * n;
   n3 = n2 * n2;
   printf("n = %d, n в квадрате = %d, n в кубе = %d\n", n, n2, n3)
   return 0;
)
```

Синтаксические ошибки

Код в листинге 2.4 содержит несколько синтаксических ошибок. Вы совершаете синтаксические ошибки, когда не следуете правилам языка C. Это можно сравнить с грамматическими ошибками в обычном тексте. В качестве примера рассмотрим следующее предложение: "Быть программные ошибки катастрофические могут".

В этом предложении используются правильные слова, однако порядок их следования нарушен. Синтаксические ошибки в языке C заключаются в том, что символы этого языка размещаются не в тех местах.

Итак, какие синтаксические ошибки допущены в программе `nogood.c`? Прежде всего, для отмечания тела функции в ней используются круглые скобки вместо фигурных, то есть правильные, в общем-то, символы языка C расставлены в неверных местах. Во-вторых, объявления должны быть такими:

```
int n, n2, n3;
```

или, по крайней мере, такими:

```
int n;
int n2;
int n3;
```

Далее, в этом примере опущена пара символов `*/`, необходимая для завершения комментария. (Также можно было бы заменить пару `/*` новой формой комментария `//`.) И, наконец, пропущена точка с запятой, необходимая для завершения оператора `printf()`.

Как обнаружить синтаксические ошибки? Во-первых, перед компиляцией вы можете просмотреть исходный код — возможно, вы обнаружите какие-либо очевидные ошибки. Во-вторых, вы можете проанализировать ошибки, найденные компилятором, так как одной из его обязанностей является именно обнаружение синтаксических ошибок. Во время компиляции такой программы компилятор сообщает об обнаруженных им ошибках, а также указывает природу и местоположение каждой ошибки.

Однако и сам компилятор может ошибаться. Истинная синтаксическая ошибка может ввести компилятор в заблуждение и заставить "думать", что он нашел другие ошибки. Например, поскольку в примере неправильно объявлены переменные `n2` и `n3`, компилятор может подумать, что он нашел еще несколько ошибок в тех местах, в которых эти переменные используются. Фактически вместо того, чтобы пытаться исправлять все обнаруженные им ошибки сразу, вам следует исправить сначала одну или две ошибки, после чего выполнить повторную компиляцию; вполне возможно, что при этом исчезнет и ряд других ошибок. Продолжайте в том же духе, пока программа не заработает. Еще одна распространенная причуда компилятора состоит в том, что он обнаруживает ошибку на строку ниже. Например, компилятор может не догадаться, что не хватает точки с запятой, пока не наступит очередь компиляции следующей строки. Таким образом, если компилятор жалуется о пропаже точки с запятой в строке, в которой уже есть один такой знак, проверьте предыдущую строку.

Семантические ошибки

Семантические ошибки — это смысловые ошибки. В качестве примера рассмотрим следующее предложение: "Бешеная инфляция думает зеленые мысли". Синтаксических ошибок оно не содержит, ибо прилагательные, существительные, глаголы и наречия находятся в нужных местах, тем не менее, само предложение не имеет смысла. В языке C вы совершаете семантическую ошибку, если соблюдаете все требования языка, но получаете неверный результат. В рассматриваемом примере присутствует одна такая ошибка:

```
n3 = n2 * n2;
```

В данном случае предполагается, что n3 представляет куб числа n, в то время как код вычисляет четвертую степень n. Компилятор не обнаруживает семантических ошибок, поскольку правила языка C не нарушены. Компилятор не способен предугадывать ваши истинные намерения. Так что обнаруживать ошибки такого вида придется самому. Один из способов заключается в сравнении того, что программа делает, с тем, что вы хотите от нее получить. Например, предположим, что вы исправили синтаксические ошибки в рассматриваемом примере, так что теперь программа приобретает вид, представленный в листинге 2.5.

Листинг 2.5. Программа `stillbad.c`

```
/* stillbad.c — программа с устраненными синтаксическими ошибками */
#include <stdio.h>
int main(void)
{
   int n, n2, n3;
/* В этой программе есть семантическая ошибка */
   n = 5;
   n2 = n * n;
   n3 = n2 * n2;
   printf("n = %d, n в квадрате = %d, n в кубе = %d\n", n, n2, n3);
   return 0;
}
```

Выходные данные этой программы имеют следующий вид:

```
n = 5, n в квадрате = 25, n в кубе = 625
```

Легко заметить, что 625 не является правильным результатом возведения числа в третью степень, стало быть, этот результат неправильный. Следующий этап предусматривает отслеживание того, как вы получили такой результат. В условиях данного примера вы, возможно, сможете выявить ошибку путем инспекции кода. В общем случае, однако, вы должны воспользоваться более систематизированным подходом. Один из методов заключается в том, что вы берете на себя роль компьютера и шаг за шагом проходите по всей программе. Воспользуемся этим методом и в данном случае.

Программа начинает свое выполнение с объявления трех переменных: n, n2 и n3. Вы можете смоделировать эту ситуацию, нарисовать три прямоугольника и присвоить им имена переменных (рис. 2.6). Далее программа присваивает переменной n значение 5. Смоделируйте это действие, записав 5 в прямоугольник n. Затем программа умножает n на n и присваивает результат переменной n2; посмотрев в прямоугольник n, вы увидите, что в нем находится значение 5, умножьте 5 на 5 и получите 25, после чего поместите 25 в прямоугольник n2. Чтобы воспроизвести следующий оператор C (`n3 = n2 * n2;`), загляните в прямоугольник n2 и там найдете 25. Умножьте 25 на 25, получите 625, и поместите его n3. Вот оно что! Вы возводите в квадрат n2 вместо того, чтобы умножить его на n.

Итак, возможно, эта процедура избыточна для данного примера, однако подобного рода пошаговое выполнение этой программы часто является наилучшим способом посмотреть, что в конечном итоге произойдет.

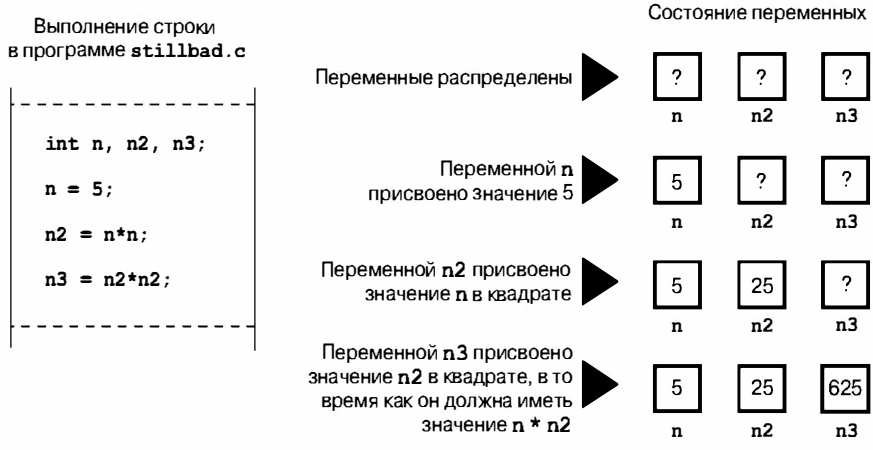


Рис. 2.6. *Трассировка программы*

Состояние программы

Выполняя пошаговый просмотр программы вручную, отслеживая каждую переменную, вы осуществляете мониторинг состояния программы. Состояние программы — это просто набор значений всех переменных в заданной точке выполнения программы. Другими словами, это моментальный снимок текущего состояния вычислений.

Мы обсудили один из методов отслеживания состояния: самостоятельно осуществить пошаговое выполнение программы. В программе, которая делает, скажем, 10 000 итераций, вы не сумеете справиться с этой задачей. Тем не менее, вы можете выполнить несколько итераций, чтобы узнать, сможет ли программа сделать то, что от нее хотите. В то же время, всегда существует возможность, что вы выполните эти шаги, как вам это нужно, а не так, как вы реализовали их в программе, так что попытайтесь неукоснительно придерживаться фактического кода. Еще один подход при выявлении семантических ошибок заключается в размещение дополнительных операторов `printf()` в разных местах программы с целью текущего контроля избранных переменных в ключевых точках программы. Наблюдение за тем, как меняются эти значения, возможно, подскажет вам, что происходит. После того, как вы добьетесь, чтобы работа программы вас удовлетворила, вы можете убрать дополнительные операторы и выполнить повторную компиляцию.

Третий метод изучения состояний программы предусматривает использование отладчиков. *Отладчик* представляет собой программу, которая позволяет выполнять другую программу в пошаговом режиме и исследовать значения программных переменных. Отладчики характеризуются различными уровнями удобства использования и сложности. Наиболее совершенные отладчики показывают исполняемую строку исходного кода. Это особенно удобно при отладке программ с альтернативными путями выполнения, поскольку легко видеть, по какому конкретному пути продвигается выполнение программы. Если ваш компилятор был установлен с отладчиком, не пожалейте сейчас времени на его изучение. Например, опробуйте его на программе, представленной в листинге 2.4.

Ключевые слова и зарезервированные идентификаторы

Ключевые слова образуют словарь языка C. Поскольку они играют в C особую роль, вы не можете их использовать, например, в качестве идентификаторов или имен переменных. Многие из этих ключевых слов описывают различные типы данных, например, `int`. Другие, такие как `if`, служат для управления порядком исполнения операторов программы. В приведенном ниже перечне ключевых слов языка C (табл. 2.2) полужирным выделены ключевые слова, введенные в употребление стандартом ISO/ANSI C90, а курсивом показаны ключевые слова, введенные стандартом C99.

Таблица 2.2. Ключевые слова языка C

Ключевые слова стандарта ISO/ANSI C			
auto	**enum**	restrict	unsigned
break	extern	return	**void**
case	float	short	**volatile**
char	for	**signed**	while
const	goto	sizeof	*_Bool*
continue	if	static	*_Complex*
default	*inline*	struct	*_Imaginary*
do	int	switch	
double	long	typedef	
else	register	union	

Если вы попытаетесь использовать какое-либо ключевое слово, скажем, для имени переменной, компилятор воспримет это как синтаксическую ошибку. Существуют и другие идентификаторы, так называемые *зарезервированные идентификаторы*, которые вы также не должны использовать для этих целей. Они не вызывают синтаксических ошибок, поскольку являются допустимыми именами. Однако язык уже использует их или зарезервировал за собой право на их использование, так что у вас могут возникнуть проблемы в случае, если вы воспользуетесь ими для каких-то других целей. Зарезервированными идентификаторами являются идентификаторы, начинающиеся с символа подчеркивания, а также имена стандартных библиотечных функций, такие как, например, `printf()`.

Ключевые понятия

Программирование представляет собой довольно трудное занятие. Оно требует абстрактного мышления на уровне идей и вместе с тем учета необходимых деталей. Вскоре вы убедитесь в том, что компилятор требует внимательного отношения к деталям. Когда вы говорите с хорошо знакомыми вам людьми, вы можете использовать то или иное слово не по правилам, совершить пару грамматических ошибок, возможно, некоторые предложения оставить неоконченными, тем не менее, ваши знакомые поймут, что вы хотите сказать. Но компилятор не допускает подобных вольностей, он придерживается принципа “почти правильно — это тоже неправильно”.

Компилятор бессилен вам чем-нибудь помочь в концептуальных вопросах, каковыми являются указанные выше, поэтому в данной книге мы стремимся заполнить этот пробел, выделяя основные идеи в каждой главе.

В этой главе основной вашей задачей должно быть понимание того, что собой представляет программа на языке С. Вы можете рассматривать программу как описание желаемого поведения компьютера, которое вы формулируете сами. Компилятор выполняет поистине кропотливую работу преобразования этого вашего описания в базовый машинный язык.

(Для того чтобы вы оценили, какую огромную работу выполняет компилятор, отметим, что он создает исполняемый файл размером в 60 Кбайт на базе исходного файла размером 1 Кбайт; большие объемы машинных кодов затрачиваются на представление в машинном языке даже простейших программ на С.) Поскольку у компьютера истинный интеллект отсутствует, вы должны представить свое описание в выражениях, понятных компилятору; такими выражениями являются формальные правила, установленные стандартом языка С. (И хотя данный подход накладывает достаточно жесткие ограничения, тем не менее, это лучше, чем составлять такое описание непосредственно на машинном языке!)

Компилятор принимает только те команды, которые представлены в специальном формате, который мы подробно описали в данной главе. Ваша обязанность как программиста заключается в том, чтобы выразить свое видение того, как программа должна работать, используя средства, которые компилятор, направляемый стандартом языка С, может успешно обработать.

Резюме

Программа на языке С состоит из одного или большего числа функций С. Каждая программа на С должна содержать функцию с именем `main()`, поскольку именно эта функция вызывается в момент запуска программы. Простая функция состоит из заголовка, за которым следует открывающая фигурная скобка, далее следуют операторы, образующие тело функции, за которой следует завершающая или *закрывающая* фигурная скобка.

Каждый оператор языка С является инструкцией компьютеру и обязательно заканчивается точкой с запятой. Оператор объявления назначает переменной имя и определяет тип данных, которые будет храниться в этой переменной. Имя переменной может служить примером идентификатора. Оператор присваивания устанавливает значение переменной или, используя более общий термин, выделяет ей пространство в памяти. Вызов функции запускает на выполнение функцию, имя которой указано в обращении. По выполнении вызванной функции программа переходит к выполнению оператора, следующего за вызовом функции.

Функция `printf()` может использоваться для печати фраз и значений переменных. Синтаксис языка — это набор правил, определяющий способ, посредством которого допустимые операторы в этом языке группируются в единую последовательность. Семантика оператора есть его смысловое значение. Компилятор позволяет вам обнаруживать синтаксические ошибки, однако синтаксические ошибки проявляются в программе лишь после того, как эта программа будет откомпилирована. Обнаружение семантических ошибок предусматривает мониторинг состояния программы, то есть, значений всех переменных на каждом шаге выполнения программы. И, наконец, ключевые слова образуют словарь языка С.

Вопросы для самоконтроля

Ответы на эти вопросы находятся в приложении А.

1. Как называются базовые модули программы на языке С?
2. Что такое синтаксическая ошибка? Приведите примеры синтаксической ошибки в контексте вашего родного языка и языка С.
3. Что такое семантическая ошибка? Приведите примеры семантической ошибки в контексте вашего родного языка и языка С.
4. Ленивец из Индианы написал и представил вам на одобрение следующую программу. Пожалуйста, помогите ему исправить допущенные в ней ошибки.

```
include studio.h
int main{void} /* программа печатает количество недель в году /*
(
int s

s := 56;
print(В году s недель.);
return 0;
```

5. Предположим, что каждый из приведенных ниже примеров является частью некоторой законченной программы. Что распечатает каждая такая часть?

   а.
   ```
   printf("Бе-е, Бе-е, Черная Овечка.");
   printf("У тебя найдется шерсть для меня?\n");
   ```

   б.
   ```
   printf("Убирайся!\Нет, толстое создание! ");
   ```

   в.
   ```
   printf("Что?\nНет/nC ума сошел?\n");
   ```

   г.
   ```
   int num;
   num = 2;
   printf("%d + %d = %d", num, num, num + num);
   ```

6. Какие из следующих слов являются ключевыми в С: `main`, `int`, `function`, `char`, =?
7. Как вы будете печатать значения `words` и `lines` в форме `Было 3020 слов и 350 строк`? В данном случае `3020` и `350` представляют значения двух указанных выше переменных.
8. Рассмотрим следующую программу:

```
#include <stdio.h>
int main(void)
{
   int a, b;

   a = 5;
   b = 2; /* строка 7 */
   b = a; /* строка 8 */
   a = b; /* строка 9 */
   printf("%d %d\n", b, a);
   return 0;
}
```

Каково состояние программы после выполнения строки 7? Строки 8? Строки 9?

Упражнения по программированию

Для изучения языка C одного чтения книг не достаточно. Вы должны попытаться написать несколько простых программ, чтобы посмотреть, так ли гладко идет написание программы, как это описано в данной главе. Мы дадим вам несколько советов, однако вы сами должны продумать решение существующих задач. Ответы на избранные упражнения по программированию вы найдете на Web-сайте издательства этой книги.

1. Напишите программу, которая использует вызов функции `printf()` для печати вашего имени и фамилии в одной строке, использует второй вызов функции `printf()`, чтобы напечатать ваше имя и фамилию в двух строках, и использует два вызова функции `printf()` для печати вашего имени и фамилии в одной строке. Выходные данные должны иметь следующий вид (при этом используются ваши персональные данные):

   ```
   Иван Иванов       ←Первый оператор печати
   Иван              ←Второй оператор печати
   Иванов            ←Все еще второй оператор печати
   Иван Иванов       ←Третий и четвертый операторы печати
   ```

2. Напишите программу, печатающую ваше имя и адрес.
3. Напишите программу, которая преобразует ваш возраст в годах в количество дней и отображает на экране оба значения. На этой стадии можно учитывать только прожитые годы и не учитывать високосные года.
4. Напишите программу, печатающую следующие выходные данные:

   ```
   Наш Билли — хороший парень!
   Наш Билли — хороший парень!
   Наш Билли — хороший парень!
   Наш Билли лучше всех!
   ```

   В этой программе в дополнение к функции `main()` следует применять функции, определенные пользователем: одна из них один раз печатает сообщение о хорошем парне, вторая печатает один раз завершающую строку.
5. Напишите программу, которая создает целочисленную переменную с именем `toes`. Программа должна присвоить переменной `toes` значение `10`. Наряду с этим, программа должна вычислить, чему равно значение удвоенного `toes` и чему равен квадрат `toes`. Программа должна напечатать все три значения, сделав соответствующие пометки.
6. Напишите программу, которая выдает следующие выходные данные:

   ```
   Улыбайся! Улыбайся! Улыбайся!
   Улыбайся! Улыбайся!
   Улыбайся!
   ```

   В программе должна быть определена функция, которая отображает строку `Улыбайся!` один раз, в то же время программа может использовать эту функцию столько раз, сколько надо.
7. Напишите программу, которая вызывает функцию с именем `one_three()`. Эта функция должна напечатать слово `один` в одной строке, вызвать функцию `two()`, а затем напечатать слово `три` в одной строке. Функция `two()` должна отобразить слово `два` в одной строке. Функция `main()` должна вывести фразу `начать сейчас:` перед вызовом функции `one_three()` и напечатать `порядок!` после ее вызова. Таким образом, выходные данные должны иметь следующий вид:

   ```
   начать сейчас:
   один
   два
   три
   порядок!
   ```

ГЛАВА 3

Представление данных в языке C

В этой главе:

- Ключевые слова: `int`, `short`, `long`, `unsigned`, `char`, `float`, `double`, `_Bool`, `_Complex`, `_Imaginary`
- Операция: `sizeof`
- Функция: `scanf()`
- Базовые типы данных в языке C
- Отличия между целочисленными данными и данными с плавающей запятой
- Написание констант и объявление переменных известных типов
- Использование функций `printf()` и `scanf()` для чтения и записи значений различных типов

Программы работают с данными. Вы вводите числа, буквы и слова в компьютер для того, чтобы он выполнил какие-нибудь действия над этой информацией. Например, вам может потребоваться, чтобы компьютер посчитал прибыль и отобразил на экране отсортированный список виноторговцев. В этой главе вы будете не просто читать о данных — вы будете на практике производить с данными различные действия, что намного интереснее.

В этой главе мы рассматриваем два больших семейства данных: целые числа и числа с плавающей запятой. В языке C имеется несколько разновидностей этих данных. Вы узнаете о том, какие типы данных существуют, как объявлять данные различных типов, как и когда их применять. Кроме того, вы поймете, чем константы отличаются от переменных, а приложенные вами усилия принесут первый успех — вы сможете написать свою первую интерактивную программу.

Демонстрационная программа

И снова мы начинаем с демонстрационной программы. Как и прежде, вы найдете несколько новых, незнакомых деталей, назначение которых будет вскоре объяснено. Общий смысл программы должен быть ясен, поэтому попробуйте скомпилировать и выполнить исходный код, представленный в листинге 3.1. Для экономии времени при вводе программы комментарии можно опустить.

Листинг 3.1. Программа `rhodium.c`

```
/* rhodium.c -- стоимость родия, вес которого равен вашему весу */
#include <stdio.h>
int main(void)
{
   float weight;      /* вес пользователя */
   float value;       /* родиевый эквивалент пользователя */
   printf("Хотите узнать свой родиевый эквивалент?\n");
   printf("Давайте подсчитаем.\n");
   printf("Пожалуйста, введите свой вес, выраженный в фунтах: ");
/* получить входные данные от пользователя */
   scanf("%f", &weight);
/* считаем, что цена родия равна $770 за тройскую унцию */
/* 14.5833 коэффициент для перевода веса, выраженного в фунтах, в тройские унции*/
   value = 770.0 * weight * 14.5833;
   printf("Ваш родиевый эквивалент составляет $%.2f.\n", value);
   printf("Вы легко можете стать достойным этого! Если цена родия падает,\n");
   printf("ешьте больше для поддержания своей стоимости.\n");
   return 0;
}
```

Сообщения об ошибках и предупреждения

Если вы введете программу неправильно и, скажем, пропустите точку с запятой, компилятор выдаст сообщение о синтаксической ошибке. Однако даже при правильном вводе программы компилятор может выдать предупреждение, подобное следующему: “Предупреждение — преобразование данных типа 'double' в тип 'float', возможна потеря данных”. Сообщение об ошибке означает, что вы сделали что-то неправильно, и программа дальше компилироваться не будет. *Предупреждение* (warning) означает, что вы ошибку не совершили, но сделали, возможно, не то, что намеревались. Предупреждение не приводит к прекращению компиляции. Данное конкретное предупреждение связано с тем, как в языке C обрабатываются числа, подобные числу 770.0. В данном примере этот вопрос не рассматривается, а смысл этого предупреждения объясняется далее в этой главе.

Когда вы будете вводить эту программу, вам, возможно, потребуется заменить число `770.0` текущей ценой на родий. Однако не нужно делать никаких манипуляций со значением `14.5833`, представляющим собой количество тройских унций в одном фунте. (Именно тройские унции используются в качестве меры веса драгоценных металлов; для измерения веса всех остальных товаров, в том числе и для измерения веса людей, как простолюдинов, так и вельможных персон, применяются фунты.)

Обратите внимание на то, что фраза, приглашающая “ввести” ваш вес, означает напечатать на клавиатуре цифры, составляющие значение вашего веса, и затем нажать клавишу <Enter> или <Return>. (Бесполезно просто напечатать цифры и ждать.) Нажатие клавиши <Enter> информирует компьютер о завершении ввода. Данная программа полагает, что на приглашение указать вес вы введете некоторое число, например, `150`, а не слова, скажем, `очень большой`. Ввод букв вместо цифр служит источником различных проблем, для устранения которых необходимо использовать оператор `if` (см. главу 7), поэтому вводите соответствующее число.

Ниже показан пример выходных данных рассматриваемой программы:

```
Хотите узнать свой родиевый эквивалент?
Давайте подсчитаем.
Пожалуйста, введите свой вес, выраженный в фунтах: 150
Ваш родиевый эквивалент составляет $1684371.12.
Вы легко можете стать достойным этого! Если цена родия падает,
ешьте больше для поддержания своей стоимости.
```

Что нового в этой программе?

В этой программе появилось несколько новых элементов языка C:

- Обратите внимание, что в программе используется объявление переменной нового типа. В предыдущих программах объявлялись только целочисленные переменные (тип `int`), а здесь добавился тип переменной с плавающей запятой (тип `float`), так что теперь вы можете обрабатывать более широкий спектр данных. Переменной типа `float` могут присваиваться вещественные числа.
- В программе демонстрируется новый способ записи констант. Теперь в роли констант выступают вещественные числа.
- Чтобы вывести значение переменной нового типа (то есть число с плавающей запятой), употребляйте в функции `printf()` спецификатор `%f`. Кроме того, в спецификаторе `%f` следует использовать модификатор `.2` для точной настройки внешнего вида выходных данных, который определяет, что данные, выводимые на экран, будут содержать два знака после десятичной точки.
- Для ввода данных в программу с клавиатуры используется функция `scanf()`. Спецификатор `%f` в функции `scanf()` означает, что с клавиатуры считывается число с плавающей запятой, а `&weight` — что вводимое число присваивается переменной с именем `weight`. Функция `scanf()` использует амперсанд (`&`), чтобы указать, где она может найти переменную `weight`. В следующей главе это рассматривается более подробно, а пока просто поверьте на слово, что он здесь необходим.
- Возможно, самая главная особенность этой программы заключается в том, что она является интерактивной. Компьютер просит вас ввести конкретную информацию, а затем использует введенное вами число. Работать с интерактивными программами намного интереснее. Но гораздо важнее то, что интерактивный подход делает программу более гибкой. Например, приведенная выше демонстрационная программа может быть использована для пересчета любого разумного веса, а не только конкретного веса, равного 150 фунтам. Эту программу не приходится переписывать каждый раз, когда она потребуется новому пользователю. Интерактивность обеспечивают функции `scanf()` и `printf()`. Функция `scanf()` считывает данные с клавиатуры и делает их доступными для программы, а функция `printf()` принимает данные от программы и выводит их на экран. Вместе обе эти две функции позволяют вам установить двусторонний обмен данными с компьютером (рис. 3.1), что делает работу с компьютером гораздо более интересной.

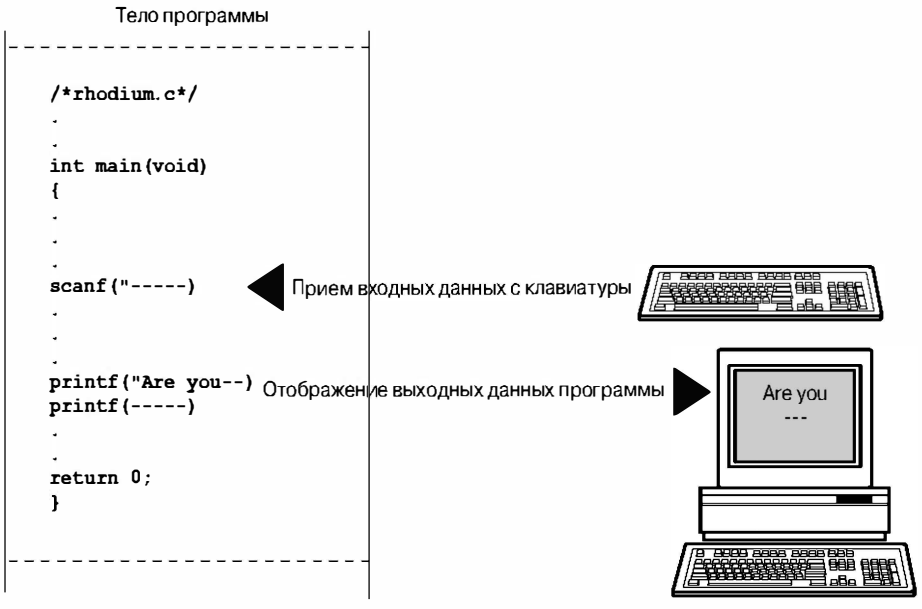


Рис. 3.1. *Функции* `scanf()` *и* `printf()` *в работе*

Из приведенного выше списка новых свойств в настоящей главе рассматриваются два элемента: переменные и константы различных типов. Остальные три свойства исследуются в главе 4; тем не менее, в данной главе мы продолжим в ограниченных масштабах использовать функции `scanf()` и `printf()`.

Переменные и константы

Компьютер под управлением программы может делать многое. Он может складывать числа, сортировать имена, воспроизводить аудио- и видеоклипы, вычислять орбиты комет, составлять списки адресатов почтовых отправлений, набирать телефонные номера, рисовать картинки, делать логические выводы и многое другое из того, что придет вам в голову. Для решения этих задач программа должна работать с *данными*: числа и символы являются той информацией, которую вы используете. Значения некоторых данных устанавливаются до начала выполнения программы и сохраняются неизменными в течение всего времени работы программы. Такие данные называются *константами*. Значения других данных могут изменяться (в частности, путем присваивания новых значений) в ходе выполнения программы. Такие данные называются *переменными*. В приведенной выше учебной программе `weight` представляет собой переменную, а `14.5833` — константу. А к какому виду данных следует отнести число `770.0`, к переменным или к константам? И в самом деле, в обычной жизни цена на родий не является постоянной величиной, но в данной программе она рассматривается как константа. Различие между переменной и константой состоит в том, что переменной можно присваивать значение или изменять его во время выполнения программы, а с константой так поступать нельзя.

Ключевые слова, обозначающие типы

Помимо различий между переменными и константами, существуют также различия между данными разных *типов*. Одни данные являются числами. Другие данные представляют собой буквы или, в общем случае, символы. Компьютер должен каким-то способом различать их и использовать соответствующими способами. Язык С различает несколько основных *типов данных*. Если данные представляют собой константы, то компилятор может, как правило, определять их тип по тому, как они выглядят: `42` есть целое число, а `42.100` — это число с плавающей запятой. Однако тип переменной должен быть указан в операторе объявления. Чуть позже мы более подробно остановимся на том, как объявлять переменные. Но сначала рассмотрим фундаментальные типы данных, существующие в языке С. В стандарте K&R C существовало семь ключевых слов, определяющих типы данных. В стандарте C90 к этому списку добавилось еще два ключевых слова. В стандарте C99 этот список пополнился еще тремя словами (табл. 3.1).

Таблица 3.1. Ключевые слова языка С

Ключевые слова исходного стандарта K&R	*Ключевые слова, добавленные стандартом C90*	*Ключевые слова, добавленные стандартом C99*
`int`	`signed`	`_Bool`
`long`	`void`	`_Complex`
`short`		`_Imaginary`
`unsigned`		
`char`		
`float`		
`double`		

Ключевое слово `int` обозначает основной класс целых чисел, используемых в языке С. Три последующих слова (`long`, `short` и `unsigned`), а также добавленное стандартом ANSI C ключевое слово `signed` представляют собой разновидности этого основного типа данных. Далее, ключевое слово `char` обозначает символьные данные, к которым относятся буквы алфавита и другие символы, такие как `#`, `$`, `%` и `*`. Тип данных `char` можно также использовать для представления небольших целых чисел. Типы данных `float`, `double` и `float double` служат для представления вещественных чисел, которые в языке С называются числами с плавающей запятой (из-за формы их представления). Тип данных `_Bool` используется для логических значений (`true` и `false`), типы данных `_Complex` и `_Imaginary` представляют, соответственно, комплексные и мнимые числа.

Все типы данных, обозначаемые этими ключевыми словами, можно разделить на два семейства в зависимости от того, как они хранятся в памяти компьютера: *целочисленные* типы данных и типы данных *с плавающей запятой*.

Биты, байты и слова

Термины *бит* (bit), *байт* (byte) и *слово* (word) могут использоваться для описания элементов компьютерных данных или элементов компьютерной памяти. Основное внимание мы уделим второму виду описания.

Минимальный элемент памяти называется *битом*. Он может хранить одно из двух значений: `0` или `1`. (Говорят также, что бит "сброшен" или "установлен".) Конечно, в одном бите много информации не запомнишь, но в компьютере имеется огромное число битов. Бит представляет собой базовый строительный блок, из которых построена память компьютера.

Байт — это наиболее часто используемый элемент памяти компьютера. Почти во всех компьютерах байт образован из восьми битов, и это является стандартным определением байта, по крайней мере, когда речь идет об измерении объема памяти. (Однако, как будет показано в разделе "Использование символов: тип `char`" далее в этой главе, в языке C применяется другое определение.) Поскольку бит может принимать значение `0` или `1`, байт обеспечивает 256 (то есть 2^8) возможных комбинаций нулей и единиц. Эти комбинации могут использоваться, например, для представления целых чисел от 0 до 255 или для представления набора символов. Такое представление может быть реализовано путем перевода числа в двоичную систему счисления, в котором для представления чисел используются только две цифры: 0 и 1 (счастливое совпадение). (Двоичный код подробно рассматривается в главе 15, но вы можете в данной главе ознакомиться с вводным материалом по этим вопросам уже сейчас.)

Слово представляет собой естественный элемент памяти для компьютеров конкретного типа. В 8-разрядных микрокомпьютерах, таких как первые компьютеры Apple, слово состояло из 8 битов. Ранние модели компьютеров, совместимые с компьютерами IBM, в которых использовался микропроцессор 80286, были 16-разрядными. Это значит, что размер слова этих машин был равен 16 битам. Такие машины, как персональные компьютеры с микропроцессорами типа Pentium и Power Macintosh, работают с 32-битовыми словами. Более мощные компьютеры могут иметь слова длиной 64 и более битов.

Целочисленные данные и данные с плавающей запятой

Вам непонятно, что такое целочисленные данные или данные с плавающей запятой? Если эти термины вам незнакомы, не огорчайтесь. Мы сейчас в общих чертах объясним, что они означают. Если вы незнакомы с битами, байтами и словами, прежде всего, внимательно изучите предыдущее примечание. Нужно ли вам знать о них все до последней детали? Вовсе нет. Вам ведь не обязательно изучать принцип работы двигателя внутреннего сгорания для того, чтобы водить автомобиль; но совсем неплохо знать немного о том, что происходит внутри компьютера или двигателя — такие знания иногда оказываются очень полезными.

Для человека различие между целыми числами и числами с плавающей запятой проявляется в способе их написания. Для компьютера это различие проявляется в способе их хранения в памяти. Рассмотрим по очереди оба эти класса данных.

Целые числа

Целое число — это число без дробной части. В языке C целое число никогда не записывается с десятичной точкой. Целыми числами являются, например, `2`, `-23` и `2456`. Такие числа, как `3.14`, `0.22` и `2.000`, не принадлежат к классу целых чисел. Целые числа хранятся в двоичной форме. Целое число 7, например, записывается в двоичной системе как `111`. Поэтому, чтобы запомнить это число в байте памяти, установите первые 5 разрядов в `0`, а последние три разряда — в `1` (рис. 3.2).

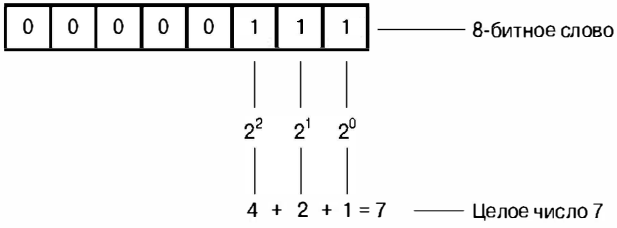


Рис. 3.2. *Хранение в памяти целого числа 7 в двоичном коде*

Числа с плавающей запятой

Число с *плавающей запятой* более или менее соответствует тому, что математики называют *вещественным числом*. К вещественным числам относятся числа, находящиеся между целыми числами. Примерами чисел с плавающей запятой могут служить: `2.75`, `3.16E7`, `7.00` и `2e-8`. Обратите внимание, что добавление десятичной точки превращает целое число в число с плавающей запятой. Таким образом, число `7` принадлежит к типу целых чисел, а в записи `7.00` оно переходит в класс чисел плавающей запятой. Существует несколько форм записи чисел с плавающей запятой. Более подробно экспоненциальную форму записи чисел мы обсудим позже. Вообще говоря, запись `3.16E7` означает, что число `3.16` необходимо умножить на 10^7, то есть на число, состоящее из единицы с семью последующими нулями. Число 7 называется *порядком* числа 10.

Ключевым моментом здесь является то, что схема, используемая для хранения числа с плавающей запятой, отличается от той, которая применяется для хранения целого числа. Число с плавающей запятой разделяется на дробную часть и порядок, хранящиеся в памяти отдельно. Поэтому число 7.00 будет храниться в памяти не в том виде, в каком хранится целое число 7, хотя оба они имеют одно и то же значение. Десятичным аналогом этого способа хранения будет запись числа 7.00 в виде 0.7E1. Здесь 0.7 — это дробная часть числа, а 1 — порядок. На рис. 3.3 представлен еще один пример хранения в памяти числа с плавающей запятой. Разумеется, для хранения в памяти компьютера используются двоичные числа и степени числа 2, а не степени числа 10. Дополнительный материал по этой теме вы найдете в главе 15.

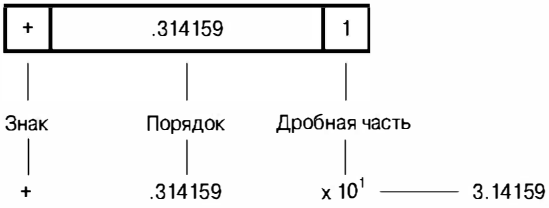


Рис. 3.3. *Хранение числа π в формате числа с плавающей запятой (десятичная версия)*

А теперь обратим внимание на различия, имеющие практическое значение:

- Целое число не имеет дробной части; число с плавающей запятой может иметь дробную часть.
- Диапазон допустимых чисел с плавающей запятой гораздо шире диапазона допустимых целых чисел. См. табл. 3.2 в конце данной главы.

- При выполнении некоторых арифметических операций с плавающей запятой, например, операции вычитания одного большого числа из другого, возможна существенная потеря точности.
- Поскольку в любом диапазоне чисел имеется бесконечное количество вещественных чисел, например, в диапазоне между 1.0 и 2.0, используемые в компьютере числа с плавающей запятой не могут представлять все числа этого диапазона. Числа с плавающей запятой часто являются приближениями истинных значений. Например, число 7.0 может быть записано как значение с плавающей запятой 6.99999 (более подробно вопрос о точности будет рассмотрен ниже).
- Обычно операции над числами плавающей запятой выполняются медленнее, чем операции над целыми числами. Однако доступны микропроцессоры, разработанные специально для выполнения операций над числами с плавающей запятой, и по быстродействию они сравнимы с операциями над целыми числами.

Базовые типы данных языка C

Рассмотрим теперь характерные особенности основных типов данных, используемых в языке C. Мы покажем, как объявлять переменные и записывать константы всех основных типов, а также каких случаях и для каких целей применять данные того или иного типа. Некоторые компиляторы языка C распознают не все возможные типы данных, поэтому выясните в соответствующей документации, какие типы данных поддерживает ваш компилятор.

Тип данных `int`

Язык C предлагает множество целочисленных типов данных, и вы, скорее всего, хотите узнать, почему одного типа оказывается не достаточно. Дело в том, что язык C дает программисту возможность выбора соответствующего типа данных для использования в каждом конкретном случае. В частности, предлагаемые языком C целые типы отличаются друг от друга диапазонами представляемых значений, а также возможностью представления отрицательных чисел. Базовым типом целочисленных данных является `int`, но если вам потребуются другие целочисленные типы, отвечающие требованиям конкретной задачи или компьютера, вы можете выбрать их.

Число типа `int` — это целое число со знаком. Это значит, что число должно быть целым, а также что оно может быть положительным, отрицательным или нулем. Диапазон возможных значений зависит от компьютерной системы. Обычно для хранения данных типа `int` используется одно машинное слово. Поэтому в компьютерах, совместимых со старыми моделями IBM PC с 16-битовыми словами, для хранения данных типа `int` выделяется 16 битов. В этих условиях значения целых чисел находятся в диапазоне от `-32768` до `+32767`. Современные персональные компьютеры оперируют 32-разрядными целыми числами и данные типа `int` соответствуют этому размеру. Примеры можно найти в табл. 3.3 далее в этой главе.

В настоящее время производство персональных компьютеров сориентировалось на выпуск 64-разрядных процессоров, которые могут свободно манипулировать еще большими целыми числами. Стандарт ANSI C требует, чтобы минимальным диапазоном возможных значений данных типа `int` являлся диапазон от `-32767` до `+32767`.

Обычно для представления знака целого числа в вычислительной системе отводится один разряд. Наиболее распространенные способы представления в памяти целых чисел со знаком рассматриваются в главе 15.

Объявление переменных типа `int`

Как было показано в главе 2, для объявления целочисленных переменных служит ключевое слово `int`. Сначала идет ключевое слово `int`, затем выбранное вами имя переменной, после которого ставится точка с запятой. Несколько переменных можно объявлять либо по отдельности, либо в одном операторе, в этом случае после ключевого слова `int` помещается список имен, причем имена отделяются друг от друга запятыми. Ниже показаны примеры допустимых объявлений переменных:

```
int erns;
int hogs, cows, goats;
```

Можно объявить каждую переменную отдельно или же, наоборот, все четыре переменные объявить в одном операторе — это дело вкуса. Результат будет тот же: выделенные области памяти с соответствующими именами для четырех переменных типа `int`.

Эти объявления создают переменные, но не присваивают им никаких значений. Как переменные получают значения? Вы уже видели два способа, с помощью которых переменные могут получать значения в программе. Во-первых, это оператор присваивания:

```
cows = 112;
```

Во-вторых, переменная может получить значение от функции, например, от `scanf()`. Теперь рассмотрим третий способ.

Инициализация переменных

Инициализировать переменную — это значит присвоить ей *начальное* значение. В языке С это можно сделать в операторе объявления. Просто после имени переменной поставьте знак операции присваивания (=) и значение, которое необходимо присвоить переменной. Вот несколько примеров:

```
int hogs = 21;
int cows = 32, goats = 14;
int dogs, cats = 94;      /* правильная, но плохая форма */
```

В последней строке инициализируется только переменная `cats`. При беглом чтении может показаться, что переменная `dogs` также инициализируется значением `94`, поэтому лучше избегать указания инициализированных и неинициализированных переменных в одном операторе объявления.

Короче говоря, эти объявления выделяют для переменных области памяти, именуют их и присваивают им начальные значения (рис. 3.4).

Ввод с клавиатуры констант типа `int`

Различные целые числа (21, 32, 14 и 94) в последнем примере представляют собой *целочисленные константы*. Когда вы вводите число без десятичной точки и без порядка, компилятор С воспринимает его как целое. Поэтому числа 22 и –44 являются целочисленными константами, а числа 22.0 и 2.2E1 — нет.

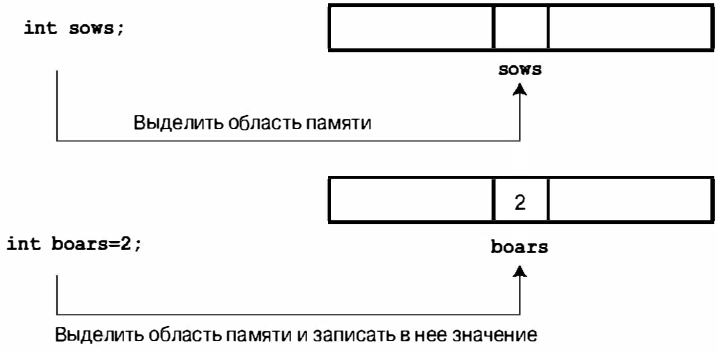


Рис. 3.4. *Определение и инициализация переменной*

Большинство целочисленных констант рассматриваются языком C как данные типа `int`. Очень большие целые числа могут трактоваться иначе (см. раздел "Константы типа `long` и `long long`", в котором рассматриваются данные типа `long int`).

Вывод на печать значений типа `int`

Для вывода на печать данных типа `int` вы можете воспользоваться функцией `printf()`. Как уже говорилось в главе 2, символы `%d` служат для указания в строке места, где будет печататься целое число. Символы `%d` называются *спецификатором формата*, поскольку они определяют формат, используемый функцией `printf()` для отображения конкретного значения. Каждому спецификатору `%d` в строке формата должно быть поставлено в соответствие значение типа `int` из списка выводимых на печать элементов. Это может быть значение переменной типа `int`, константа типа `int` или любое другое выражения типа `int`. Программист должен следить за тем, чтобы количество спецификаторов формата равнялось числу значений типа `int`, так как компилятор не обнаруживает подобного рода ошибок.

В листинге 3.2 представлена простая программа, которая сначала выполняет инициализацию переменной, а затем выводит на печать значение этой переменной, значение константы и простого выражения. Программа также демонстрирует, к каким последствиям может привести невнимательность в этих вопросах.

Листинг 3.2. Программа `print1.c`

```
/*  print1.c  -- демонстрирует некоторые свойства функции printf() */
#include <stdio.h>
int main (void)
{
   int ten =  10;
   int two =  2;

   printf("Выполняется правильно: ") ;
   printf("%d  минус %d равно %d\n" ten, 2, ten - two);
   printf("Выполняется неправильно: ") ;
   printf("%d минус %d is %d\n", ten );       // пропущены 2 аргумента

   return 0;
}
```

После компиляции и выполнения программа выведет на экран следующую информацию:

```
Выполняется правильно: 10 минус 2 равно 8
Выполняется неправильно: 10 минус 10 равно 2
```

В первой строке выходных данных первый спецификатор `%d` представляет переменную `ten` типа `int`, второй — константу `2` типа `int` и третий — значение выражения `ten - two`, также типа `int`. Во второй строке первому спецификатору `%d` также соответствует значение переменной `ten`, зато для двух последующих спецификаторов `%d` соответствующих значений нет, и программа использует случайные значения, находящиеся в близлежащей памяти! (На своем компьютере вы можете получить совсем не те числа, которые использует рассматриваемая программа. Однако не только содержимое памяти может быть ошибочным, дело еще в том, что различные компиляторы по-разному манипулируют ячейками памяти.)

Вас может огорчить тот факт, что компилятор не способен обнаружить столь очевидную ошибку. Причина заключается в функции `printf()`. Большинство функций принимает четко определенное количество аргументов, и компилятор может проверить, правильное ли число аргументов вы указали. В то же время функция `printf()` может принимать один, два, три и большее число аргументов, и это обстоятельство не позволяет компилятору использовать обычные методы обнаружения такого рода ошибок. Необходимо тщательно проверять, равно ли количество спецификаторов формата в функции `printf()` количеству значений, которые подлежат отображению на экране.

Восьмеричные и шестнадцатеричные числа

Обычно в языке С предполагается, что целочисленная константа представляет собой десятичное число (основанием системы счисления является 10). Однако в программировании широко используются и восьмеричные (основание — число 8), и шестнадцатеричные (основание — число 16) числа. Поскольку 8 и 16 — это степени числа 2, а не 10, восьмеричная и шестнадцатеричная системы счисления более удобны для представления чисел, используемых в компьютерах. Например, число 65536, которое часто всплывает в 16-разрядных компьютерах, в шестнадцатеричной системе выглядит как 10000. Наряду с этим, каждая цифра шестнадцатеричного числа соответствует в точности 4 разрядам. Например, шестнадцатеричная цифра 3 — это 0011, а шестнадцатеричная цифра 5 — это 0101.

Таким образом, шестнадцатеричному значению 35 соответствует битовая комбинация 0011 0101, а шестнадцатеричному значению 53 — битовая комбинация 0101 0011. Это соответствие позволяет облегчить переход от шестнадцатеричного представления числа к двоичному представлению (основание — число 2) и обратно. Но каким образом компьютер может определить, является ли число 10000 десятичным, шестнадцатеричным или восьмеричным? На используемую систему счисления в языке С указывают специальные префиксы. Префикс `0x` или `0X` означает, что вы определяете шестнадцатеричное число, поэтому 16 в шестнадцатеричной системе записывается как `0x10` или `0X10`. Аналогично, префикс `0` означает, что задается восьмеричное число. Например, в языке С десятичное число 16 записывается в восьмеричной системе как `020`. Более полно эти альтернативные системы счисления (восьмеричная и шестнадцатеричная) обсуждаются в главе 15.

Вы должны понимать, что различные системы счисления применяются для удобства программистов. Они не влияют на то, как числа хранятся в памяти компьютера. Иначе говоря, вы можете написать 16, 020 или 0x10, тем не менее, это число в любом случае будет храниться в памяти в одном и том же виде — в двоичном коде, который используется внутри компьютера.

Отображение восьмеричных и шестнадцатеричных чисел

Язык C предоставляет вам возможность не только записывать число в любой из описанных выше трех систем счисления, но и отображать его в любой из этих трех систем. Чтобы отобразить на экране целое число в восьмеричном, а не десятичном виде, вместо спецификатора %d необходимо применять спецификатор %o. Для отображения целого числа в шестнадцатеричном виде служит спецификатор %x. Если вы хотите вывести на экран префикс языка C, воспользуйтесь спецификаторами %#o, %#x и %#X, что позволит вывести, соответственно, префиксы 0, 0x и 0X. В листинге 3.3 показан небольшой пример. (Напомним, что вам, возможно, придется вставить в программу оператор вызова функции getchar(); при работе в некоторых интегрированных средах разработки он предотвращает немедленное закрытие активного окна программы.)

Листинг 3.3. Программа bases.c

```
/* bases.c — распечатывает число 100 в десятичной, восьмеричной
   и шестнадцатеричной системах счисления */
#include <stdio.h>
int main (void)
{
   int x = 100;
   printf("десятичное = %d; восьмеричное = %o; шестнадцатеричное = %x\n",
          x, x, x);
   printf("десятичное = %d; восьмеричное = %#o; шестнадцатеричное = %#x\n",
          x, x, x);
   return 0;
}
```

В результате компиляции и выполнение этой программы были получены следующие выходные данные:

```
десятичное = 100; восьмеричное = 144; шестнадцатеричное = 64
десятичное = 100; восьмеричное = 0144; шестнадцатеричное = 0x64
```

Одно и то же значение отображается в трех различных системах счисления. Все преобразования выполняет функция printf(). Обратите внимание, что выводимые на экран данные отображаются без префиксов 0 или 0x, если вы не включили знак # как часть спецификатора.

Другие целочисленные типы

Если вы просто изучаете язык C, то при работе с целыми числами в большинстве случаев вам, скорее всего, вполне достаточно будет типа int. Однако для полноты картины мы рассмотрим сейчас и другие формы целых чисел. При желании вы можете пропустить этот раздел и сразу перейти к изучению данных типа char в разделе “Использование символов: тип char”, возвращаясь к данному разделу по мере необходимости.

В языке С используются три ключевых слова-прилагательных, обозначающих модификации основного типа целочисленных данных (то есть типа int): short, long и unsigned.

- Данные типа short int (или, короче, типа short) могут занимать меньший объем памяти, чем данные типа int, и таким образом экономить память в случае, когда используются только небольшие числа. Данные типа short, равно как и данные типа int, являются данными со знаком.
- Данные типа long int (или long) могут занимать больший объем памяти, чем данные типа int, и тем самым обеспечивать вам возможность представлять большие целочисленные значения. Данные типа long, так же как и данные типа int, являются данными со знаком.
- Данные типа long long int, или long long (оба типа введены стандартом С99), могут использовать больше памяти, чем тип long, тем самым обеспечивая возможность представлять еще большие целочисленные значения. Как и тип int, тип long long является типом со знаком.
- Данные типа unsigned int (или просто unsigned), используется только для представления неотрицательных чисел. Этот тип смещает диапазон представляемых чисел в сторону положительных чисел. Например, 16-разрядный тип unsigned int обеспечивает диапазон представляемых значений от 0 до 65535 вместо диапазона от -32768 до 32767. Разряд, который использовался для представления знака, теперь становится еще одной двоичной цифрой, благодаря чему расширяется диапазон представляемых неотрицательных чисел.
- Данные типа unsigned long int, или unsigned long, а также unsigned short int, или unsigned short, признаются стандартом С90 как допустимые. К этому списку стандарт С99 добавляет тип unsigned long long int, или unsigned long long.
- Ключевое слово signed может использоваться с любыми типами со знаком с тем, чтобы ваши намерения стали прозрачными. Например, short, short int, signed short и signed short int представляют собой имена одного и того же типа данных.

Объявление целочисленных данных различных типов

Другие типы целых чисел объявляются так же, как и тип int. Приведенный ниже список содержит несколько примеров. Не все ранние компиляторы языка С распознают последние три типа; завершающий пример допустим только в рамках стандарта С99.

```
long int estine;
long johns;
short int erns;
short ribs;
unsigned int s_count;
unsigned players;
unsigned long headcount;
unsigned short yesvotes;
long long ago;
```

Зачем нужно так много целочисленных типов?

Почему мы говорим, что типа long и short "могут" использовать больше или меньше памяти, чем тип int? Потому что язык С гарантирует только то, что тип short не длиннее типа int и что тип long не короче, чем тип int. Идея заключается в том, чтобы подобрать тип данных для конкретной машины. На персональном компьютере IBM PC, работающем под управлением операционной системы Windows 3.1, например, как тип int, так и тип short являются 16-разрядными типами данных, в то время как long представляет собой 32-разрядный тип данных. С другой стороны, на машине с Windows XP или на Macintosh PowerPC тип short содержит 16 разрядов, а оба типа int и long являются 32-разрядными. Естественный размер слова, которыми оперируют микропроцессоры Pentium, PowerPC G3 или G4 составляет 32 разряда. Поскольку такое слово обеспечивает представление чисел, превышающих два миллиарда (как показано в табл. 3.3), разработчики языка С для различных комбинаций этих процессоров и операционных систем не видели необходимости в чем-то более крупном, в связи с чем тип long оказался таким же, как и int. Для многих приложений такие большие целые числа вовсе не нужны, поэтому был предложен тип short, обеспечивающий экономию памяти. С другой стороны, первые модели персонального компьютера IBM PC поддерживали только 16-битовое слово, и поэтому возникла необходимость в большем типе целочисленных данных, в частности, в типе long.

В настоящее время, когда все более широкое распространение получают 64-разрядные процессоры, такие как IBM Itanium, AMD Opteron и PowerPC G5, ощущается необходимость в 64-разрядных целых числах, и именно это обстоятельство стало причиной для использования типа long long.

Сегодня тип long long предназначается для представления 64-разрядных чисел, тип long — для 32-разрядных чисел, тип short — для 16-разрядных чисел, а тип int — для представления как 16-разрядных, так и 32-разрядных чисел, в зависимости от естественного размера слова машины. В принципе, однако, указанные выше четыре типа данных могут представлять четыре различных размера.

Стандарт языка С устанавливает ориентиры, определяющие минимально допустимый размер каждого базового типа данных. Минимальный диапазон возможных значений как типа short, так и типа int, находится в пределах от –32 767 до 32 767, что соответствует 16-разрядному слову, а минимальный диапазон типа long находится в пределах от –2 147 483 647 до 2 147 483 647, что соответствует 32-разрядному слову. Что касается типов unsigned short и unsigned int, то минимальный диапазон охватывает числа от 0 до 65 535, а для типа unsigned long минимальный диапазон находится в пределах от 0 до 4 294 967 295. Тип long long предназначен для поддержки 64-разрядных объектов данных. Минимальный диапазон чисел представляемых этим типом данных достаточно внушителен и простирается от –9 223 372 036 854 775 807 до 9 223 372 036 854 775 807, а минимальный диапазон типа unsigned long long охватывает числа от 0 до 18 446 744 073 709 551 615.

В каких случаях вы используете конкретные варианты типа int? Сначала рассмотрим типы без знака. Прежде всего, они используются для подсчета, поскольку в этих случаях нет необходимости в отрицательных числах, беззнаковые типы могут принимать большие положительные значения, чем данные со знаком.

Данные типа long применяйте в тех случаях, когда тип long предоставляет вам возможность работы с нужными числами, а тип int не способен этого сделать.

В то же время, в тех системах, в которых тип long больше, чем тип int, использование типа long может замедлить вычисления, в силу этого обстоятельства не следует употреблять тип long, если в этом нет необходимости. Еще один момент: если вы пишете программу для машины, у которой типы int и long одного и того же размера, а вам нужно работать с 32-разрядными целыми числами, вам следует выбирать тип long вместо int, чтобы ваша программа работала корректно, будучи перенесенной на 16-разрядную машину.

Аналогично, применяйте тип long long, если вам приходится иметь дело с 64-разрядными целочисленными значениями. В некоторых компьютерах уже используются 64-разрядные процессоры, а 64-разрядная обработка в серверах, в рабочих станциях и даже в настольных компьютерах вскоре может стать обычной практикой.

Пользуйтесь типом short в целях экономии памяти, например, в тех случаях, когда вам нужна 16-разрядная величина в системе, в которой типом int является 32-разрядный тип данных. Обычно задача экономии памяти возникает только в тех случаях, когда ваша программа использует массивы целых чисел, которые можно рассматривать как крупные по сравнению с доступной памятью системы. Еще одна причина в использовании типа short состоит в том, что по своим размерам он соответствует размерам регистров, используемых конкретными компонентами системы.

Целочисленное переполнение

Что произойдет, если целое число окажется больше, чем допускает выбранный для этого числа тип? Давайте присвоим целочисленной переменной максимально возможное целое значение, прибавим к нему какое-либо целое число и посмотрим, чем это закончится. Выполним эту операцию как над целыми со знаком, так и с целыми без знака. (В вызове функции printf() для отображения значений типа unsigned int применяется спецификатор %u.)

```
/* toobig.c - превышение максимально возможного значения int
   в вашей системе */
#include <stdio.h>
int main(void)
{
   int i = 2147483647;
   unsigned int j = 4294967295;
   printf("%d %d %d\n", i, i+1, i+2);
   printf("%u %u %u\n", j, j+1, j+2);
   return 0;
}
```

В нашей системе был получен следующий результат:

```
2147483647 -2147483648 -2147483647
4294967295 0 1
```

Целое число без знака j выступает в роли счетчика расстояния, пройденного автомобилем. Когда этот счетчик достигает максимального значения, он сбрасывает это значение и начинает новый отсчет с нуля. Целое число i ведет себя аналогично. Основное различие заключается в том, что переменная j типа unsigned int, подобно счетчику расстояния, начинается с 0, в то время как переменная i типа int — с -2 147 483 648. Обратите внимание, что вы ничего не знали о том, что значение i превзошло максимально возможное значение (произошло переполнение).

Чтобы составить соответствующие таблицы, вам придется предусмотреть в этой программе специальный код.

Описанное здесь поведение программы определяется правилами языка C, регламентирующими работу с типами без знака. Стандарт не указывает, как должны вести себя типы со знаком. Описанное в рамках рассмотренного примера поведение является обычным, однако вам, возможно, еще придется столкнуться с поведением другого рода.

Константы типа long и long long

Обычно, когда вы используете в коде программы число, подобное, например, 2345, оно обычно сохраняется в памяти как относящееся к типу `int`. Что произойдет, если вы используете, например, число 1 000 000, в своей системе, в которой тип `int` не может запомнить столь большое число? В этом случае компилятор рассматривает его как число типа `long int`, предполагая, что этого типа будет достаточно. Если число превосходит максимально возможное значение типа `long`, C рассматривает его как значение типа `unsigned long`. Если и этого оказывается не достаточно, C интерпретирует его как значение типа `long long` или `unsigned long long`, если, разумеется, упомянутые типы доступны.

Восьмеричные и шестнадцатеричные константы интерпретируются как значения типа `int`, пока это значение слишком велико. Затем компилятор примеривает к этому числу тип `unsigned int`. Если и это не помогает, компилятор последовательно пробует применить типы `long`, `unsigned long`, `long long` и `unsigned long long`.

Иногда вы хотите, чтобы компилятор сохранил небольшое число как целое значение типа `long`. Подобная необходимость может возникнуть, например, во время программирования на IBM PC, когда явно используются адреса памяти. Наряду с этим, некоторые стандартные функции языка C требуют значений типа `long`. Чтобы небольшая константа интерпретировалась как значение типа `long`, вы можете сопроводить ее буквой `l` (буква "L" в нижнем регистре) или `L` в качестве суффикса. Вторая форма предпочтительнее, поскольку она не выглядит похоже на цифру 1. В силу этого обстоятельства система с 16-разрядным типом `int` и 32-разрядным типом `long` интерпретирует целое число `7` как 16-разрядное, а целое число `7L` — как 32-разрядное. Суффиксы `l` и `L` также могут использоваться в восьмеричных и шестнадцатеричных числах, например, `020L` и `0x10L`. Аналогично, в системах, поддерживающих тип `long long`, вы можете воспользоваться суффиксами `ll` и `LL`, дабы указать значение типа `long long`, например, `3LL`. Добавьте `u` или `U` к суффиксу в случае использования типа `unsigned long long`, как случае `5ull` или `10LLU` или `6LLU` или `9Ull`.

Печать типов short, long, long long и unsigned

Для вывода чисел типа `unsigned int` на печать применяется спецификатор `%u`. Чтобы вывести значение типа `long`, используется спецификатор формата `%ld`. Если типы `int` и `long` имеют в вашей системе один и тот же размер, вполне достаточно одного спецификатора `%d`, однако ваша программа не будет работать должным образом, если ее перенести в систему, у которой два эти типы имеют разные размеры, поэтому для типа `long` следует пользоваться спецификатором `%ld`. Вы можете также указывать префикс `l` вместе с префиксами `x` и `o`. Благодаря этому вы можете примерить спецификатор `%lx` для печати целого числа типа `long` в шестнадцатеричном формате и спецификатор `%lo` — для печати в восьмеричном формате.

Обратите внимание, что если язык С позволяет использовать как буквы верхнего, так и буквы нижнего регистров в качестве суффиксов констант, то в спецификаторах формата допускаются только буквы нижнего регистра.

В языке С доступны дополнительные форматы функции `printf()`. Во-первых, вы можете использовать префикс `h` для значений типа `short`. По этой причине спецификатор `%hd` отображает целое число типа `short` в десятичной форме, а спецификатор `%ho` отображает это же число в восьмеричной форме. Префиксы `h` и `l` могут использоваться с префиксом `u` для типов без знака. Например, вы могли бы воспользоваться обозначением `%lu` для печати значений типов `unsigned long`. В листинге 3.4 представлен соответствующий пример. Системы, поддерживающие типы `long long`, используют спецификаторы `%lld` и `%llu` для версий со знаком и версий без знака. В главе 4 изучение спецификаторов формата будет продолжено.

Листинг 3.4. Программа `print2.c`

```
/* print2.c - дальнейшее изучение свойств функции printf() */
#include <stdio.h>
int main(void)
{
   unsigned int un = 3000000000;    /* Система с 32-разрядным типом int */
   short end = 200;                 /* и 16-разрядным типом short */
   long big = 65537;
   long long verybig = 12345678908642;

   printf("un = %u, но не %d\n", un, un);
   printf("end = %hd и %d\n", end, end);
   printf("big = %ld, но не %hd\n", big, big);
   printf("verybig = %lld, но не %ld\n", verybig, verybig);

   return 0;
}
```

Вот как выглядят выходные данные на одной из систем:

```
un = 3000000000, но не -1294967296
end = 200 и 200
big = 65537, но не 1
verybig = 12345678908642, но не 1942899938
```

Этот пример показывает, что использование неправильной спецификации может привести к непредсказуемым результатам. Прежде всего, обратите внимание на то, что использование спецификатора `%d` для переменной без знака `un` вызывает появление отрицательного числа!

Это объясняется тем, что значение 3 000 000 000 без знака и значение –129 496 296 со знаком имеют одно и то же внутреннее представление в памяти нашей системы. (В главе 15 это свойство рассматривается подробно.) Таким образом, если вы сообщите функции `printf()` о том, что значение является числом без знака, она напечатает одно значение, а если вы сообщите ей, что значение представляет собой число со знаком — то другое значение. Подобное поведение имеет место в тех случаях, когда величина со знаком превосходит максимально допустимое значение.

Небольшие положительные значения, например, 96, сохраняется и отображается так же, как и типы со знаком и без знака.

Далее, обратите внимание на то обстоятельство, что переменная end типа short отображается одинаково в тех случаях, когда вы указываете функции printf(), что переменная end имеет тип short (спецификатор %hd) и когда задаете тип int (спецификатор %d). Это объясняется тем, что во время передачи аргумента функции C автоматически расширяет значение типа short до значения типа int. При этом у вас могут возникнуть два вопроса: почему это преобразование имеет место, и для чего используется модификатор h? Ответ на первый вопрос выглядит следующим образом: тип int выбирался с целью обеспечения наиболее эффективной работы компьютера. Следовательно, на компьютере, для которого типы short и int имеют разные размеры, передача значения как типа int может выполняться быстрее. Ответ на второй вопрос звучит так: вы можете использовать модификатор h, чтобы показать, какой вид примет целое значение, будучи усеченным до типа short. Третья строка выходных данных может служить иллюстрацией этого момента.

Число 65537, записанное в двоичном формате как 32-разрядное число, имеет такой вид: 00000000000000010000000000000001. Применив спецификатор %hd, мы заставляем функцию printf() просматривать только 16 последних разрядов числа, поэтому данное число она отображает как 1. Аналогично, завершающая строка выходных данных показывает полное значение величины verybig, после чего это значение сохраняется в последних 32 разрядах, на что указывает спецификатор %ld.

Как упоминалось ранее, программист отвечает за то, чтобы количество спецификаторов соответствовало количеству отображаемых значений. Из сказанного выше следует, что программист несет ответственность также и за правильный выбор спецификаторов, соответствующих типу отображаемых значений.

Соответствие типам спецификаторов `printf()`

Не забывайте убедиться в том, что на каждую отображаемую величину в вашем операторе printf() имеется один спецификатор формата. Проверьте также, чтобы тип каждого спецификатора формата соответствовал типу отображаемого значения.

Использование символов: тип `char`

Тип данных char применяется для хранения символов, таких как буквы и знаки препинания, однако в техническом аспекте он является также целочисленным. Почему? Да потому, что тип char фактически хранит целые числа, а не символы. При работе с символами компьютер использует числовые коды, то есть определенные целые числа представляют определенные символы. В США наиболее часто используемым кодом является код ASCII (American Standard Code for Information Interchange — американский стандартный код обмена информацией). Именно этот код принят в данной книге. В нем, например, целое значение 65 представляет букву *A* верхнего регистра. Таким образом, чтобы сохранить букву *A*, вы фактически должны записать целое число 65. (Многие мэйнфреймы IBM используют другой код, получивший название EBCDIC (Extended Binary Coded Decimal Interchange Code — расширенный двоично-десятичный код обмена информацией), хотя в принципе это одно и то же. В компьютерных системах за пределами США могут применяться совершенно другие коды.)

Стандартный код ASCII воспринимает числовые значения в диапазоне от 0 до 127. Этот диапазон достаточно мал, и чтобы охватить его, достаточно всего лишь 7 разрядов. Тип `char` обычно определяется как 8-разрядная единица памяти, следовательно, ее более чем достаточно для того, чтобы вместить стандартный код ASCII. Многие системы, такие как, например, IBM PC и Apple Macintosh, используют расширенные коды ASCII (разные для этих двух систем), которые не выходят за пределы 8 разрядов. В общем смысле, язык С гарантирует, что тип `char` достаточно большой, чтобы представлять базовый набор символов для систем, в которых реализован С.

Многие наборы символов состоят из более чем 127 или даже 255 значений. Например, существует набор символов "Japanese kanji" для японских иероглифов. В рамках коммерческой инициативы Unicode был создана система, представляющая различные наборы символов, используемых в различных частях мира, которая в настоящее время содержит более 96 000 символов. Организация ISO и комиссия IEC (International Electrotechnical Commission — Международная электротехническая комиссия) вместе разработали стандарт наборов символов, получивший название ISO/IEC 10646. К счастью, с помощью стандарта ISO/IEC 10646 удалось сохранить совместимость стандарта Unicode с более широким стандартом ISO/IEC 10646.

Платформа, использующая один из этих наборов в качестве своего базового набора символов, могла употреблять 16-разрядное и даже 32-разрядное представление типа `char`. В языке С байт определяется как число разрядов, используемых для представления типа `char`. В документации по языку С сказано, что в таких системах байт должен содержать 16 или 32 разряда, но не 8 разрядов.

Объявление переменных типа `char`

Как и следовало ожидать, переменные типа `char` объявляются так же, как и другие переменные. Вот несколько примеров таких объявлений:

```
char response;
char itable, latan;
```

Приведенный код создает три переменных типа `char`: `response`, `itable` и `latan`.

Символьные константы и инициализация

Предположим, что вы хотите инициализировать символьную константу буквой *A*. Компьютерные языки должны облегчить решение этой задачи, по этой причине вы не должны помнить все коды ASCII, да вам это и не нужно. Вы можете присвоить символ `A` переменной `grade` с помощью следующей процедуры инициализации:

```
char grade = 'A';
```

Одиночная буква, заключенная в одиночные кавычки, представляет собой *символьную константу* языка С. Когда в поле зрения компилятора попадает конструкция `'A'`, он переводит ее в соответствующее кодовое значение. Одиночные кавычки здесь очень важны. Рассмотрим еще один пример:

```
char broiled;   /* объявление переменной типа char */
broiled = 'T';  /* Правильно */
broiled = T;    /* Неправильно! Компилятор думает, что T является переменной */
broiled = "T";  /* Неправильно! Компилятор думает, что "T" является строкой */
```

Если опустить кавычки, то компилятор подумает, что T представляет имя переменной. Если вы примените двойные кавычки, он подумает, что вы используете строку. Мы обсудим строки в главе 4.

Поскольку символы на самом деле хранятся как числовые значения, вы можете также употреблять числовые коды для присваивания значений:

```
char grade = 65; /* правильно в контексте ASCII, но это плохой стиль */
```

В данном примере 65 имеет тип int, однако, поскольку это значение меньше максимального значения типа char, оно может быть присвоено переменной grade без каких-либо осложнений. Поскольку 65 представляет собой ASCII-код буквы *A*, в условиях данного примера переменной grade присваивается значение A. Тем не менее, обратите внимание, что в рассматриваемом примере предполагается использование в системе кодировки ASCII. Указание 'A' вместо 65 позволяет получить программный код, который работает на любой системе. Поэтому намного проще употреблять символьные константы, чем значения числовых кодов.

Несколько странным является тот факт, что C рассматривает символьные константы как тип int, а не char. Например, в системе с 32-разрядым типом int и с 8-разрядным типом char, код

```
char grade = 'B';
```

представляет 'B' как числовое значение 66, хранящееся в 32-разрядной ячейке памяти, но при этом переменная grade с трудом умещает значение 66 в 8-разрядную ячейку. Эта характеристика символьных констант позволяет определять такую константу в виде 'FATE', при этом четыре отдельных 8-разрядных ASCII-кода помещаются в 32-разрядную ячейку. В то же время, попытка присвоить такую символьную константу переменной типа char приводит к тому, что используются только последние 8 разрядов, в силу чего данная переменная получает значение 'E'.

Непечатаемые символы

Технология, использующая одиночные кавычки, хороша для символов, цифр и знаков препинания, однако если внимательно просмотреть таблицу кодов ASCII, можно найти непечатаемые символы. Например, некоторые из них представляют собой такие действия, как возврат на одну позицию влево, переход на следующую строку или выдачу звукового сигнала терминалом или встроенным динамиком. Как все это можно представить? Язык C предлагает три способа.

Первый способ уже был описан, это использование ASCII-кодов. Например, значением ASCII-кода для символа звукового сигнала является 7, так что вы можете использовать следующий оператор:

```
char beep = 7;
```

Второй способ представления неудобных символов в языке C предполагает применение специальных последовательностей символов. Такие последовательности называются *управляющими последовательностями*. В табл. 3.2 приводится перечень управляющих последовательностей вместе с их описаниями.

Таблица 3.2. Управляющие последовательности

Последовательность	*Описание*
\a	Предупреждение (стандарт ANSI C).
\b	Возврат на одну позицию влево.
\f	Перевод страницы.
\n	Новая строка.
\r	Возврат каретки.
\t	Горизонтальная табуляция.
\v	Вертикальная табуляция.
\\	Обратная косая черта (\).
\'	Одиночная кавычка (').
\"	Двойная кавычка (").
\?	Знак вопроса (?).
\0oo	Восьмеричное значение (o представляет восьмеричную цифру).
\xhh	Шестнадцатеричное значение (o представляет шестнадцатеричную цифру).

Когда символьным переменным присваиваются управляющие последовательности, последние должны быть заключены в одиночные кавычки. Например, можно записать такой оператор:

```
char nerf = '\n';
```

а затем распечатать переменную nerf, что обеспечит передвижение на следующую строку на принтере или на экране монитора.

Теперь более подробно изучим, что делает каждая управляющая последовательность. Символ предупреждения (\a), введенный стандартом C90, вызывает появление звукового или визуального предупреждающего сигнала. Природа предупреждающего сигнала зависит от оборудования, чаще других используется звуковой сигнал. (В некоторых системах предупреждающий символ не никак не проявляется.) Стандарт ANSI требует, чтобы предупреждающий сигнал не менял активной позиции. Под *активной позицией* в стандарте понимается место в устройстве отображения (экран, телетайп, печатающее устройство и так далее), в котором должен появиться следующий символ, если бы не было предупреждающего символа. Короче говоря, активная позиция есть обобщение понятия экранного курсора, с которым вы, скорее всего, привыкли работать. Использование предупреждающего символа в программе, отображенной на экране, должно вызвать звуковой сигнал без перемещения экранного курсора.

Далее, управляющие последовательности \b, \f, \n, \r, \t и \v представляют собой обычные символы управления выходным устройством. Их проще всего описывать с учетом их воздействия на активную позицию. Возврат на одну позицию влево (\b) перемещает активную позицию назад на один символ текущей строки. Символ перевода страницы (\f) переносит активную позицию в начало следующей страницы. Символ новой строки (\n) перемещает активную позицию в начало следующей строки.

Символ возврата каретки (\r) переносит активную позицию в начало текущей строки. Символ горизонтальной табуляции (\t) перемещает активную позицию в следующую точку горизонтальной табуляции (обычно эти точки находятся в позициях 1, 9, 17, 25 и так далее). Символ вертикальной табуляции (\v) перемещает активную позицию в следующую точку вертикальной табуляции.

Эти управляющие последовательности не обязательно работают на всех устройствах отображения. Например, символы перевода страницы и вертикальной табуляции не приводят к перемещению курсора и вызывают появление на экране случайных символов, однако, они работают в соответствие с описанием, будучи переданными на принтер.

Следующие три управляющие последовательности (\\, \' и \") обеспечивают возможность использования символов \, ', и " в качестве символьных констант. (Так как эти символы служат для определения символьных констант как части команды printf(), то может возникнуть путаница, если вы воспользуетесь ими обычным способом.) Предположим, что вы хотите распечатать следующую строку:

```
Gramps sez, "a \ is a backslash."
```

Необходимо подготовить следующий программный код:

```
printf("Gramps sez, \"a \\ is a backslash.\"\n");
```

Две последних формы (\0oo и \xhh) — это специальные представления ASCII-кода. Чтобы представить тот или иной символ в виде его восьмеричного ASCII-кода, нужно поставить перед ним обратную косую черту (\) и заключить всю эту конструкцию в одиночные кавычки. Например, если ваш компилятор не распознает символ предупреждения (\a), вы можете воспользоваться соответствующим ему ASCII-кодом:

```
beep = '\007';
```

Вы можете опустить нулевые старшие разряды, так что запись '\07' или даже '\7' будет правильной. Такая форма записи означает, что числа должны интерпретироваться как восьмеричные, даже если отсутствует ведущий 0.

Начиная со стандарта C90, C предлагает третий вариант — использование шестнадцатеричной формы для представления символьных констант. В этом случае за обратной косой чертой следуют символы x или X и от одной до трех шестнадцатеричных цифр. Например, символу <Ctrl+P> соответствует шестнадцатеричный ASCII-код 10 (16 в десятичной системе счисления), следовательно, его можно представить как '\x10' или '\X010'. На рис. 3.5 показаны некоторые из представительных целых типов. Когда вы используете код ASCII, обращайте внимание на различие между числами и символами чисел (цифры). Например, символ 4 представлен в коде ASCII значением 52. Запись '4' представляет символ 4, но не числовое значение.

В этом месте у вас могут возникнуть три вопроса:

- *Почему в последнем примере управляющие последовательности не заключены в одиночные кавычки* (printf("Gramps sez, \"a \\ is a backslash.\"\n");)? Когда конкретный символ (не имеет значения, является ли он управляющей последовательностью или нет) есть часть строки символов, заключенной в двойные кавычки, не помещайте его в одиночные кавычки. Обратите внимание, что ни один из символов, использованных в этом примере (G, r, a, m, p, s и так далее), не заключены в одиночные кавычки. Строка символов, заключенная в двойные

кавычки, называется *символьной строкой*. (Строки изучаются в главе 4.) Аналогично, оператор `printf("Hello!\007\n");` распечатает `Hello!` и выдаст звуковой сигнал, в то же время оператор `printf("Hello!7\n");` распечатает `Hello!7`. Цифры, не являющиеся частью управляющей последовательности, рассматриваются как обычные символы, которые нужно распечатать.

- *Когда используются ASCII-коды, а когда управляющие последовательности?* Если у вас имеется возможность выбора между применением одной из специальных управляющих последовательностей, скажем `'\f'`, и эквивалентного ASCII-кода, например, `'\014'`, отдавайте предпочтение `'\f'`. Во-первых, при таком представлении легче понять ее смысл. Во-вторых, такая запись обладает лучшей переносимостью. Если вы работаете с системой, в которой не используется кодировка ASCII, последовательность `'\f'`, тем не менее, будет работать должным образом.
- *Если нужно использовать цифровой код, почему необходимо указывать, скажем,* `'\032'` *вместо* `032`? Во-первых, использование записи `'\032'` вместо `032` позволит другому пользователю, читающему ваш программный код, понять, что вы хотите представить код соответствующего символа. Во-вторых, управляющая последовательность, такая как `\032`, может быть встроена в некоторую часть строки на C таким же способом, как и `\007` в первом пункте.

Примеры целочисленных констант

Тип	*Шестнадцатеричный*	*Восьмеричный*	*Десятичный*
`char`	`\0x41`	`\0101`	Отсутствует
`int`	`0x41`	`0101`	`65`
`unsigned int`	`0x41u`	`0101u`	`65u`
`long`	`0x41L`	`0101L`	`65L`
`unsigned long`	`0x41UL`	`0101UL`	`65UL`
`long long`	`0x41LL`	`0101LL`	`65LL`
`unsigned long long`	`0x41ULL`	`0101ULL`	`65ULL`

Рис. 3.5. *Виды записи целочисленных констант семейства* `int`

Печатаемые символы

Функция `printf()` использует спецификатор `%c`, чтобы показать, что должен распечатываться символ. Вспомните, что символьная переменная хранится как однобайтовое целое значение. По этой причине, если вы печатаете значение переменной типа `char` с обычным спецификатором `%d`, вы получите целое число. Спецификатор формата `%c` заставляет функцию `printf()` отобразить символ, который имеет в качестве кодового значение это целое число. В листинге 3.5 приводится программа, которая представляет переменную типа `char` двумя способами.

Листинг 3.5. Программа charcode.c

```
/* charcode.c - отображает кодовое значение символа */
#include <stdio.h>
int main(void)
{
   char ch;

   printf("Введите какой-нибудь символ.\n");
   scanf("%c", &ch);                    /* пользователь вводит символ */
   printf("Код символа %c равен %d.\n", ch, ch);

   return 0;
}
```

Вот как выглядит пример выполнения этой программы:

```
Введите какой-нибудь символ.
C
Код символа C равен 67.
```

При работе с программой не забывайте нажать клавишу <Enter> или <Return> после ввода символа. Функция `scanf()` затем выбирает символ, который был введен с клавиатуры, а амперсанд (`&`) означает, что этот символ присваивается переменной `ch`. Затем функция `printf()` дважды распечатывает значение переменной `ch`, первый раз как символ (на что указывает код спецификатора `%c`), а второй раз как десятичное целое число (на что указывает код спецификатора `%d`). Обратите внимание на то обстоятельство, что спецификаторы функции `printf()` определяют, как будут отображаться данные, но не то, как они хранятся в памяти (рис. 3.6).

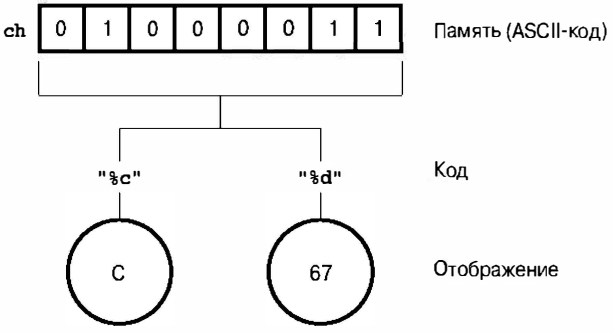


Рис. 3.6. *Отображение данных на экране и хранение их в памяти*

Со знаком или без знака?

В некоторых реализациях языка C данные типа `char` имеют тип со знаком. Это значит, что данные типа `char` могут принимать значения в диапазоне от –128 до 127. В других реализациях данные типа `char` выступают как данные без знака и могут принимать значения в диапазоне от 0 до 255. В описании вашего компилятора должно быть указано, к какой разновидности принадлежит тип `char`, либо вы это можете узнать, изучив заголовочный файл `limits.h`, который будет рассматриваться в следующей главе.

В рамках стандарта C90 язык C позволяет использовать ключевые слова signed и unsigned с типом char. Далее, независимо от того, какими являются данные тип char по умолчанию, данные со знаком будут иметь тип signed char, а данные без знака — тип unsigned char. Эти версии типа char полезны, если упомянутый тип применяется для работы с небольшими целыми числами. При работе с символами используйте стандартный тип char без модификаторов.

Тип _Bool

Тип _Bool был добавлен стандартом C99 и он используется для представления булевских значений, то есть, логических значений true (истина) и false (ложь). Поскольку язык C использует значение 1 для представления значения true и 0 — для представления значения false, тип _Bool по существу есть целый тип, однако такой целый тип, который в принципе требует 1 бит памяти, поскольку этого достаточно, чтобы охватить весь диапазон от 0 до 1.

Переносимые типы: inttypes.h

Существуют ли еще какие-то целочисленные типы? Таких типов больше нет, в то же время возможны другие названия существующих целочисленных типов, которыми вы можете пользоваться. Можно подумать, что эти названия более чем адекватно представляют существующие типы целочисленных данных, однако с основными названиями типов у вас могут возникнуть проблемы. Тот факт, что переменная имеет тип int, ничего не говорит о том, сколько разрядов он содержит, пока вы не ознакомитесь с документацией по вашей системе. Чтобы обойти эту проблему, стандарт C99 предлагает альтернативный набор имен, который точно определяет, с какими данными вы имеете дело. Например, имя int16_t представляет 16-разрядный целочисленный тип со знаком, в то время как имя uint32_t — 32-разрядный целочисленный тип без знака.

Чтобы эти имена можно было использовать в программах, включите в нее заголовочный файл inttypes.h. (Следует отметить, что во время подготовки данного издания некоторые компиляторы еще не поддерживали эту возможность.) Этот файл использует средство typedef (его краткое описание впервые дается в главе 5) для создания имен новых типов.

Например, он делает имя uint32_t синонимом, или альтернативным именем, стандартного типа с нужными характеристиками, возможно, это unsigned int в одной системе и unsigned long — в другой. Ваш компилятор обеспечит совместимость любого заголовочного файла с вычислительной системой, на которой вы работаете. Эти новые обозначения называются *типами с точной шириной*. Обратите внимание, что в отличие от типа int, тип uint32_t не является ключевым словом, таким образом, компилятор не распознает его, пока вы не включите в программу заголовочный файл inttypes.h.

Одна из возможных проблем, возникающих при попытке установить типы с точной шириной, заключается в том, что та или иная конкретная система может не поддерживать некоторые выбранные вами типы, так что нельзя гарантировать доступность, скажем, типа int8_t (8-разрядное целое со знаком). Чтобы избежать этой проблемы, стандарт C99 определяет второй набор имен, который гарантирует, что тип

достаточно велик, чтобы соответствовать требованиям спецификации, и что нет других типов, которые также могут обеспечить выполнение соответствующей работы и которые были бы меньше. Эти типы называются *типами с минимальной шириной*. Например, int_least8_t представляет собой альтернативное имя наименьшего доступного типа, который может принять 8-разрядное целое значение со знаком. Если бы наименьший тип в той или иной конкретной системе был бы 8-разрядным, то объявлять тип int8_t не имеет смысла. В то же время, может оказаться доступным тип int_least8_t и, возможно, он реализован как 16-разрядное целое.

Разумеется, некоторых программистов больше интересует быстродействие, чем расход памяти. Для них стандарт C99 определяет набор типов, которые позволяют достичь максимальной скорости вычислений. Эти типы называются *высокоскоростными типами с минимальной шириной*. Например, int_fast8_t может быть объявлено как альтернативное имя целочисленного типа данных вашей системы, который обеспечивает высокоскоростные вычисления для 8-разрядных значений со знаком. И, наконец, некоторых программистов может устроить только максимально возможный в системе целочисленный тип; этот тип представляет имя intmax_t и он может принять любое допустимое значение целого числа со знаком. Аналогично, uintmax_t представляет тип наибольшего допустимого целочисленного значения без знака. Фактически эти типы должны быть больше, чем long long и unsigned long, поскольку реализациям C предоставляется возможность определять типы, которые не относятся к числу обязательных.

Стандарт C99 вводит не только эти новые, переносимые имена типов, но также оказывает помощь при вводе и выводе. Например, функция printf() требует специальных спецификаторов для конкретных типов. Таким образом, что вы должны сделать, чтобы отобразить значение типа int32_t, если она может потребовать спецификатора %d для одного определения и спецификатора %ld для другого? Стандарт C99 предлагает несколько макрокоманд (их описание приводится в главе 4), которые используются для отображения переносимых типов. Например, заголовочный файл inttypes.h определяет PRId16 как строку, представляющую соответствующий спецификатор (например, hd или d) для 16-разрядного значения со знаком. В листинге 3.6 представлен краткий пример, демонстрирующий использование переносимых типов и соответствующих им спецификаторов.

Листинг 3.6. Программа altnames.c

```
/* altnames.c — переносимые имена для целочисленных типов */
#include <stdio.h>
#include <inttypes.h>   // поддерживает переносимые типы
int main(void)
{
   int16_t me16;        // me16 - это 16-разрядная переменная со знаком
   me16 = 4593;
   printf("Сначала предположим, что int16_t имеет тип short: ");
   printf("me16 = %hd\n", me16);
   printf("Далее не будем делать никаких предположений.\n");
   printf("Вместо этого воспользуйтесь \"макрокомандой\" из файла inttypes.h: ");
   printf("me16 = %" PRId16 "\n", me16);
   return 0;
}
```

В завершающем аргументе функции printf(), PRId16 замещается своим определением "hd" из файла inttypes.h, в результате чего соответствующая строка программы принимает такой вид:

```
printf("me16 = %" "hd" "\n", me16);
```

В то же время С формирует из нескольких последовательных строк, заключенных в двойные кавычки, одну строку в кавычках, в результате чего эта строка программы принимает следующий вид:

```
printf("me16 = %hd\n", me16);
```

Ниже показаны выходные данные программы; обратите внимание на то, что в рассматриваемом примере также используется управляющая последовательность \" для вывода двойных кавычек:

```
Сначала предположим, что int16_t имеет тип short: me16 = 4593
Далее не будем делать никаких предположений.
Вместо этого воспользуйтесь "макрокомандой" из файла inttypes.h: me16 = 4593
```

В справочном разделе VI (приложение Б) приводится полное описание добавлений заголовочного файла inttypes.h, а также список всех макрокоманд спецификаторов.

Поддержка стандарта С99

Поставщики компиляторов языка С подходили к реализации средств, введенных новым стандартом С99, с разной степенью готовности и с разной оперативностью. В период подготовки данной книги некоторые компиляторы пока не поддерживали заголовочный файл inttypes.h и соответствующие средства.

Данные типа float, double и long double

Разнообразие целочисленных типов в большинстве случаев облегчает разработку программного обеспечения и способствует повышению его качества. В то же время программы, ориентированные на работу в таких предметных областях, как математика и финансы, часто оперируют числами с плавающей запятой. В языке С такие числа имеют тип float, double или long double. Они соответствуют данным типа real в таких языках программирования, как FORTRAN и Pascal. Как уже говорилось выше, применение чисел с плавающей запятой дает вам возможность работать с гораздо большими диапазонами чисел, включая десятичные дроби. Представление чисел с плавающей запятой во многом подобно научной форме записи, которая используется учеными для записи очень больших и очень маленьких чисел. Рассмотрим эту форму записи.

В научной форме записи числа представлены как десятичные числа, умноженные на соответствующую степень числа 10. Рассмотрим несколько примеров.

Число	*Научная форма записи*	*Экспоненциальная форма записи*
1 000 000 000	1.0×109	1.0e9
123 000	1.23×105	1.23e5
322.56	3.2256×102	3.2256e2
0.000056	5.6×10-5	5.6e-5

В первом столбце показана обычная форма записи числа, во втором столбце — научная форма записи, а в третьем — экспоненциальная форма записи, которая представляет собой научную форму записи, обычно используемую при работе с компьютерами, при этом за обозначением e следует показатель степени 10. На рис. 3.7 приводится еще несколько примеров чисел с плавающей запятой.

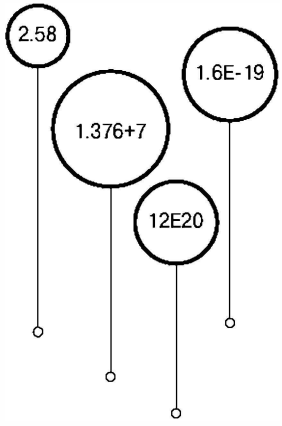


Рис. 3.7. *Примеры чисел с плавающей запятой*

Стандарт языка C требует, чтобы значения типа `float` имели, по меньшей мере, шесть значащих цифр после точки и представляли диапазон чисел, по меньшей мере, от 10^{-37} до 10^{+37}. Первое требование, например, означает, что тип `float` должен представлять, по меньшей мере, первые шесть цифр такого числа, как 33.333333. Второе требование по достоинству оценят те, кто оперирует такими величинами, как масса солнца (2.0e30 килограмм), электрический заряд протона (1.6e-19 кулона) или сумма государственного долга. Для хранения чисел с плавающей запятой системы используют 32 разряда. Восемь разрядов отводятся под значение экспоненты и ее знака, остальные 24 разряда служат для представления неэкспоненциальной части числа, называемой *мантиссой* или *значащей частью числа*, и ее знака.

Для представления чисел с плавающей запятой в языке C имеется также тип `double` (он обеспечивает двойную точность). Минимальный диапазон возможных значений для данных типа `double` тот же, что и для типа `float`, а минимальное количество значащих цифр увеличено до 10. Для представления данных типа `double` обычно используется 64 разряда вместо 32. В некоторых системах используются все 32 дополнительных разряда неэкспоненциальной части. Это увеличивает количество значащих цифр и уменьшает ошибку округления. В других системах часть этих разрядов применяется для увеличения количества значащих цифр экспоненты, благодаря чему расширяется диапазон возможных значений этого типа. При каждом из этих подходов количество значащих цифр не бывает меньше 13, что с избытком удовлетворяет минимальным требованиям стандарта.

Язык C допускает третий тип данных с плавающей запятой: `long double`. Этот тип вводит с целью достижения еще большей точности, чем точность, обеспечиваемая типом `double`. Однако C гарантирует только то, что точность типа `long double`, по меньшей мере, не уступает точности типа `double`.

Объявление переменных с плавающей запятой

Объявление и инициализация переменных с плавающей запятой осуществляется так же, как и родственных элементов из семейства целочисленных типов. Рассмотрим несколько примеров:

```
float noah, jonah;
double trouble;
float planck = 6.63e-34;
long double gnp;
```

Константы с плавающей запятой

При записи константы с плавающей запятой вы можете выбрать одну из нескольких возможностей. В основной форме записи константа с плавающей запятой представляет собой некоторую последовательность цифр со знаком, включающая и десятичную точку, за которой следует буква e или E, а за ними следует экспонента со знаком, представляющая целую степень числа 10. Вот два примера правильной записи констант с плавающей запятой:

```
-1.56E+12
2.87e-3
```

Знак плюс вы везде можете опускать. Вы можете также опустить десятичную точку (2E5) или экспоненциальную часть (19.28), но не то и другое одновременно. Вы можете обойтись без дробной части (3.E16) или целой части (.45E-6), но не без обоих компонентов сразу (это уже чересчур!). Вот еще несколько допустимых констант с плавающей запятой:

```
3.14159
.2
4e16
.8E-5
100.
```

В константах с плавающей запятой не должны присутствовать пробелы.

Например, вот недопустимая константа: `1.56 E+12`

По умолчанию компилятор полагает, что константы с плавающей запятой имеют двойную точность. Предположим, например, что some представляет собой переменную с плавающей запятой, и у вас имеется следующий оператор:

```
some = 4.0 * 2.0;
```

Тогда значения 4.0 и 2.0 запоминаются как данные типа double, при этом под каждое из них отводятся (обычно) 64 бита памяти. Для вычисления произведение применяется арифметика с двойной точностью, и только после выполнения операции умножения результат приводится к размеру double. Это обеспечивает более высокую точность вычислений, однако замедляет вычисления.

В языке С имеются средства, которые позволяют изменить такое поведение компилятора, установленное по умолчанию, путем использования суффикса f или F, что заставит компилятор рассматривать константу с плавающей запятой как тип; примерами могут служить 2.3f и 9.11E9F. Суффиксы l или L определяют константу как число типа long double, например, 54.3l и 4.32e4L. Обратите внимание на то, что букву L труднее перепутать с 1 (единица), чем букву l. Если число с плавающей запятой не содержит суффикса, это число имеет тип double.

Стандарт С99 ввел новый формат для представления констант с плавающей запятой. Он использует шестнадцатеричный префикс (0x или 0X) с шестнадцатеричными цифрами, p или P вместо e или E и экспоненту, которая является степенью числа 2, а не 10. Такое число может принимать следующий вид:

```
0xa.1fp10
```

a — это 10, .1f — $^{1}/_{16}$-ая плюс $^{15}/_{256}$-ая и p10 — 2^{10} или 1024, что в сумме дает 10364.0 в десятичном эквиваленте.

Не все компиляторы C поддерживают это средство, введенное стандартом C99.

Печать значений с плавающей запятой

Функция `printf()` использует спецификатор формата `%f` для печати чисел типа `float` и `double` в десятичном представлении и спецификатор `%e` для печати в экспоненциальном представлении. Если ваша система поддерживает шестнадцатеричный формат чисел с плавающей запятой, введенный стандартом C99, вы можете использовать `a` или `A` вместо `e` или `E`. Тип `long double` требует спецификаторов `%Lf`, `%Le` и `%La` для печати данных этого типа. Обратите внимание, что как тип `float`, так и тип `double` используют спецификаторы `%f`, `%e` или `%a` для вывода. Это объясняется тем, что язык C автоматически расширяет значения типа `float` до типа `double`, когда они передаются в качестве аргументов любой функции, такой как, например, `printf()`, в прототипе которой тип аргумента явно не определен. Код в листинге 3.7 служит иллюстрацией такого поведения.

Листинг 3.7. Программа `showf_pt.c`

```
/* showf_pt.c — отображает значение типа float двумя способами */
#include <stdio.h>
int main(void)
{
   float aboat = 32000.0;
   double abet = 2.14e9;
   long double dip = 5.32e-5;

   printf("%f может быть записано как %e\n", aboat, aboat);
   printf("%f может быть записано как %e\n", abet, abet);
   printf("%f может быть записано как %e\n", dip, dip);

   return 0;
}
```

Выполнение этой программы дает следующие результаты:

```
32000.000000 может быть записано как 3.200000e+04
2140000000.000000 может быть записано как 2.140000e+09
0.000053 может быть записано как 5.320000e-05
```

Этот пример может служить иллюстрацией вывода данных по умолчанию. В следующей главе мы обсудим, как управлять внешним видом этих выходных данных за счет установки ширины поля и количества позиций справа от десятичной точки.

Переполнение и потеря значимости в операциях с плавающей запятой

Предполагая, что наибольшее возможное значение типа `float` равно примерно 3.4E38, выполните следующие операции:

```
float toobig = 3.4E38 * 100.0f;
printf("%e\n", toobig);
```

Что произойдет в данном случае? Это пример *переполнения* (overflow), когда в результате вычислений получается слишком большое число, чтобы его можно было пра-

вильно представить. Поведение системы в таких случаях обычно не определено, но в рассматриваемом случае С присваивает переменной toobig специальное значение, которое обозначает *бесконечность* (infinity), и функция printf() в качестве значения отображает либо inf, либо infinity (либо другие разновидности).

Что можно сказать о делении очень малых чисел? В этом случае ситуация более сложная. Вспомните, что число типа float хранится в виде сочетания показателя степени и значащей части, или *мантиссы*. В рассматриваемом случае это число, имеющее минимально возможный показатель степени, а также наименьшее значение, которое использует все доступные биты, отведенные для представления мантиссы. Именно это число является наименьшим числом, представленным с наибольшей точностью, которая достижима для значения типа float. Теперь разделим его на 2. В общем случае это приводит к уменьшению показателя степени, но в данном случае показатель уже достиг своего нижнего предела. В такой ситуации компьютер сдвигает разряды мантиссы вправо, освобождая первую позицию и теряя последнюю двоичную цифру. Аналогичная картина возникает, если выбрать 10 в качестве основания системы счисления, взять число с четырьмя значащими цифрами, например, 0.1234E-10, и разделить его на 10, получив при этом 0.0123E-10. Вы получите ответ, но в процессе деления вы потеряете цифру. Такая ситуация называется *потерей значимости* (underflow), а язык С рассматривает значения с плавающей запятой, которые утратили полную точность типа, как *субнормальные*. Таким образом, деление наименьшего положительного значения с плавающей запятой на 2 дает в результате субнормальное значение. Если вы выполните деление на достаточно большое значение, вы потеряете все цифры и получите в результате 0. В настоящее время библиотека С может предоставить в ваше распоряжение функции, которые дают вам возможность проверить, не дают ли ваши вычисления результаты в виде субнормальных значений.

Существует еще одно специальное значение с плавающей запятой: NaN ("not-a-number" – "не число"). Например, вы передаете функции asin() некоторое значение, а она возвращает угол, для которого переданное значение является синусом. Однако значение синуса не может быть больше 1, в силу чего функция не определена для значений, превышающих 1. В таких случаях функция возвращает значение NaN, которое функция printf() отображает как nan, NaN или как-то похоже.

Ошибки округления данных с плавающей запятой

Возьмем некоторое число, прибавим к нему 1 и вычтем из суммы исходное число. Что мы получим в результате? Мы получим 1. Вычислительные операции, выполняемые над числами с плавающей запятой, как показывает приведенный ниже пример, могут дать другой результат:

```
/* floaterr.c – служит иллюстрацией ошибки округления */
#include <stdio.h>
int main(void)
{
   float a,b;
   b = 2.0e20 + 1.0;
   a = b - 2.0e20;
   printf("%f \n", a);
   return 0;
}
```

В результате выполнения этой программы получены следующие результаты:

```
0.000000 ← старая версия компилятора gcc в операционной системе Linux
-13584010575872.000000 ← Turbo C 1.5
4008175468544.000000   ← CodeWarrior 9.0, MSVC++ 7.1
```

Причина появления таких странных результатов состоит в том, что компьютер не следит за тем, чтобы под числа с плавающей запятой было отведено столько десятичных позиций, сколько нужно для правильного выполнения арифметических операций. Число 2.0e20 представлено цифрой 2, за которой следует 20 нулей, прибавляя 1, вы пытаетесь изменить 21-ю цифру. Чтобы эта операция была выполнена правильно, программа должна иметь возможность сохранять число, состоящее из 21 цифры. Обычно число с плавающей запятой представлено шестью или семью цифрами, умноженных на основание системы счисления в соответствующей степени, увеличивающей или уменьшающей это числовое значение. Эта попытка обречена на неудачу. С другой стороны, если вы используете число 2.0e4 вместо 2.0e20, вы получите правильный ответ, поскольку вы пытаетесь изменить пятую цифру, а числа с плавающей запятой обладают достаточной точностью для правильного выполнения такой операции.

Комплексные и мнимые типы

Во многих научных и технических расчетах используются комплексные и мнимые числа. Стандарт C99 поддерживает эти числа, правда с некоторыми оговорками. В рамках автономных реализаций, например, при реализации встроенных процессоров, нет необходимости во всех этих типах. (Микропроцессор видеомагнитофона, возможно, не нуждается в комплексных числах во время выполнения своих функций.) В свою очередь мнимые типы не относятся к числу обязательных. Существуют три комплексных типа, получивших названия `float _Complex`, `double _Complex` и `long double _Complex`. Переменная типа `float _Complex`, например, будет содержать два значения типа `float`, одно из них представляет действительную часть комплексного числа, а другая представляет мнимую его часть. Аналогично, существуют три мнимых типа, названные `float _Imaginary`, `double _Imaginary` и `long double _Imaginary`.

Включение заголовочного файла `complex.h` позволяет вам делать подстановки слова `complex` (комплексное) вместо `_Complex` и слова `imaginary` (мнимое) вместо `_Imaginary`, и появляется возможность использования символа `I` для представления квадратного корня из –1.

За пределами базовых типов

Список фундаментальных типов данных завершен. Для одних из вас этот список может показаться слишком длинным. В то же время многие другие могут подумать, что необходимы новые типы. Что можно сказать о типе символьной строки? В языке C нет такого типа, в то же время он обеспечивает успешную работу со строками. Впервые со строками вы ознакомитесь в главе 4.

В C присутствуют также и другие типы, являющиеся производными от базовых типов. Эти типы включают массивы, указатели, структуры и объединения. И хотя все они являются предметом обсуждения в других главах, тем не менее, некоторые указатели просочились в примеры, рассмотренные в настоящей главе. (*Указатель* (pointer) указывает на место в памяти, в котором хранится переменная или какой-то другой объект данных. Префикс `&`, который используется в функции `scanf()`, создает указатель, сообщающий этой функции, куда следует поместить информацию.)

Сводка: базовые типы данных

Ключевые слова:

Названия основных типов данных образуются с помощью 11 ключевых слов: `int`, `long`, `short`, `unsigned`, `char`, `float`, `double`, `signed`, `_Bool`, `_Complex` и `_Imaginary`.

Целые числа со знаком:

Это могут быть как положительные, так и отрицательные числа:

- `int` — основной тип целочисленный тип данных в конкретной вычислительной системе. Язык C отводит под тип `int` не менее 16 разрядов.
- `short` или `short int` — максимальное целое число типа `short` не превосходит наибольшего целочисленного значения типа `int`. Язык C гарантирует, что под тип `short` будут отведены, по меньшей мере, 16 разрядов.
- `long` или `long int` — может хранить целое число, которое, по меньшей мере, не меньше наибольшего числа типа `int` или больше его. Язык C гарантирует, что под тип `long` отводятся 32 разряда.
- `long long` или `long long int` — этот тип может быть целым числом, которое, по меньшей мере, не меньше наибольшего числа типа `long`, а, возможно, и больше его. Под тип `long long` отводятся, по меньшей мере, 64 разряда памяти.

Обычно тип `long` имеет большую длину, чем тип `short`, а длина типа `int` совпадает с длиной одного из этих типов. Например, системы на персональных компьютерах, функционирующих под управлением DOS, имеют 16-разрядные типы `short` и `int` и 32-разрядный тип `long`, системы, работающие на базе Windows 95, предлагают 16-разрядный тип `short` и 32-разрядные типы `int` и `long`. При желании вы можете использовать ключевое слово `signed` с любым из типов со знаком, тем самым явно указывая на то, что этот типы со знаком.

Целые без знака:

Эти типы могут хранить только нулевое или положительные значения. В силу этого обстоятельства увеличивается диапазон наибольшего возможного положительного числа.

Ставьте ключевое слово `unsigned` перед нужным типом: `unsigned int`, `unsigned long`, `unsigned short`. Одно ключевое `unsigned` означает то же, что и `unsigned int`.

Символы:

Существуют типографские символы, такие как `A`, `&` и `+`. По определению тип `char` использует 1 байт памяти для представления конкретного символа. Исторически сложилось так, что этот байт символа в большинстве случаев имеет длину 8 разрядов, но она может быть 16 разрядов и даже больше, если возникнет необходимость представить базовый набор символов.

- `char` — ключевое слово для этого типа данных. В одних реализациях используется тип `char` со знаком, в то время как в других реализациях используют тип `char` без знака. Язык C предоставляет возможность использовать ключевые слова `signed` и `unsigned` с тем, чтобы вы могли задать нужную форму данных.

Булевские значения:

Булевский тип представляет значения `true` (истина) и `false` (ложь); язык C использует 1 для представления `true` и 0 для представления `false`.

- `_Bool` — ключевое слово для этого типа. Это значение `int` без знака, оно должно быть достаточно большим, чтобы соответствовать диапазону от 0 до 1.

Вещественные числа с плавающей запятой:

Числа с плавающей запятой могут принимать положительные и отрицательные значения:

- `float` — основной тип данных с плавающей запятой в вычислительной системе; он может хранить, по меньшей мере, шесть значащих цифр.
- `double` — этот тип (возможно) служит представления больших, чем тип `float`, чисел с плавающей запятой. Он допускает больше значащих цифр (по меньшей мере, 10, но обычно их больше) и, возможно, большие значения показателя степени.
- `long double` — этот тип (возможно) служит для представления еще больших чисел с плавающей запятой. Он допускает большее число значащих цифр и, возможно, больших значений показателей степени, чем тип `double`.

Комплексные и мнимые числа с плавающей запятой:

Мнимые типы не принадлежат к числу базовых. В основу вещественных и мнимых компонентов положены соответствующие вещественные типы:

- `float _Complex`
- `double _Complex`
- `long double _Complex`
- `float _Imaginary`
- `double _Imaginary`
- `long double _Imaginary`

Сводка: объявление простой переменной

1. Выберите нужный вам тип данных.
2. Выберите имя для этой переменной, используя с этой целью все допустимые символы.
3. Для объявления переменной используйте следующий формат:

   спецификатор-типа имя-переменной;

   Компонент *спецификатор-типа* образуется из одного или большего числа ключевых слов типов; вот пример такого объявления:

   ```
   int erest;
   unsigned short cash;
   ```

4. Вы можете объявить сразу несколько переменных одного и того же типа, отделяя имена переменных друг от друга запятыми. Ниже приведен пример такого объявления:

   ```
   char ch, init, ans;
   ```

5. Вы можете инициализировать переменную в операторе объявления:

   ```
   float mass = 6.0E24;
   ```

Размеры типов

В таблицах 3.3 и 3.4 приведены размеры данных различных типов в некоторых широко используемых операционных средах языка C. (В некоторых таких средах вам предоставляется выбор размеров данных.) Каков размер данных в вашей системе? Попробуйте выполнить программу, код которой показан в листинге 3.8, чтобы получить ответ на этот вопрос.

Таблица 3.3. Размеры целочисленных типов (в битах) для некоторых известных вычислительных систем

Тип	*Macintosh Metrowerks CW (По умолчанию)*	*Linux на ПК*	*IBM PC с Windows XP и Windows NT*	*ANSI C Minimum*
`char`	8	8	8	8
`int`	32	32	32	16
`short`	16	16	16	16
`long`	32	32	32	32
`long long`	64	64	64	64

Таблица 3.4. Информация, характеризующая типы с плавающей запятой, используемые в некоторых известных системах

Тип	*Macintosh Metrowerks CW (По умолчанию)*	*Linux на ПК*	*IBM PC с Windows XP и Windows NT*	*ANSI C Minimum*
`float`	6 цифр от –37 до 38	6 цифр от –37 до 38	6 цифр от –37 до 38	6 цифр от –37 до 37
`double`	18 цифр от –4931 до 4932	15 цифр от –307 до 308	15 цифр от –307 до 308	10 цифр от –37 до 37
`long double`	18 цифр от –4931 до 4932	18 цифр от –4931 до 4932	18 цифр от –4931 до 4932	10 цифр от –37 до 37

Для каждого типа верхняя строка показывает количество значащих цифр, а вторая строка — диапазон, в котором показатель степени принимает значения (по основанию 10).

Листинг 3.8. Программа `typesize.c`

```
/* typesize.c – распечатка размеров типов */
#include <stdio.h>
int main(void)
{
/* Стандарт c99 предусматривает спецификатор %zd для размеров типов */
   printf("Тип int имеет размер %u байт(а,ов).\n", sizeof(int));
   printf("Тип char имеет размер %u байт(а,ов).\n", sizeof(char));
   printf("Тип long имеет размер %u байт(а,ов).\n", sizeof(long));
   printf("Тип double имеет размер %u байт(а,ов).\n", sizeof(double));
   return 0;
}
```

В языке C имеется встроенная операция `sizeof`, которая возвращает размер типа в байтах. (Некоторые компиляторы требуют спецификатора `%lu` вместо `%u` для вывода значений, возвращаемых `sizeof`. Это объясняется тем, что C оставляет определенную свободу в отношении фактического целочисленного типа без знака, который использует функция `sizeof` при возврате результатов. Стандарт C99 предлагает использовать спецификатор `%zd` для этого типа, и вы должны пользоваться именно указанным спецификатором в тех случаях, когда ваш компилятор его поддерживает.)

Выходные данные, полученные в результате выполнения программы, представленной в листинге 3.8, имеют следующий вид:

```
Тип int имеет размер 4 байт(а,ов).
Тип char имеет размер 1 байт(а,ов).
Тип long имеет размер 4 байт(а,ов).
Тип double имеет размер 8 байт(а,ов).
```

Эта программа определяет размеры только четырех типов, но вы легко можете приспособить ее с таким расчетом, чтобы она определяла размеры любого другого интересующего вас типа. Обратите внимание на тот факт, что тип `char` обязательно должен иметь размер 1 байт, поскольку C определяет размер одного байта через тип `char`. Следовательно, в системе с 16-разрядным типом `char` и 64-разрядным типом `double` функция `sizeof` сообщит, что тип `double` имеет размер 4 байта. Вы можете просмотреть заголовочные файлы `limits.h` и `float.h`, чтобы получить более подробную информацию о пределах типов. (В следующей главе мы продолжим анализ этих двух файлов.)

Между прочим, обратите внимание на то, что последний оператор `printf()` программы занимает две строки. Вы можете разбить этот оператор на любое количество частей при условии, что разбиение не проходит внутри раздела оператора, заключенного в кавычки, или в середине какого-то слова.

Использование типов данных

При разработке программы обращайте внимание на то, какие переменные вам требуются, и какие типы должны иметь эти переменные. Скорее всего, для представления чисел вы выберете `int` или, возможно, `float`, и `char` — для символов. Объявляйте их в начале функций, которые их используют. Выбирайте такие имена переменных, которые в той или иной степени отражают назначение хранящейся в них информации. Инициализируя ту или иную переменную, следите за тем, чтобы тип присваиваемого значения соответствовал типу переменной. Ниже представлены соответствующие примеры:

```
int apples = 3;          /* Правильно */
int oranges = 3.0;       /* Неправильная форма */
```

В отношении несовпадения типов C более либерален, чем, скажем, такой язык программирования, как Pascal. Компиляторы C допускает второй вид инициализации, но при этом они выдают сообщения об ошибке, особенно в тех случаях, когда вы установили высокий уровень предупреждений. Не следует потакать плохим привычкам.

Когда вы инициализируете переменную одного числового типа значением другого числового типа, C преобразует тип этого значения в тип переменной. Это означает, что может произойти потеря некоторых данных. В качестве примера рассмотрим следующие инициализации:

```
int cost = 12.99;        /* переменная int инициализируется
                            значением типа double */
float pi = 3.1415926536; /* переменная типа float инициализируется
                            значением типа double */
```

Первое объявление присваивает значение 12 переменной cost; при преобразовании значений с плавающей запятой в целые значения компилятор С вместо округления просто отбрасывает дробную часть числа (выполняет *усечение*). Во втором объявлении происходит некоторая потеря точности, поскольку для типа float гарантируется только точность в пределах шести цифр. Компиляторы могут (но не обязаны) выдавать предупреждающие сообщения, когда вы выполняете такую инициализацию. Вы можете столкнуться с такой проблемой во время компиляции программы, показанной в листинге 3.1.

Многие программисты и организации придерживаются систематических соглашений, регламентирующих присвоения имен переменным, согласно которым имя указывает на тип переменных. Например, вы можете воспользоваться префиксом i_ для указания типа int, и префиксом us_, чтобы указать тип unsigned short, откуда следует, что i_smart немедленно распознается как переменная типа int, а us_verysmart — как переменная типа unsigned short.

Аргументы и ошибки при их использовании

Ранее в этой главе уже отмечалась необходимость корректного использования функции printf(), и сейчас мы еще раз подчеркиваем важность такого подхода. Как уже говорилось выше, элементы информации, передаваемые функции, называются *аргументами*. Например, вызов функции printf("Добро пожаловать.") содержит один аргумент — "Добро пожаловать.". Последовательность символов в кавычках, например, "Добро пожаловать.", называется *строкой*. Мы будем изучать строки в главе 4. В настоящее время важным моментом является то, что строка, даже если она содержит несколько слов и знаков препинания, рассматривается как один аргумент.

Аналогично, вызов функции scanf("%d", &weight) содержит два аргумента: "%d" и &weight. Для отделения аргументов функции друг от друга в С используются запятые. Функции printf() и scanf() отличаются от других функций тем, что они не ограничиваются заранее установленным количеством аргументов. Например, мы вызывали функцию printf() с одним, двумя и даже тремя аргументами. Чтобы программа работала должным образом, она должна знать, сколько аргументов получает функция. Функции printf() и scanf() используют первые аргументы для того, чтобы указать, сколько дополнительных аргументов будет передаваться. Дело в том, что каждая спецификация формата в начальной строке указывает на наличие еще одного дополнительного аргумента. Например, показанный ниже оператор имеет два спецификатора формата, %d и %d:

```
printf("%d котов съедают %d банок тунца\n", cats, cans);
```

В этом случае наличие двух спецификаторов означает, что функция должна принять два аргумента, и в самом деле, дальше следуют два эти аргумента — cats и cans.

Обязанность программиста заключается в том, чтобы убедиться, что количество спецификаций формата соответствует числу дополнительных аргументов, и что тип спецификатора соответствует типу значения. В настоящее время в языке С имеется механизм прототипирования функций, который проверяет, правильно ли задано число аргументов функции и имеют ли эти аргументы соответствующий тип, однако он не

работает в случае функций `printf()` и `scanf()`, так как при каждом вызове они принимают разное число аргументов. А что произойдет, если программист не справится с этой обязанностью? В качестве примера рассмотрим программу, представленную в листинге 3.9.

Листинг 3.9. Программа `badcount.c`

```
/* badcount.c — неверное число аргументов */
#include <stdio.h>
int main(void)
{
   int f = 4;
   int g = 5;
   float h = 5.0f;
   printf("%d\n", f, g);      /* избыточное число аргументов */
   printf("%d %d\n",f);       /* недостаточное число аргументов */
   printf("%d %f\n", h, g);   /* неправильные типы спецификаторов */
   return 0;
}
```

Выходные данные, полученные в результате выполнения этой программы в системе Microsoft Visual C++ 7.1 (под управлением Windows XP), имеют следующий вид:

```
4
4 34603777
0 0.000000
```

Далее, выходные данные, полученные в результате выполнения этой программы в системе Digital Mars (под управлением Windows XP), имеют такой вид:

```
4
4 4239476
0 0.000000
```

И, наконец, выходные данные, полученные в результате выполнения этой программы в среде разработки Metrowerks CodeWarrior Development Studio 9 (под управлением Macintosh OS X), имеют вид:

```
4
4 3327456
1075052544 0.000000
```

Обратите внимание, что использование спецификатора `%d` для отображения значения `float` не приводит к преобразованию значения `float` в ближайшее значение `int`; наоборот, то, что при этом отображается, похоже на информационный мусор.

Аналогично, применение спецификатора `%f` для отображения значения `int` не приводит к преобразованию целочисленного значения в значение с плавающей запятой. Наряду с этим, результаты, которые вы получаете в случае недостаточного количества аргументов или в случае неправильно указанных типов аргументов, отличаются от платформы к платформе.

Ни один из компиляторов, которые мы использовали для выполнения данной программы, не предъявил никаких претензий к программному коду. Не было также никаких сообщений об ошибках в процессе выполнения программы. Некоторые компиля-

торы вполне могут обнаружить подобного рода ошибки, однако стандарт языка С от них этого не требует. В силу этого обстоятельства, компьютер может и не выявить таких ошибок, но поскольку во всех других отношениях программа ведет себя безупречно, вы, возможно, и сами не заметите этих ошибок. Если программа не печатает ожидаемое число значений или печатает неожиданные значения, проверьте, использовали ли вы нужное количество аргументов в вызове функции `printf()`.

(Между прочим, программа `lint`, осуществляющая проверку синтаксиса в системе Unix, которая намного привередливее, чем компилятор Unix, отметила наличие ошибочных аргументов функции `printf()`.)

Еще один пример: управляющие последовательности

В качестве примера выполним еще одну программу печати, в которой используются некоторые специальные управляющие последовательности символов языка С. В частности, программа, представленная в листинге 3.10, показывает, как работают возврат на одну позицию влево (`\b`), табуляция (`\t`) и возврат каретки (`\r`). Эти понятия возникли с тех времен, когда компьютеры использовали для вывода телетайпы, а сегодня они не всегда успешно приживаются в современных графических интерфейсах. Например, представленная ниже программа не работает так, как здесь описано, в некоторых реализациях языка С на компьютерах Macintosh.

Листинг 3.10. Программа `escape.c`

```
/* escape.c – использование символов управляющих последовательностей */
#include <stdio.h>
int main(void)
{
   float salary;
   printf("\aВведите предпочитаемую вами сумму месячного жалования:");/* 1
*/
   printf(" $_______\b\b\b\b\b\b\b"); /* 2 */
   scanf("%f", &salary);
   printf("\n\t$%.2f в месяц соответствует $%.2f в год.", salary,
      salary * 12.0); /* 3 */
   printf("\Ого!\n"); /* 4 */
   return 0;
}
```

Каким будет результат выполнения этой программы

Пройдемся по этой программе и посмотрим, как она будет работать в реализации языка С, соответствующей стандарту ANSI С. Первый оператор `printf()` (идет под номером 1) подает звуковой сигнал (вызванный последовательностью `\a`), а затем печатает следующую фразу:

```
Введите предпочитаемую вами сумму месячного жалования:
```

Поскольку в конце строки отсутствует последовательность \n, курсор устанавливается в позиции, следующей за двоеточием.

Второй оператор printf() продолжает печать с того места, где закончил работу первый, таким образом, после того, как он завершит свою работу, на экране будет отображена следующая фраза:

```
Введите желаемую вами сумму месячного жалования: $_______
```

Пробел между двоеточием и знаком доллара появился здесь в связи с тем, что строка во втором операторе начинается с пробела. Результатом семи символов возврата на одну позицию будет перемещение курсора на семь позиций влево. Курсор проходит через семь символов подчеркивания и располагается непосредственно после знака доллара. Как правило, возврат на одну позицию влево не приводит к удалению символа, через который проходит курсов, но в некоторых системах может быть применен деструктивный возврат на одну позицию (забой), и результат выполнения этой простой программы окажется другим.

В этом месте вы вводите с клавиатуры свой ответ, скажем 2000.00. Теперь строка принимает следующий вид:

```
Введите предпочитаемую вами сумму месячного жалования: $2000.00
```

Символы, которые вы вводите с клавиатуры, заменяют символы подчеркивания, после чего вы нажимаете клавишу <Enter> (или <Return>), чтобы закончить ввод вашего ответа, после чего курсор переместится в начало следующей строки.

Выходные данные третьего оператора printf() начинаются с последовательности \n\t. Символ перевода строки перемещает курсор в начало следующей строки. Символ табуляции перемещает курсор в следующую позицию табуляции в этой строке, обычно, но не обязательно, в столбец 9. Затем печатается остальная строка. По завершении выполнения этого оператора на экране появляются следующие данные:

```
Введите предпочитаемую вами сумму месячного жалования: $2000.00
        $2000.00 в месяц соответствует $24000.00 в год.
```

Поскольку оператор printf() не использует символ перевода строки, курсор остается в позиции непосредственно после завершающей точки.

Четвертый оператор printf() начинается с последовательности \r. Она помещает курсор в начало текущей строки. С этой позиции отображается фраза Ого!, а \n переводит курсор на следующую строку. В окончательном варианте выходные данные на экране имеют вид:

```
Введите предпочитаемую вами сумму месячного жалования: $2000.00
Ого!    $2000.00 в месяц равно $24000.00 в год.
```

Сброс буфера выходных данных

Когда фактически функция printf() отсылает выходные данные на экран? Первоначально операторы printf() пересылают выходные данные в промежуточную область хранения данных, называемую *буфером*. Время от времени материал, хранящийся в буфере, пересылается на экран. Правила стандартного C, определяющие, когда выходные данные пересылаются из буфера на экран, понятны: они пересылаются на экран, как только буфер будет заполнен, когда появляется символ новой строки или когда предполагается ввод данных. (Пересылка выходных данных из буфера на экран

или в файл называется *сбросом буфера*.) Например, два первых оператора не заполняют буфер и не содержат символа новой строки, но непосредственно за ними следует оператор `scanf()`, который запрашивает ввод. Это инициирует пересылку выходных данных оператора `printf()` на экран.

Вы можете столкнуться с более ранними реализациями, в которых оператор `scanf()` не вызывает очистки буфера, что, в свою очередь, приводит к тому, что программа начинает поиск ваших входных данных, не выводя на экран приглашения. В этом случае вы можете воспользоваться символом новой строки для очистки буфера. Программный код можно изменить следующим образом:

```
printf("Введите предпочитаемую вами сумму месячного жалования:\n");
scanf("%f", &salary);
```

Этот программный код работает независимо от того, производит ли предстоящий ввод данных очистку буфера или нет. Однако он устанавливает курсор на следующей строке, благодаря чему входные данные не будут размещаться в той же строке, в которой находится приглашение. Другим решением является использование функции `fflush()`, описанной в главе 13.

Ключевые понятия

В языке C реализовано удивительно большое количество числовых типов. Этот факт отражает намерение языка C не создавать дополнительных трудностей на пути программиста. Вместо постулирования, скажем, того факта, что одного целочисленного типа вполне достаточно, C пытает предоставить программисту возможность выбора типа (со знаком или без) и размера, которые наилучшим образом соответствуют потребностям конкретной программы.

Числа с плавающей запятой фундаментально отличаются от целых чисел на конкретном компьютере. Они хранятся и обрабатываются различными способами. Два 32-разрядных элемента памяти могут содержать один и тот же набор битов, но если один из них будет интерпретирован как значение с плавающей запятой, а другой как значение типа `long`, они будут представлять совершенно разные и абсолютно не связанные между собой значения. Например, на если персональном компьютере вы рассмотрите набор битов, представляющих число 256.0 с плавающей запятой и попытаетесь интерпретировать его как значение типа `long`, вы получите 113246208. Язык C позволяет вам писать выражения со смешанными типами данных, но при этом он выполняет автоматическое приведение типов с тем, чтобы фактические вычисления выполнялись над данными одного типа.

В памяти компьютера символы представлены числовыми кодами. Код ASCII получил наибольшее распространение в США, однако язык C поддерживает использование и других кодов. Символьная константа — это символьное представление числовых кодов, используемых в компьютерных системах — она состоит из символов, заключенных в одиночные кавычки, например, `'A'`.

Резюме

В языке C имеется большое разнообразие типов данных. Основные типы данных подразделяются на две категории: целочисленные типы данных и данные с плавающей запятой. Двумя отличительными особенностями целочисленных типов являются объем

памяти, выделяемый для размещения данных того или иного типа, и знак числа, то есть, со знаком данный тип или без знака. Наименьшим целочисленным типом является `char`, который, в зависимости от реализации, может быть со знаком или без знака. Вы можете использовать `signed char` и `unsigned char`, чтобы явно определить тот тип, какой вам нужен, однако обычно вы выбираете эти типы, когда работаете с небольшими целыми числами, а не с символьными кодами. К другим целочисленным типом относятся `short`, `int`, `long` и `long long`. Язык C гарантирует, что каждый из этих типов не меньше, чем предшествующий тип. Каждый из этих типов есть тип со знаком, но вы можете использовать ключевое слово `unsigned` для создания соответствующего типа без знака: `unsigned short`, `unsigned int`, `unsigned long` и `unsigned long long`. Либо вы можете добавить модификатор `signed` с тем, чтобы явно объявить, что данный тип является типом со знаком. Наконец, существует также тип `_Bool`, представляющий собой тип без знака, способный принимать значения `0` и `1`, которые представляют значения `false` и `true`.

Тремя типами с плавающей запятой являются `float`, `double` и новый для стандарта ANSI C тип `long double`. Каждый из них, по меньшей мере, не меньше предыдущего типа. При необходимости та или иная реализация может поддерживать комплексные и мнимые типа, используя с этой целью ключевые слова `_Complex` и `_Imaginary` в сочетании с ключевыми словам типа с плавающей запятой. Например, могут существовать типы `double _Complex` и тип `float _Imaginary`.

Целые числа могут быть представлены в десятичной, восьмеричной и шестнадцатеричной форме. Ведущий символ `0` показывает, что число представлено в восьмеричной форме, ведущие символы `0x` или `0X` указывают, что это шестнадцатеричное число. Например, `32`, `040` и `0x20` — это десятичное, восьмеричное и шестнадцатеричное представления одного и того же значения. Суффикс `l` или `L` указывает, что значение имеет тип `long`, а `ll` или `LL` — тип `long long`.

Символьные константы представляются путем заключения символа в одиночные кавычки, например `'Q'`, `'8'` и `'$'`. Управляющие последовательности в языке C, такие как `'\n'`, представляют определенные непечатаемые символы. Вы можете использовать форму `'\007'` для представления символа в коде ASCII.

Числа с плавающей запятой могут быть записаны в форме с фиксированной десятичной точкой, например, `9393.912` или в экспоненциальном представлении, например, `7.38E10`.

Функция `printf()` позволяет печатать различные типы значений, используя спецификаторы, которые в своей простейшей форме состоят из знака процента и буквы, указывающей тип, например, `%d` или `%f`.

Вопросы для самоконтроля

Ответы на эти вопросы находятся в приложении A.

1. Какие типы данных вы будете использовать для каждого из следующих типов данных?
   а. Население Парижа.
   б. Стоимость копии фильма на DVD-диске.
   в. Буква, которая чаще других встречается в данной главе.
   г. Количество раз, сколько эта буква встречается в данной главе.
2. Назовите причины, по каким вы будете применять тип `long` вместо типа `int`?

3. Какие переносимые типы вы можете использовать, чтобы получить 32-разрядное целое число со знаком? Приведите аргументы в пользу вашего выбора.
4. Определите тип и значение каждой из следующих констант:
   а. `'\b'`
   б. `1066`
   в. `99.44`
   г. `0XAA`
   д. `2.0e30`
5. Некто написал программу с ошибками. Помогите найти эти ошибки.

```
include <stdio.h>
main
(
   float g; h;
   float tax, rate;
   g = e21;
   tax = rate*g;
)
```

6. Определите тип данных (каким он используется в операторах объявления) и формат спецификатора функции `printf()` для каждой из следующих констант:

	Константа	*Тип*	*Спецификатор*
а.	`12`		
б.	`0X3`		
в.	`'C'`		
г.	`2.34E07`		
д.	`'\040'`		
е.	`7.0`		
ж.	`6L`		
з.	`6.0f`		

7. Определите тип данных (каким он используется в операторах объявления) и формат спецификатора функции `printf()` для каждой из следующих констант (предполагается, что используется 16-разрядный тип `int`):

	Константа	*Тип*	*Спецификатор*
а.	`12`		
б.	`2.9e05L`		
в.	`'s'`		
г.	`100000`		
д.	`'\n'`		
е.	`20.0f`		
ж.	`0x44`		

8. Предположим, что программа начинается со следующих объявлений:

```
int imate = 2;
long shot = 53456;
char grade = 'A';
float log = 2.71828;
```

Вставьте нужный тип спецификатора в следующие операторы printf():

```
printf("Ставка на %__ равна %__ к 1.\n", imate, shot);
printf("Рейтинг %__ не соответствует %__ позиции.\n", log, grade);
```

9. Предположим, что ch представляет собой переменную типа char. Покажите, как присвоить этой переменной символ возврата каретки, используя для этой цели соответствующую управляющую последовательность, десятичное значение, восьмеричную символьную константу и шестнадцатеричную символьную константу (предполагается использование значений в коде ASCII).

10. Исправьте приведенную ниже довольно-таки глупую программу. (Символом / в языке C обозначается операция деления.)

```
void main(int) / Эта программа не содержит ошибок /
{
   cows, legs integer;
   printf("Сколько коровьих ног вы насчитали?\n);
   scanf("%c", legs);
   cows = legs / 4;
   printf("Отсюда следует, что имеется %f коров.\n", cows)
}
```

11. Определите, что представляют собой каждая из следующих управляющих последовательностей:

а. \n
б. \\
в. \"
г. \t

Упражнения по программированию

1. Экспериментальным путем определите, как ваша система решает проблему переполнения при выполнении операций над целыми числами, над числами с плавающей запятой и проблему потери значимости при выполнении операций над числами с плавающей запятой, то есть, напишите программу, которая сталкивается такими проблемами.

2. Напишите программу, которая приглашает ввести некоторое значение в коде ASCII, например, 66, а затем печатает символ, которому соответствует введенный код.

3. Напишите программу, которая выдает предупредительный звуковой сигнал, а затем выводит на печать следующий текст:

Испуганная внезапно раздавшимся звуком, Салли воскликнула: "Что это было?"

4. Напишите программу, которая считывает некоторое число с плавающей запятой и печатает его сначала в десятичном представлении, а затем в экспоненциальном представлении. Выходные данные должны быть отображены в следующем формате (фактическое число отображаемых цифр зависит от конкретной вычислительной системы):

   ```
   Введено число 21.290000 или 2.129000e+001.
   ```

5. В году примерно 3.156×10^7 секунд. Напишите программу, которая приглашает ввести возраст в годах, а затем выводит на экран эквивалентное значение в секундах.
6. Масса одной молекулы воды приблизительно равна 3.0×10^{-23} грамм. Кварта воды весит примерно 950 грамм. Напишите программу, которая приглашает ввести некоторое значение объема воды в квартах и отображает количество молекул воды в этом объеме.
7. В дюйме 2.54 сантиметра. Напишите программу, которая приглашает вас ввести рост в дюймах, после чего выводит на экран этот рост в сантиметрах. Либо, если вам так больше нравится, запрашивает рост в сантиметрах и переводит его в дюймы.

ГЛАВА 4

Символьные строки и форматированный ввод–вывод

В этой главе:

- Функция: `strlen()`
- Ключевые слова: `const`
- Символьные строки
- Создание и хранение символьных строк
- Использование функций `printf()` и `scanf()` для чтения и отображения символьных строк
- Использование функции `strlen()` для измерения длины строки
- Директива `#define` препроцессора C и модификатор `const` стандарта ANSI C, используемые для создания символьных констант

Основное внимание в этой главе будет уделено вопросам ввода и вывода. Изучив данную главу, вы сможете наделить свои программы индивидуальными свойствами, сделав их интерактивными и воспользовавшись символьными строками. Кроме того, будут более детально рассмотрены такие функции ввода-вывода языка C, как `printf()` и `scanf()`. Эти две функции представляют собой программные инструменты, предназначенные для связи с пользователями и для форматирования выходных данных в соответствие с существующими потребностями и вкусами. Наконец, вы вкратце ознакомитесь с таким важным средством языка C, как препроцессор (процессор предварительной обработки), и узнаете, как следует определять и использовать символические константы.

Вводная программа

Скорее всего, сейчас, подобно тому, как в начале других глав, вы ожидаете анализа очередной простой учебной программы. В листинге 4.1 представлена именно такая программа, которая реализует диалог с пользователем. Чтобы внести некоторое разнообразие, в данной программе использован новый стиль комментариев, рекомендуемый стандартом C99.

Листинг 4.1. Программа `talkback.c`

```
// talkback.c — довольно настырная информативная программа
#include  <stdio.h>
#include  <string.h>      // для прототипа функции strlen()
#define DENSITY 62.4      // плотность человека в фунтах на кубический фут
int main()
{
   float weight, volume;
   int size, letters;
   char name[40];         // имя представляет собой массив из 40 символов
   printf("Здравствуйте! Как вас зовут?\n");
   scanf("%s", name);
   printf("%s, сколько вы весите в фунтах?\n", name);
   scanf("%f", &weight);
   size = sizeof name;
   letters = strlen(name);
   volume = weight / DENSITY;
   printf("Хорошо, %s, ваш объем составляет %2.2f кубических футов.\n",
          name, volume);
   printf("К тому же ваше имя состоит из %d символов,\n",
          letters);
   printf("и мы располагаем %d байтами для его сохранения.\n", size);
   return 0;
}
```

Запустив на выполнение программу `talkback`, получаем следующий результат:

```
Здравствуйте! Как вас зовут?
Шарла
Шарла, сколько вы весите в фунтах?
139
Хорошо, Шарла, ваш объем составляет 2.23 кубических футов.
К тому же ваше имя состоит из шести символов,
и мы располагаем 40 байтами для его сохранения.
```

Отметим основные особенности этой программы, с которыми вы, возможно, сталкиваетесь впервые:

- Для хранения символьных строк в программе используется *массив*, в котором хранится *символьная строка*. Имя пользователя считывается в массив, который в этом случае представляет собой набор из 40 последовательных байтов памяти, причем каждый из них способен запомнить одно символьное значение.
- Рассматриваемая программа использует *спецификацию преобразования* `%s` для обработки ввода и вывода строки. Обратите внимание, что переменная `name`, в отличие от переменной `weight`, не использует префикс `&` в тех случаях, когда она указывается в функции `scanf()`. (Далее вы увидите, что как значение `&weight`, так и значение `name` представляют собой адреса.)
- В программе используется препроцессор C, который дает возможность определить символьную константу `DENSITY` для представления значения `62.4`.
- Для вычисления длины строки в данной программе применяется функция `strlen()`.

Подход языка C к процедурам ввода-вывода может показаться несколько усложненным по сравнению, скажем, с языком Basic. Однако благодаря этой сложности достигается более совершенное управление вводом-выводом и более высокая эффективность программы. Как только вы привыкнете к этому, новые возможности покажутся просто удивительными.

Проведем исследование этих новых идей.

Строки символов: введение

Символьная строка представляет собой последовательность из одного или большего числа символов, например:

```
"Это длинная строка символов."
```

Двойные кавычки не являются частью строки. Они сообщают компилятору, что охватывают строку, в то время как одиночные кавычки идентифицируют конкретный символ.

Массив значений типа `char` и нулевой символ

В языке C нет никакого специального типа для представления строк. Вместо этого строки хранятся в массивах значений типа `char`. Символы, образующие строки, хранятся в смежных ячейках памяти, по одному символу в ячейке, массив состоит из смежных ячеек памяти, так что строка размещается в массиве естественным образом (рис. 4.1).

Рис. 4.1. *Строка в массиве*

Обратите внимание, что на рис. 4.1 в последней позиции массива показан символ `\0`. Это *нулевой* (*null*) *символ*, и язык C использует его для того, чтобы отметить конец строки. Нулевой символ — это не цифра нуль; это непечатаемый символ, значение которого в кодировке ASCII (или эквивалентной) равно `0`. Все строки языка C всегда сохраняются с этим символом в конце. То, что в конце массива всегда проставляется нулевой символ, означает, что массив должен иметь, по крайней мере, на одну ячейку больше, чем количество символов, которое вы хотите сохранить.

Что же представляет собой массив? Вы можете рассматривать массив как несколько ячеек памяти, расположенных в ряд. Если вы предпочитаете более формальную и точную формулировку, то массив — это упорядоченная последовательность элементов данных одного типа. В рассматриваемом примере создается массив из 40 ячеек памяти, или *элементов*, каждый из которых может хранить одно значение типа `char`. При этом используется следующее объявление:

```
char name[40];
```

Квадратные скобки после `name` говорят о том, что это массив. Число `40` внутри скобок показывает число элементов в массиве. Ключевое слово `char` идентифицирует тип каждого элемента (рис. 4.2).

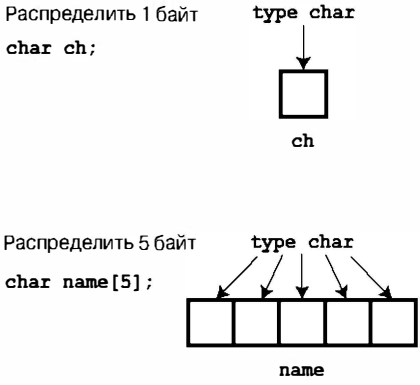


Рис. 4.2. *Объявление переменной и объявление массива*

Использование символьных строк теперь кажется очень сложным! Вам нужно создать массив, разместить символы строки друг за другом, к тому же не забыть добавить символ \0 в конец строки. К счастью, компьютер и сам может выполнить большинство этих деликатных действий.

Использование строк

На примере программы, представленной в листинге 4.2, можно убедиться, как просто на самом деле пользоваться строками.

Листинг 4.2. Программа `praise1.c`

```
/* praise1.c -- использует различные представления строк */
#include <stdio.h>
#define PRAISE "Какое прекрасное имя!"

int main(void)
{
   char name[40];

   printf("Как вас зовут?\n");
   scanf("%s", name);
   printf("Здравствуйте, %s. %s\n", name, PRAISE);

   return 0;
}
```

Спецификатор `%s` дает задание функции `printf()` распечатать строку. Спецификатор `%s` появляется дважды, поскольку программа печатает две строки: одна хранится в массиве `name`, а другая представлена константой `PRAISE`. Выполнение программы `praisel.c` дает на выходе следующий результат:

```
Как вас зовут?
Хилари Баббис
Здравствуйте, Хилари! Какое прекрасное имя!
```

Вам не нужно самому помещать нулевой символ в массив name. Эту задачу для вас выполняет функция scanf() при вводе. При этом нет необходимости добавлять нулевой символ в строковую константу PRAISE. Ниже мы рассмотрим действия оператора #define, а пока просто обратите внимание на то, что двойные кавычки, в которые заключается текст, следующий непосредственно за именем PRAISE, идентифицируют этот текст как строку. Компилятор сам позаботится о добавлении нулевого символа.

Обратите внимание (и это важно) на то обстоятельство, что функция scanf() читает только имя Хилари, а не Хилари Бабблс. После того, как функция scanf() начинает считывать входные данные, она останавливает чтение на первом же из встречных так называемых *"пробельных" символов*, то есть, символов пробела, табуляции и новой строки. Следовательно, рассматриваемая программа останавливает просмотр имени в тот момент, когда наталкивается на символ пробела между словами Хилари и Бабблс. Вообще говоря, функция scanf() используется со спецификатором %s только для чтения одиночных слов, а не целых фраз, таких как строка. В языке C доступны и другие функции, предназначенные для ввода данных, например, функция gets(), выполняющая обработку строк. В последующих главах мы подвергнем эти функции более глубоким исследованиям.

Различия между строками и символами

Строковая константа "x" — это отнюдь не то же самое, что и символьная константа 'x'. Одно из различий заключается в том, что 'x' принадлежит к базовому типу (char), а "x" — это производный тип, массив значений типа char. Второе различие состоит в том, что "x" на самом деле представляет собой строку, состоящую из двух символов, именно, из 'x' и '\0', как показано на рис. 4.3.

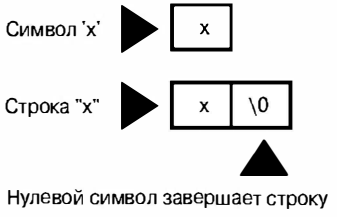


Рис. 4.3. *Символ 'x' и строка "x"*

Функция strlen()

В предыдущей главе кратко затрагивалась операция sizeof, которая определяет размеры объектов в байтах. Функция strlen() определяет длину строки в символах. Поскольку для размещения одного символа требуется один байт, можно предположить, что применительно к строке обе операции дадут одинаковый результат, но это не так. В качестве примера рассмотрим код в листинге 4.3.

Листинг 4.3. Программа praise2.c

```
/* praise2.c */
#include <stdio.h>
#include <string.h>          /* предоставляет прототип strlenf() */
#define PRAISE "Какое прекрасное имя!"
int main(void)
{
   char name[40];
   printf("Как вас зовут?\n");
   scanf("%s", name);
   printf("Здравствуйте, %s. %s\n", name, PRAISE);
```

```
    printf("Ваше имя состоит из %d символов и занимает %d ячеек памяти.\n",
         strlen(name), sizeof name);
    printf("Хвалебная фраза содержит %d символов",
         strlen(PRAISE));
    printf("и занимает %d ячеек памяти.\n",sizeof PRAISE);
    return 0;
}
```

Если вы используете версию компилятора, не поддерживающую ANSI C, вам потребуется убрать строку:

```
#include <string.h>
```

Заголовочный файл `string.h` содержит прототипы функций для нескольких функций обработки строк, в числе которых и `strlen()`. Этот файл мы еще будем рассматривать в главе 11. (Между прочим, некоторые системы UNIX, разработанные до появления стандарта ANSI, используют заголовочный файл `strings.h` вместо `string.h`, в котором содержатся объявления строковых функций.)

В общем случае C делит библиотеку функций на семейства логически связанных функций и предоставляет отдельный заголовочный файл для каждого семейства. Например, функции `printf()` и `scanf()` принадлежат семейству стандартных функций ввода-вывода и используют `stdio.h` в качестве своего заголовочного файла. Функция `strlen()` объединяет вокруг себя несколько других функций обработки строк, таких как функции копирования строк, поиск по строкам, и для этого семейства отводится заголовочный файл `string.h`.

Следует обратить внимание на то, что в листинге 4.3 используются два метода обработки длинных операторов `printf()`. Первый метод распространяет действие одного оператора `printf()` на две строки (можно разбить строку на два аргумента, но не в середине строки; то есть, не между кавычками). Второй метод подразумевает применение двух операторов `printf()`, чтобы печатать только одну строку. Символ новой строки (`\n`) присутствует только во втором операторе. В результате выполнения программы получается следующий обмен данными:

```
Как вас зовут?
Морган Баттеркап
Здравствуйте, Морган. Какое прекрасное имя!
Ваше имя состоит из шести букв и занимает 29 ячеек памяти.
Хвалебная фраза содержит 16 символов и занимает 21 ячеек памяти.
```

Посмотрите, что происходит. Массив `name` содержит 40 ячеек памяти, и операция `sizeof` сообщает об этом факте. Однако для размещения имени `Морган` необходимо только первых шесть ячеек, и об этом сообщает функция `strlen()`. Седьмая ячейка в массиве `name` содержит нулевой символ, и об этом также сообщает функция `strlen()` после завершения подсчета символов. Иллюстрацией этих идей служит рис. 4.4.

Когда вы приступаете к манипуляциям с `PRAISE`, вы обнаруживаете, что функция `strlen()` снова подсчитывает точное количество символов в строке (включая пробелы и знаки препинания). Операция `sizeof` выдает количество символов на единицу больше, так как она учитывает нулевой символ, проставленный в конце строки. Вы не задаете компьютеру, какой объем памяти нужно зарезервировать для размещения фразы. Он сам должен подсчитывать количество символов, заключенных в двойные кавычки.

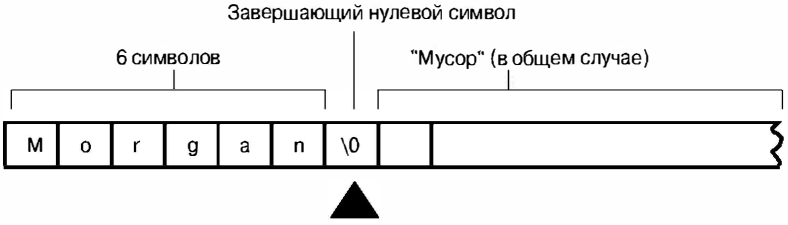


Рис. 4.4. *Функция* `strlen()` *знает, когда остановиться*

С другой стороны, в предыдущей главе операция `sizeof` употреблялась с круглыми скобками, а в этой главе мы их не видим. Используете ли вы круглые скобки или нет, зависит от того, хотите ли вы получить размер типа или конкретное числовое значение. Круглые скобки требуются для определения размера типов, но необязательны для подсчета размеров конкретных переменных. Иначе говоря, допускается конструкция `sizeof(char)` или `sizeof(float)`, но также разрешается `sizeof name` или `sizeof 6.28`. Тем не менее, в этих случаях правильным будет и применение круглых скобок, например, `sizeof (6.28)`.

В последнем примере функции `strlen()` и `sizeof` использовались с довольно тривиальной целью, а именно, с целью удовлетворить возможное любопытство пользователей. На самом деле функции `strlen()` и `sizeof` — это важные инструментальные средства программирования. Например, функция `strlen()` весьма полезна во всех программах, работающих с символами и строками, как будет показано в главе 11.

Теперь перейдем к рассмотрению оператора `#define`.

Константы и препроцессор C

Иногда вам приходится использовать константы в программе. Например, длину окружности можно вычислить по формуле:

```
circumference = 3.14159 * diameter;
```

Здесь константа `3.14159` представляет собой общеизвестную константу π. Чтобы воспользоваться этой константой, просто введите ее фактическое значение, как в приведенном выше примере. Однако существуют серьезные основания для того, чтобы использовать вместо числа *символьную константу*. Другими словами, вы можете воспользоваться оператором, подобным показанному в приведенном ниже примере, и заставить компьютер подставлять фактическое значение позже:

```
circumference = pi * diameter;
```

В чем заключаются преимущества использования символической константы? Во-первых, имя является более информативным, нежели число. Сравните два следующих утверждения:

```
owed = 0.015 * housevalue;
// задолженность = 0.015 * рыночная-стоимость-дома;
owed = taxrate * housevalue;
// задолженность = ставка-налогового-обложения * рыночная-стоимость-дома;
```

Если вы читаете длинную программу, то вторая версия лучше воспринимается, чем первая.

Кроме того, предположим, что вы используете константу в нескольких местах программы, и возникает необходимость изменить ее значение. В конце концов, даже налоговые отчисления иногда меняются. В таком случае достаточно всего лишь изменить определение символической константы, а не искать и менять соответствующим образом значение этой константы в разных местах программы.

Все это так, но как установить символьную константу? Один из способов заключается в том, чтобы объявить переменную и присвоить ей значение, равное требуемой константе. Вы можете сделать это следующим образом:

```
float taxrate;
taxrate = 0.015;
```

В этом случае мы имеем символьное имя, но при этом `taxrate` остается переменной, и ваша программа может случайно изменить ее значение. К счастью, в языке C доступны и более удачные реализации.

Намного более оригинальная идея предусматривает использование препроцессора C. В главе 2 вы уже видели, как препроцессор использует директиву `#include` для включения информации из другого файла. Наряду с этим препроцессор позволяет определять константы. Просто добавьте в верхнюю часть файла, содержащего вашу программу, строку, подобную следующей:

```
#define TAXRATE 0.015
```

После компиляции вашей программы значение `0.015` будет подставлено повсюду, где вы указываете `TAXRATE`. Это называется *подстановкой во время компиляции*. К моменту выполнения программы все подстановки уже выполнены (рис. 4.5). Константы, определенные таким образом, часто называются *символическими константами* или *литералами*.

Обратите внимание на формат. Сначала идет директива `#define`. За ней следует символическое имя (`TAXRATE`) константы, после чего идет значение (`0.015`) константы. Обратите внимание на то, что рассматриваемая конструкция не использует знак =. Обобщенный формат выглядит следующим образом:

```
#define NAME value
```

Вы можете заменить `NAME` на символическое имя, выбранное по собственному усмотрению, и указать соответствующее значение для `value`. Точка с запятой в этом случае не используется, поскольку это механизм замены, а не оператор языка C. Почему имя `TAXRATE` представлено заглавными буквами? Представление имен констант в символах верхнего регистра — это давно сложившаяся традиция в языке C. Если вы встречаете такое имя в недрах программы, вы сразу же осознаете, что перед вами константа, а не переменная. Представление имен констант заглавными буквами — еще один способ повысить удобочитаемость программы. Разумеется, ваши программы будут работать, даже если вы не воспользуетесь заглавными буквами для представления констант, но лучше все-таки взять этот прием на вооружение.

Другой, менее распространенный способ выделения имен констант заключается в использовании префиксов `c_` или `k_` для выделения констант, при этом возникают имена наподобие `c_level` или `k_line`.

Имена, выбираемые для символических констант, должны удовлетворять тем же правилам, что и имена переменных. Вы можете использовать символы верхнего и нижнего регистров, цифры и знак подчеркивания. Первый символ не может быть цифрой. В листинге 4.4 показан простой пример.

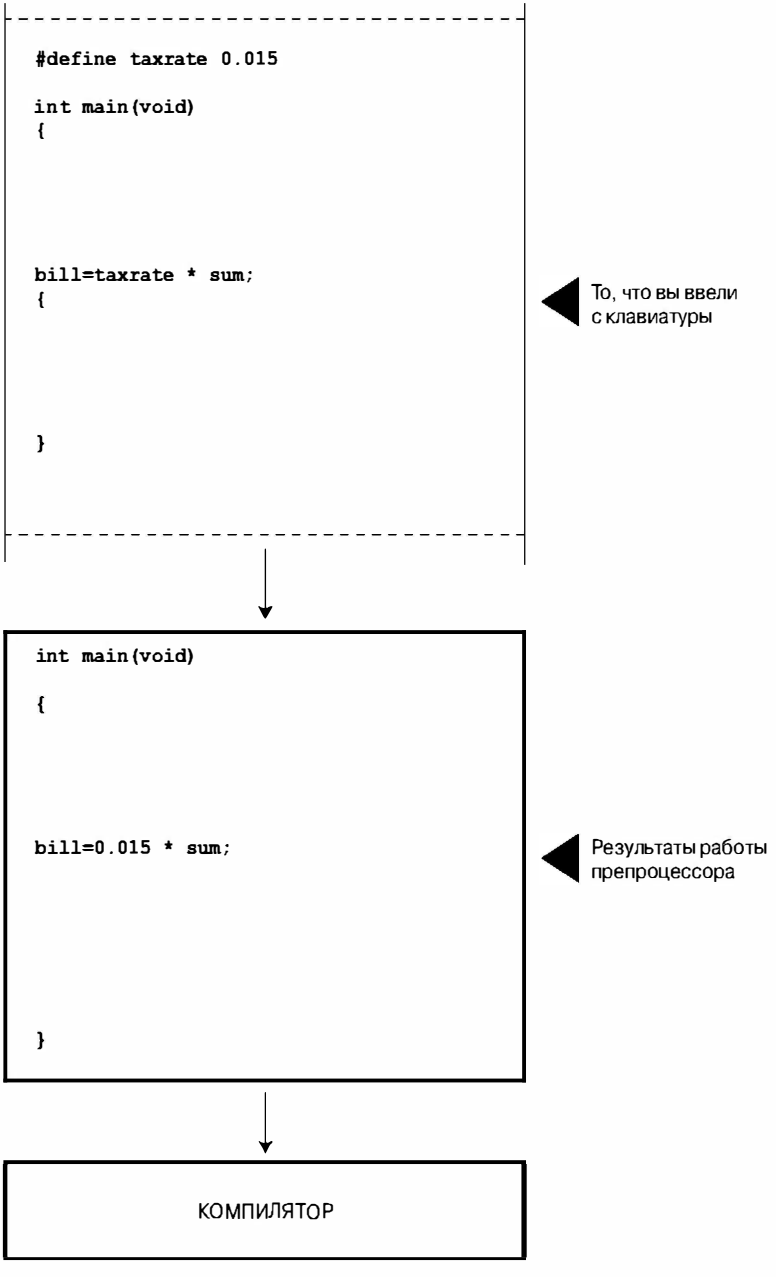


Рис. 4.5. *То, что вы ввели с клавиатуры, и то, что было откомпилировано*

Листинг 4.4. Программа `pizza.c`

```
/* pizza.c -- использует константы, определенные в контексте пиццы */
#include <stdio.h>
#define PI 3.14159
int main(void)
{
   float area, circum, radius;
   printf("Каков радиус вашей пиццы?\n");
   scanf("%f", &radius);
   area = PI * radius * radius;
   circum = 2.0 * PI *radius;
   printf("Базовые параметры вашей пиццы:\n");
   printf("длина окружности = %1.2f, площадь = %1.2f\n", circum, area);
   return 0;
}
```

Спецификатор `%1.2f` в операторе `printf()` требует округления до двух десятичных позиций при выводе. Конечно, эта программа может не отражать основные параметры пиццы, вызывающие ваш интерес, но она заполняет небольшую нишу в мире программ, касающихся пиццы. Рассмотрим пример выполнения этой программы:

```
Каков радиус вашей пиццы?
6.0
Базовые параметры вашей пиццы:
длина окружности = 37.70, площадь = 113.10
```

Оператор `#define` также может применяться для объявления символических и строчных констант. С этой целью достаточно воспользоваться одиночными кавычками для первых и двойными кавычками для последних. Ниже приводятся примеры допустимого объявления констант:

```
#define   BEEP '\a'
#define   TEE 'T'
#define   ESC '\033'
#define   OOPS "Наконец-то вы сделали это!"
```

Следует еще раз подчеркнуть, что все то, что следует за символическим именем, замещает его. Не допускайте следующую распространенную ошибку:

```
/* следующее объявление некорректно */
#define TOES = 20
```

Если вы сделали это объявление, константа `TOES` примет значение `= 20`, а не просто `20`. В этом случае такой оператор, как

```
digits = fingers + TOES;
```

получит неправильное представление:

```
digits = fingers  + = 20;
```

Модификатор const

Стандарт C90 обеспечивает еще один способ создания символических констант, который предусматривает использование ключевого слова const для преобразования объявления переменной в объявление константы:

```
const int MONTHS = 12; // MONTHS - символическая константа со значением 12
```

Благодаря такому объявлению константа MONTHS становится значением только для чтения. Иначе говоря, вы можете вывести на экран значение MONTHS и задействовать его в вычислениях, но вы не можете изменять значение MONTHS. Этот новый подход более гибок, чем использование конструкции #define; в главе 13 обсуждается этот, а также и другие способы использования констант.

На самом деле в языке C имеется еще и третий способ создания символических констант — использование средства enum, которое рассматривается в главе 14.

Работа с символическими константами

Заголовочные файлы limits.h и float.h содержат подробную информацию относительно ограничений размеров, соответственно, для целочисленных типов и типов с плавающей запятой. Каждый файл определяет ту или иную последовательность символических констант, которые вы можете использовать в своих программах. Например, файл limits.h содержит строки, подобные следующим:

```
#define INT_MAX   +32767
#define INT_MIN   -32768
```

Эти константы представляют максимально и минимально возможные значения типа int. Если ваша система использовала 32-разрядное значение int, этот файл обеспечивает различные значения таких символических констант. Этот файл определяет минимальные и максимальные значения для всех целочисленных типов. Если вы включаете файл limits.h в программу, то можете пользоваться следующим кодом:

```
printf("Максимальное значение типа int для данной системы = %d\n", INT_MAX);
```

Если ваша система использовала четырехбайтное значение int, то файл limits.h, который поставляется в комплекте с этой системой, предоставляет определения значений INT_MAX и INT_MIN, соответствующих пределам четырехбайтного типа int. В табл. 4.1 приводится список некоторых констант, определения которых содержатся в файле limits.h.

Подобным образом в файле float.h определяются константы типа FLT_DIG и DBL_DIG, которые представляют собой количество значащих чисел, обеспечиваемых типами float и double. В табл. 4.2 перечислены некоторые константы, которые можно найти в файле float.h. (Вы можете воспользоваться текстовым редактором, чтобы открыть и ознакомиться с содержимым заголовочного файла float.h, который присутствует в вашей системе.) Рассматриваемый пример относится к типу float. Эквивалентные константы определены для типов double и long double, при этом DBL и LDBL заменены в имени на FLT.

(Таблица предполагает, что в системе числа с плавающей запятой представлены в терминах степени 2.)

Таблица 4.1. Некоторые символические константы, определения которых содержатся в файле `limits.h`

Символическая константа	*Что представляет*
`CHAR_BIT`	Количество битов в типе `char`.
`CHAR_MAX`	Максимальное значение типа `char`.
`CHAR_MIN`	Минимальное значение типа `char`.
`SCHAR_MAX`	Максимальное значение типа `signed char`.
`SCHAR_MIN`	Минимальное значение типа `signed char`.
`UCHAR_MAX`	Максимальное значение типа `unsigned char`.
`SHRT_MAX`	Максимальное значение типа `short`.
`SHRT_MIN`	Минимальное значение типа `short`.
`USHRT_MAX`	Максимальное значение типа `unsigned short`.
`INT_MAX`	Максимальное значение типа `int`.
`INT_MIN`	Минимальное значение типа `int`.
`UINT_MAX`	Максимальное значение типа `unsigned int`.
`LONG_MAX`	Максимальное значение типа `long`.
`LONG_MlN`	Минимальное значение типа `long`.
`ULONG_MAX`	Максимальное значение типа `unsigned long`.
`LLONG_MAX`	Максимальное значение типа `long long`.
`LLONG_MIN`	Минимальное значение типа `long long`.
`ULLONG_MAX`	Максимальное значение типа `unsigned long long`.

Таблица 4.2. Некоторые символические константы из файла `limits.h`

Символическая константа	*Что представляет*
`FLT_MANT_DIG`	Число разрядов в мантиссе типа `float`.
`FLT_DIG`	Минимальное число значащих десятичных цифр типа `float`.
`FLT_MIN_10_EXP`	Минимальное значение отрицательного десятичного порядка числа для типа `float` с полным набором значащих цифр.
`FLT_MAX_10_EXP`	Максимальное значение положительного десятичного порядка числа для типа `float`.
`FLT_MIN`	Минимальное значение для положительного числа типа `float`, сохраняющее полную точность.
`FLT_MAX`	Максимальное значение для положительного числа типа `float`.
`FLT_EPSILON`	Различие между `1.00` и наименьшим значением `float`, превышающим `1.00`.

Листинг 4.5 служит иллюстрацией использования данных из заголовочных файлов float.h и limits.h. (Обратите внимание, что многие современные компиляторы поддерживают стандарт C99 еще не в полной мере, поэтому могут не распознать идентификатор LONG_MIN.)

Листинг 4.5. Программа defines.c

```
// defines.c -- использует определенные константы из файла limit.h и тип float.
#include <stdio.h>
#include <limits.h>           // целочисленные пределы
#include <float.h>            // пределы для чисел с плавающей запятой
int main(void)
{
   printf("Некоторые пределы для данной системы:\n");
   printf("Наибольшее значение типа int: %d\n", INT_MAX);
   printf("Наименьшее значение типа long long: %lld\n", LLONG_MIN);
   printf("Один байт = %d разрядов в данной системе.\n", CHAR_BIT);
   printf("Наибольшее значение типа double: %e\n", DBL_MAX);
   printf("Наименьшее нормальное значение типа float: %e\n", FLT_MIN);
   printf("Точность значений типа float = %d знаков \n", FLT_DIG);
   printf("Различие между 1.00 и наименьшим значением float, превышающим
1.00 = %e\n", FLT_EPSILON);
return 0;
}
```

Вот как выглядят выходные данные программы defines.c:

```
Некоторые пределы для данной системы:
Наибольшее значение типа int: 2147483647
Наименьшее значение типа long long: -9223372036854775808
Один байт = 8 разрядов в данной системе.
Наибольшее значение типа double: 1,797693e+308
Наименьшее нормальное значение типа float: 1,175494e-38
Точность значений типа float = 6 знаков
Различие между 1.00 и наименьшим значением float, превышающим 1.00 =
1.192093e-07
```

Препроцессор C — очень полезное и удобное инструментальное средство, поэтому применяйте его везде, где возможно. В этой книге вы ознакомитесь с другими случаями его использования.

Исследование и использование функций printf() и scanf()

Функции printf() и scanf() обеспечивают возможность общения пользователя с программой. Они называются функциями *ввода-вывода*. Эти функции не единственные, которые вы можете использовать для этих целей в языке C, в то же время они обладают множеством дополнительных возможностей. Исторически эти функции, подобно другим функциям в библиотеке C, *не были* частью определения языка C. Первоначально C предоставлял реализацию процедур ввода-вывода разработчикам компилятора;

такой подход давал возможность учитывать особенности конкретных машин при выполнении операций ввода-вывода. В интересах сохранения совместимости различные реализации выполнялись с включенными в них функциями `scanf()` и `printf()`. Однако между версиями встречались и случайные несоответствия. Стандарты C90 и C99 описывает стандартные версии этих функций, и мы будем следовать требованиям этих стандартов.

Несмотря на то что `printf()` — это функция вывода, а `scanf()` — функция ввода, в их работе много общего; так, например, каждая из них использует управляющую строку и список аргументов. Далее мы покажем, как они работают, сначала рассмотрим функцию `printf()`, а затем и функцию `scanf()`.

Функция `printf()`

Инструкции, которые вы даете `printf()`, обращаясь к ней с требованием печати переменной, зависят от типа этой переменной. Например, ранее мы использовали форму записи `%d` при печати целого числа и `%c` — при печати символа. Эти указания называются *спецификациями преобразования*, поскольку они определяют, каким образом преобразуются данные в форму, пригодную для вывода. Мы приведем список спецификаций преобразования, которые стандарт ANSI C предусматривает для функции `printf()`, и затем покажем, как следует использовать наиболее употребляемые из них. В табл. 4.3 перечислены спецификаторы преобразования и типы вывода, который они обеспечивают.

Таблица 4.3. Спецификаторы преобразования и результат вывода на печать

Символическая константа	*Выходные данные*
`%a`	Число с плавающей запятой, шестнадцатеричные цифры и p-представление (стандарт C99).
`%A`	Число с плавающей запятой, шестнадцатеричные цифры и P-представление (стандарт C99).
`%c`	Одиночный символ.
`%d`	Десятичное целое число со знаком.
`%e`	Число с плавающей запятой, экспоненциальное представление (e-представление).
`%E`	Число с плавающей запятой, E-представление.
`%f`	Число с плавающей запятой, десятичное представление.
`%g`	Используется спецификатор `%f` или `%e`, в зависимости от значения. Стиль `%e` используется, если показатель степени меньше –4 либо больше или равен заданной точности.
`%G`	Используется спецификатор `%f` или `%e`, в зависимости от значения. Стиль `%e` используется, если показатель степени меньше –4 либо больше или равен заданной точности.
`%i`	Десятичное целое число со знаком (то же, что и спецификатор `%d`).

Окончание табл. 4.3

Символическая константа	*Выходные данные*
`%o`	Восьмеричное целое число без знака.
`%p`	Указатель.
`%s`	Строка символов.
`%u`	Десятичное целое число без знака.
`%x`	Шестнадцатеричное целое число без знака, используются шестнадцатеричные цифры `0f`.
`%X`	Шестнадцатеричное целое число без знака, используются шестнадцатеричные цифры `0f`.
`%%`	Печать знака процента.

Использование функции `printf()`

В листинге 4.6 представлена программа, в которой используются некоторые рассмотренные выше спецификаторы.

Листинг 4.6. Программа `printout.c`

```
/* printout.c -- использует спецификаторы преобразования */
#include  <stdio.h>
#define PI 3.141593
int main(void)
{
   int number = 5;
   float espresso = 13.5;
   int cost = 3100;
   printf("%d администраторов выпили %f чашек кофе эспрессо.\n", number,
         espresso);
   printf("Значение pi равно %f.\n", PI);
   printf("До свидания! Ваше искусство слишком дорого мне обходится,\n");
   printf("%c%d\n", '$', 2 * cost);
   return 0;
}
```

В результате выполнения этой программы были получены следующие выходные данные:

```
5 администраторов выпили 13.500000 чашек эспрессо.
Значение pi равно 3.141593.
До свидания! Ваше искусство слишком дорого мне обходится,
$6200
```

Формат использования функции `printf()` имеет вид:

```
printf(управляющая-строка, элемент1, элемент2, ...);
```

Где *элемент1*, *элемент2* и так далее — это элементы, которые нужно напечатать. Они могут быть переменными либо константами, либо даже выражениями, значение которых вычисляется при выводе на печать, а *управляющая-строка* — это строка символов, описывающая, каким образом необходимо выполнить печать элементов. Как уже говорилось в главе 3, управляющая строка должна содержать спецификатор преобразования для каждого выводимого элемента. Например, рассмотрим следующий оператор:

```
printf("%d администраторов выпили %f чашек кофе эспрессо.\n", number,
       espresso);
```

Здесь *управляющая-строка* представляет собой фразу, заключенную в двойные кавычки. Она содержит два спецификатора преобразования, соответственно, для переменных `number` и `espresso` — двух выводимых на печать элементов. На рис. 4.6 показан другой пример применения оператора `printf()`.

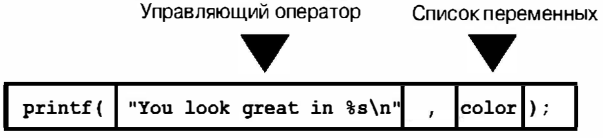


Рис. 4.6. *Аргументы функции* `printf()`

Вот еще одна строка из того же примера:

```
printf("Значение pi равно %f.\n", PI);
```

В этот момент времени список состоит только из одного элемента — символической константы `PI`. Как вы можете видеть на рис. 4.7, *управляющая-строка* содержит два различных вида информации:

- Символы, которые печатаются на самом деле.
- Спецификаторы преобразования.

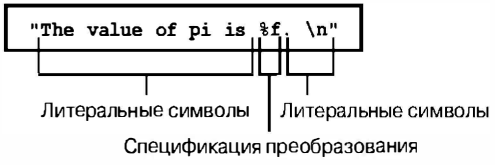


Рис. 4.7. *Анатомия управляющей строки*

Внимание!

Не забудьте, что необходимо предусмотреть по одной спецификации преобразования для каждого элемента в приведенной ниже *управляющей-строке*. Будет очень печально, если вы не выполните это основное требование! Никогда не поступайте так:

```
printf("Выпало Squids %d из %d.\n", score1);
```

Отсутствует значение для второго спецификатора `%d`. То, к чему приведет подобная небрежность, зависит от вашей системы, но в лучшем случае вы получите на экране абракадабру.

Если необходимо вывести на печать только какую-то фразу, вам не нужны никакие спецификации преобразования. Если вы хотите распечатать только данные, вы можете обойтись текущим комментарием. Для этого вполне достаточно одного из двух следующих операторов из листинга 4.6:

```
printf("До свидания! Ваше искусство слишком дорого мне обходится,\n");
printf("%c%d\n", '$', 2 * cost);
```

Обратите внимание на второй оператор, в котором первый элемент в списке для печати представляет собой константу, а не переменную, в то же время второй элемент — операция умножения. Это иллюстрирует тот факт, что функция `printf()` использует значения независимо от того, представляют ли они собой переменные, константы или выражения.

Поскольку функция `printf()` использует символ `%` для идентификации спецификации преобразования, то возникает небольшая проблема, когда вы захотите распечатать сам символ `%`. Если вы просто укажете одиночный знак `%`, компилятор подумает, что вы неверно задали спецификацию преобразования. Выход из этой ситуации достаточно прост: в таких случаях вы должны воспользоваться двумя символами `%`, как показано ниже:

```
pc = 2*6;
printf("Только %d%% припасов Мэри были съедобными.\n", pc);
```

В результате выполнения этого фрагмента программы получим следующий результат:

```
Только 12% припасов Мэри были съедобными.
```

Модификаторы спецификации преобразования для функции `printf()`

Вы можете изменить базовую спецификацию преобразования, вставляя модификаторы между знаком `%` и символом, определяющим преобразования. В таблицах 4.4 и 4.5 приводятся списки символов, которые можно размещать в этих местах. Если вы используете более одного модификатора, они должны располагаться в том же порядке, в каком они представлены в табл. 4.4. Не все комбинации допустимы. Таблицы отражают стандарт C99; используемая вами реализация, возможно, еще не поддерживает все представленные в этих таблицах варианты.

Таблица 4.4. Модификаторы функции `printf()`

Модификатор	*Значение*
флаг	Пять флагов (`-`, `+`, пробел, `#` и `0`) описаны в табл. 4.5. В функции могут быть представлены несколько флагов или же вообще ни одного.
	Пример: `"%-10d"`
цифра (ы)	Минимальная ширина поля. Если выводимое на печать число или строка не помещаются таком поле, используется поле большей ширины.
	Пример: `"%4d"`

Окончание табл. 4.4

Модификатор	*Значение*
.*цифра (ы)*	Точность. Для преобразований `%e`, `%E` и `%f` указывается количество цифр, которые будут распечатаны справа от десятичного числа. Для преобразований `%g` и `%G` задается максимальное количество значащих цифр. Для преобразования `%s` определяется максимальное количество символов, которое может быть распечатано. Для целочисленных преобразований указывается минимальное количество воспроизводимых при печати цифр, ведущие нули используются в случае необходимости установки соответствия с этим минимумом. Если используется только точка (`.`), то имеется в виду, что далее следует нуль, следовательно, `%.f` — то же, что и `%.0f`. Пример: `"%5.2f"` выводит значение типа `float` в поле шириной пять символов и двумя цифрами после десятичной точки.
`h`	Используется со спецификатором целочисленного преобразования для отображения значений типа `short int` или `unsigned short int`. Примеры: `"%hu"`, `"%hx"` и `"%6.4hd"`.
`hh`	Используется со спецификатором целочисленного преобразования для отображения значений типа `signed char` или `unsigned char`. Примеры: `"%hhu"`, `"%hhx"` и `"%6.4hhd"`.
`j`	Используется при целочисленном преобразовании со спецификатором для отображения значений `intmax_t` или `uintmax_t`. Примеры: `"%jd"` и `"%8jX"`.
`l`	Используется со спецификатором целочисленного преобразования для отображения значений типа `long int` или `unsigned long int`. Примеры: `"%ld"` и `"%8lu"`.
`ll`	Используется со спецификатором целочисленного преобразования для отображения значений типа `long long int` или `unsigned long long int` (стандарт C99). Примеры: `"%lld"` и `"%8llu"`.
`L`	Используется со спецификатором преобразовании значений с плавающей запятой для отображения значений типа `long double`. Примеры: `"%Lf"` и `"%10.4Le"`.
`t`	Используется со спецификатором целочисленного преобразования для отображения значений `ptrdiff_t`. Этот тип соответствует различию между двумя указателями (стандарт C99). Примеры: `"%td"` и `"%12ti"`.
`z`	Используется со спецификатором целочисленного преобразования для отображения значений `size_t`. Этот тип возвращается функцией `sizeof` (стандарт C99). Примеры: `"%zd"` и `"%12zx"`.

Преобразование аргументов типа `float`

Существуют спецификаторы преобразования для вывода на печать типов `double` и `long double`. В то же время такой спецификатор для типа `float` отсутствует. Причина состоит в том, что в классическом языке C или K&R C, значения типа `float` автоматически преобразовывались к типу `double` перед использованием в выражениях или в виде аргумента. В общем случае ANSI C не предусматривает автоматического преобразования `float` в `double`. Однако, для того, чтобы обеспечить правильную работу огромного количества существующих программ, которые разрабатывались с расчетом на то, что аргументы типа `float` могут быть преобразованы в тип `double`, все аргументы `float`, используемые функцией `printf()`, равно как и в другими функциями C, для которых не создаются явные прототипы, автоматически преобразуются к типу `double`. Поэтому в каждом из двух случаев, как в классическом K&R C, так и в ANSI C, нет специального спецификатора преобразования, который требуется для вывода на печать элементов типа `float`.

Таблица 4.5. Флаги функции `printf()`

Флаг	*Значение*
`-`	Элемент выравнивается влево, то есть содержимое будет напечатано, начиная с левого края. Пример: `"% -20s"`.
`+`	Значения со знаком печатаются со знаком +, если они положительные, и со знаком –, если отрицательные. Пример: `"% +6.2f"`.
Пробел	Значения со знаком печатаются с ведущим пробелом (но без знака), если они положительны, и со знаком –, если они отрицательные. Флаг + перекрывает пробел. Пример: `"% 6.2f"`.
`#`	Использует альтернативную форму для спецификации преобразования. Выводит ведущий `0` для формы `%o` и ведущий `0x` или `0X` для форм `%x` и `%X`. Для всех форм с плавающей запятой флаг # гарантирует, что символ десятичной точки будет напечатан, даже если за ним не следуют цифры. Для форм `%g` и `%G` это предотвращает удаление завершающих нулей. Примеры: `"%#o"`, `"%#8.0f"` и `"%+#10.3E"`.
`0`	Для числовых форм функция заполняет поле по всей ширине ведущими нулями вместо пробелов. Этот флаг игнорируется, если присутствует флаг - или если указана точность для целочисленной формы. Примеры: `"%010d"` и `"%08.3f"`.

Примеры использования модификаторов и флагов

Рассмотрим, как работают описанные выше модификаторы. Начнем с того, что проанализируем, как влияет модификатора ширины поля на отображение целого числа. Рассмотрим программу, показанную в листинге 4.7.

Листинг 4.7. Программа `width.c`

```
/* width.c -- ширина поля */
#include <stdio.h>
#define PAGES 931
int main(void)
{
   printf("*%d*\n", PAGES);
   printf{"*%2d*\n", PAGES);
   printf("*%10d*\n", PAGES);
   printf("*%-10d*\n", PAGES);
   return 0;
}
```

Программа из листинга 4.7 выводит на печать одно и то же число четыре раза, но с применением четырех различных спецификаций преобразования. Звездочка (*) служит для уазания, где начинается и где заканчивается каждое поле. Выходные данные имеют следующий вид:

```
*931*
*931*
*          931*
*931          *
```

Первая спецификация преобразования представлена конструкцией `%d` без модификаторов. Она генерирует поле с той же шириной, какую имеет распечатываемое целое число. Этот вариант принимается по умолчанию, то есть число будет напечатано именно в таком виде, если не представлены дальнейшие инструкции. Вторая спецификация преобразования представлена конструкцией `%2d`. Она устанавливает ширину поля равной 2, но, поскольку в рассматриваемом примере целое число имеет три значащих цифры, поле расширяется автоматически, чтобы уместить это число. Следующая спецификация преобразования — `%10d`. Она генерирует поле шириной 10 пробелов, при этом мы имеем семь пробелов и три цифры между звездочками с числом, а само число смещено в правый конец поля. Последней спецификацией является `%-10d`. Она также генерирует поле шириной в 10 символов, а знак - означает, что число помещается с левого края, в соответствии со сказанным выше. После того, как вы приобретете необходимые навыки работы, вы поймете, насколько удобна такая система. Наряду с этим она обеспечивает высокую степень контроля над внешним видом вашего вывода. Попробуйте изменить значение `PAGES`, чтобы посмотреть, как печатаются числа с различным количеством цифр.

Теперь рассмотрим некоторые форматы чисел с плавающей запятой. Наберите, откомпилируйте и выполните программу, представленную на листинге 4.8.

Листинг 4.8. Программа `floats.c`

```
/* Программа floats.c -- некоторые комбинации типов с плавающей запятой
#include <stdio.h>
int main(void)
{
   const double RENT = 3852.99;              // константа в стиле const
   printf("*%f*\n", RENT);
```

```
    printf("*%e*\n", RENT);
    printf("*%4.2f*\n", RENT);
    printf("*%3.1f*\n", RENT);
    printf("*%10.3f*\n", RENT);
    printf("*%10.3e*\n", RENT);
    printf("*%+4.2f*\n", RENT);
    printf("*%010.2f*\n", RENT);
    return 0;
}
```

На сей раз программа использует ключевое слово `const`, чтобы создать символическую константу. Получается следующий вывод:

```
*3852.990000*
*3.852990e+03*
*3852.99*
*3853.0*
*  3852.990*
* 3.853e+03*
*+3852.99*
*0003852.99*
```

Начнем с варианта, заданного по умолчанию, а именно — с `%f`. В этом случае имеется два значения, принятые по умолчанию: ширина поля и количество цифр справа от десятичной точки. Второе значение, принятое по умолчанию, — это шесть цифр и ширина поля, необходимая для того, чтобы вместить число.

Далее в программе идет значение, принятое по умолчанию для `%e`. В условиях этого варианта печатается одна цифра слева от десятичной точки и резервируются шесть позиций справа от нее. Получается довольно-таки много цифр! Чтобы исправить это положение, нужно задать количество десятичных позиций справа от десятичной точки, и следующие четыре примера в этом разделе служат иллюстрацией этого решения. Обратите внимание на то, как в четвертом и шестом примере выполняется округление выходных данных.

Наконец, флаг + означает, что числовые данные будут выводиться вместе с их алгебраическими знаками, которым в данном случае является знак +, а флаг `0` приводит к заполнению ведущими нулями на всю ширину поля. Обратите внимание, что в спецификаторе `%010` первый `0` — это флаг, а остальные цифры (`10`) — ширина поля.

Можете изменить значение `RENT`, чтобы посмотреть, как печатаются значения, для которых задаются различные размеры поля. Программа, показанная в листинге 4.9, демонстрирует еще несколько возможных комбинаций.

Листинг 4.9. Программа `flags.c`

```
/* flags.c -- иллюстрация применения некоторых форматирующих флагов */
#include <stdio.h>
int main(void)
{
    printf("%x %X %#x\n", 31, 31, 31);
    printf("**%d**% d**% d**\n", 42, 42, -42);
    printf("**%5d**%5.3d**%05d**%05.3d**\n", 6, 6, 6, 6);
    return 0;
}
```

Получаем следующий вывод:

```
1f 1F 0x1f
**42** 42**-42**
**    6**  006**00006**   006**
```

Прежде всего примем к сведению, что 1f представляет собой шестнадцатеричный эквивалент десятичного числа 31. Спецификатор x обеспечивает формат 1f, а спецификатор X — 1F. Использование флага # приводит к тому, что на выходе мы получим исходные данные в формате 0x.

Вторая строка выходных данных программы служит иллюстрацией того факта, что использование пробела в спецификаторе приводит к появлению ведущего пробела для положительных, но не для отрицательных значений. Это обстоятельство можно использовать выравнивания выходных данных, поскольку положительные и отрицательные значения с одинаковым количеством значащих цифр будут напечатаны в полях одинаковой ширины.

Третья строка служит иллюстрацией того факта, что использование спецификатора точности (%5.3d) в целочисленной форме приводит к печати такого количества ведущих нулей, которое достаточно для представления числа минимальным количеством цифр (три в рассматриваемом случае). Однако применение флага 0 приводит к наполнению представления числа ведущими нулями, количество которых достаточно для заполнения числом всей ширины поля. Наконец, если вы используете вместе флаг 0 и спецификатор точности, то флаг 0 игнорируется.

Теперь исследуем некоторые из параметров строки. Рассмотрим пример, показанный в листинге 4.10.

Листинг 4.10. Программа `strings.c`

```
/* strings.c -- форматирование строки */
#include <stdio. h>
#define BLURB "Authentic imitation!"
int main(void)
{
   printf{"/%2s/\n", BLURB);
   printf("/%24s/\n", BLURB);
   printft"/%24.5s/\n", BLURB);
   printf ("/%-24.5s/\n", BLURB);
   return 0;
}
```

Выходные данный этой программы имеют вид:

```
/Authentic imitation!/
/    Authentic imitation!/
/                   Authe/
/Authe                   /
```

Обратите внимание, что поле расширено настолько, чтобы вместить все описанные символы. Обратите также внимание на то, как спецификатор точности ограничивает выводимое количество символов. Запись .5 в спецификаторе формата сообщает функции printf() о том, что нужно выводить только пять символов. И снова напомним, что модификатор - выравнивает текст по левому краю.

Применение полученных знаний на практике

Вы только что просмотрели несколько примеров. Каким должен быть оператор вывода на печать текста в следующей форме:

```
Семья NAME может стать богаче на XXX.XX долларов!
```

Здесь NAME и XXX.XX представляют собой значения, для которых в программе будут зарезервированы переменные, скажем, name[40] и cash.

Предлагаем одно из возможных решений:

```
printf ("Семья %s может стать богаче на %.2f долларов!\n",name,cash);
```

Что преобразует спецификация преобразования

Теперь более подробно рассмотрим, что именно преобразует спецификация преобразования. Она преобразует значение, представленное в компьютере в некотором двоичном формате, в последовательность символов (строка), предназначенных для отображения. Например, число 76 может быть представлено в компьютере как двоичное число 01001100. Спецификатор преобразования %d превращает эту двоичную последовательность в символы 7 и 6, тем самым отображая число 76. Преобразование %x превращает ту же двоичную последовательность (01001100) в шестнадцатеричное представление 4c. А спецификатор %c приводит то же значение к символьному представлению L.

Термин *преобразование* иногда приводит к недоразумениям, поскольку может возникнуть неправильное предположение, будто исходное значение заменяется преобразованным значением. Преобразующие спецификации по существу являются спецификациями перевода; %d означает, что нужно "перевести данное значение в десятичное целочисленное текстовое представление и вывести это представление на печать".

Несовпадающие преобразования

Естественно, необходимо согласовать спецификацию преобразования с типом данных, выводимых на печать. Часто в этих случаях доступно несколько вариантов. Например, если вы хотите распечатать тип int, вы можете использовать либо спецификатор %d, либо %x, либо %o. Все эти спецификаторы предполагают, что вы выводите на печать значение типа int; сами же они просто обеспечивают различные представления этого значения. Точно так же вы можете использовать %f, %e или %g, чтобы представить тип double.

А что произойдет, если вы не согласовали спецификацию преобразования с типом? Вы уже видели в предыдущей главе, что такие несоответствия могут привести к возникновению проблем. Об этом никогда не следует забывать, поэтому программа, представленная в листинге 4.11, предлагает еще несколько примеров несоответствий при работе с семейством целочисленных типов.

Листинг 4.11. Программа intconv.c

```
/* intconv.c -- некоторые несогласованные преобразования целочисленных типов */
#include <stdio.h>
#define PAGES 336
#define WORDS 65618
```

```
int main(void)
{
   short num = PAGES;
   short mnum = -PAGES;
   printf("num как тип short и тип unsigned short: %hd %hu\n", num, num);
   printf("-num как тип short и тип unsigned short: %hd %hu\n", mnum, mnum);
   printf("num как тип int и тип char: %d %c\n", num, num);
   printf("WORDS как тип int, short и char: %d %hd %c \n", WORDS, WORDS, WORDS);
   return 0;
}
```

В нашей системе были получены следующие результаты:

```
num как тип short и тип unsigned short: 336 336
-num как тип short и тип unsigned short: -336 65200
num как тип int и тип char: 336 P
WORDS тип int, short и char: 65618 82 R
```

Посмотрев на первую строку, вы можете заметить, что спецификаторы `%hd` и `%hu` приводят к выводу числа `336` в качестве значения переменной `num`; тут нет никаких проблем. Вариант `ninum` со спецификатором `%u` (без знака) приводит к выводу значения `65200`, а не `336`, как можно было ожидать. Такой результат был получен вследствие способа представления значений типа `short int` со знаком, реализованного в вашей системе. Во-первых, они имеют размер 2 байта. Во-вторых, система использует метод, называемый *поразрядным дополнением до двойки* для представления целых чисел со знаком. В этом методе числа от 0 до 32767 представляют сами себя, а числа от 32768 до 65535 — отрицательные числа, причем 65535 представляет число –1, 65534 — число –2 и так далее. Поэтому `-336` представляется как `65536 - 336`, или `65200`. Итак, число 65200 представляет -336, когда интерпретируется как тип `signed int`, и 65200, когда интерпретируется как `unsigned int`. Будьте осторожны! Одно число может интерпретироваться как два различных значения. Не все системы используют этот метод для представления отрицательных целых чисел. Тем не менее, мы приходим к следующему выводу: не рассчитывайте на то, что в результате преобразования `%u` просто исчезнет знак у числа без каких-либо других последствий.

Вторая строка показывает, что может случиться, если вы пытаетесь преобразовать целое значение, превышающее 255, в символ. В данной системе тип `short int` занимает 2 байта, а тип `char` — 1 байт. Когда функция `printf()` печатает 336 с указанием спецификатора `%c`, она обращает внимание только на первый байт из двух байтов, хранящих число 336. Подобное усечение (рис. 4.8) равнозначно делению целого числа на 256 с сохранением остатка. В этом случае появляется остаток 80, который представляет собой ASCII-значение символа `P`. Выражаясь формально, число интерпретируется *по модулю 256*, что означает использование остатка от деления числа на 256.

В заключение мы попытались распечатать целое число (65618), превышающее максимально допустимое для типа `short int` (32767), реализованное в нашей системе. И снова компьютер применяет операции деления по модулю. Число 65616 в силу своего размера сохраняется в нашей системе как 4-байтовое значение `int`. Когда мы печатаем это число, используя спецификатор `%hd`, функция `printf()` использует только последние 2 байта. Это равносильно использованию остатка от деления этого числа на 65536. В этом случае получаем остаток, равный 82.

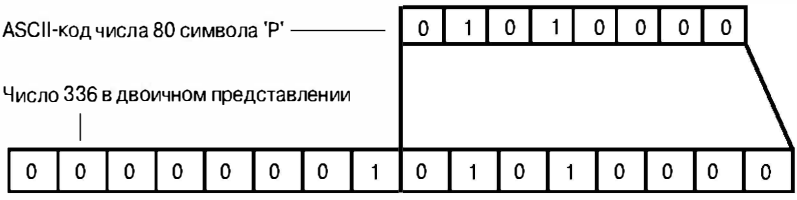


Рис. 4.8. *Интерпретация числа 336 как символа*

Остаток, находящийся между 32767 и 65536, был бы напечатан как отрицательное число в силу соответствующего способа хранения отрицательных чисел. Системы с другими размерами целых чисел придерживались бы той же линии поведения, но числовые значения были бы другими.

Когда вы начинаете смешивать значения, имеющие тип целого числа и тип числа с плавающей запятой, то результаты оказываются еще более причудливыми. Рассмотрим, например, код в листинге 4.12.

Листинг 4.12. Программа `floatcnv.c`

```
/* floatcnv.c -- несогласованные преобразования с плавающей запятой */
#include <stdio.h>
int main(void)
{
   float n1 = 3.0;
   double n2 = 3.0;
   long n3 = 2000000000;
   long n4 = 1234567890;

   printf("%.1e %. 1e %.1e %.1e\n", n1, n2, n3, n4);
   printf("%ld  %ld\n", n3, n4);
   printf("%ld  %ld  %ld   %ld\n", n1, n2, n3, n4);

   return 0;
}
```

В нашей системе код из листинга 4.12 сгенерировал следующий вывод:

```
3.0e+00 3.0e+00 3.1e+046 1.7e+266
2000000000 1234567890
0 1074266112 0 1074266112
```

Первая строка выходных данных этой программы показывает, что при использовании спецификатора `%e` не происходит преобразование целого числа в число с плавающей запятой. Посмотрим, например, что случается, если вы пробуете печатать значение `n3` (типа `long`) с указанием спецификатора `%e`. Во-первых, спецификатор `%e` настраивает функцию `printf()` на число типа `double`, которое является 8-байтовым значением в нашей системе. Когда функция `printf()` сталкивается с переменной `n3`, которая представлена 4-байтовым значением в нашей системе, эта функция рассматривает также смежный блок памяти длиной 4 байта. Таким образом, функция рассматривает 8-байтовый модуль, в который фактически вложено значение `n3`. Во-вторых, она интерпретирует биты этого модуля как число с плавающей запятой. Некоторые биты, например, интерпретируются как показатель степени.

Поэтому, даже если бы n3 имел правильное число битов, они бы интерпретировались по-разному при использовании спецификаторов %e и %ld. Итоговый результат является бессмысленным.

Первая строка также иллюстрирует то, о чем мы говорили выше: тип float преобразовывается к типу double, когда передается в качестве параметра функции printf(). В этой системе тип с плавающей запятой занимает 4 байта, в то же время тип переменной n1 был расширен до 8 байтов, чтобы функция printf() могла отображать правильный результат.

Вторая строка вывода показывает, что функция printf() может выводить значения n3 и n4 правильно, если использован корректный спецификатор.

Третья строка выходных данных показывает, что даже правильно выбранный спецификатор может приводить к неверным результатам, если функция printf() имеет несоответствия где-нибудь в другом месте. Как и следовало ожидать, попытка вывода на печать значения с плавающей запятой с использованием спецификатора %ld приводит к неудаче, однако и в данном случае попытка вывода на печать значения типа long с указанием спецификатора %ld не удается! Проблема связана с тем, как C передает информацию той или иной функции. Конкретные подробности этих неудач зависят от реализации; на врезке "Передача аргументов" эти вопросы будут рассмотрены более подробно.

Передача аргументов

Механизм передачи аргументов зависит от реализации. Вот как передача аргументов происходит в нашей системе. Вызов функции имеет следующий вид:

```
printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);
```

Показанный вызов сообщает компьютеру о том, что ему передаются значения переменных n1, n2, n3 и n4. Это делается путем помещения их в область памяти, называемую *стеком*. Когда компьютер помещает эти значения в стек, он руководствуется типами переменных, а не спецификаторами преобразования. Соответственно, для n1 он отводит 8 байтов стека (тип float преобразован в тип double). Точно так же он отводит еще 8 байтов для переменной n2, после чего выделяет по 4 байта для переменных n3 и n4. Затем управление передается функции printf(). Эта функция читает значения из стека, причем в соответствии со спецификаторами преобразования. Спецификатор %ld указывает, что функция printf() должна читать 4 байта, так что printf() читает первые 4 байта в стеке как свое первое значение. Это только первая половина значения n1, и она интерпретируется как целочисленное значение типа long. Следующий спецификатор %ld задает чтение еще 4 байтов; это только вторая половина значения n1 и она интерпретируется как второе целочисленное значение типа long (рис. 4.9). Точно так же третий и четвертый спецификаторы %ld приводят к тому, что первая и вторая половины значения n2 читаются и интерпретируются как очередные два числа типа long, таким образом, хотя мы и используем правильные спецификаторы для переменных n3 и n4, функция printf() читает не те байты, что надо.

```
float n1;  /* Передаются как тип double */
double n2;
long n3, n4;
...
printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);
```

Рис. 4.9. *Передача аргументов*

Возвращаемое значение функции `printf()`

Как было сказано в главе 2, функция в языке C в общем случае имеет возвращаемое значение — то, что она вычисляет и возвращает в вызывающую программу. Например, в библиотеке C доступна функция `sqrt()`, которая принимает число в качестве аргумента и возвращает его квадратный корень. Возвращаемое значение может быть присвоено некоторой переменной, может использоваться в вычислениях, может передаваться как аргумент, короче говоря, им можно манипулировать как любым другим значением. Функция `printf()` также имеет возвращаемое значение: в C она возвращает количество символов, выведенных на печать. Если произошла ошибка вывода, функция `printf()` возвращает то или иное отрицательное значение. (Некоторые версии `printf()` имеют другие возвращаемые значения.)

Возвращаемое значение для функции `printf()` не является ее главной целью, связанной с печатью выходных данных, и обычно не используется. Одна причина, по которой вы могли бы использовать возвращаемое значение, состоит в том, чтобы про-

верить наличие ошибки вывода. Контроль ошибок чаще осуществляется при записи в файл, чем при выводе на экран. Например, если запись на гибкий диск невозможна по причине нехватки места, можно было в этом случае запустить программу, которая выполнила бы какое-то действие, скажем, выдала звуковой сигнал в течение 30 секунд. Для начала вы должны научиться работать с условным оператором `if`. Простой пример, представленный в листинге 4.13, показывает, как можно работать с возвращаемым значением.

Листинг 4.13. Программа `prntval.c`

```
/* prntval.c -- обнаружение возвращаемого значения printf() */
#include <stdio.h>
int main(void)
{
   int bph2o = 212;
   int rv;
   rv  =  printf("%d градусов по Фаренгейту соответствуют точке кипения
воды.\n", bph2o);
   printf("Функция printf() выводит %d символов.\n",
           rv);
   return 0;
}
```

Выходные данные этой программы имеют вид:

```
212 градусов по Фаренгейту соответствуют точке кипения воды.
Функция printf() выводит 32 символов.
```

Сначала программа использовала форму `rv = printf (...);` для присвоения возвращаемого значения переменной `rv`. Этот оператор выполняет две задачи: выводит информацию и присваивает значение переменной. Во-вторых, обратите внимание, что итоговый результат включает все напечатанные символы, в том числе пробелы и невидимый символ новой строки.

Печать длинных строк

Иногда приходится сталкиваться с достаточно длинными операторами `printf()`, которые не умещаются в одной строке. Поскольку C игнорирует пробельные символы (пробелы, символы табуляции и символы новой строки), кроме случаев, когда они используются внутри строк, длинные операторы можно размещать в нескольких строках. Например, в листинге 4.13 оператор печати размещается в двух строках:

```
printf("Функция printf() печатает %d символов.\n",
        rv);
```

Строка разорвана между запятой и переменной `rv`. Чтобы показать читателю, что строка имеет продолжение, в этом примере `rv` представлена с отступом вправо. Язык C игнорирует избыточные пробелы.

В то же время вы не можете разрывать строку, заключенную в кавычки. Предположим, что вы пытаетесь сделать что-то в этом роде:

```
printf("Функция printf() печатает %d
         символов.\n", rv);
```

В этом случае компилятор сообщит о том, что вы используете недопустимый символ в строковой константе. Вы можете включить в строку символ \n, чтобы обозначить символ новой строки, но это не будет фактический символ новой строки, порожденный нажатием клавиши <Enter> (<Return>) посреди строки.

Если необходимо разбить строку на части, вы можете действовать тремя способами, как показано в листинге 4.14.

Листинг 4.14. Программа `longstrg.c`

```
/* longstrg.c -- печать длинных строк */
#include <stdio.h >
int  main(void)
{
   printf("Вот один из способов вывода ");
   printf("длинных строк.\n");
   printf("Второй способ вывода \
длинных строк.\n");
   printf("Новейший способ вывода "
          "длинных строк. \n");              /* стандарт ANSI C */
   return 0;
}
```

В результате выполнения программы получается следующий результат:

```
Вот один из способов вывода длинных строк.
Второй способ вывода длинных строк.
Новейший способ вывода длинных строк.
```

Первый метод предусматривает использование нескольких операторов printf(). Поскольку первая напечатанная строка не заканчивается символом \n, вторая строка продолжается с конца первой.

Второй метод состоит в том, чтобы завершить первую строку комбинацией обратной косой черты и нажатия клавиши <Enter>. Это приводит к тому, что текст на экране начинается с новой строки, но без включения символа новой строки в саму строку. Другими словами, эффект состоит в том, что предыдущая строка не прерывается, а продолжается на следующей строке. Однако следующая строка должна начинаться с крайней левой позиции, как показано в листинге 4.14. Если вы сдвинете эту строку вправо, скажем, на пять пробелов, то эти пять пробелов станут частью строки.

Третий метод, введенный стандартом ANSI C, называется конкатенацией или сцеплением строк. Если одна строковая константа, заключенная в кавычки, следует за другой такой константой и обе строки разделены пробелом, C рассматривает эту комбинацию как единую строку, таким образом, следующие три формы эквивалентны:

```
printf("Привет юным влюбленным, где бы они ни были.");
printf("Привет юным"    "влюбленным"   ", где бы они ни были.");
printf("Привет юным влюбленным"
       ", где бы они ни были.");
```

Во всех этих методах вы должны включить любые требуемые пробелы в строки: "юным" "влюбленным" приводит к строке "юнымвлюбленным", а комбинация "юным " "влюбленным" дает в результате "юным влюбленным".

Использование функции scanf()

Теперь перейдем от вывода к вводу и исследуем функцию scanf(). Библиотека C содержит несколько функций ввода, и scanf() является самой универсальной из них, поскольку она может читать несколько форматов. Разумеется, результат ввода с клавиатуры — это текст, поскольку клавиши генерируют текстовые символы: буквы, цифры и знаки препинания. Когда вы хотите ввести, скажем, целое число 2004, вы вводите с клавиатуры символы 2 0 0 и 4.

Если вы хотите сохранить его как числовое значение, а не как строку, ваша программа должна выполнить посимвольное преобразование строки в числовое значение — именно это функция scanf() и делает! Она преобразует строковый ввод в различные формы: целые числа, числа с плавающей запятой, символы и строки C. Эта операция обратная по действию функции printf(), которая преобразует целые числа, числа с плавающей запятой, символы и строки C в текст, который затем отображается на экране.

Подобно функции printf(), scanf() использует управляющую строку, сопровождаемую списком параметров. Управляющая строка указывает, к каким типам данных должен быть преобразован вводимый поток символов. Главное различие между ними заключается в списке параметров. Функция printf() использует имена переменных, константы и выражения, а функция scanf() — указатели на переменные. По счастливому стечению обстоятельств, для использования этой функции не требуется ничего знать об указателях. Помните только следующие два простых правила:

- Если вы используете функцию scanf() для чтения значения одного из базовых типов, перед именем такой переменной ставьте знак &.
- Если вы используете функцию scanf() для чтения строки в символьный массив, не знак & не нужен.

В листинге 14.15 показана короткая программа, служащая иллюстрацией упомянутых правил.

Листинг 4.15. Программа input.c

```
// input.c -- когда использовать знак &
#include <stdio.h>

int main(void)
{
   int age;                          // переменная
   float assets;                     // переменная
   char pet[30];                     // строка

   printf("Введите информацию о своем возрасте, о сумме в банке и о любимом
животном.\n");
   scanf ("%d %f", &age, &assets);   // используйте знак &

   scanf("%s", pet);              // не используйте & для строкового массива
   printf("%d $%.2f %s\n", age, assets, pet);

   return 0;
}
```

Диалог с программой выглядит примерно так:

```
Введите информацию о своем возрасте, о сумме в банке и о любимом
животном.
38
92360.88 Шарик
38 $92360.88 Шарик
```

Функция `scanf()` использует символы, называемые пробельными (символы новой строки, табуляции и пробела), чтобы решить, как разделить ввод на отдельные поля. Это соответствует последовательным применениям спецификаций преобразования к последовательным полям, с пропуском пробельных символов, которые находятся между ними. Обратите внимание на то, как ввод распространяется на две строки. Вы могли точно так же использовать одну или пять строк, пока в вашем распоряжении имеется, по меньшей мере, один символ перевода строки, пробела или табуляции между каждым элементом. Единственное исключение представляет собой спецификатор преобразования `%c`, который читает каждый следующий символ, даже если этот символ является пробельным. Вскоре мы вернемся к этой теме.

Функция `scanf()` использует в основном тот же набор спецификаторов преобразования, что и функция `printf()`. Главное различие состоит в том, что функция `printf()` использует спецификаторы `%f`, `%e`, `%E`, `%g` и `%G` для обоих типов `float` и `double`, тогда как функция `scanf()` применяет их только для типа с плавающей запятой `float`, требуя модификатор `l` для типа `double`. В табл. 4.6 перечислены основные спецификаторы преобразования, как это описано в стандарте C99.

Таблица 4.6. Спецификаторы преобразования стандарта ANSI C для функции `scanf()`

Спецификатор преобразования	*Значение*
`%c`	Интерпретирует ввод как символ.
`%d`	Интерпретирует ввод как десятичное целое число со знаком.
`%e`, `%f`, `%g`, `%a`	Интерпретирует ввод как число с плавающей запятой (`%a` определено стандартом C99).
`%E`, `%F`, `%G`, `%a`	Интерпретирует ввод как число с плавающей запятой (`%a` определено стандартом C99).
`%i`	Интерпретирует ввод как десятичное целое число со знаком.
`%o`	Интерпретирует ввод как восьмеричное целое число со знаком.
`%p`	Интерпретирует ввод как указатель (адрес).
`%s`	Интерпретирует ввод как строку; ввод начинается с первого символа, не являющегося пробельным, и включает все символы до следующего пробельного символа.
`%u`	Интерпретирует ввод как десятичное целое число без знака.
`%x`, `%X`	Интерпретирует ввод как шестнадцатеричное целое число со знаком.

Вы также можете использовать модификаторы в спецификаторах преобразования, перечисленных в табл. 4.6. Модификаторы размещаются между знаком процента и

символом преобразования. Если вы используете в спецификаторе более одного модификатора, они должны появляться в том же самом порядке, как показано в табл. 4.7.

Таблица 4.7. Модификаторы преобразований функции `scanf()`

Модификатор	*Значение*
*	Подавить присваивание (см. текст). Пример: `"%*d"`.
цифра (ы)	Максимальная ширина поля; ввод останавливается, когда достигнута максимальная ширина поля или если обнаружен первый служебный символ, каким бы он ни был. Пример: `"%10s"`.
`hh`	Читает целое число как `signed char` или `unsigned char`. Примеры: `"%hhd"`, `"%hhu"`.
`ll`	Читает целочисленное число как `long long` или как `long long` без знака (стандарт C99). Примеры: `"%lld"`, `"%llu"`.
`h`, `l` или `L`	Спецификаторы `"%hd"` и `"%hi"` указывают, что значение будет сохранено с типом `short int`. Спецификаторы `"%ho"`, `"%hx"` и `"%hu"` определяют, что значение будет сохранено с типом unsigned `short int`. Спецификаторы `"%ld"` и `"%li"` указывают, что значение будет сохранено с типом long. `"%lo"`, `"%lx"` и `"%lu"` определяют, что значение будет сохранено с типом `unsigned long`. Спецификаторы `"%le"`, `"%lf"` и `"%lg"` указывают, что значение будет сохранено с типом `double`. Использование модификатора `L` вместо `l` в комбинациях с `e`, `f` и `g` обеспечивает то, что значение будет сохранено с типом `long double`. В отсутствие этих модификаторов `d`, `i`, `o` и `x` указывают на тип `int`, а `e`, `f` и `g` — на тип `float`.

Как видите, вполне возможно использование спецификаторов преобразования, при этом в таблицах опущены некоторые их свойства. Эти свойства, прежде всего, облегчают чтение выбранных данных из источников, имеющих жесткий формат, таких как перфокарты или другие записи данных. Поскольку в этой книге функция `scanf()` используется, в первую очередь, как удобное средство для ввода данных в программу в интерактивном режиме, мы не будем обсуждать экзотические особенности, понятные только профессионалам.

Ввод с помощью функции `scanf()`

Рассмотрим более внимательно, как функция `scanf()` читает символы входного потока данных. Предположим, что вы используете спецификатор `%d`, чтобы читать целое число. Функция `scanf()` начинает читать поток ввода в посимвольном режиме. Она пропускает пробельные символы (пробелы, символы табуляции и символы перевода строки) до тех пор, пока не столкнется с символом, отличным от пробельного.

Поскольку функция предпринимает попытку читать целое число, функция scanf() надеется получить символ цифры или, возможно, символ знака (+ или -). Если она встречает цифру или знак, то запоминает знак и читает следующий символ. Если это цифра, она сохраняет цифру и читает следующий символ. Функция scanf() продолжает читать и сохранять символы, пока не столкнется с нецифровым символом. Тогда функция приходит к заключению, что она достигла конца очередного целого числа. Функция scanf() возвращает этот нецифровой символ обратно в поток ввода. Это означает, что в следующий раз, когда программа начнет читать поток ввода, она начнет его с ранее отвергнутого нецифрового символа. Наконец, функция scanf() вычисляет числовое значение, соответствующее цифрам, которые она считала, и помещает это значение в специально предназначенную переменную.

Если вы используете всю ширину поля, функция scanf() прекращает чтение при достижении конца поля или на первом пробельном символе, в зависимости от того, что встретится раньше.

А что случится, если первый символ, отличный от пробельного, не является цифрой, а скажем, представляет собой символ A? Тогда функция scanf() тут же останавливается и помещает символ A (или что бы там ни было) обратно в поток ввода. Специально предназначенной переменной никакое значение не присваивается, и в следующий раз, когда программа возобновит чтение потока ввода, она начнет его с символа A.

Если ваша программа работает только со спецификаторами %d, функция scanf() никогда не пропустит этот символ A. Кроме того, если вы используете scanf() с несколькими спецификаторами, ANSI C требует, чтобы функция прекращала чтение ввода при первой же ошибке.

Чтение потока ввода с использованием других числовых спецификаторов происходит так же, как в случае применения спецификатора %d. Главное различие между ними заключается в том, что функция scanf() может распознавать больше символов как часть числа. Например, спецификатор %x требует, чтобы функция scanf() распознавала символы a–f и A–F как шестнадцатеричные цифры. Спецификаторы с плавающей запятой требуют, чтобы функция scanf() распознавала десятичные точки, экспоненциальную форму записи и новую p-нотацию.

Если вы используете спецификатор %s, то допускается любой символ, отличный от пробельного, поэтому функция scanf() пропускает пробельные символы до появления первого непробельного символа, после чего сохраняет все неслужебные символы вплоть до следующего появления пробельного символа. Это означает, что спецификатор %s заставляет функцию scanf() читать одиночные слова, то есть строки, которые не содержат пробельных символов. Если вы используете всю ширину поля, функция scanf() прекращает чтение в конце поля или на первом пробельном символе. Вы не можете использовать всю ширину поля, чтобы заставить функцию scanf() читать более одного слова при наличии одного спецификатора %s. Завершающий штрих: когда функция scanf() помещает строку в специально предназначенный для этой цели массив, она то добавляет завершающий символ '\0' с тем, чтобы сделать содержимое массива строкой C.

Если вы указываете спецификатор %c, то все вводимые символы запоминаются в исходном виде. Если следующим вводимым символом является пробел или символ новой строки, то пробел или символ новой строки присваивается указанной переменной; пробельный символ не опускается.

По сути дела функцию scanf() нельзя считать наиболее часто используемой функцией ввода в C. Она рассматривается здесь в силу своей универсальности (она может читать все многообразие типов данных), тем не менее, в C доступны несколько других функций ввода, такие как getchar() и gets(), которые больше подходят для выполнения ряда конкретных задач, например, чтения одиночных символов или чтения строк, содержащих пробелы. Мы рассмотрим некоторые из этих функций в главах 7, 11 и 13. В то же время, если вам нужно ввести целое число или десятичную дробь, символ либо строку, вы можете воспользоваться услугами функции scanf().

Обычные символы в форматирующей строке

Функция scanf() предоставляет возможность помещать обычные символы в форматирующую строку. Обычные символы, отличные от пробела, должны быть согласованы с форматом вводимой строки. Например, предположим, что вы случайно помещаете запятую между двумя спецификаторами:

```
scanf("%d,%d", &n, &m);
```

Функция scanf() интерпретирует эту строку следующим образом: нужно напечатать число, затем напечатать запятую, а затем второе число. То есть вы должны ввести два целых числа следующим образом:

```
88, 121
```

Поскольку запятая стоит в форматирующей строке непосредственно за спецификатором %d, вы должны напечатать ее сразу после числа 88. Однако, так как функция scanf() пропускает пробельный символ, предшествующий целому числу, вы могли напечатать пробел или символ новой строки после запятой при вводе. То есть, варианты

```
88, 121
```

и

```
88,
121
```

также были бы приемлемыми.

Пробел в форматирующей строке означает, что нужно пропускать любой пробельный символ перед следующим элементом ввода. Например, оператор

```
scanf ("%d,%d", &n, &m);
```

принял бы любую из следующих входных строк:

```
88,121
88 ,121
88 , 121
```

Обратите внимание, что понятие “любой пробельный символ” включает в себя отсутствие пробельного символа как частный случай.

За исключением %c, все остальные спецификаторы автоматически пропускают пробельный символ, предшествующий вводимому значению, так что оператор scanf("%d%d", &n, &m) ведет себя так же, как и scanf ("%d %d ", &n, &m). В случае спецификатора %c включение пробела в форматирующую строку изменяет это поведение. Например, если в форматирующей строке спецификатору %c предшествует пробел, то функция scanf() пропускает все до появления первого непробельного символа. Таким образом, оператор scanf ("%c", &ch) читает первый значащий сим-

вол, с которым сталкивается при вводе, а scanf(" %c", &ch) читает первый встреченный ею непробельный символ.

Возвращаемое значение функции scanf()

Функция scanf() возвращает количество элементов, которые она успешно прочитала. Если она не прочитала никаких элементов, что бывает в тех случаях, когда вы печатаете нечисловую строку, а scanf() ожидает число, она возвращает 0. scanf() возвращает EOF, когда обнаруживает условие, известное как "конец файла". (EOF — это специальное значение, определенное в файле stdio.h. Как правило, директива #define присваивает EOF значение -1.) Мы рассмотрим признак конца файла в главе 6, а вопросы использования возвращаемого значения функции scanf() — далее в этой главе. После того как вы изучите операторы if и while, вы сможете использовать возвращаемое значение функции scanf(), чтобы обнаруживать и обрабатывать несогласованный ввод.

Модификатор * и его использование в функциях printf() и scanf()

Модификатор * используется в функциях printf() и scanf() для изменения значения спецификатора, причем весьма несхожими способами. Сначала посмотрим, что модификатор * может сделать для функции printf().

Предположим, что вы не хотите фиксировать ширину поля заранее, но хотите, чтобы ее определила сама программа. Вы можете сделать это, воспользовавшись модификатором * вместо числа, задающего ширину поля, но при этом вам придется с помощью аргумента функции сообщить, какой должна быть ширина поля. Иначе говоря, если вы работаете со спецификатором преобразования %*d, список аргументов должен включать значение для * *и* значение для d. Эта методика также может быть использована применительно к значениям с плавающей запятой с тем, чтобы задавать точность, а также ширину поля. В листинге 4.16 показан небольшой пример, демонстрирующий, как все это работает.

Листинг 4.16. Программа varwid.c

```
/* varwid.c -- использование поля вывода переменной ширины */
#include <stdio.h>
int main(void)
{
   unsigned  width, precision;
   int number = 256;
   double weight = 242.5;
   printf("Какова  ширина  поля?\n");
   scanf("%d", &width);
   printf("Значение равно: %*d:\n", width, number);
   printf("Теперь введите ширину и точность:\n");
   scanf("%d  %d", &width, &precision);
   printf("Вес = %*.*f\n", width, precision, weight);
   printf("Готово!\n");
   return 0;
}
```

Переменная width определяет ширину поля, а переменная number — это число, которое должно быть напечатано. Поскольку модификатор * предшествует d в спецификаторе, значение width предшествует значению number в списке параметров функции printf(). Точно так же значения width и precision предоставляют необходимую информацию для форматирования при печати значения weight. Ниже показан результат выполнения этой учебной программы:

```
Какова ширина поля?
6
Значение равно: 256:
Теперь введите ширину и точность:
8 3
Вес =  242.500
Готово!
```

В рассматриваемом случае ответ на первый вопрос был 6, так что ширина использованного поля задана равной 6. Точно так же в результате второго ответа была установлена ширина поля 8 с 3 цифрами справа от десятичной точки. В общем случае программа сама может принять решение в отношении значений этих переменных после того, как ей становится известным значение weight.

В случае функции scanf() модификатор * выполняет совсем другую работу. Когда модификатор * помещен между знаком % и символом спецификатора, это заставляет функцию пропустить соответствующий ввод. В листинге 4.17 представлен соответствующий пример.

Листинг 4.17. Программа skip2.c

```
/* skip2.c -- игнорирует первые два целых числа из потока ввода */
#include <stdio.h>
int main(void)
{
   int n;
   printf("Введите три целых числа:\n");
   scanf("%*d %*d %d",fin);
   printf("Последним целым числом было %d\n", n);
   return 0;
}
```

Функция scanf() в листинге 4.17 выполняет следующее: пропускает два целых числа и копирует третье целое число в переменную n. В результате выполнения программы получаются такие выходные данные:

```
Введите три целых числа:
2004 2005 2006
Последнее целое число будет 2006
```

Это средство пропуска полезно в тех случаях, когда, например, необходимо прочесть конкретный столбец файла, в котором данные упорядочены в виде однородных столбцов.

Советы по использованию функции printf()

Определение фиксированной ширины поля полезно, когда вы хотите выводить данные в форме столбцов. Поскольку заданная по умолчанию ширина поля есть всего лишь ширина числа, выраженная в числах, повторное использование, скажем, такого оператора

```
printf("%d %d %d\n", val1, val2, val3);
```

приводит к печати данных в виде выровненных столбцов, даже если числа в столбце имели различные размеры. Например, вывод может выглядеть следующим образом:

```
   12     234      1222
    4       5        23
22334    2322     10001
```

(В данном случае предполагается, что значение переменных подвергалось изменениям в промежутке между выполнением операторов печати.)

Вывод может быть упорядочен, если использовать достаточно большую фиксированную ширину поля. Например, использование оператора

```
printf("%9d %9d %9d\n", val1, val2, val3);
```

позволило бы получить следующие выходные данные:

```
   12     234      1222
    4       5        23
22334    2322     10001
```

Включение пробела между одним спецификатором преобразования и следующим спецификатором гарантирует, что одно число никогда не будет наложено на другое, даже если оно выйдет за границы своего собственного поля. Это происходит потому, что в управляющей строке печатаются обычные символы, включая пробелы.

С другой стороны, если число должно быть включено в фразу, часто бывает удобно определить поле таким же или меньшим по размеру, чем ширина ожидаемого числа. Благодаря такому подходу число размещается правильно, без ненужных пробелов. Например, оператор

```
printf("Каунт Беппо пробежал %.2f миль за 3 часа.\n", distance);
```

печатает следующую фразу:

```
Каунт Беппо пробежал 10.22 миль за 3 часа.
```

Изменение спецификации преобразования на `%10.2f` привело бы к такому результату:

```
Каунт Беппо пробежал      10.22 миль за 3 часа.
```

Ключевые понятия

В языке C тип `char` представляет единичный символ. Для представления последовательности символов в C используются символьные строки. Одной из форм строки является символическая константа, в которой символы заключены в двойные кавычки; примером может служить строка `"Удачи, друзья!"`. Вы можете хранить строку в массиве символов, который размещается в смежных байтах памяти компьютера.

Символьные строки, выраженные как символические константы либо хранящиеся в массиве символов, оканчиваются символом, который не выводится на печать и называется *нулевым* (*null*) символом.

Плодотворной оказалась идея представлять числовые константы в программе символическими, либо посредством директивы #define, либо с помощью ключевого слова const. Символические константы делают программу удобочитаемой и легкой для сопровождения и внесения изменений.

Стандартные функции ввода и вывода scanf() и printf() языка C используют систему, в которой вы должны согласовать спецификаторы в первом аргументе со значениями в последующих аргументах. Согласование, скажем, спецификатора типа int, такого как %d, со значением float приводит к непредсказуемым результатам. Вы должны внимательно следить за тем, чтобы количество и типы спецификаторов были согласованы с остальными аргументами функций. Что касается функции scanf(), то не забывайте проставить перед именем переменной префикс в виде адресной операции (&).

Пробельные символы (символы табуляции, пробела и новой строки) играют решающую роль в том, как scanf() видит данные ввода. За исключением режима ввода, задаваемого спецификатором %c (который читает только следующий символ), при чтении входных данных функция scanf() пропускает все символы пробела до первого непробельного символа. Далее она продолжает чтение символов до тех пор, пока не обнаружит пробельный символ либо пока не обнаружит символ, имеющий тип, отличный от заданного. Теперь посмотрим, что происходит, если мы подадим на ввод одну и ту же строку, но при различных режимах работы функции scanf(). Начнем со следующей входной строки:

```
-13.45e12#  0
```

Прежде всего, предположим, что используется режим %d; в этом случае функция scanf() прочтет три символа (-13) и остановится на точке, рассматривая ее как следующий входной символ. Затем функция scanf() преобразует последовательность символов -13 в соответствующее целочисленное значение и сохраняет его в переменной назначения типа int. Далее, в режиме %f функция scanf() читает ту же строку как последовательность символов -13.45E12 и останавливает чтение на символе #, оставляя его для следующей операции ввода. Затем она преобразует последовательность символов -13.45E12 в соответствующее значение с плавающей запятой и сохраняет его в переменной типа float. Читая ту же строку в режиме %s, функция scanf() прочитает последовательность символов -13.45E12# и останавливается на пробеле, оставляя его для следующей операции ввода. Затем она сохраняет коды всех этих десяти символов в массиве назначения, добавив в конец массива нулевой символ. Наконец, при чтении этой же строки в режиме %s функция scanf() прочтет и сохранит первый символ, в данном случае это пробел.

Резюме

Строка — это последовательность символов, рассматриваемая как единый блок. В языке C строки представлены последовательностями символов, завершающимися нулевым символом, ASCII-код которого равен 0. Строки могут сохраняться в символьных массивах. Массив представляет собой последовательность элементов, имеющих один и тот же тип.

Чтобы объявить массив с именем name, содержащий 30 элементов типа char, воспользуйтесь следующей конструкцией:

```
char name[30];
```

Позаботьтесь о том, чтобы система выделила такое количество элементов, которое достаточно для того, чтобы вместить строку целиком, включая нулевой символ.

Строковые константы создаются путем заключения строки в двойные кавычки: "Это пример строки".

Функцию strlen() можно использовать для определения длины строки (без нулевого символа в конце). Функция scanf(), будучи вызванной вместе со спецификатором %s, может применяться для чтения строк, состоящих из одного слова.

Препроцессор языка C осуществляет поиск в исходном тексте программы директив для препроцессора, которые начинаются с символа #, и действует в соответствии с этими директивами до компиляции программы. Директива #include заставляет процессор добавлять содержимое другого файла в ваш файл в то место, где расположена эта директива. Директива #define позволяет определять явные константы, то есть задавать символические представления констант. Заголовочные файлы limits.h и float.h используют директиву #define, чтобы определить набор констант, представляющих различные свойства цело численных типов и чисел с плавающей запятой. Для создания символических констант вы также можете использовать модификатор const.

Функции printf() и scanf() обеспечивают универсальную поддержку ввода и вывода. Каждая из них использует управляющую строку, содержащую вложенные спецификаторы преобразования, которые указывают количество и типы элементов данных для чтения или вывода на печать. Вы можете также использовать спецификаторы преобразования, чтобы управлять внешним видом выходных данных: шириной поля, количеством десятичных позиций и размещением в пределах поля.

Вопросы для самоконтроля

Ответы на эти вопросы находятся в приложении А.

1. Запустите программу из листинга 4.1 еще раз и введите имя и фамилию, когда программа предложит ввести имя. Что происходит в этом случае? Почему?
2. Предположим, что каждый из следующих примеров является частью законченной программы. Что печатать будет каждая такая часть?

   а.
   ```
   printf("Он продал эту картину за $%2.2f .\n", 2.345e2);
   ```

   б.
   ```
   printf("%c%c%c\n", 'H', 105, '\41');
   ```

   в.
   ```
   #define Q "Его Гамлет был хорош и без намека на вульгарность."
   printf("%s\nсодержит %d символов.\n", Q, strlen(Q));
   ```

   г.
   ```
   printf("Является ли %2.2e тем же, что и %2.2f?\n",
          1201.0, 1201.0);
   ```

3. Какие изменения необходимо сделать в пункте 2.в, чтобы строка Q была напечатана без кавычек?
4. Попытайтесь найти ошибку в следующем коде:

```
define B booboo
define X 10
main(int)
{
   int  age;
   char name;
   printf{"Введите свое имя,");
   scanf("%s", name);
   printf("Хорошо, %C, а сколько вам лет?\n", name);
   scanf("%£", age);
   xp = age + X;
   printf ("Неужели, %s! Вам должно быть, по меньшей мере,
%d лет.\n", B, xp);
   rerun 0;
}
```

5. Предположим, что программа начинается так:

```
#define BOOK "Война и мир"
int main(void)
{
   float cost = 12.99;
   float percent = 80.0;
```

Напишите оператор `printf()`, который использует `BOOK`, `cost` и `percent`, чтобы напечатать следующее:

```
Этот экземпляр книги "Война и мир" стоит $12.99.
Это 80% от цены прайс-листа.
```

6. Какие спецификаторы преобразования вы бы использовали, чтобы напечатать:
   а. Десятичное целое число с шириной поля, равной количеству цифр этого числа.
   б. Шестнадцатеричное целое число в форме 8A в поле шириной 4 символа.
   в. Число с плавающей запятой в форме 232.346 с шириной поля 10 символов.
   г. Число с плавающей запятой в форме 2.33e+002 с шириной поля 12 символов.
   д. Строку, выровненную по левому краю в поле шириной 30 символов.
7. Какие спецификаторы преобразования вы бы использовали, чтобы напечатать перечисленные ниже выражения?
   а. Целое число типа `unsigned long` в поле шириной 15 символов.
   б. Шестнадцатеричное целое число в форме 0x8a в поле шириной 4 символа.
   в. Число с плавающей запятой в форме 2.33E+02, выровненное по левому краю в поле шириной 12 символов.
   г. Число с плавающей запятой в форме +232.346 в поле шириной 10 символов.
   д. Первые 8 символов строки в поле шириной 8 символов.
8. Какие спецификаторы преобразования вы бы использовали, чтобы печатать перечисленные ниже выражения?
   а. Десятичное целое число, имеющее минимум 4 цифры в поле шириной 6 символов.

б. Восьмеричное целое число в поле, ширина которого будет задаваться в списке параметров.

в. Символ в поле шириной 2 символа.

г. Число с плавающей запятой в форме +3.13 в поле шириной, которая равна количеству символов в этом числе.

д. Первые пять символов в строке, выровненной по левому краю поля шириной 7 символов.

9. Для каждой из следующих входных строк напишите оператор `scanf()`, чтобы прочитать их. Объявите также переменные или массивы, используемые в операторе.

   а. 101

   б. 22.32 и 8.34E-09

   в. linguini

   г. catch 22

   д. catch 22 (но пропустить catch)

10. Что такое пробельный символ?

11. Предположим, что вы предпочитаете пользоваться круглыми скобками вместо фигурных скобок в своих программах. Насколько хорошо бы работали следующие конструкции?

```
#define ( {
#define ) }
```

Упражнения по программированию

1. Напишите программу, которая запрашивает имя и фамилию, а затем печатает их в формате "*фамилия, имя*".

2. Напишите программу, которая запрашивает имя и выполняет с ним следующие действия:

   а. Печатает его заключенным в двойные кавычки.

   б. Печатает его в поле шириной 20 символов, при этом все поле заключается в кавычки.

   в. Печатает его с левого края поля шириной 20 символов, при этом все поле заключается в кавычки.

   г. Печатает его в поле шириной, на три символа превышающем длину имени.

3. Напишите программу, которая читает число с плавающей запятой и печатает его сначала в десятичной, а затем в экспоненциальной форме. Предусмотрите вывод в следующих формах (количество цифр показателя степени в вашей системе может быть другим).

   а. Вводом является `21.3` или `2.1e+001`.

   б. Вводом является `+21.290` или `2.129E+001`.

4. Напишите программу, которая запрашивает рост в дюймах и имя, после чего отображает полученную информацию в следующей форме:

   ```
   Ларри, ваш рост составляет 6.208 футов
   ```

   Используйте тип `float`, а также операцию деления `/`. Запросите рост в сантиметрах и выразите в метрах, если вам так удобнее.

5. Напишите программу, которая запрашивает имя пользователя и затем его фамилию. Сделайте так, чтобы она печатала введенные имена в одной строке и количество символов в каждом слове в следующей строке. Выровняйте каждое количество символов по концу соответствующего имени, как показано ниже:

   ```
   Melissa Honeybee
         7        8
   ```

   Затем сделайте так, чтобы она печатала ту же самую информацию, но с количеством символов, выровненным по началу каждого слова:

   ```
   Melissa Honeybee
   7       8
   ```

6. Напишите программу, которая присваивает переменной типа `double` значение `1.0/3.0` и переменной типа `float` значение `1.0/3.0`. Выведите на экран каждый результат три раза — в первом случае с четырьмя цифрами справа от десятичной точки, во втором случае — с двенадцатью цифрами и в третьем случае — с шестнадцатью цифрами. Включите также в программу заголовочный файл `float.h` и выведите на экран значения `FLT_DIG` и `DBL_DIG`. Совместимы ли выведенные значения со значением `1.0/0.3`?

7. Напишите программу, которая просит пользователя ввести количество преодоленных миль и количество галлонов израсходованного бензина. Затем эта программа должна рассчитать и отобразить на экране количество миль, пройденных на одном галлоне горючего, с одним знаком после десятичной точки. Далее, используя тот факт, что один галлон приблизительно равен 3.785 литра и одна миля — 1.609 километра, ваша программа должна перевести значение в милях на галлон в литры на 100 километров (обычную европейскую меру измерения потребления горючего) и вывести результат с одним знаком после десятичной точки. (Обратите внимание, что американская схема измеряет затраты горючего, необходимого для преодоления заданного расстояния, тогда как европейская схема измеряет пройденный путь на единицу горючего.) Используйте символические константы (определенные с помощью `const` или `#define`) для этих двух параметров преобразования.

ГЛАВА 5

Операции, выражения и операторы

В этой главе:

- Ключевые слова: `while`, `typedef`
- Операции: = + - * / % ++ -- (тип)
- Множественные операции языка C, в том числе и те, которые используются для реализации обычных арифметических операций
- Приоритеты операций и значение терминов *"оператор"* и *"выражение"*
- Удобный в использовании оператор цикла `while`
- Составные операторы, автоматическое преобразование типов и приведение типов
- Написание функций, использующих аргументы

Сейчас, когда вы уже ознакомились со способами представления данных, проведем анализ способов обработки данных. Для этих целей язык C предлагает множество различных операций. Вы можете выполнять арифметические операции, сравнивать значения, обновлять значения переменных, логически комбинировать отношения и делать многое другое. Начнем с арифметических операций — сложения, вычитания, умножения и деления.

Другим аспектом обработки данных является организация ваших программ с тем, чтобы они выполняли правильные действия в нужном порядке. C обладает несколькими языковыми средствами, которые помогают решить эту задачу. Одним из этих средств являются циклы, и в этой главе вы получите первое представление о них. Цикл позволяет повторять действия, делая программу более интересной и повышая ее возможности.

Введение в циклы

В листинге 5.1 представлена демонстрационная программа, выполняющая несложные арифметические операции по вычислению длины ступни в дюймах, для которой подходит обувь (мужская) размера 9. Чтобы вы в полной мере смогли оценить преимущества циклов, эта версия программы иллюстрирует ограничения программирования, не использующего циклов.

Листинг 5.1. Программа `shoes1.c`

```
/* shoes1.c -- преобразует размер обуви в дюймы */
#include <stdio.h>
#define ADJUST 7.64
#define SCALE 0.325
int main(void)
{
   double shoe, foot;
   shoe = 9.0;
   foot = SCALE * shoe + ADJUST;
   printf("Размер обуви (мужской)    длина ступни\n");
   printf("%10.1f %20.2f дюймов\n", shoe, foot);
   return 0;
}
```

Мы, наконец, получили желанную программу с операциями умножения и сложения. Она берет размер вашего ботинка (если вы носите размер 9) и сообщает вам, каков размер вашей ступни в дюймах. Вы можете сказать: “Я вполне могу решить эту программу вручную быстрее, чем вы введете эту программу с клавиатуры”. Правильное замечание. Одноразовая программа, которая переводит в дюймы только один размер, представляет собой напрасную трату времени и усилий. Вы можете сделать программу более полезной, если реализуете в ней интерактивность, но и в этом случае могучий потенциал компьютера не будет задействован в полной мере.

Вам нужно заставить компьютер выполнять повторные вычисления для некоторой последовательности размеров обуви. В конце концов, это одна из причин применения компьютеров для арифметических вычислений. Язык C предлагает несколько методов выполнения повторных вычислений, и здесь мы рассмотрим один из них. Этот метод, получивший название *цикла* `while`, позволяет обеспечить более эффективное использование операций. В листинге 5.2 показан усовершенствованный вариант программы определения длины стопы по размеру обуви.

Листинг 5.2. Программа `shoes2.c`

```
/* shoes2.c -- вычисляет длину стопы для нескольких размеров обуви */
#include <stdio.h>
#define ADJUST 7.64
#define SCALE 0.325
int main(void)
{
   double shoe, foot;
   printf("Размер обуви (мужской)    длина ступни\n");
   shoe = 3.0;
   while (shoe < 18.5)           /* начало цикла while */
   {                             /* начало блока       */
      foot = SCALE*shoe + ADJUST;
      printf("%10.1f %20.2f дюймов\n", shoe, foot);
      shoe = shoe + 1.0;
   }                             /* конец блока        */
   printf("Если обувь подходит, носите ее.\n");
   return 0;
}
```

Вот как выглядит сжатая версия выходных данных программы shoes2.c:

```
Размер обуви (мужской)      длина ступни
   3.0                       8.62 дюймов
   4.0                       8.94 дюймов
   ...                       ...
  17.0                      13.16 дюймов
  18.0                      13.49 дюймов
Если обувь подходит, носите ее.
```

(По сути, значения констант для этого преобразования были получены во время неофициального визита в обувной магазин. Единственная метка для обуви, которая лежала на прилавке, была предназначена для мужской обуви. Тем, кого интересуют размеры женской обуви, придется самим отправиться в обувной магазин. Кроме того, программа делает нереалистичные предположения о том, что существующая система размеров обуви рациональна и единообразна.)

Вот как работает цикл while. Когда программа впервые попадает на оператор цикла while, она проверяет, принимает ли условие в круглых скобках значение true. В этом случае выражение этого условия имеет вид:

```
shoe < 18.5
```

Символ < означает "меньше чем". Переменная shoe инициализирована значением 3.0, что, естественно, меньше 18.5. В силу этого обстоятельства условие принимает значение true, и программа переходит к следующему оператору, который преобразует размер в дюймы. Затем она печатает результат. Следующий оператор увеличивает значение shoe на 1.0, в результате чего эта переменная получает значение 4.0:

```
shoe = shoe + 1.0;
```

В этом месте программа возвращается к порции кода while, чтобы проверить условие. Но почему именно в этом месте? Потому, что в следующей строке находится закрывающая фигурная скобка (}), а код использует пару скобок ({}) для обозначения границ цикла while. Операторы, которые находятся между двумя фигурными скобками, используются повторно. Раздел программы, расположенный внутри фигурных скобок, и сами фигурные скобки, называются блоком. Однако вернемся к программе. Значение 4 меньше 18.5, следовательно, вся совокупность команд, заключенных в фигурные скобки (блок), следующая за while, повторяется. (Пользуясь компьютерным жаргоном, программа "просматривает эти операторы в цикле".) Это продолжается до тех пор, пока переменная shoe не достигнет значения 19.0. Тогда условие

```
shoe < 18.5
```

получает значение false, так как 19.0 не меньше 18.5. Как только это произойдет, управление передается первому оператору, следующему за циклом while. В рассматриваемом случае это завершающий оператор printf().

Вы легко можете приспособить эту программу для выполнения других преобразований. Например, назначьте константе SCALE значение 1.8, а константе ADJUST — 32.0, и вы получите программу, которая преобразует значение температуры по Цельсию в значение по Фаренгейту. Назначьте константе SCALE значение 0.6214, а ADJUST — 0, и вы получите преобразование километров в мили. Внося описанные модификации, естественно, измените содержание выходных сообщений, дабы избежать путаницы.

Цикл while предоставляет в ваше распоряжение удобное и гибкое средство управления ходом выполнения программы.

А сейчас перейдем к изучению фундаментальных операций, которыми вы можете воспользоваться в своих программах.

Фундаментальные операции

Язык C использует операции для представления арифметических действий. Например, операция + означает, что две величины, находящиеся по обе стороны знака +, складываются. Если термин *операция* покажется вам странным, согласитесь с тем, что вещи подобного рода нужно каким-то образом называть. "Операция" выглядит ничуть не хуже, чем, скажем, "эта вещь" или "арифметический транзактор". Теперь рассмотрим базовые арифметические операции: =, +, -, * и /. (В языке C операция возведения в степень отсутствует. Однако библиотека стандартных математических функций C предлагает для этих целей функцию `pow()`. Например, `pow(3.5, 2.2)` возвращает значение 3.5, возведенное в степень 2.2.)

Операция присваивания: =

В языке C знак равенства не означает "равно". Это операция присваивания значения. Например, оператор

```
bmw = 2006;
```

присваивает значение 2006 переменной с именем `bmw`. Иначе говоря, элемент, расположенный слева от знака = представляет собой *имя* переменной, а элемент справа — значение, присваиваемое этой переменной. Символ = называется *операцией присваивания*. Снова хотим вас предупредить, не думайте, что эта строка означает "переменная `bmw` равна 2006". Напротив, она означает "присвоить переменной `bmw` значение 2006". В этой операции действие направлено справа налево. Возможно, это различие между именем переменной и значением переменной покажется непринципиальным, однако, рассмотрим следующий довольно-таки общий оператор:

```
i = i + 1;
```

С математической точки зрения этот оператор не имеет смысла. Если вы прибавите 1 к конечному числу, результат не может быть "равен" числу, к которому вы прибавляете 1, однако как компьютерный оператор присваивания он имеет смысл и прекрасно работает. Он означает следующее: "Извлечь значение переменной с именем `i`, добавить к этому значению 1, а затем назначить это новое значение переменной `i`", как показано на рис. 5.1.

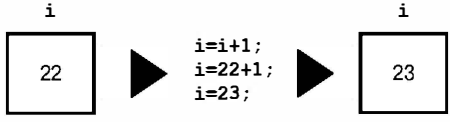


Рис. 5.1. *Оператор* `i = i + 1;`

Оператор наподобие

```
2006 = bmw;
```

в языке C не имеет смысла (и, разумеется, не является правильным), так как 2006 — это константа. Вы не можете присвоить значение константе, она сама всегда *является* значением. Имея дело с компьютером, всегда помните, что элемент слева от знака = должен быть именем какой-то переменной. По сути, слева должна находиться ссылка на ячейку памяти. Простейший способ предполагает использование имени перемен-

ной, однако, как вы увидите далее, для определения соответствующей ячейки памяти можно применять "указатели". В общем случае C употребляет термин *модифицируемое l-значение* для выделения тех логических сущностей, которым вы можете присваивать значения. Фраза "модифицируемое l-значение", возможно, не несет пока четкого содержательного смысла, поэтому рассмотрим несколько определений.

Немного терминологии: объекты данных, l-значения, r-значения и операнды

Объект данных — это общий термин, обозначающий область хранения данных, которая может быть использована для удержания значений. Например, область хранения данных для переменных или массивов представляет собой объект данных. В C используется понятие *l-значение* для обозначения имени или выражения, которое определяет конкретный объект данных. Имя переменной, например, является l-значением, следовательно, объект означает фактическую область хранения данных, однако, l-значение есть метка, которая служит для определения местоположения этого участка памяти.

Не все объекты могут менять свои значения, поэтому в C имеется термин "модифицируемое *l-значение*" для обозначения тех объектов, которые могут менять свое значение. По этой причине левая часть оператора присваивания должна быть модифицируемым l-значением. На самом деле, "*l*" в l-значении происходит от слова "left" ("левое"), поскольку модифицируемые l-значения могут указываться в левой стороне операторов присваивания.

Термин "*r-значение*" означает числа, которые могут быть присвоены модифицируемым l-значениям. Например, рассмотрим следующий оператор:

```
bmw = 2006;
```

В данном случае bmw — это модифицируемое l-значение, а 2006 — r-значение. Как вы уже, должно быть, догадались, "*r*" в r-значении происходит от слова "right" ("правое"). R-значение может быть константой, переменной или любым другим выражением, которое дает в результате значение. По мере изучения имен вещей вырисовывается подходящий термин для того, что мы называем "элементом" (примером может служить фраза "элемент слева от знака ="), этот термин обозначает операнд. Операндом является то, чем оперирует операция. Например, поедание бутерброда вы можете описать как выполнение операции "поедать" к операнду "бутерброд"; точно так же вы можете сказать, что левый операнд операции = должен быть модифицируемым l-значением.

Базовая операция присваивания в языке C отличается оригинальностью на фоне других его операций. Выполним программу, показанную в листинге 5.3.

Листинг 5.3. Программа `golf.c`

```
/* golf.c -- карта итоговых результатов игры в гольф */
#include <stdio.h>
int main(void)
{
   int jane, tarzan, cheeta;
   cheeta = tarzan = jane = 68;
   printf("                    чита тарзан джейн\n");
   printf("Счет первого раунда %4d %8d %7d\n",cheeta,tarzan,jane);
   return 0;
}
```

Многие языки программирования не допускают тройного присваивания значений, сделанного в этой программе, но C считает это обычным делом. Присваивания выполняются справа налево. Прежде всего, значение 68 получает переменная jane, затем tarzan и в завершение это значение присваивается переменной cheeta. Получаем следующие выходные данные:

```
                    чита  тарзан  джейн
Счет первого раунда   68      68     68
```

Операция сложения: +

Операция сложения приводит к тому, что два значения с обеих сторон знака + образуют сумму. Например, оператор

```
printf("%d", 4 + 20);
```

выводит число 24, но не выражение

```
4 + 20
```

Складываемые значения (операнды) могут быть как переменными, так и константами. В силу этого, оператор

```
income = salary + bribes;
```

заставляет извлечь значения двух переменных в правой части оператора, выполнить их сложение, а результат сложения присвоить переменной income.

Операция вычитания: –

Операция вычитания состоит в том, что число, стоящее в операторе после знака –, вычитается из числа, стоящего перед этим знаком. Оператор

```
takehome = 224.00 – 24.00;
```

присваивает переменной takehome значение 200.0.

Операции + и – называются бинарными, или двухместными, это означает, что они выполняются над *двумя* операндами.

Операции знака: – и +

Знак "минус" может использоваться для указания или изменения алгебраического знака конкретного значения. Например, следующая последовательность:

```
rocky = –12;
smokey = –rocky;
```

завершается тем, что переменная smokey получает значение 12.

Когда подобным образом используется знак "минус", он называется унарной (одноместной) операцией, это означает, что она выполняется над одним операндом (рис. 5.2).

Стандарт C90 вводит в язык C унарную операцию +. Она не меняет значения или знака операнда, она просто позволяет использовать такие операторы, как

```
dozen = +12;
```

и при этом не получать сообщений об ошибке. Раньше такая конструкция не допускалась.

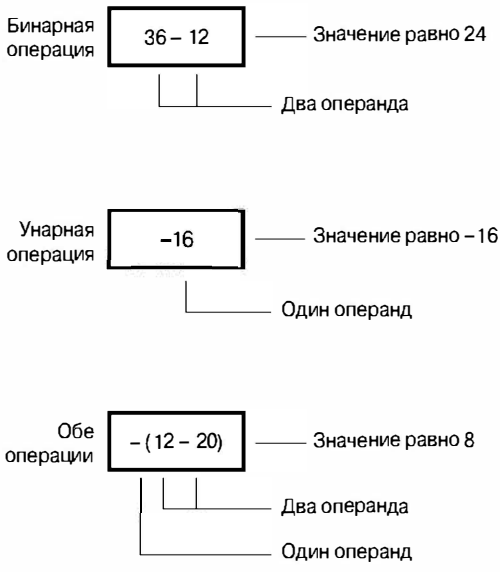


Рис. 5.2. *Унарные и бинарные операции*

Операция умножения: *

Умножение обозначается символом *. Оператор

```
cm = 2.54 * inch;
```

умножает переменную `inch` на `2.54` и присваивает результат умножения переменной `cm`.

Между прочим, не желаете ли вы составить таблицу квадратов значений? В языке C отсутствует функция возведения в квадрат, но как показано в листинге 5.4, для этого вы можете воспользоваться операцией умножения и вычислить квадраты соответствующих величин.

Листинг 5.4. Программа `squares.c`

```
/* squares.c -- генерирует таблицу квадратов 20 значений */
#include <stdio.h>
int main(void)
{
   int num = 1;
   while (num < 21)
   {
      printf("%4d %6d\n", num, num * num);
      num = num + 1;
   }
   return 0;
}
```

Эта программа печатает первые 20 целых чисел и их квадраты, в чем вы можете убедиться сами. Рассмотрим более интересный пример.

Экспоненциальный рост

Вы, скорее всего, слышали историю о могущественном правителе, который хотел вознаградить мудреца, оказавшего ему большую услугу. Когда мудреца спросили, что он желает получить, он указал на шахматную доску и попросил положить одно пшеничное зернышко на первую клетку, два зернышка на вторую клетку, четыре — на третью, восемь — на четвертую и так далее. Правитель, не имеющий понятия в математике, был поражен скромностью притязаний мудреца, ибо был готов предложить ему большие богатства. Мудрец сыграл с правителем злую шутку, как показывает программа в листинге 5.5. Она вычисляет, сколько зернышек приходится на каждую клетку и подсчитывает общую сумму. Так как вы вряд ли следите за объемами ежегодных урожаев пшеницы в Соединенных Штатах, программа сравнивает промежуточные суммы с урожаем, собираемым каждый год в этой стране.

Листинг 5.5. Программа `wheat.c`

```
/* wheat.c -- экспоненциальный рост */
#include <stdio.h>
#define SQUARES 64          /* количество квадратов шахматной доски */
#define CROP 1E15           /* урожай пшеницы в США в зерньшках */
int main(void)
{
   double current, total;
   int count = 1;

   printf("квадрат    добавлено    итого     ");
   printf("процент от \n");
   printf("           зерен        зерен     ");
   printf("урожая в США\n");
   total = current = 1.0;     /* начинаем с одного зерньшка */
   printf("%4d %13.2e %12.2e %12.2e\n", count, current, total, total/CROP);
   while (count < SQUARES)
   {
      count = count + 1;
      current = 2.0 * current;
          /* удвоить количество зерен на следующем квадрате */
      total = total + current;   /* обновить итоговую сумму */
      printf("%4d %13.2e %12.2e %12.2e\n", count, current, total, total/CROP);
   }
   printf("Вот и все.\n");

   return 0;
}
```

Вначале выходные данные не должны были вызывать у правителя беспокойство:

```
квадрат  добавлено    итого     процент от
         зерен        зерен     урожая в США
   1     1.00e+00    1.00e+00    1.00e-15
   2     2.00e+00    3.00e+00    3.00e-15
   3     4.00e+00    7.00e+00    7.00e-15
   4     8.00e+00    1.50e+01    1.50e-14
   5     1.60e+01    3.10e+01    3.10e-14
```

```
   6     3.20e+01    6.30e+01    6.30e-14
   7     6.40e+01    1.27e+02    1.27e-13
   8     1.28e+02    2.55e+02    2.55e-13
   9     2.56e+02    5.11e+02    5.11e-13
  10     5.12e+02    1.02e+03    1.02e-12
```

На четырех клетках мудрец получил немного более тысячи зерен пшеницы, однако, посмотрите, что получилось на клетке 50!

```
  50     5.63e+14    1.13e+15    1.13e+00
```

Добыча мудреца превосходит весь ежегодный урожай США! Если вы хотите, что произойдет на 64-й ячейке, выполните эту программу сами.

Приведенный пример служит красноречивой иллюстрацией явления экспоненциального роста. Население мира и использование энергетических ресурсов растет по тому же закону.

Операция деления: /

В языке C символ / используется для обозначения деления. Значение слева от символа / делится на значение, указанное справа. Например, следующее выражение дает в результате `4.0`:

```
four = 12.0/3.0;
```

Действия операции деления на целочисленных типах отличаются от действий на типах с плавающей запятой. При делении типов с плавающей запятой получаем ответ в виде числа с плавающей запятой, в то время как деление целых чисел дает целочисленный результат. Целое число не может иметь дробной части, и это обстоятельство делает деление 5 на 3 неточным, поскольку ответ не содержит дробной части. В C любая дробная часть, полученная при делении двух целых чисел, отбрасывается. Этот процесс называется *усечением*.

Выполните программу, представленную в листинге 5.6, чтобы посмотреть, как работает усечение и чем отличается деление целых чисел от деления чисел с плавающей запятой.

Листинг 5.6. Программа `divide.c`

```
/* divide.c -- деление, каким мы его знаем */
#include <stdio.h>
int main(void)
{
   printf("Целочисленное деление:       5/4   равно %d \n", 5/4);
   printf("Целочисленное деление:       6/3   равно %d \n", 6/3);
   printf("Целочисленное  деление:      7/4   равно %d \n", 7/4);
   printf("Деление с плавающей запятой: 7./4. равно %1.2f \n", 7./4.);
   printf("Смешанное деление:           7./4  равно %1.2f \n", 7./4);
   return 0;
}
```

Листинг 5.6 включает случай “смешанных типов”, когда значение с плавающей запятой делится на целое число. C является более либеральным языком программирования по сравнению с некоторыми другими и позволяет вам выполнять такие опера-

ции, однако в обычных ситуациях желательно избегать смешивания типов. Теперь рассмотрим результаты выполнения этой программы:

```
Целочисленное деление:        5/4   равно 1
Целочисленное деление:        6/3   равно 2
Целочисленное  деление:       7/4   равно 1
Деление с плавающей запятой: 7./4. равно 1.75
Смешанное деление:            7./4  равно 1.75
```

Обратите внимание, что деление целых чисел не выполняет округления до ближайшего целого, оно в любом случае отбрасывает дробную часть (то есть выполняет усечение). Если вы смешиваете в одной операции целые числа и числа с плавающей запятой, вы получаете такой же ответ, как и в случае деления чисел с плавающей запятой. На самом деле компьютер не способен делить число с плавающей запятой на целое число, и в силу этого обстоятельства компилятор приводит оба операнда к единому типу. В рассматриваемом случае целое число преобразуется в число с плавающей запятой до того, как будет выполнена операция деления.

До появления стандарта C99 язык C предоставлял разработчикам реализаций некоторую свободу при решении вопроса, как выполняется деление с использованием отрицательных чисел. Следует иметь в виду, что процедура округления предусматривает определение наибольшего целого значения, меньшего или равного числу с плавающей запятой. Разумеется, 3 удовлетворяет этому требованию, если его сравнивать с числом 3.8. Но как быть в случае –3.8? Метод наибольшего целого числа предполагает его округление до –4, поскольку –4 меньше, чем –3.8. Однако в другой трактовке процесс округления просто отбрасывает дробную часть, то есть происходит *усечение в направлении нуля*, в результате которого число –3.8 преобразуется в –3. До появления стандарта C99 одни реализации использовали первый подход, другие — второй. Однако стандарт C99 однозначно требует усечения в направлении нуля, следовательно, –3.8 преобразуется к –3.

Свойства операции деления целых чисел оказываются очень удобными для решения некоторых классов задач, и вы вскоре увидите соответствующий пример. Во-первых, существует еще один важный аспект: что произойдет, если вы включите сразу несколько операций в один оператор? Это наша следующая тема.

Приоритеты операций

Рассмотрим следующую строку кода:

```
butter = 25.0 + 60.0 * n / SCALE;
```

В этом операторе присутствуют операции сложения, умножения и деления. Какая из них выполнится первой? Возможно, следует к `25.0` прибавить `60.0`, полученный результат `85.0`, умножить на `n`, после чего полученное произведение разделить на `SCALE`? Или же `60.0` необходимо умножить на `n`, к полученному произведению прибавить `25.0`, после чего результат сложения разделить на `SCALE`? Возможно, существует какой-то другой порядок выполнения арифметических операций? Пусть `n` равно `6.0`, а `SCALE` равно `2.0`. Если вы выполните рассматриваемый оператор с упомянутыми значениями, то в условиях первого подхода вы получите результат `255`. Второй подход даст в результате `192.5`. Программа на C, по-видимому, использует какой-то другой порядок следования операций, ибо она присваивает переменной `butter` значение `205.0`.

Разумеется, порядок выполнения различных операций приводит к различиям, следовательно, в языке C должны быть определены однозначные правила, и выбор того, что должно делаться в первую очередь, должен осуществляться в соответствие с этими правилами. Язык C решает эту проблему, устанавливая порядок выполнения операций. Каждой операции назначается соответствующий приоритет. Как и в обычной арифметике, умножение и деление имеют более высокий приоритет, чем сложение и вычитание, поэтому они выполняются первыми. Но что делать, если две операции имеют один и тот же приоритет? Если они совместно используют один и тот же операнд, они выполняются в порядке, в каком они следуют в операторе. Для большинства операций определяется порядок следования слева направо. (Операция = является в этом случае исключением.) По этой причине в операторе

```
butter = 25.0 + 60.0 * n / SCALE;
```

устанавливается следующий порядок выполнения операций:

`60.0 * n`	Первая операция * или / в выражении (исходя из предположения, что n равно 6, так что `60.0 * n` дает в результате `360.0`).
`360.0 / SCALE`	Затем идет вторая операция * или / в выражении.
`25.0 + 180`	В завершение (поскольку SCALE равно `2.0`) первая операция + или – в выражении дает в результате `205.0`.

Многие предпочитают представлять порядок вычислений в виде диаграммы, получившей название дерева выражения. На рис. 5.3 представлен пример такой диаграммы. Диаграмма показывает, как исходное выражение постепенно сводится к одному значению.

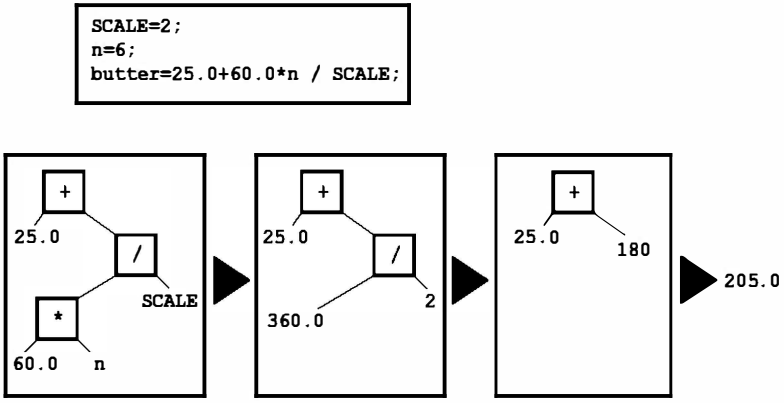


Рис. 5.3. *Деревья выражений, показывающие операции, операнды и порядок выполнения вычислений*

А что если вы захотите, чтобы операция сложения выполнялась до операции деления? Тогда вы должны поступить так, как это сделано в следующей строке:

```
flour = (25.0 + 60.0 * n) / SCALE;
```

То, что заключено в круглые скобки, выполняется первым. Внутри круглых скобок действуют обычные правила. Например, сначала выполняется умножение, а затем

сложение. Эти операции позволяют вычислить выражение в круглых скобках. Теперь результат этих вычислений делится на SCALE.

Использованные нами на текущий момент операции сведены в табл. 5.1.

Таблица 5.1. Операции в порядке уменьшения приоритетов

Операции	*Ассоциативность*
()	Слева направо
+ - (унарные)	Справа налево
* /	Слева направо
+ - (бинарные)	Слева направо
=	Справа налево

Обратите внимание, что две формы использования знака "минус" имеют различные приоритеты, это же справедливо и для знака "плюс". В столбце ассоциативности показано, как операции связаны со своими операндами. Например, унарная операция - ассоциируется с числовым значением, стоящим справа, в то время как при делении операнд слева делится на операнд справа.

Приоритеты и порядок вычисления

Приоритеты операций являются жизненно важным правилом для определения порядка вычисления выражения, в то же время нет необходимости в определении всей последовательности вычислений. Язык C предоставляет возможность выбора при реализации. Рассмотрим следующий оператор:

```
y = 6 * 12 + 5 * 20;
```

Приоритет устанавливает порядок вычислений в тех случаях, когда две операции совместно используют один и тот же операнд. Например, 12 является операндом как для операции *, так и для операции +, а приоритеты операций говорят, что первым должно выполняться умножение. Аналогично, приоритет показывает, что 5 используется в операции умножения, но не в операции сложения. Короче говоря, операции умножения 6 * 12 и 5 * 20 выполняются до того, как начнут выполняться операция сложения. При этом приоритет не устанавливает, какая из двух операций умножения выполняется первой. Язык C оставляет выбор за реализацией, поскольку один выбор может оказаться более эффективным для одного типа аппаратных средств, а другой выбор обеспечивает более высокую производительность на другом оборудовании. В любом случае, рассматриваемое выражение сводится к вычислению 72 + 100, следовательно, выбор не отражается на конечном результате в условиях этого конкретного примера. Однако вы можете возразить: "Умножение выполняется слева направо. Не означает ли это, что самое левое умножение исполняется первым?" (Возможно, вы этого и не скажете, но кто-нибудь когда-нибудь обязательно это скажет.) Правило ассоциативности применяется к операциям, которые совместно используют конкретный операнд. Например, в выражении 12 / 3 * 2 операции / и *, обладающие одним и тем же приоритетом, совместно используют операнд 3.

В силу этого обстоятельства применяется правило "слева направо", и выражение сводится к вычислению произведения `4 * 2`, что в результате дает `8`. (Обход справа налево дает `12 / 6` или `2`. В этом случае выбор имеет существенное значение.) В предыдущем примере две операции `*` не претендуют на общий операнд, в связи с чем правило "слева направо" не применяется.

Проверка приоритета операций

Попытаемся применить эти правила в условиях более сложного примера (листинг 5.7).

Листинг 5.7. Программа `rules.c`

```
/* rules.c -- проверка, как работают приоритеты */
#include <stdio.h>
int main(void)
{
   int top, score;
   top = score = -(2 + 5) * 6 + (4 + 3 * (2 + 3));
   printf("top = %d \n", top);
   return 0;
}
```

Какое значение распечатает эта программа? Сначала вычислите его вручную, а затем выполните программу или прочитайте приведенное ниже пояснение, дабы проверить свой ответ.

Прежде всего, наивысший приоритет присваивается вычислению в круглых скобках. Какое выражение в скобках, `-(2 + 5) * 6` или `(4 + 3 * (2 + 3))` вычисляется первым, зависит, как мы только что установили, от конкретной реализации. Любой выбор в этом примере приводит к одному и тому же результату, так что пусть выражение слева вычисляется первым. Более высокий приоритет выражения в скобках означает, что в подвыражении `-(2 + 5) * 6` вы сначала вычисляете сумму `(2 + 5)`, которая равна `7`. Далее вы применяете к `7` унарную операцию минус, получая при этом `-7`. Теперь данное выражение принимает вид

```
top = score = -7 * 6 + (4 + 3 * (2 + 3))
```

На следующем шаге следует вычислить `2 + 3`. Выражение принимает вид

```
top = score = -7 * 6 + (4 + 3 * 5)
```

Поскольку приоритет операции `*` выше, чем у `+`, выражение сводится к

```
top = score = -7 * 6 + (4 + 15)
```

а затем к

```
top = score = -7 * 6 + 19
```

Умножим `-7` на `6` и получим следующее выражение:

```
top = score = -42 + 19
```

Выполнив сложение, получим

```
top = score = -23
```

Теперь переменной `score` присваивается значение `-23`, и, наконец, `top` получает значение `-23`. Вспомните, что операция `=` выполняется справа налево.

Некоторые дополнительные операции

В языке C имеется около 40 операций, и некоторые из них употребляются гораздо чаще, чем другие. Те, с которыми мы уже ознакомились, относятся к наиболее часто используемым операциям, однако к этому списку можно добавить еще четыре очень полезных операции.

Операция `sizeof` и тип `size_t`

С операцией `sizeof` мы уже имели дело в главе 3. Вспомните, что операция `sizeof` возвращает размер ее операнда в байтах. (Вспомните также, что байт в языке C по определению является размерностью типа `char`. В прошлом в качестве такой размерности чаще всего использовалась порция в 8 бит, в то же время для представления некоторых наборов символов требовалось более одного байта.) Операндом может быть конкретный объект данных, такой как имя переменной, либо это может быть тип. Если это тип, например, `float`, операнд должен быть заключен в круглые скобки. В примере, представленном в листинге 5.8, показаны обе формы операндов.

Листинг 5.8. Программа `sizeof.c`

```
// sizeof.c -- использование операции sizeof
// Применение модификатора %z стандарта C99 -- попытайтесь воспользоваться
// %u или %lu, если вам не достаточно %zd
#include <stdio.h>
int main(void)
{
   int n = 0;
   size_t intsize;

   intsize = sizeof (int);
   printf("n = %d, n состоит из %zd байтов; все значения int имеют %zd байтов.\n",
          n, sizeof n, intsize );

   return 0;
}
```

Язык C утверждает, что операция `sizeof` возвращает значение типа `size_t`. Это тип целого числа без знака, но не абсолютно новый тип. Вместо этого, как и переносимые типы (`int32_t` и прочие), он определяется в терминах стандартных типов. В C имеется механизм `typedef` (подробно обсуждается в главе 14), который позволяет создавать альтернативные имена для существующих типов. Например,

```
typedef double real;
```

делает `real` еще одним именем для `double`. Теперь вы можете объявить переменную типа `real`:

```
real deal;     // используется typedef
```

Компилятор, который сталкивается со словом `real`, вспоминает, что оператор `typedef` сделал `real` альтернативным именем для `double` и создает `deal` как переменную типа `double`. Аналогично, система заголовочных файлов языка C может использовать `typedef`, чтобы сделать `size_t` синонимом `unsigned int` в одной системе и

unsigned long в другой. Таким образом, когда вы используете тип size_t, компилятор выполняет подстановку стандартного типа, который работает в вашей системе.

Стандарт C99 идет на шаг дальше и представляет %zd в качестве спецификатора функции printf() для вывода значения size_t. Если в вашей системе %zd не реализован, можно попытаться использовать вместо него %u или %lu.

Операция деления по модулю: %

Операция деления по модулю применяется в целочисленной арифметике. Ее результатом является остаток от деления целого числа, стоящего слева от знака операции, на число, расположенное справа от него. Например, 13 % 5 (читается как "13 по модулю 5") дает в результате 3, поскольку 5 содержится в 13 дважды с остатком, равным 3. Не пытайтесь выполнять эту операцию над числами с плавающей запятой. Она просто не работает.

На первый взгляд эта операция может кому-то показаться экзотическим инструментальным средством, предназначенным только для математиков, но, по сути дела, это очень удобная и полезная операция. Обычно она используется, чтобы помочь управлять ходом выполнения программы. Предположим, например, вы работаете над программой подготовки счетов, которая начисляет дополнительную плату каждые третий месяц. Для этого достаточно разделить номер месяца по модулю 3 (то есть, month % 3) и проверить, не равен ли результат 0. Если равен, то программа включает дополнительную плату. После того, как в главе 7 вы изучите, как работает оператор if, вы оцените все преимущества этой операции.

В листинге 5.9 предлагается еще один пример применения операции %. В нем также показан еще один способ использования цикла while.

Листинг 5.9. Программа min_sec.c

```
// min_sec.c -- переводит секунды в минуты и секунды
#include <stdio.h>
#define SEC_PER_MIN 60          // число секунд в минуты
int main(void)
{
   int sec, min, left;

   printf("Перевод секунд в минуты и секунды!\n");
   printf("Введите количество секунд (<=0 для выхода):\n");
   scanf("%d", &sec);           // читать количество секунд

   while (sec > 0)
   {
      min = sec / SEC_PER_MIN;  // усеченное количество минут
      left = sec % SEC_PER_MIN; // число количество в остатке
      printf("%d секунд - это %d минут %d секунд.\n", sec, min, left);
      printf("Введите следующее значение (<=0 для выхода):\n");
      scanf("%d", &sec);
   }

   printf("Сделано!\n");

   return 0;
}
```

Вот как выглядит пример выходных данных:

```
Перевод секунд в минуты и секунды!
Введите количество секунд (<=0 для выхода):
154
154 секунд - это 2 минут 34 секунд.
Введите следующее значение (<=0 для выхода):
567
567 секунд - это 9 минут 27 секунд.
Введите следующее значение (<=0 для выхода):
0
Сделано!
```

В коде из листинга 5.2 используется счетчик для управления циклом `while`. Как только счетчик превысит заданное значение, цикл завершается. Однако код в листинге 5.9 использует функцию `scanf()`, чтобы назначить новые значения переменной `sec`. Цикл продолжается до тех пор, пока это значение положительно. Когда пользователь вводит ноль или отрицательное значение, цикл завершается. Важной особенностью программы в обоих случаях является то, что каждая итерация цикла обновляет значение проверяемой переменной.

Что произойдет, если будет введено отрицательное значение? До того, как стандарт C99 установил для целочисленного деления правило "усечения в направлении нуля", существовало несколько возможностей. В случае реализации этого правила вы получаете отрицательное значение при делении по модулю, если первых операнд отрицательный, во всех остальных случаях вы получаете положительное значение:

`11 / 5` равно 2 и `11 % 5` равно 1

`11 / -5` равно 2 и `11 % -2` равно 1

`-11 / -5` равно 2 и `-11 % -5` равно -1

`-11 / 5` равно -2 и `-11 % 5` равно -1

Если поведение вашей системы будет другим, значит, она не соответствует требованиям стандарта C99. В любом случае, стандарт фактически утверждает, что если `a` и `b` являются целочисленными, то вы можете вычислить `a%b` путем вычитания `(a/b)*b` из `a`. Например, таким способом вы можете вычислить значение `-11%5`:

```
-11 - (-11/5) * 5 = -11 -(-2)*5 = -11 -(-10) = -1
```

Операции инкремента и декремента: ++ и --

Операция *инкремента* решает простую задачу; она увеличивает (инкрементирует) значение своего операнда на 1. Существует две разновидности этой операции. В первом случае символы ++ идут непосредственно перед переменной, это так называемая *префиксная* форма. Во втором случае символы ++ следуют сразу за переменной; это *постфиксная* форма. Эти две формы отличаются друг от друга по моменту выполнения инкремента. Сначала мы рассмотрим, в чем эти формы совпадают, а затем обратимся к различиям. Короткий пример, представленный в листинге 5.10, демонстрирует, как работает операция инкремента.

Листинг 5.10. Программа `add_one.c`

```
/* add_one.c -- инкремент: префиксный и постфиксный */
#include <stdio.h>
int main(void)
{
   int ultra = 0, super = 0;
   while (super < 5)
   {
      super++;
      ++ultra;
      printf("super = %d, ultra = %d \n", super, ultra);
   }
   return 0;
}
```

В результате выполнения программы `add_one.c` получены следующие выходные данные:

```
super = 1, ultra = 1
super = 2, ultra = 2
super = 3, ultra = 3
super = 4, ultra = 4
super = 5, ultra = 5
```

Программа дважды просчитала до пяти в одно и то же время. Вы могли бы получить тот же результат, заменив две операции инкремента следующими операторами присваивания:

```
super = super + 1;
ultra = ultra + 1;
```

Это очень простые операторы. Зачем создавать еще один такой же оператор, не говоря уже о двух, пусть даже в более компактном виде? Одна из причин заключается в том, что компактная форма позволяет создавать программы удобочитаемыми и более понятными. Такие операторы придают программе изящество и делают ее элегантной, а с такой программой всегда приятно иметь дело. Например, можно представить часть программы `shoes2.c` (листинг 5.2) в следующем виде:

```
shoe = 3.0;
while (shoe < 18.5)
{
   foot = SCALE * size + ADJUST;
   printf("%10.1f %20.2f дюймов\n", shoe, foot);
   ++shoe;
}
```

Тем не менее, и в этом случае вы не до конца воспользовались всеми преимуществами операции инкремента. Вы можете сделать этот фрагмент программы еще более компактным:

```
shoe = 2.0;
while (++shoe < 18.5)
{
   foot = SCALE*shoe + ADJUST;
   printf("%10.1f %20.2f дюймов\n", shoe, foot);
}
```

В данном случае процесс инкремента и сравнения цикла while объединены в одном выражении. Конструкции этого типа так часто используются в C, что заслуживают более пристального внимания. Во-первых, как работает такая конструкция? Достаточно просто. Значение переменной shoe каждый раз увеличивается на 1, а затем сравнивается с 18.5. Если ее величина меньше 18.5, то операторы, заключенные в фигурные скобки, выполняются один раз. Затем shoe опять увеличивается на 1, и цикл продолжается до тех пор, пока значение shoe не станет достаточно большим. Потребуется уменьшить начальное значение shoe с 3.0 до 2.0, чтобы скомпенсировать инкремент переменной shoe, сделанный перед тем, как первый раз было вычислено значение переменной foot (см. рис. 5.4).

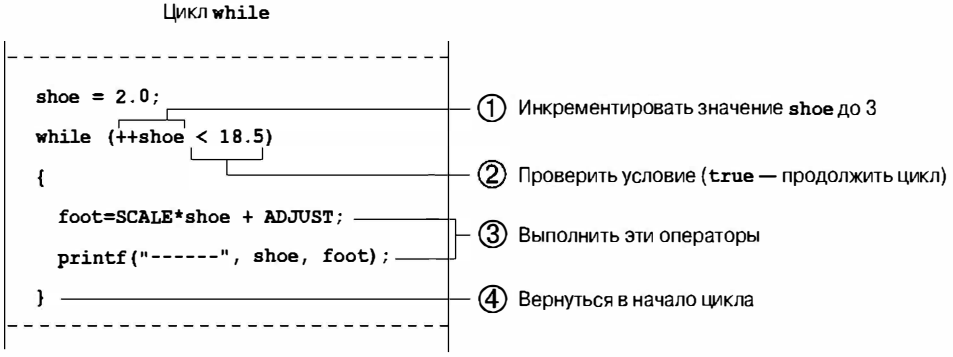


Рис. 5.4. *Одна итерация цикла*

Во-вторых, в чем преимущества такого подхода? Прежде всего, в компактности программы. Но что важнее — он объединяет в одном месте два процесса, которые управляют выполнением цикла. Первичный процесс — это проверка конца цикла: нужно ли продолжать выполнение цикла? В рассматриваемом случае проверка выясняет, меньше ли значение переменной size, чем 18.5. Вторичный процесс меняет значение объекта проверки, в данном случае размер обуви увеличивается на единицу.

Предположим, что вы не предусмотрели изменение размера обуви. Тогда значение переменной shoe всегда будет меньше 18.5, а цикл никогда не завершится. Угодивший в бесконечный цикл компьютер раз за разом будет выводить одну и ту же строку. В конце концов, вы потеряете интерес к выходным данным и будете вынуждены каким-нибудь способом прервать выполнение программы. То обстоятельство, что и проверка условия цикла, и изменение параметра цикла находятся в одном, а не в разных местах, лишний раз напоминает о том, что необходимо изменить значение параметра цикла.

Недостаток сочетания двух операций в одном выражении проявляется в том, что восприятие программного кода в этом случае затрудняется, а вероятность возникновения программных ошибок возрастает.

Еще одно достоинство операции инкремента состоит в том, что при ее использовании несколько возрастает эффективность кода в машинном языке, поскольку она подобна фактическим командам машинного языка. Однако, по мере того, как поставщики программного обеспечения создают все более эффективные компиляторы языка C, это преимущество постепенно сходит на нет. Сообразительный компилятор может распознавать, что операцию x = x + 1 следует трактовать как ++x.

И, наконец, эти операции имеют дополнительную особенность, которая может оказаться полезной в некоторых деликатных ситуациях. Чтобы узнать, что это за особенность, попытайтесь выполнить программу, представленную в листинге 5.11.

Листинг 5.11. Программа `post_pre.c`

```
/* post_pre.c -- постфиксная и префиксная формы */
#include <stdio.h>
int main(void)
{
   int a = 1, b = 1;
   int aplus, plusb;

   aplus = a++; /* постфиксная форма */
   plusb = ++b; /* префиксная форма */
   printf("a   aplus   b   plusb \n");
   printf("%1d %5d %5d %5d\n", a, aplus, b, plusb);

   return 0;
}
```

Если вы и ваш компилятор все выполнили правильно, то должны получить следующий результат:

```
a   aplus   b   plusb
2     1     2     2
```

Как и предполагалось, значения переменных `a` и `b` увеличились на 1. В то же время, значение переменной `aplus` — это значение `a`, которое эта переменная имела *до того*, как произошло изменение `a`, а значение переменной `plusb` — это значение `b` *после того*, как произошло изменение `b`. Именно в этом и состоит различие между префиксной и постфиксной формами операции инкремента (рис. 5.5).

```
aplus = a++; /* постфиксная форма: значение a меняется после того,
                как оно использовано */
plusb = ++b; /* префиксная форма: значение a меняется до того,
                как оно использовано */
```

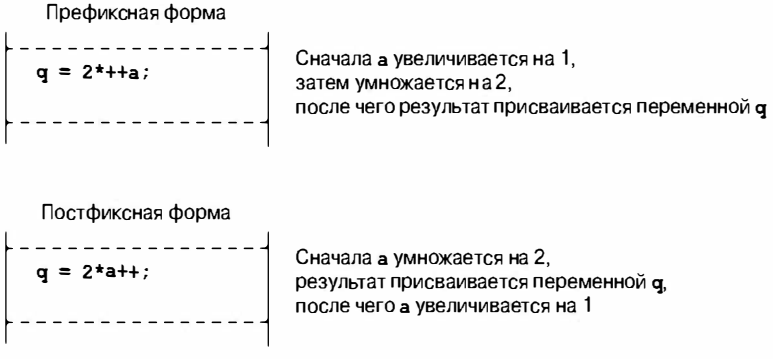


Рис. 5.5. *Префиксная и постфиксная формы инкремента*

В тех случаях, когда одна из этих операций инкремента использовалась сама по себе, как, например, в одиночном операторе `ego++;`, не имеет значения, какой ее форме вы отдали предпочтение. Однако выбор имеет смысл, когда сама операция и ее операнд являются частью некоторого выражения, например, такого, какое вы только что видели в операторах присваивания. В ситуациях подобного рода вы должны иметь четкое представление о том, какой результат вы хотите получить. В качестве примера напомним, что мы намеревались использовать следующую конструкцию:

```
while (++shoe < 18.5)
```

Такая проверка условия конца цикла позволяет получить таблицу до размера 18. Если вы воспользуетесь операцией `shoe++` вместо `++shoe`, размер таблицы возрастет до 19, поскольку значение `shoe` должно быть увеличено после сравнения, но никак не до сравнения. Разумеется, вы всегда можете возвратиться к менее элегантной форме

```
shoe = shoe + 1;
```

но тогда никто не поверит, что вы являетесь истинным приверженцем языка C.

Читая эту книгу, вы должны обратить особое внимание на примеры операции инкремента. Спросите себя, использовали ли вы где-нибудь префиксную и постфиксную формы попеременно или обстоятельства ставили вас перед конкретным выбором. Возможно, более разумной является политика вообще избегать кода, в котором имеет смысл, какую форму вы изберете, префиксную или постфиксную. Например, вместо

```
b = ++i; // получим другое значение b, если использовать i++
```

использовать

```
++i;     // строка 1
b = i;   // b получит то же значение, как если бы в строке 1 было i++
```

С другой стороны, иногда ради шутки можно позволить себе немного безрассудства, так что данная книга не всегда следует этому благоразумному совету.

Декремент: --

Для каждой формы операции инкремента существует соответствующая операция декремента. При этом вместо ++ применяется обозначение --.

```
--count;   /* префиксная форма операции декремента */
count--;   /* постфиксная форма операции декремента */
```

Листинг 5.12 служит иллюстрацией того, что иногда компьютер может быть безнадежным лириком.

Листинг 5.12. Программа `bottles.c`

```
#include <stdio.h>
#define MAX 100
int main(void)
{
   int count = MAX + 1;
   while (--count > 0) {
      printf("%d бутылок минеральной воды на полке, "
             "%d бутылок минеральной воды!\n", count, count);
```

```
        printf("Возьмите одну из них и пустите по кругу,\n");
        printf("%d бутылок минеральной воды!\n\n", count - 1);
    }
    return 0;
}
```

Выходные данные начинаются со следующих фраз:

```
100 бутылок минеральной воды на полке, 100 бутылок минеральной воды!
Возьмите одну из них и пустите по кругу,
99 бутылок минеральной воды!

99 бутылок минеральной воды на полке, 99 бутылок минеральной воды!
Возьмите одну из них и пустите по кругу,
98 бутылок минеральной воды!
```

Все это продолжается некоторое время и закачивается следующим образом:

```
1 бутылок минеральной воды на полке, 1 бутылок минеральной воды!
Возьмите одну из них и пустите по кругу,
0 бутылок минеральной воды!
```

Как вы можете убедиться сами, у нашего безнадежного лирика не все в порядке с грамматикой, но это можно исправить, если воспользоваться уловными операторами, которые рассматриваются в главе 7.

Кстати, операция > означает "больше чем". Как и операция < ("меньше чем"), она является операцией отношения. Более подробно операции отношений рассматриваются в главе 6.

Приоритеты операций

Операции инкремента и декремента имеют очень высокий приоритет; только скобки обладают более высоким уровнем приоритета. В силу этого, `x*y++` означает `(x)*(y++)`, но никак не `(x*y)++`, что весьма кстати, ибо последнее выражение не имеет смысла. Операции инкремента и декремента применяются только к *переменным* (или, в общем случае, к модифицируемым l-значениям), а произведение `x*y` само по себе не является переменной, хотя сомножители таковыми являются.

Не путайте приоритеты этих двух операций с порядком вычисления. Предположим, мы имеем следующую последовательность операторов:

```
y = 2;
n = 3;
nextnum = (y + n++)*6;
```

Какое значение получит переменная `nextnum`? Подстановка значений дает следующее

```
nextnum = (2 + 3)*6 = 5*6 = 30
```

Значение `n` инкрементируется и получает значение 4 только после того, как эта переменная будет использована в выражении. Приоритет операции говорит о том, что операция ++ применяется только к `n`, но не к `y + n`. Кроме того, он указывает, когда значение `n` используется для вычисления выражения, но природа операции приращения определяет момент изменения значения `n`.

Когда n++ присутствует в выражении как его часть, можно считать, что она означает "использовать n, а затем увеличить его значение на единицу". С другой стороны, ++n означает "увеличить значение n на единицу, а затем его использовать".

Не будьте слишком самоуверенными

Вы можете попасть в весьма неловкое положение, если попытаетесь применять операцию инкремента там, где надо и где не надо. Например, вы, возможно, подумаете, что можно улучшить программу squares.c (листинг 5.4) для вывода на печать целых чисел, заменив цикл while в указанной программе следующим циклом:

```
while (num < 21)
   {
   printf("%10d %10d\n", num, num*num++);
   }
```

На первый взгляд это вполне оправдано. Вы печатаете число num, умножаете его само на себя, чтобы получить квадрат этого числа, а затем увеличиваете значение num на 1. По сути, эта программа может даже работать в некоторых системах, однако не на всех. Проблема заключается в том, что когда функция printf() готова принять очередные значения для последующего вывода на печать, она может вычислить сначала последний аргумент и увеличить значение num на единицу, прежде чем переходить к следующему аргументу. Поэтому, вместо того, чтобы печатать

```
5       25
```

она может напечатать

```
6       25
```

Она даже может работать справа налево, используя 5 в качестве крайнего правого num и 6 в качестве следующих двух, в результате чего получатся следующие выходные данные:

```
6       30
```

Компилятор языка C может выбирать сам, какой аргумент функции должен быть вычислен первым. Такая свобода выбора повышает производительность компилятора, но в то же время она может стать причиной проблем, если операция инкремента применяется к аргументу функции.

Другим возможным источником неприятностей может стать оператор следующего вида:

```
ans = num/2 + 5*(1 + num++);
```

И снова проблема состоит в том, что компилятор может выполнять действия не в том порядке, в каком вы предполагаете. Вы, возможно, подумаете, что он сначала найдет выражение num/2 и только потом перейдет к вычислению другой части выражения, однако он вполне может первым вычислить последний элемент, увеличить значение num, и использовать это новое значение для вычисления num/2. На этот счет не существует никаких гарантий.

Еще одним возможным источником проблем может стать и такая конструкция:

```
n = 3;
y = n++ + n++;
```

Разумеется, после выполнения этого оператора значение переменной n увеличится на 2, зато значение y определяется неоднозначно. Компилятор может использовать старое значение n дважды при вычислении значения y, а затем дважды увеличить значение n на единицу.

При этом y принимает значение 6, а n — значение 5, либо он может использовать старое значение один раз, увеличить значение n один раз, использовать это значение для второго n в выражении, а затем инкрементировать n второй раз. В этом случае y принимает значение 7, а n — значение 5. Возможен и тот, и другой вариант. А если называть вещи своими именами, то результат этих вычислений не определен, и, следовательно, это означает, что стандарт языка C не способен определить, каким должен быть результат.

Вы можете легко избежать описанных выше проблем:

- Не применяйте операции инкремента и декремента к переменной, которая является частью более одного аргумента функции.
- Не применяйте операции инкремента и декремента к переменной, которая появляется в выражении более одного раза.

С другой стороны, в языке C имеются средства, которые обеспечивают правильное выполнение операций инкремента и декремента. Мы вернемся к этой теме, когда будем обсуждать точки оценки в разделе "Побочные эффекты и точки оценки" далее в этой главе.

Выражения и операторы

Мы использовали термины "*выражение*" и "*оператор*" на протяжении нескольких первых глав, а теперь настало время глубже обсудить смысл этих терминов. Операторы образуют основные элементы программы на C, а большая часть операторов состоит из выражений. Отсюда следует, что в первую очередь нам нужно подробно изучить выражения.

Выражения

Выражение представляет собой некоторую комбинацию операций и операндов. (Вспомните, что операнд есть то, над чем выполняется операция.) Простейшим выражением является отдельный операнд, он может служить отправной точкой для построения более сложных выражений. Ниже представлено несколько примеров выражений:

```
4
-6
4+21
a*(b + c/d)/20
q = 5*2
x = ++q % 3
q > 3
```

Легко заметить, что операндами могут быть константы, переменные и их сочетания. Некоторые выражения представляют собой комбинации выражений меньших

размеров, которые мы назовем *подвыражениями*. Например, в четвертом примере `c/d` выступает в качестве подвыражения.

Каждое выражение имеет значение

Важное свойство языка C состоит в том, что каждое выражение в этом языке имеет значение. Чтобы найти это значение, нужно выполнить операции в порядке, определенном приоритетами выражений. Значения нескольких первых приведенных выше выражений очевидны, однако что можно сказать о выражениях со знаком =? Эти выражения просто принимают те же значения, что и переменные слева от знака =. Поэтому выражение `q=5*2` как единое целое получает значение `10`. А что можно сказать о выражении `q > 3`? Такие выражения отношения получают значение `1`, если выражение истинно, и `0`, если оно ложно. Рассмотрим несколько выражений и их значения:

Выражение	*Значение*
`-4 + 6`	2
`c = 3 + 8`	11
`5 > 3`	1
`6 + (c = 3 + 8)`	17

Последнее выражение не может не показаться странным! Однако оно вполне допустимо в C (тем не менее, использовать подобные выражения не рекомендуется), и представляет собой сумму двух выражений, каждое из которых имеет собственное значение.

Операторы

Операторы служат основными строительными блоками программы. Программа — это последовательность операторов с необходимыми знаками пунктуации. Оператор есть законченная команда компьютеру. В языке C операторы распознаются по точке с запятой в конце. В силу этого обстоятельства

```
legs = 4
```

это всего лишь выражение (которое может быть частью другого выражения), в то же время

```
legs = 4;
```

является оператором.

Какой будет команда в завершенном виде? Во-первых, C трактует любое выражение как оператор, если только оно оканчивается точкой с запятой. По этой причине C не отвергает строки, подобные показанным ниже:

```
8;
3 + 4;
```

Однако эти операторы не производят никаких действий в программе и, по сути дела, не могут рассматриваться как операторы, имеющие хоть какой-нибудь смысл.

Обычно операторы меняют значения переменных и вызывают функции:

```
x = 25;
++x;
y = sqrt(x);
```

Хотя оператор (или, по меньшей мере, оператор, имеющий смыл) — это завершенная команда, не все завершенные команды являются операторами. Рассмотрим следующий оператор:

```
x = 6 + (y = 5);
```

В него входит подвыражение `y = 5`, представляющее завершенную команду, но это только часть оператора. Поскольку завершенная команда не обязательно есть оператор, для распознавания команд, которые на самом деле представляют собой операторы, необходима точка с запятой.

До сих пор мы сталкивались с четырьмя видами операторов. В листинге 5.13 представлен короткий пример, в котором используются все эти четыре вида операторов.

Листинг 5.13. Программа addemup.c

```
/* addemup.c -- четыре вида операторов */
#include <stdio.h>
int main(void)              /* находит сумму первых 20 натуральных чисел */
{
   int count, sum;          /* оператор объявления                        */
   count = 0;               /* оператор присваивания                      */
   sum = 0;                 /* то же самое                                */
   while (count++ < 20)     /* оператор цикла while                       */
      sum = sum + count;    /* операторы                                  */
   printf("sum = %d\n", sum); /* оператор вызова функции                  */
   return 0;
}
```

Рассмотрим листинг 5.13 более внимательно. На данный момент вы должны быть достаточно хорошо знакомы с оператором объявления. Тем не менее, следует вспомнить, что он вводит в употребление имена и типы переменных, а также выделяет под них память. Обратите внимание, что оператор объявления не является оператором выражения. То есть, если вы удалите из объявления точку с запятой, вы получите нечто такое, что не является выражением и не имеет значения:

```
int port    /* это не выражение, оно не имеет значения */
```

Оператор присваивания — это "рабочая лошадка" многих программ; он присваивает значения переменной. Этот оператор состоит из имени переменной, за которым следует операция присваивания (=), далее идет выражение, сопровождаемое точкой с запятой. Обратите внимание, что оператор `while` в примере содержит в себе оператор присваивания. Оператор присваивания представляет собой пример оператора выражения.

Оператор вызова функции заставляет функцию делать то, для чего она предназначена. В рассматриваемом примере функция `printf()` вызывается для того, чтобы распечатать некоторые результаты.

В операторе while имеются три различные части, как показано на рис. 5.6. Первой частью является ключевой слово while. Далее, в круглые скобки заключено проверяемое условие. В завершение имеется оператор, который выполняется, если условие истинно. В цикле присутствует только один оператор. Это может быть простой оператор, как это имеет место в рассматриваемом примере, в таком случае фигурные скобки не нужны для его обозначения, либо это может быть составной оператор, как в некоторых примерах, рассмотренных выше, в которых фигурные скобки были нужны. О составных операторах речь пойдет несколько позже.

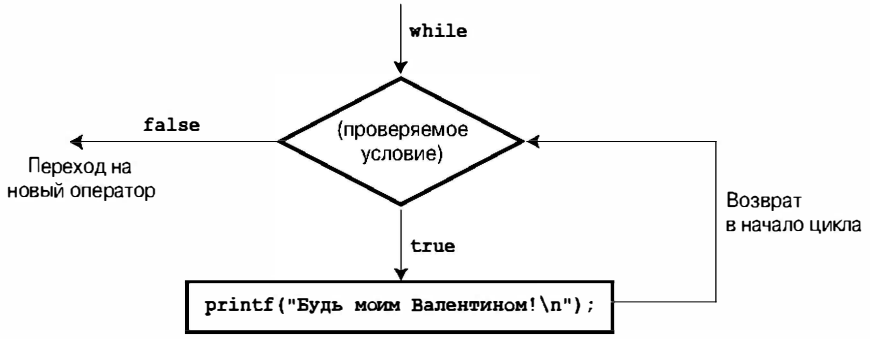


Рис. 5.6. *Структура простого цикла* while

Оператор while принадлежит к классу операторов, который иногда называют структурированными операторами, поскольку они обладают более сложной структурой, чем простой оператор присваивания. В последующих главах вы столкнетесь с множеством других структурированных операторов.

Побочные эффекты и точки оценки

Введем несколько новых терминов языка C. *Побочный эффект* представляет собой модификацию некоторого объекта данных или файла. Например, побочный эффект оператора `states = 50;` заключается в том, что он присваивает переменной states значение 50. Но почему "побочный эффект"? Это, скорее, является основным назначением этого оператора! Тем не менее, что касается языка C, то основная цель операторов состоит в вычислении выражений. Введите в C выражение `4 + 6`, и C вычислит его значение, равное 10. Введите выражение `states = 50`, и C вычислит его значение, которое равно 50. Вычисление этого выражения имеет своим побочным эффектом то, что переменная states изменяет свое прежнее значение на 50. Операции инкремента и декремента, как и операция присваивания, имеют свои побочные эффекты и используются, прежде всего, из-за наличия этого побочного эффекта.

Точка оценки — это точка в ходе выполнения программы, в которой производится вычисление всех побочных эффектов, прежде чем переходить к следующему действию. В языке C точку оценки обозначает точка с запятой в операторах. Это означает, что все изменения, вызванные операциями присваивания, инкремента и декремента в некотором операторе должны быть выполнены до того, как программа перейдет к следующему оператору. Некоторые операции, которые будут рассматриваться в следующих главах, также имеют точки оценки. Кроме того, конец любого полного оператора также является точкой оценки.

Что такое полное выражение? *Полное выражение* — это выражение, которое не является подвыражением более крупного выражения. Примерами полных выражений могут служить выражения оператора присваивания, а также выражения, используемые в условии проверки цикла `while`.

Точки оценки позволяют выяснить, когда происходит инкрементирование в постфиксной форме. Рассмотрим в качестве примера следующий код:

```
while (guests++ < 10)
printf("%d \n", guests);
```

Иногда начинающие программисты полагают, что фраза "использовать значение, а затем инкрементировать его" означает в данном контексте увеличение значения переменной после того, как она будет использована в операторе `printf()`. Однако выражение `guests++ < 10` представляет собой полное выражение, так как оно является проверяемым условием цикла `while`, поэтому конец этого выражения является точкой оценки. Следовательно, язык C гарантирует, что побочный эффект (приращение значения `guests`) произойдет до того, как программа перейдет к выполнению `printf()`. Однако, использование постфиксной формы обеспечивает то, что переменная `guests` получит приращение после того, как произойдет сравнение ее со значением 10.

Теперь рассмотрим следующий оператор:

```
y = (4 + x++) + (6 + x++);
```

Выражение `4 + x++` не является полным выражением, следовательно, язык C не гарантирует, что значение `x` будет инкрементировано сразу после того, как будет вычислено подвыражение `4 + x++`. В данном случае полное выражение представлено целым оператором присваивания, при этом точка с запятой отмечает точку оценки, так что C может гарантировать только то, что значение `x` будет увеличено на единицу дважды к моменту перехода программы к выполнению следующего оператора. Язык C не уточняет, будет ли значение `x` инкрементировано после вычисления каждого подвыражения или после вычисления всех выражений, именно поэтому вы должны избегать операторов подобного рода.

Составные операторы (блоки)

Составной оператор — это два или большее число операторов, объединенных в единую группу с помощью фигурных скобок; его еще называют *блоком*. В программе `shoes2.c` блок используется для того, чтобы оператор `while` мог выполнить сразу несколько операторов как один. Рассмотрим следующие фрагменты программ:

```
/* фрагмент 1 */
index = 0;
while (index++ < 10)
   sam = 10 * index + 2;
printf("sam = %d\n", sam);
```

```
/* фрагмент 2 */
index = 0;
while (index++ < 10)
{
   sam = 10 * index + 2;
   printf("sam = %d\n", sam);
}
```

Внутри фрагмента 1 в цикл while входит только оператор присваивания. В отсутствие фигурных скобок область действия оператора while распространяется от ключевого слова while до следующей точки с запятой. Функция printf() вызывается только один раз — по завершении цикла.

В рамках фрагмента 2 наличие фигурных скобок гарантирует, что оба оператора являются частью цикла while, а функция printf() вызывается при каждом выполнении цикла. Что касается структуры оператора while, то весь составной оператор рассматривается как единый оператор (рис. 5.7).

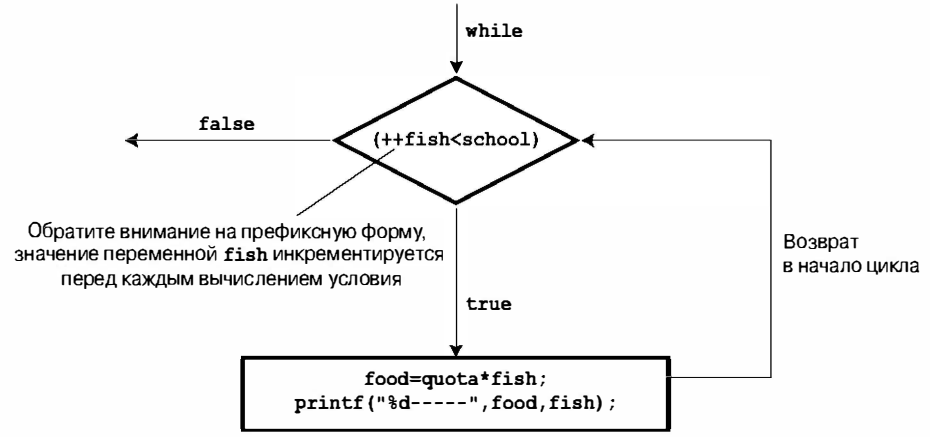


Рис. 5.7. *Цикл while с составным оператором*

Советы касательно стиля

Еще раз рассмотрим оба фрагмента, в которых присутствует цикл while, и внимательно присмотримся к тому, как используются отступы от левого поля в каждом из этих циклов. Для компилятора отступы в строке не имеют никакого значения, принимая решение относительно того, как интерпретировать заданную команду, он учитывает только фигурные скобки и знания о структуре цикла while, которые в него заложены. Отступы в данном случае служат для того, чтобы облегчить зрительное восприятие организации программы.

Приведенный выше пример демонстрирует широко распространенный способ расстановки фигурных скобок в случае использования оператора блока или, иначе говоря, составного оператора. Другой, не менее распространенный способ выглядит следующим образом:

```
while (index++ < 10) {
   sam = 10*index + 2;
   printf("sam = %d\n", sam);
}
```

Этот стиль акцентирует внимание на принадлежности блока к циклу while. Предыдущий стиль акцентирует внимание на том, что несколько операторов образуют блок. Повторим еще раз: что касается компилятора, то оба способа идентичны.

Подводя итоги, скажем: используйте отступы как инструмент, позволяющий сделать структуру программы более понятной для читателя.

Сводка: выражения и операторы

Выражения:

Выражение представляет собой некоторую комбинацию операций и операндов. Простейшим выражением является константа или переменная без операции, например, `22` или `beebop`.

Более сложные выражениями являются `55 + 22` и `vap = 2 * (vip + (vup = 4))`.

Операторы:

Оператор — это команда компьютеру. Операторы бывают простыми и составными. *Простые операторы* завершаются точкой с запятой, как показано в следующих примерах:

Оператор объявления: `int toes;`

Оператор присваивания: `toes = 12;`

Оператор вызова функции: `printf("%d\n", toes);`

Структурированный оператор:
```
while (toes < 20)
   toes = toes + 2;
```

Пустой (NULL) оператор: `; /* ничего не делает */`

Составные операторы, или *блоки*, состоят из одного или большего числа операторов (которые сами могут быть составными операторами), заключенных в фигурные скобки. Приведенный ниже пример оператора `while` содержит составной оператор:

```
while (years < 100)
{
   wisdom = wisdom * 1.05;
   printf("%d %d\n", years, wisdom);
   years = years + 1;
}
```

Преобразования типов

Операторы и выражения в общем случае должны использовать одни и те же типы выражений и констант. Если вы, однако, в одном и том же выражении употребляете разные типы, то выполнение программы на C не останавливается, как это имеет место, скажем, с программами на языке Pascal. Вместо этого используется набор правил, обеспечивающих автоматическое преобразование типов данных. С одной стороны, это очень удобно, но в то же время это может стать источником проблем, особенно если вы допускаете смешивание типов по причине невнимательности. (Программа `lint`, входящая в состав многих Unix-систем, выполняет проверку на наличие конфликтов между типами. Многие компиляторы C, не предназначенные для работы под управлением Unix, будут сообщать о возможных проблемах, если вы выберете высокий уровень защиты от ошибок.) Программистам рекомендуется хотя бы в общих чертах ориентироваться в правилах преобразования типов.

Ниже описаны базовые правила преобразования типов данных.

1. Если в выражение входят типы `char` и `short`, причем оба они могут быть как `signed` (со знаком) так и `unsigned` (без знака) они автоматически преобразуются в тип `int` или, при необходимости, в `unsigned int`. (Если тип `short` имеет тот же размер, что и `int`, то размер типа `unsigned short` больше, чем размер `int`; в этом случае `unsigned short` преобразуется в `unsigned int`.) Согласно

правилам K&R C, но не в текущей версии языка C, тип `float` автоматически преобразуется в `double`. Поскольку эти преобразования приводят к появлению более крупного размера данных, они называются *повышением типа*.

2. Если какая-либо операция выполняется над данными двух типов, оба типа приводятся к высшему из двух этих типов.
3. Последовательность типов, упорядоченных по принципу от высшего к низшему, выглядит так: `long double`, `double`, `float`, `unsigned long long`, `long long`, `unsigned long`, `long`, `unsigned int` и `int`. Возможно одно исключение, когда `long` и `int` имеют один и тот же размер, в этом случае `unsigned int` превосходит `long`. Типы `short` и `char` не присутствуют в этом списке, поскольку они уже могли быть повышены до `int` или, возможно, до `unsigned int`.
4. В операторе присваивания окончательный результат вычислений преобразуется к типу переменной, которой присваивается этот результат. Процесс может привести к повышению типа в соответствии с правилом 1 или к *понижению типа*, вследствие чего значение приводится к более низкому типу.
5. При использовании в качестве аргументов типов `char` и `short` преобразуются к `int`, а `float` — к `double`. Это автоматическое повышение типов может быть отменено с помощью прототипирования функций, как будет показано в главе 9.

Повышение в общем случае представляет собой гладкий процесс, протекающий без осложнений, в то время как понижение типа может привести к серьезным последствиям. Причина проста: тип более низкого типа может оказаться недостаточным, чтобы содержать в памяти число полностью. 8-разрядная переменная типа `char` может принимать целочисленное значение `101`, но не целое число `22334`. Когда типы с плавающей запятой приводятся с понижением к целому типу, они усекаются либо округляются до нуля. Это означает, что два разных числа `23.12` и `23.99` в результате усечения принимают одно и то же значение `23` и что `-23.5` усекается до `-23`. В листинге 5.14 показано, как работают эти правила.

Листинг 5.14. Программа `convert.c`

```
/* convert.c -- автоматическое преобразование типов */
#include <stdio.h>
int main(void)
{
   char ch;
   int i;
   float fl;
   fl = i = ch = 'C';                                    /* строка 9  */
   printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl); /* строка 10 */
   ch = ch + 1;                                          /* строка 11 */
   i = fl + 2 * ch;                                      /* строка 12 */
   fl = 2.0 * ch + i;                                    /* строка 13 */
   printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl); /* строка 14 */
   ch = 5212205.17; /* строка 15 */
   printf("Теперь ch = %c\n", ch);
   return 0;
}
```

Выполнив программу convert.c, получим следующие результаты:

```
ch = C, i = 67, fl = 67.00
ch = D, i = 203, fl = 339.00
Теперь ch = -
```

В этой системе, в которой реализованы 8-разрядный тип char и 32-разрядный тип int, происходит следующее:

- **Строки 9 и 10** — символ 'C' запоминается как однобайтное ASCII-значение в переменной ch. Целочисленная переменная i получает целое значение, которое является преобразованием символа 'C' в целое число, равное 67, и хранится в 4 байтах памяти. И, наконец, переменная fl получает значение 67.00, которое является результатом преобразования целого числа 67 в число с плавающей запятой.
- **Строки 11 и 14** — значение 'C' символьной переменной преобразуется в целое число 67, к которому затем прибавляется 1. Полученное в результате 4-байтовое целое число 68 усекается до 1 байта и запоминается в переменной ch. При печати с использованием спецификатора %c число 68 интерпретируется как ASCII-код символа 'D'.
- **Строки 12 и 14** — при умножении на 2 значение переменной ch преобразуется в 4-байтовое целое (68). Полученное при этом целое значение (136) преобразуется в число с плавающей запятой, чтобы затем можно было сложить с fl. Результат (203.00f) преобразуется к типу int и запоминается в i.
- **Строки 13 и 14** — значение переменной ch ('D', или 68) преобразуется в тип с плавающей запятой, после чего становится возможным его умножение на 2.0. Значение i (203) преобразуется в значение с плавающей запятой, чтобы можно было выполнить сложение, а результат (339.00) запоминается в переменной fl.
- **Строки 15 и 16** — в этих строках предпринимается попытка понижения типа, когда переменной ch присваивается очень большое значение. После усечения ch получает значение, которое в коде ASCII интерпретируется как символ дефиса.

Операция приведения

По мере возможностей вы должны избегать автоматического преобразования типов, особенно понижения типов, но иногда такие преобразования оправданы, естественно, при условии, что принимаются все меры предосторожности. Преобразования типов, которые мы обсуждали до сих пор, выполняются автоматически. В то же время вы можете потребовать выполнения именно того типа преобразований, какой вам нужен, либо документировать факт, что вы знаете, что выполняете преобразование типа. Метод выполнения таких преобразований известен как *приведение типов* и предусматривает выполнение следующей процедуры: перед заданной величиной в круглых скобках проставляется имя требуемого типа данных. Скобки и имя в совокупности представляют собой операцию приведения типа. В общем случае операция приведения типа записывается в следующем виде:

(*тип*)

Вместо слова *тип* указывается фактически необходимый тип, например, long.

Рассмотрим две приведенные ниже строки кода, в которых mice — это переменная типа int. Вторая строка содержит два приведения к типу int.

```
mice = 1.6 + 1.7;
mice = (int) 1.6 + (int) 1.7;
```

В первом примере применяется автоматическое преобразование типов. Сначала складываются числа 1.6 и 1.7, что в сумме дает 3.3. Это число затем преобразуется за счет усечения в целое число 3 с целью согласования с переменной типа int. Во втором примере число 1.6, равно как и 1.7, перед сложением преобразуются в целое число (1), в результате чего переменной mice присваивается значение 1+1, или 2. По сути дела, ни одна из форм ничем не лучше другой, вам следует выбрать соответствующую форму, исходя из требований контекста программируемой задачи.

В повседневной практике старайтесь избегать перемешивания типов (по этой причине в некоторых языках программирования оно не допускается), но встречаются случаи, когда оно полезно. Философия языка C заключается в том, чтобы не устанавливать дополнительных барьеров на своем пути, возложив на вас всю ответственность за злоупотребление предоставляемой вам свободой.

Сводка: операции и операторы языка C

Ниже перечислены операции, которые обсуждались выше:

Операция присваивания:

=	Присваивает значение справа от знака операции переменной, указанной слева от знака.

Арифметические операции:

+	Прибавляет значение справа от знака операции к значению слева от знака.
-	Вычитает значение справа от знака операции из значения слева от знака.
-	Унарная операция; изменяет знак значения справа от знака операции.
*	Умножает значение справа от знака операции на значение слева от знака.
/	Делит значение слева от знака операции на значение справа от знака. Результат усекается, если оба операнда являются целочисленными.
%	Результат этой операции представляет собой остаток от деления значения слева от знака на значение справа от знака (выполняется только над целыми числами).
++	Прибавляет 1 к значению переменной справа от знака операции (префиксная форма) или к значению слева от знака операции (постфиксная форма).
--	Аналогична операции ++, но не добавляет, а вычитает 1.

Операции различного назначения:

sizeof	Возвращает размер в байтах операнда справа от знака операции. Таким оператором может быть спецификатор типа, заключенный в круглые скобки, например, sizeof (float), или это может быть имя конкретной переменной, массива и тому подобное, например sizeof foo.
(*тип*)	Как оператор приведения типа, преобразует соответствующее значение, стоящее справа, к типу, определенному ключевым словом в круглых скобках. Например, (float) 9 преобразует целое число 9 в число с плавающей запятой 9.0.

Функции с аргументами

К настоящему моменту вы уже знаете, что такое аргументы функций и как они используются. Следующий шаг на пути к мастерству в работе с функциями заключается в том, чтобы научиться писать свои собственные функции, которые используют аргументы. Рассмотрим, в чем состоит это искусство. (На этой стадии, возможно, имеет смысл освежить в памяти пример функции `butler()`, рассмотренный в конце главы 2; этот пример показывает, как нужно писать функции без аргументов.) Программа, представленная в листинге 5.15, содержит функцию `pound()`, которая печатает заданное количество знаков фунта (#). Этот пример также проливает свет на некоторые аспекты преобразования типов данных.

Листинг 5.15. Программа pound.c

```
/* pound.c -- определяет функцию с одним аргументом */
#include <stdio.h>
void pound(int n);   /* прототип ANSI */

int main(void)
{
   int times = 5;
   char ch = '!';    /* ASCII-код равен 33                                  */
   float f = 6.0;

   pound(times);     /* аргумент типа int                                   */
   pound(ch);        /* тип char автоматически приводится к типу int        */
   pound((int) f);   /* приведение типа f -> int                            */
   return 0;
}

void pound(int n)    /* заголовок функции в стиле ANSI                      */
{                    /* говорит, что она принимает один аргумент типа int*/
   while (n-- > 0)
      printf("#");
   printf("\n");
}
```

В результате выполнения этой программы получены следующие выходные данные:

```
#####
##################################
######
```

Сначала рассмотрим заголовок функции:

```
void pound(int n)
```

Если для выполнения функции аргументы не требуются, в круглых скобках заголовка функции будет присутствовать ключевое слово `void`. Поскольку рассматриваемая функция принимает один аргумент типа `int`, в круглых скобках содержится объявление переменной типа `int` с именем `n`. Вы можете воспользоваться любым именем, которое соответствует правилам именования языка C.

Операция объявления аргумента приводит к созданию переменной, называемой *формальным аргументом* или *формальным параметром*. В данном случае формальным па-

раметром является переменная типа int с именем n. Вызов функции, такой как, например, pound(10), приводит к тому, что переменной n присваивается значение 10. В рассматриваемой программе вызов pound(times) присваивает значение times (5) переменной n. Мы говорим, что вызов функции (или обращение к функции) *передает* значение, а само это значение называется *фактическим аргументом* или *фактическим параметром*, следовательно, вызов функции pound(10) передает фактический аргумент 10 данной функции, при этом 10 присваивается формальному аргументу (переменной n). Иначе говоря, значение переменной times в функции main() копируется в новую переменную n функции pound().

Аргументы и параметры

И хотя термины *аргумент* и *параметр* часто употребляются в одном и том же смысле, стандарт C99 рекомендует пользоваться термином *аргумент* вместо фактических аргументов или фактических параметров, а термином *параметр* — вместо формальных параметров и формальных аргументов. Учитывая это соглашение, мы можем утверждать, что параметры — это переменные, а аргументы — это значения, которыми снабжаются вызовы функций и которые присваиваются соответствующим параметрам.

Имена переменных являются локальными по отношению к функции. Это значит, что имя, объявленное в какой-либо функции, не вступает в конфликт с таким же именем, объявленным в любом другом месте. Если вы использовали переменную times вместо n в функции pound(), то в функции main() будет создана переменная, отличная от times. То есть, у вас появляются две переменных с одним именем, но программа хорошо знает, к какой функции относится та или иная переменная.

Теперь рассмотрим вызовы функций. Первым вызовом является pound(times), и как мы уже отмечали, данный вызов приводит к тому, что значение переменной times, равное 5, присваивается переменной n. В результате эта функция выводит на печать пять знаков фунта и знак новой строки. Второй вызов этой функции выглядит как pound(ch). В данном случае переменная ch имеет тип char. Она инициализируется символом !, который в системах с кодировкой ASCII означает, что ch получила числовое значение 33. Автоматическое повышение типа char до int преобразует его в данной системе со значения 33, которое размещается в одном байте, до значения 33, размещенного в четырех байтах, так что теперь значение 33 приобрело нужную форму, чтобы его можно было использовать в данной функции. Последний вызов, pound((int) f), использует преобразование типов для приведения типа переменной f типа float к типу, соответствующему назначению этого аргумента.

Предположим, что вы опустили приведение типа. В современном варианте C программа выполнит приведение типа автоматически. Это объясняется тем, что в начале файла находится прототип ANSI:

```
void pound(int n);   /* прототип ANSI */
```

Прототип — это объявление функции, в котором дается описание возвращаемого значения функции и всех ее аргументов. Рассматриваемый прототип сообщает следующие сведения о функции pound():

- У рассматриваемой функции возвращаемое значение отсутствует.
- Эта функция принимает один аргумент, каковым является значение типа int.

Поскольку компилятор просматривает этот прототип раньше, чем функция `pound()` будет использована в функции `main()`, он к тому моменту будет знать, какой вид аргумента должна иметь функции `pound()`, и он вставляет в программу преобразование типов, если в этом есть необходимость, чтобы фактический аргумент соответствовал прототипу в плане типов. Например, вызов функции `pound(3.859)` будет преобразован в `pound(3)`.

Демонстрационная программа

В листинге 5.16 представлена полезная программа (для ограниченной, но активной части человечества), которая служит иллюстрацией нескольких идей, рассматриваемых в данной главе. Она выглядит достаточно длинной, однако все вычисления выполняются в шести строках кода, расположенных ближе к концу. Основной объем программы предназначен для обмена информацией между компьютером и пользователем. Мы снабдили ее множеством комментариев, чтобы сделать ее действия очевидными. Внимательно ознакомьтесь с этой программой, после чего мы поможем вам снять несколько вопросов.

Листинг 5.16. Программа `running.c`

```
// running.c -- программа, полезная для тех, кто занимается бегом
#include <stdio.h>
const int S_PER_M = 60;       // количество секунд в минуте
const int S_PER_H = 3600;     // количество секунд в часе
const double M_PER_K = 0.62137; // количество миль в километре
int main(void)
{
   double distk, distm;       // дистанция пробега в километрах и милях
   double rate;               // средняя скорость в милях в час
   int min, sec;              // время пробега в минутах и секундах
   int time;                  // время пробега только в секундах
   double mtime;              // время пробега одной мили в секундах
   int mmin, msec;            // время пробега одной мили в минутах и секундах

   printf("Эта программа пересчитывает время пробега дистанции в
метрической системе\n");
   printf("во время пробега одной мили и вычисляет вашу среднюю\n");
   printf("скорость в милях в час.\n");
   printf("Введите дистанцию пробега в километрах.\n");
   scanf("%lf", &distk);        // %lf для типа double
   printf("Далее введите время в минутах и секундах.\n");
   printf("Начните с ввода минут.\n");
   scanf("%d", &min);
   printf("Теперь введите секунды.\n");
   scanf("%d", &sec);
// переводит время в секунды
   time = S_PER_M * min + sec;
// переводит километры в мили
   distm = M_PER_K * distk;
// мили в секунду умножить на секунды в час = количество миль в час
   rate = distm / time * S_PER_H;
```

```
// время/расстояние = время пробега одной мили
   mtime = (double) time / distm;
   mmin = (int) mtime / S_PER_M; // вычисление полного количества минут
   msec = (int) mtime % S_PER_M; // вычисление остатка в секундах
   printf("Вы пробежали %1.2f км (%1.2f мили) за %d мин, %d сек.\n",
      distk, distm, min, sec);
   printf("Такая скорость соответствует пробегу одной мили за %d мин, ",
      mmin);
   printf("%d sec.\nВаша средняя скорость равнялась %1.2f миль в час.\n",
      msec, rate);
   return 0;
}
```

В программе, представленной в листинге 5.16, используется тот же подход, который применялся ранее в программе `min_sec` для завершающего перевода результата в минуты и секунды, при этом данная программа применяет преобразования типов. С какой целью? Да потому, что той части программы, которая выполняет пересчет секунд в минуты, требуются целочисленные аргументы, а при преобразовании метрических данных в мили применяются числа с плавающей запятой. Мы использовали операцию приведения, чтобы сделать эти преобразования явными.

По правде говоря, эту программу можно было написать, используя только автоматические преобразования типов. Фактически мы так и поступили, применив операцию приведения переменной `mtime` к типу `int` с тем, чтобы при вычислении времени все операнды имели целый тип. Тем не менее, эту версию программы не удалось выполнить на одной из 11 систем, на которых мы пытались ее запускать. Компилятор той системы (устаревшая и вышедшая из употребления версия) оказался неспособным следовать правилам языка C. Использование приведения типов делают ваши намерения более понятными не только для читателей, но, по-видимому, и для компиляторов тоже.

Вот один из результатов исполнения рассматриваемой программы:

```
Эта программа пересчитывает время пробега дистанции в метрической системе
во время пробега одной и мили вычисляет вашу среднюю
скорость в милях в час.
Введите дистанцию пробега в километрах.
10.0
Далее введите время в минутах и секундах.
Начните с ввода минут.
36
Теперь введите секунды.
23
Вы пробежали 10.00 км (6.21 мили) за 36 мин, 23 сек.
Такая скорость соответствует пробегу одной мили за 5 мин, 51 сек.
Ваша средняя скорость равнялась 10.25 миль в час
```

Ключевые понятия

Язык C использует операции с целью оказания различного рода услуг. Каждую операцию можно характеризовать некоторым количеством операндов, необходимых для ее выполнения, ее приоритетом и ассоциативностью. Два последних качества опреде-

ляют, какие операции применяются первыми и когда две операции совместно используют тот или иной операнд. Операции в комбинациях с конкретными значениями образуют выражения, и каждое выражение в языке C имеет значение. Если вы не учитываете приоритет и ассоциативность операции, вы можете построить недопустимые выражения или такие выражения, которые дают результаты, отличные от тех, которые вы ожидаете, что отнюдь не содействует вашей репутации как квалифицированного программиста.

Язык C позволяет вам записывать выражения, комбинируя различные типы данных. В то же время арифметические операции требуют, чтобы операнды были одного и того же типа, и в силу этого C выполняет автоматическое преобразование типов. Тем не менее, хороший тон в программировании гласит: не следует полагаться на автоматическое преобразование типов. Вместо этого осуществляйте явный выбор типов, используя переменные необходимых типов, либо применяйте операции приведения типов. Поступая подобным образом, вы избежите неприятных сюрпризов со стороны автоматического преобразования типов данных.

Резюме

В языке C существует множество операций, таких как операции присваивания и арифметические операции, рассмотренные в данной главе. В общем случае, *операция* выполняется над одним или большим числом операндов с целью получить соответствующее значение. Операции, которые выполняются в отношении одного операнда, подобные знаку "минус" или `sizeof`, называются *унарными* (или *одноместными*) *операциями*. Операции, использующие два операнда, такие как операции сложения и умножения, называются *бинарными* (или *двухместными*) *операциями*.

Выражения представляют собой комбинации операций и операндов. В C каждое выражение имеет значение, в том числе выражения присваивания и выражения сравнения. Правила *приоритетов операций* помогают определить, в каком порядке группируются элементы при вычислении выражений. Когда две операции претендуют на использование какого-либо операнда, операция, имеющая более высокий приоритет, выполняется над ним первой. Если эти операции обладают равными приоритетами, какая них применяется первой, определяет ассоциативность (слева направо или справа налево).

Операторы — это завешенные команды компьютеру, они распознаются в C по точке с запятой, проставленной в конце. До сих пор вам приходилось иметь дело с операторами объявлений, операторами присваивания, операторами вызова функций и управляющими операторами. Операторы, заключенные в фигурные скобки, образуют *составной оператор*, или *блок*. Конкретным примером управляющего оператора является цикл `while`, который повторно выполняет операторы до тех пор, пока проверяемое условие остается истинным.

В языке C многие преобразования типов выполняются автоматически. Типы `char` и `short` повышаются до типа `double` всякий раз, когда используются в аргументах функций. В версии K&R C (но не в версии ANSI C) тип `float` повышается до `double`, когда присутствует в выражении. Когда значение одного типа присваивается переменной другого типа, то эта величина приводится к тому же типу, что и переменная. Когда большие типы (в смысле занимаемой памяти) приводятся к меньшим типам

(например, long к short или double к float), возможна потеря данных. В случае выполнения арифметических операций над операндами разных типов меньшие типы приводятся к большим в соответствии с правилами, изложенными в данной главе.

Когда вы даете определение функции с одним аргументом, вы объявляете *переменную*, или *формальный аргумент*, в определении функции. Затем значение, передаваемое в вызове функции, присваивается этой переменной, которая может теперь использоваться внутри данной функции.

Вопросы для самоконтроля

1. Предположим, что все переменные имеют тип int. Найдите значения каждой из следующих переменных:
   а. `x = (2 + 3) * 6;`
   б. `x = (12 + 6)/2*3;`
   в. `y = x = (2 + 3)/4;`
   г. `y = 3 + 2*(x = 7/2);`
2. Предположим, что все переменные имеют тип int. Найдите значения каждой из следующих переменных:
   а. `x = (int) 3.8 + 3.3;`
   б. `x = (2 + 3) * 10.5;`
   в. `x = 3 / 5 * 22.0;`
   г. `x = 22.0 * 3 / 5;`
3. У вас возникли подозрения, что в приведенной ниже программе присутствуют ошибки. Сможете ли вы их обнаружить?

```
int main(void)
{
   int i = 1,
   float n;
   printf("Будьте осторожны! Далее идет последовательность дробей!\n");
   while (i < 30)
      n = 1/i;
      printf(" %f", n);
   printf("Вот и все!\n");
   return;
}
```

4. Ниже приводится альтернативный вариант программы из листинга 5.9. Он отличается более простым кодом, поскольку два оператора scanf() из листинга 5.9 в нем заменены одним оператором scanf(). Почему этот вариант программы хуже, чем исходный?

```
#include <stdio.h>
#define S_TO_M 60
int main(void)
{
   int sec, min, left;
```

```
    printf("Эта программа переводит секунды в минуты и ");
    printf("секунды.\n");
    printf("Введите количество секунд.\n");
    printf("Для завершения программы введите 0.\n");
    while (sec > 0) {
       scanf("%d", &sec);
       min = sec/S_TO_M;
       left = sec % S_TO_M;
       printf("%d секунд равно %d минут, %d секунд. \n", sec, min, left);
       printf("Следующий ввод данных?\n");
    }
    printf("Работа завершена.\n");
    return 0;
}
```

5. Что распечатает следующая программа?

```
#include <stdio.h>
#define FORMAT "%s! C — это круто!\n"
int main(void)
{
   int num = 10;
   printf(FORMAT,FORMAT);
   printf("%d\n", ++num);
   printf("%d\n", num++);
   printf("%d\n", num--);
   printf("%d\n", num);
   return 0;
}
```

6. Что распечатает следующая программа?

```
#include <stdio.h>
int main(void)
{
   char c1, c2;
   int diff;
   float num;
   c1 = 'S';
   c2 = 'O';
   diff = c1 - c2;
   num = diff;
   printf("%c%c%c:%d %3.2f\n", c1, c2, c1, diff, num);
   return 0;
}
```

7. Что распечатает эта программа?

```
#include <stdio.h>
#define TEN 10
int main(void)
{
   int n = 0;
```

```
    while (n++ < TEN)
       printf("%5d", n);
    printf("\n");
    return 0;
}
```

8. Внесите в последнюю программу такие изменения, чтобы она вместо цифр печатала буквы от *a* до *g*.
9. Если приведенные ниже фрагменты были частью единой программы, что бы они печатали?

а.
```
int x = 0;
while (++x < 3)
printf("%4d", x);
```

б.
```
int x = 100;
while (x++ < 103)
     printf("%4d\n",x);
     printf("%4d\n",x);
```

в.
```
char ch = 's';
while (ch < 'w')
{
     printf("%c", ch);
     ch++;
}
printf("%c\n",ch);
```

10. Что будет печатать следующая программа?

```
#define MESG "COMPUTER BYTES DOG"
#include <stdio.h>
int main(void)
{
   int n = 0;
   while ( n < 5 )
     printf("%s\n", MESG);
     n++;
   printf("Вот и все.\n");
   return 0;
}
```

11. Напишите операторы, которые выполняют следующие действия (или, другими словами, имеет следующие побочные эффекты):
    а. Увеличивает значение переменной `x` на `10`.
    б. Увеличивает значение переменной `x` на `1`.
    в. Присваивает удвоенную сумму `a` и `b` переменной `c`.
    г. Присваивает `a` плюс дважды `b` переменной `c`.
12. Напишите операторы, которые выполняют следующие действия:
    а. Уменьшает значение переменной `x` на `1`.
    б. Присваивает `m` остатов от деления `n` на `k`.
    в. Делит `q` на `b`, вычитает `a` присваивает результат `p`.
    г. Присваивает переменной `x` результат деления суммы значений переменных `a` и `b` на произведение `c` и `d`.

Упражнения по программированию

1. Напишите программу, которая переводит время в минутах во время в часах и минутах. Воспользуйтесь инструкциями `#define` или `const` для создания символьной константы со значением 60. Используйте цикл `while`, чтобы обеспечить пользователю возможность повторного ввода значений и для прекращения цикла, если вводится значение времени, меньшее или равное нулю.
2. Напишите программу, которая запрашивает ввод целого числа, а затем печатает все целые числа, начиная (и включая) с этого числа и до числа, большего введенной величины на 10 включительно. (Иначе говоря, если вводится число 5, то выходными данными являются числа от 5 до 15 включительно.) Обязательно отделите друг от друга значения выходных данных пробелом либо символами табуляции или новой строки.
3. Напишите программу, которая требует от пользователя ввести количество дней, а затем переводит это значение в количество недель и дней. Например, она переводит 18 дней в 2 недели и 4 дня. Отобразите результаты в следующем формате:

   ```
   18 дней составляют 2 недели и 4 дня.
   ```

   Используйте цикла `while`, чтобы дать пользователю возможность многократного ввода количества дней и закончить цикл, для чего пользователь должен ввести неположительное значение, например, `0` или `-20`.
4. Напишите программу, которая просит пользователя ввести высоту в сантиметрах, после чего отображает высоту в сантиметрах, а также в футах и дюймах. Разрешается также выводить дробные части сантиметров и дюймов, а программа должна обеспечить пользователю возможность продолжать ввод значений высоты до тех пор, пока не будет введено неположительное значение. Пример выполнения этой программы должен иметь следующий вид:

   ```
   Введите высоту в сантиметрах: 182
   182.0 см = 5 футов, 11.7 дюймов
   Введите высоту в сантиметрах (<=0 для выхода из программы): 168
   168.0 см = 5 футов, 6.1 дюймов
   Введите высоту в сантиметрах (<=0 для выхода из программы): 0
   Работа завершена.
   ```
5. Внесите изменения в программу `addemup.c` (листинг 5.13), которая вычисляет сумму 20 первых целых чисел. (При желании вы можете рассматривать `addemup.c` как программу, которая вычисляет, какую сумму денег вы получите за 20 дней, если в первый день вы получаете $1, во второй день — $2, в третий день — $3 и так далее.) Внесите в эту программу изменения, благодаря которым вы можете в интерактивном режиме указать, насколько далеко зайдут вычисления. Другими словами, замените целое число 20 переменной, значение которой программа может вводить.
6. Теперь внесите такие изменения в программу из упражнения 5, чтобы она могла вычислять сумму квадратов целых чисел. (Или, если вам так больше нравится, программа вычисляет, какую сумму денег вы получите, если в первый день

вам заплатят $1, во второй день — $4, в третий день — $9 и так далее. Такая интерпретация намного симпатичнее!) В языке C нет функции возведения в квадрат, но в этом случае вам поможет сознание того, что квадратом числа `n` является `n * n`.

7. Напишите программу, которая запрашивает ввод числа типа `float` и распечатывает значение куба этого числа. С этой целью воспользуйтесь функцией, которая возводит заданное значение в куб и распечатывает результат, но эту функцию вы должны реализовать самостоятельно. Программа `main()` должна передать вводимое значение этой функции.

8. Напишите программу, которая требует от пользователя ввести значение температуры по Фаренгейту. Программа должна считывать значение температуры как число типа `double` и передать его как аргумент пользовательской функции с именем `Temperatures()`. Эта функция должна вычислять эквивалентные значения температуры по Цельсию и по Кельвину и отображать на экране все три значения температуры с точностью до двух позиций справа от десятичной точки. Функция должна сопоставлять каждое значение с температурной шкалой, которую оно представляет. Формула перевода температуры по Фаренгейту в температуру по Цельсию имеет вид:

   ```
   Температура-по-Цельсию = 1.8 * Температура-по-Фаренгейту + 32.0
   ```

   В шкале Кельвина, которая обычно применяется в научных применениях, 0 представляет абсолютный нуль, минимальный предел возможных температур. Формула перевода температуры по Цельсию в температуру по Фаренгейту имеет вид:

   ```
   Температура-по-Кельвину = Температура-по-Цельсию + 273.16
   ```

   Функция `Temperatures()` должна использовать `const` для создания символических представлений трех констант, которые используются для перевода. Функция `main()` должна использовать цикл, чтобы предоставить пользователю возможность многократного ввода значений температуры. Программа завершает работу, когда будет введен символ `q` или какое-то другое нечисловое значение.

ГЛАВА 6

Управляющие операторы: циклы

В этой главе:

- Ключевые слова: `for`, `while`, `do while`
- Операции: `<`, `>`, `>=`, `<=`, `!=`, `==`, `+=`, `*=`, `-=`, `/=`, `%=`
- Функции: `fabs()`
- Три структуры циклов в C — `while`, `for` и `do while`
- Использование операций отношения для создания выражений, управляющих этими циклами
- Другие операции
- Массивы, которые часто используются с циклами
- Написание функций, имеющих возвращаемые значения

Мощный, интеллектуальный, универсальный и полезный! Многие, несомненно, хотели бы заслужить такие эпитеты. При наличии языка C существует, по меньшей мере, определенный шанс, что наши программы заслужат подобную оценку. Все дело заключается в правильном управлении ходом выполнении программы. В соответствии с положениями теории вычислительных систем (это наука о компьютерах, но не наука, развивающаяся благодаря компьютерам ... пока что), хороший язык программирования должен обеспечивать следующие три формы управления ходом выполнения программы:

- Выполнение последовательности операторов.
- Многократное выполнение заданной последовательности операторов, пока не будет удовлетворено некоторое условие (цикл).
- Использование проверок с целью выбора одной из нескольких альтернативных последовательностей действий (условный переход).

Первая форма вам хорошо знакома; все рассмотренные выше программы представляли собой те или иные последовательности операторов. Цикл `while` является одним из примеров второй формы. В этой главе вы более подробно ознакомитесь с циклом `while` наряду с двумя другими циклическими структурами — `for` и `do while`. Третья и последняя форма, осуществляя выбор из нескольких возможных последовательностей действий, делает программу намного более “интеллектуальной” и существенно повышает эффективность использования компьютера. Как ни печально, но вам

придется подождать, пока не прочитаете целую главу, прежде чем вам будет доверено такое могущество. Эта глава служит также введением в массивы, поскольку именно к ним вы сможете применить приобретенные знания о циклах. Наряду с этим, в этой главе продолжается наращивание ваших знаний, касающихся функций. Начнем с того, что возобновим изучение цикла `while`.

Продолжение изучения цикла `while`

Вы уже немного знакомы с циклом `while`, и сейчас повторим то, что уже известно, на примере программы, которая суммирует целые числа, вводимые с клавиатуры (см. листинг 6.1). В этом примере для прекращения ввода данных используется возвращаемое значение функции `scanf()`.

Листинг 6.1. Программа `summing.c`

```
/* summing.c -- суммирует целые числа, вводимые в интерактивном режиме */
#include <stdio.h>
int main(void)
{
   long num;
   long sum = 0L; /* инициализация переменной sum нулем */
   int status;
   printf("Введите целое число для последующего суммирования ");
   printf("(или q для завершения программы): ");
   status = scanf("%ld", &num);
   while (status == 1) /* == обозначает равенство */
   {
      sum = sum + num;
      printf("Введите следующее целое число (или q для завершения программы): ");
      status = scanf("%ld", &num);
   }
   printf("Сумма введенных целых чисел равна %ld.\n", sum);
   return 0;
}
```

Программа, показанная в листинге 6.1, использует тип `long`, что позволяет вводить большие числа. Во избежание противоречий программа инициализирует переменную `sum` значением `0L` (нуль типа `long`), а не значением `0` (нуль типа `int`), даже несмотря на то, что свойство автоматического преобразования типов в языке C позволяет использовать просто `0`.

Ниже показан пример выполнения этой программы:

```
Введите целое число для последующего суммирования (или q для завершения
программы): 44
Введите следующее целое число (или q для завершения программы): 33
Введите следующее целое число (или q для завершения программы): 88
Введите следующее целое число (или q для завершения программы): 121
Введите следующее целое число (или q для завершения программы): q
Сумма введенных целых чисел равна 286.
```

Комментарии по программе

Рассмотрим сначала цикл while. Проверяемым условием этого цикла является:

```
status == 1
```

В языке C с помощью == представляется операция равенства, то есть это выражение проверяет, равно ли 1 значение переменной status. Не путайте это условие с выражением status = 1, которое присваивает 1 переменной status. Если проверяемым условием является status == 1, то цикл повторяется до тех пор, пока переменная status сохраняет значение 1. На каждой итерации цикл прибавляет значение num к значению переменной sum, благодаря чему в переменной sum накапливается текущее значение суммы. Когда переменная status получит значение, отличное от 1, цикл завершается и программа выводит окончательное значение sum.

Чтобы программа работала правильно, на каждой итерации цикла она должна получать для переменной num новое значение и на каждой итерации она должна устанавливать новое значение переменной status. Программа решает эту задачу, используя два свойства функции scanf(). Во-первых, программа вызывает scanf() для считывания нового значения переменной num. Во-вторых, она использует значение, возвращаемое функцией scanf(), чтобы сообщить, была ли попытка считывания успешной. Как отмечалось в главе 4, scanf() возвращает количество успешно считанных элементов. Если scanf() успешно считывает целое число, она помещает его в переменную num и возвращает значение 1, которое присваивается переменной status. (Обратите внимание, что входное значение передается переменной num, но не переменной status.) Это приводит к обновлению как значения num, так и значения status, и цикл while выходит на очередную итерацию. Если вы ответите на приглашение вводом нечисловой величины, такой как, например, символ q, то функция scanf() не обнаружит целого числа при вводе, следовательно, возвращаемое ею значение и значение status будут равны 0. На этом выполнение цикла завершается. Входной символ q, в силу того факта, что он не является числом, возвращается во входную очередь, он вообще не читается. (По сути дела, любой нечисловой ввод, а не только символ q, вызывает завершение цикла, тем не менее, пользователю предлагается ввести q, поскольку это проще, чем просить ввести любое нечисловое значение.)

Если перед выполнением преобразования значения функция scanf() столкнется с какой-то проблемой (например, обнаружит символ конца файла или случится сбой оборудования), она возвращает специальное значение EOF ("End Of File" — конец файла), которое обычно определяется как -1. Это значение также вызовет завершение цикла.

Такое двойное использование функции scanf() позволяет избежать трудно решаемой проблемы интерактивного ввода в цикле: как сообщить циклу о том, когда он должен прекратиться? Предположим, например, что функция scanf() не имеет возвращаемого значения. В таком случае единственное, что подвергается изменению на каждой итерации — это значение переменной num. Вы можете использовать значение num, чтобы прекратить выполнение цикла, указав, например, выражение num > 0 или num != 0 в качестве проверяемого условия, но в этом случае вы лишаетесь возможности использовать отдельные вводимые значения, такие как -3 или 0. Вместо этого вы могли бы добавить в цикл новый код, например запрос "Намерены ли вы продолжать? <да/нет>" на каждой итерации, а затем проверять, ввел ли пользователь ответ "да" на

этот вопрос. Это выглядит несколько неуклюже, к тому же, замедляется ввод. Использование возвращаемого значения `scanf()` позволяет избежать этих проблем.

Теперь подробнее рассмотрим структуру этой программы. Кратко описать ее можно следующим образом:

```
инициализировать переменную sum значением 0
выдача пользователю приглашения на ввод
считывание входных данных
пока входное значение представляет собой целое число,
   прибавить входное значение к значению sum,
   выдать пользователю приглашение на ввод,
   затем прочитать следующий ввод
по завершении ввода печать значения переменной sum
```

Фактически это описание служит примером применения *псевдокода*, который представляет собой способ упрощенного описания программы на естественном языке, который в какой-то мере моделирует формы машинного языка. Псевдокод весьма полезен при разработке логики конкретной программы. Как только логика покажется вам безошибочной, можете приступать к переводу псевдокода в программный код. Одно из преимуществ псевдокода заключается в том, что он позволяет сконцентрировать усилия на отработке логики и организации программы, не беспокоясь до поры о том, как выразить идеи на языке программирования. Например, в приведенном выше псевдокоде блоки выделяются с помощью отступов, при этом неважно, что синтаксис языка C требует применения фигурных скобок. Еще одно достоинство псевдокода состоит в том, что псевдокод не привязан к конкретному языку программирования, благодаря чему один и тот же псевдокод может быть переведен на различные языки.

Как бы то ни было, но поскольку `while` является циклом с предусловием, программа должна получить входное величину и проверить значение переменной `status` *до того*, как она войдет в тело цикла. Вот почему функция `scanf()` стоит в программе перед `while`. Чтобы цикл продолжался, необходимо, чтобы внутри него был оператор чтения, который мог бы определить состояние следующей входной величины. Поэтому в конец цикла `while` поставлен оператор `scanf()`; он подготавливает цикл к следующей итерации. Вы можете рассматривать приведенный ниже псевдокод как стандартный формат цикла:

```
получение первого значения и выполнение его проверки
пока проверка проходит успешно
   обработать это значение
   получить следующее значение
```

Цикл считывания в стиле C

Код в листинге 6.1 мог быть написан на языке программирования Pascal, BASIC или FORTRAN в соответствии с замыслом, сформулированном в приведенном выше псевдокоде. Язык C, однако, обеспечивает более короткую запись. Конструкция

```
status = scanf("%ld", &num);
while (status == 1)
{
   /* действия, выполняемые в цикле */
   status = scanf("%ld", &num);
}
```

может быть заменена следующей:

```
while (scanf("%ld", &num) == 1)
{
   /* действия, выполняемые в цикле */
}
```

Вторая форма использует функцию `scanf()` одновременно двумя различными способами. Во-первых, в результате вызова функции, если он завершился успешно, входное значение помещается в переменную `num`. Во-вторых, значение, возвращаемое этой функцией (оно равно `1` или `0` и не является значением переменной `num`) управляет циклом. Поскольку условие цикла проверяется на каждой итерации, то и функция `scanf()` вызывается на каждой итерации, при этом будет введено новое значение `num` и проведена новая проверка. Другими словами, свойства синтаксиса языка C позволяют заменить стандартный формат цикла следующей компактной версией:

```
пока получение и проверка значения завершается успешно
   выполнить обработку этого значения
```

А теперь рассмотрим оператор `while` более формально.

Оператор `while`

Общая форма оператора `while` имеет следующий вид:

```
while (выражение)
   оператор
```

Часть *оператор* может быть простым выражением, завершающимся точкой с запятой, либо представлять собой составной оператор, заключенный в фигурные скобки.

В примерах, которые рассматривались до сих пор, в качестве *выражения* использовались условные выражения, то есть *выражением* служило сравнение значений. В общем случае вы можете использовать любое выражение. Если *выражение* дает в результате истину (или, в общем случае, принимает ненулевое значение), *оператор* выполняется один раз, после чего выражение проверяется снова. Цикл проверок и выполнений повторяется до тех пор, пока *выражение* не станет ложным (нулем). Каждая последовательность проверки и выполнения называется итерацией (рис. 6.1).

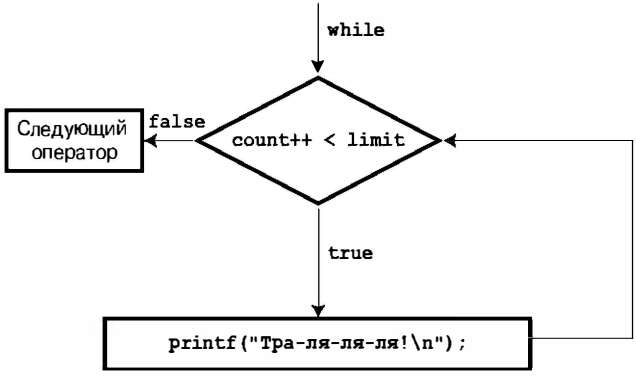


Рис. 6.1. *Структура цикла* `while`

Завершение цикла while

Мы подошли к наиболее существенному моменту, характеризующему циклы while: при создании цикла while должно быть предусмотрено нечто, что меняет значение проверяемого выражения таким образом, чтобы оно в конечном итоге становилось ложным. В противном случае цикл никогда не заканчивается. (На самом деле вы можете воспользоваться операторами break и if, чтобы завершить цикл, но пока о них ничего не знаете.) Рассмотрим следующий пример:

```
index = 1;
while (index < 5)
   printf("Доброе утро!\n");
```

Предыдущий фрагмент программы печатает это бодрое сообщение бесконечное число раз. Почему так получается? Да потому, что ничто в цикле не меняет первоначального значения индекса, равного 1. Теперь рассмотрим такой фрагмент:

```
index = 1;
while (--index < 5)
   printf("Доброе утро!\n");
```

Он оказывается ненамного лучше первого. Значение переменной index изменяется, но не в том направлении! Эта версия кода в конечном итоге приведет к окончанию цикла, но только после того, когда значение переменной index упадет ниже минимального отрицательного числа и станет максимальным положительным значением. (Программа toobig.c в главе 3 служит иллюстрацией того, что сложение максимального положительного числа с 1 обычно в результате приводит к появлению отрицательного числа; аналогично, вычитание 1 из минимального отрицательного числа дает в результате положительное значение.)

Когда цикл завершается?

Важно понимать, что решение прервать выполнение цикла или продолжить его, принимается только в момент, когда завершена проверка условия цикла. В качестве примера рассмотрим программу, представленную в листинге 6.2.

Листинг 6.2. Программа when.c

```
// when.c -- когда цикл завершается?
#include <stdio.h>
int main(void)
{
   int n = 5;
   while (n < 7)                        // строка 7
   {
      printf("n = %d\n", n);
      n++;                              // строка 10
      printf("Теперь n = %d\n", n);     // строка 11
   }
   printf("Цикл завершен.\n");
   return 0;
}
```

В результате выполнения программы, показанной в листинге 6.2, получаются следующие выходные данные:

```
n = 5
Теперь n = 6
n = 6
Теперь n = 7
Цикл завершен.
```

Переменная n сначала получает значение 7 в строке 10 во время выполнения второй итерации цикла. Однако программа на этом не завершается. Она всего лишь завершает очередную итерацию (строка 11) и выходит из цикла только после того, как условное выражение в строке 7 будет проверено в третий раз. (Переменная n принимает значение 5 во время первой проверки и 6 — во время второй.)

Оператор while: цикл с предусловием

Цикл while — это условный цикл, использующий входное условие (или предусловие). Цикл называется "условным", поскольку выполнение его операторной части зависит от условия, представленного условным выражением, таким как, например, (index < 5). Это выражение представляет собой предусловие, поскольку оно должно быть выполнено, прежде чем управление будет передано телу цикла. В ситуациях подобного рода управление никогда не войдет в тело цикла, так как условие ложно с самого начала:

```
index = 10;
while (index++ < 5)
   printf("Удачного дня!\n");
```

Поменяем первую строку на

```
index = 3;
```

и цикл начнет выполняться.

Синтаксические особенности

При использовании оператора while всегда следует иметь в виду, что в цикл в качестве его составной части, расположенной непосредственно за проверяемым условием, может входить только один оператор, простой или составной. Отступы от начала строки предназначены для помощи читателю, а не для компьютера. Листинг 6.3 демонстрирует, что может произойти, если вы забудете об этом.

Листинг 6.3. Программа while1.c

```
/* while1.c -- правильно расставляйте фигурные скобки */
/* Неправильное кодирование может стать причиной появления бесконечных циклов*/
#include <stdio.h>
int main(void)
{
   int n = 0;
   while (n < 3)
      printf("n равно %d\n", n);
      n++;
   printf("Это все, на что способна данная программа\n");
   return 0;
}
```

Программа, представленная в листинге 6.3, генерирует следующие выходные данные:

```
n равно 0
n равно 0
n равно 0
n равно 0
n равно 0
```

(... и так далее, пока вы каким-то радикальным способом не остановите ее выполнение.)

Хотя в этом примере оператор `n++;` вынесен в отдельную строку, он не заключен в фигурные скобки вместе с предшествующим оператором. Поэтому в состав цикла входит только оператор печати, который следует непосредственно за проверяемым условием. Переменная `n` никогда не изменится, условие `n < 3` навсегда останется истинным, и вы получите цикл, который будет продолжать печатать известие о том, что `n` равно `0`, пока вы не прервете выполнение программы. Это пример бесконечного цикла, из которого нельзя выйти без постороннего вмешательства.

Всегда помните, что сам по себе оператор `while`, даже если в нем используется составной оператор, с позиций синтаксиса рассматривается как один оператор. Такой оператор выполняет все, что находится справа от ключевого слова `while` до первой точки с запятой, или в случае использования составного оператора, до закрывающей фигурной скобки.

Расставляя точки с запятой, соблюдайте осторожность. В качестве примера рассмотрим программу в листинге 6.4.

Листинг 6.4. Программа `while2.c`

```
/* while2.c -- правильно расставляйте точки с запятой */
#include <stdio.h>
int main(void)
{
   int n = 0;
   while (n++ < 3);                  /* строка 7 */
      printf("n равно %d\n", n);     /* строка 8 */
   printf("Это все, что может сделать данная программа.\n");
   return 0;
}
```

В результате выполнения программы, показанной в листинге 6.4, получены следующие результаты:

```
n равно 4
Это все, что может сделать данная программа.
```

Как уже было сказано выше, цикл заканчивается первым оператором, простым или составным, который следует непосредственно за проверяемым условием. Поскольку в строке 7 точка с запятой следует сразу за проверяемым условием, цикл заканчивается в этом месте, поскольку отдельно стоящая точка с запятой рассматривается как оператор. Оператор печати в строке 8 не относится к циклу, следовательно, значение `n` обновляется на каждой итерации, однако печать выполняется только по выходу из цикла.

В рассматриваемом примере после проверяемого условия следует пустой оператор, то есть оператор, который не выполняет никаких действий. Время от времени программисты намеренно используют оператор while с пустым оператором как временную задержку или в связи с тем, что вся полезная работа выполняется во время проверки условия. Например, предположим, что вы хотите пропустить ввод до первого символа, который не является пробелом или цифрой. Вы можете воспользоваться таким циклом:

```
while (scanf("%d", &num) == 1)
   ;       /* пропустить ввод целого числа */
```

Поскольку функция scanf() считывает целое число, она возвращает 1, и цикл продолжается. Обратите внимание, что для наглядности мы поместили точку с запятой (пустой оператор) не в первой строке, а в строке ниже. Это повышает удобочитаемость текста программы и в то же время говорит о том, что пустой оператор включен в программу не случайно. Еще лучше пользоваться в таких случаях оператором continue, который будем рассматриваться в следующей главе.

Что больше: использование операций и выражений отношения

Поскольку правильность выполнения цикла while часто зависит от проверяемых выражений, с помощью которых выполняются сравнения, эти выражения заслуживают пристального внимания. Эти выражения получили название выражений отношения, а операции, в которых они появляются, называются операциями отношения. Вы уже пользовались несколькими такими операциями, а в табл. 6.1 приводится полный список операций отношения в C. Эта таблица охватывает фактически все возможные числовые отношения. (Числа, даже если это комплексные числа, не так сложны, как люди со всеми их комплексами.)

Таблица 6.1. Операции отношений

Операция	*Значение*
<	Меньше
<=	Меньше или равно
==	Равно
>=	Больше или равно
>	Больше
!=	Не равно

Операции отношений используются для построения выражений отношения, употребляемых в операторах while и в других операторах языка C, которые мы будем изучать позже. Эти операторы проверяют, является ли конкретное выражение истинным или ложным. Вот три не связанных между собой оператора, содержащие примеры выражений отношений. Мы надеемся, что их смысл должен быть понятен.

```
while (number < 6)
{
   printf("Введенное число очень мало.\n");
   scanf("%d", &number);
}
while (ch != '$')
{
   count++;
   scanf("%c", &ch);
}
while (scanf("%f", &num) == 1)
   sum = sum + num;
```

Обратите внимание на второй цикл — в условных выражениях могут использоваться также и символы. Для сравнений применяются их машинные коды (которыми, как мы полагаем, были коды ASCII). Однако вы не можете использовать операции отношений для сравнения строк. В главе 11 вы узнаете, какие языковые средства используется для сравнения строк.

Операции отношений могут также применяться и к числам с плавающей запятой. Однако необходимо иметь в виду, что при сравнении чисел с плавающей запятой можно пользоваться только операциями < и >. Это объясняется тем, что ошибки округления могут привести к тому, что числа окажутся неравными, хотя по логике программы они должны быть равны. Например, вполне очевидно, что произведение чисел 3 и 1/3 равно 1.0. Но если представить 1/3 в виде десятичной дроби с шестью значащими числами, то произведением будет .999999, что в точности не равно 1. Функция `fabs()`, объявленная в заголовочном файле `math.h`, может оказаться весьма полезной при проверке условий, в которых используются числа с плавающей запятой. Эта функция возвращает абсолютное значение величины с плавающей запятой, то есть, значение без алгебраического знака. Например, вы можете проверить, насколько некоторое число близко к желаемому результату, используя код, подобный показанному в листинге 6.5.

Листинг 6.5. Программа `cmpflt.c`

```
// cmpflt.c -- сравнение чисел с плавающей запятой
#include <math.h>
#include <stdio.h>
int main(void)
{
   const double ANSWER = 3.14159;
   double response;
   printf("Каково значение числа pi?\n");
   scanf("%lf", &response);
   while (fabs(response - ANSWER) > 0.0001)
   {
      printf("Введите значение повторно!\n");
      scanf("%lf", &response);
   }
   printf("Требуемая точность достигнута!\n");
   return 0;
}
```

Этот цикл продолжает уточнять ответ до тех пор, пока разность между ответом и правильным значением не окажется в пределах 0.0001:

```
Каково значение числа pi?
3.14
Введите значение повторно!
3.1416
Требуемая точность достигнута!
```

Каждое условное выражение получает оценку "истина" (`true`) или "ложь" (`false`) (но никогда "может быть"). При этом возникает интересный вопрос.

Что такое истина?

Ответ на этот извечный вопрос вы можете получить, по крайней мере, когда дело касается языка C. Напомним, что выражение в C всегда имеет значение. Это утверждение остается в силе и для выражений отношения, как показывает пример, представленный в листинге 6.6. В этом примере на печать выводятся значения двух выражений отношения, одно из которых истинное, а другое — ложное.

Листинг 6.6. Программа `t_and_f.c`

```
/* t_and_f.c -- истинные и ложные выражения в языке C */
#include <stdio.h>
int main(void)
{
   int true_val, false_val;

   true_val = (10 > 2);      /* значение истинного отношения */
   false_val = (10 == 2);    /* значения ложного отношения */
   printf("true = %d; false = %d \n", true_val, false_val);

   return 0;
}
```

Код в листинге 6.6 присваивает значения двух выражений отношения двум переменным. Во избежание недоразумений переменной `true_val` присваивается значение истинного отношения, а переменной `false_val` — значение ложного отношения. Выполнив эту программу, получаем следующий простой результат:

```
true = 1; false = 0
```

Так вот в чем дело! В языке C истинное выражение имеет значение `1`, а ложное выражение — `0`. И действительно, некоторые программы на C используют следующую конструкцию для циклов, которые по замыслу их авторов должны выполняться, поскольку `1` всегда означает `true`:

```
while (1)
{

}
```

Какой еще может быть истина?

Если вы можете использовать 1 или 0 в качестве проверяемого условия в операторе while, можете ли вы применять для этих целей другие числа? Если да, то что при этом происходит? Давайте немного поэкспериментируем с программой из листинга 6.7.

Листинг 6.7. Программа truth.c

```
// truth.c -- какие значения обозначают истину?
#include <stdio.h>
int main(void)
{
   int n = 3;
   while (n)
      printf("%2d есть true\n", n--);
   printf("%2d есть false\n", n);
   n = -3;
   while (n)
      printf("%2d есть true\n", n++);
   printf("%2d есть false\n", n);
   return 0;
}
```

Получаем следующие результаты :

```
3  есть true
2  есть true
1  есть true
0  есть false
-3 есть true
-2 есть true
-1 есть true
0  есть false
```

Первый цикл выполняется, когда переменная n принимает значения 3, 2 и 1, но прекращает выполняться, как только n становится равной 0. Аналогично, второй цикл выполняется, когда переменная n принимает значения -3, -2 и -1, но прекращает выполняться, как только n становится равной 0. В общих словах, *все* ненулевые значения рассматриваются как истинные (true), и только 0 считается ложным (false). Язык C предлагает довольно-таки широкое толкование истины!

С другой стороны, можно утверждать, что цикл while выполняется до тех пор, пока вычисление его проверяемого условия дает в результате ненулевое значение. Это обстоятельство позволяет перевести проверяемое условие на числовую основу вместо использования значений “истина” и “ложь”. Не упускайте из виду, что условное выражение принимает значение 1, если оно истинно, и 0, если ложно, таким образом, все выражения этого типа являются числовыми.

Многие программисты, работающие на C, используют это свойство условий проверки. Например, конструкцию while(goats != 0) можно заменить на while(goats), поскольку как выражение (goats != 0), так и выражение (goats) принимают значение 0, или false, только когда переменная goats равна 0.

Первая форма, по-видимому, более понятна тем, кто только что приступил к изучению языка, в то время как вторая форма представляет собой идиому, которая используется профессиональными программистами, работающими на C. Вам придется немного поэкспериментировать с формой `while (goats)`, чтобы она впоследствии казалась вполне естественной.

Трудности при употреблении понятия "истина"

Достаточно либеральное толкование языком C понятия истины чревато осложнениями. Например, внесем одно малозаметное на первый взгляд изменение в программу, представленную в листинге 6.1, и получим программу, показанную в листинге 6.8.

Листинг 6.8. Программа `trouble.c`

```
// trouble.c -- неправильное использование знака =
// приводит к возникновению бесконечных циклов
#include <stdio.h>
int main(void)
{
   long num;
   long sum = 0L;
   int status;
   printf("Введите целое число для последующего суммирования ");
   printf("(или q для завершения программы): ");
   status = scanf("%ld", &num);
   while (status = 1)
   {
      sum = sum + num;
      printf("Введите следующее целое число (или q для завершения программы): ");
      status = scanf("%ld", &num);
   }
   printf("Сумма введенных целых чисел равна %ld.\n", sum);
   return 0;
}
```

Выполнение программы из листинга 6.8 дает следующие выходные результаты:

```
Введите целое число для последующего суммирования (или q для завершения
программы): 20
Введите следующее целое число (или q для завершения программы): 5
Введите следующее целое число (или q для завершения программы): 30
Введите следующее целое число (или q для завершения программы): q
Введите следующее целое число (или q для завершения программы):
Введите следующее целое число (или q для завершения программы):
Введите следующее целое число (или q для завершения программы):
Введите следующее целое число (или q для завершения программы):
```

(... и так далее, пока вы каким-то радикальным способом не остановите выполнение программы.)

Столь неутешительные результаты вызвало изменение, внесенное в проверяемое условие оператора `while`, когда выражение `status == 1` было заменено выражением

`status = 1`. Второй оператор — это оператор присваивания, который назначает переменной `status` значение `1`. Более того, значение оператора присваивания представляет собой значение его левой части, так что `status = 1` имеет то же числовое значение, равное `1`. Следовательно, для всех практических целей использование этого цикла `while` дает тот же результат, что и оператор `while (1)`; то есть, это бесконечный цикл. Когда вы вводите `q`, переменная `status` получает значение `0`, однако во время проверки конца цикла переменная `status` получает значение `1` и инициирует выполнение очередной итерации.

Вы, скорее всего, захотите узнать, почему из-за того, что программа выполняется в цикле, пользователь не имеет возможности ввести дальнейшие входные данные после того, как он ввел символ `q`. Когда функции `scanf()` не удается прочитать входные данные в той форме, на которую она настроена, она оставляет этот не соответствующий ее требованиям ввод на месте, чтобы прочитать его в следующий раз. Когда `scanf()` предпринимает попытку читать символ `q` как целое число и терпит при этом неудачу, она оставляет `q` на месте. Во время выполнения следующей итерации цикла функция `scanf()` пытается читать в том месте, где она читала последний раз, а именно, там, где остался символ `q`. И снова `scanf()` не может прочитать символ `q` как целое число, так что этот пример не только устанавливает бесконечный цикл, он также демонстрирует цикл бесконечных неудач, словом, печальный результат. Хорошо еще, что компьютеры пока еще лишены чувств. Использовать глупые команды компьютеру не лучше и не хуже попыток успешно предсказывать положение на фондовой бирже в течение следующих 10 лет.

Не употребляйте знак = вместо ==. В некоторых языках программирования (например, в BASIC) используется один и тот же символ для представления операции присваивания и проверки на равенство, хотя эти операции совершенно разные (рис. 6.2).

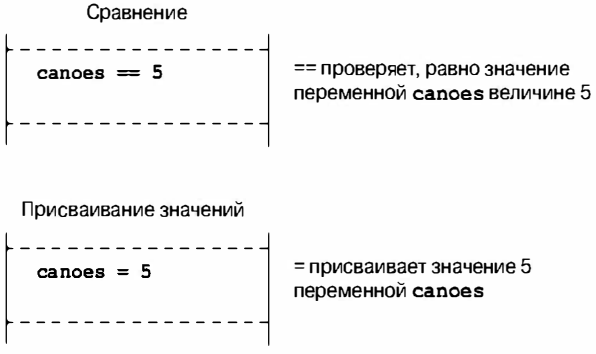


Рис. 6.2. *Операция отношения == и операция присваивания =*

Операция присваивания устанавливает значение переменной, указанной слева от знака операции. В то же время операция проверки на равенство проверяет, существует ли равенство между левой и правой частями операции. Она не меняет значения переменной, указанной в левой части операции, если таковая там имеется.

Рассмотрим соответствующий пример:

```
canoes = 5   ←Присваивает переменной canoes значение 5
canoes == 5  ←Проверяет, равно ли значение переменной canoes величине 5
```

Соблюдайте осторожность при выборе операции. Компилятор разрешит вам использовать неправильную форму, но тогда вы получите совсем не тот результат, на который рассчитываете. (Тем не менее, такое множество программистов так часто неправильно применяло операцию =, что большая часть компиляторов выводит на экран предупреждающие сообщения — вполне достаточно, чтобы отбить охоту ею пользоваться.) Если одна из сравниваемых величин является константой, вы можете поместить ее слева от знака операции сравнения, чтобы помочь выявить ошибку:

```
canoes = 5   ←Синтаксическая ошибка
5 == canoes  ←Проверяет, равно ли значение переменной canoes величине 5
```

Дело в том, что нельзя присваивать значение константе, и в силу этого компилятор рассматривает такую операцию присваивания как синтаксическую ошибку. Многие профессиональные программисты первой ставят константу в выражениях проверки на предмет равенства.

Подводя итоги, отметим, что операции отношения используются для создания условных выражений. Выражения отношения получают значение 1, если они истинны, и 0, если ложны. В операторах (таких как `while` и `if`), в которых обычно присутствуют условные выражения для определения условий проверки, могут использоваться любые выражения, при этом ненулевые выражения интерпретируются как "истина", а нулевые — как "ложь".

Новый тип _Bool

Переменные, предназначенные для представления значений `true` и `false`, традиционно в языке C имеют тип `int`. Специально для переменных этого вида стандарт C99 вводит тип `_Bool`. Тип получил свое название в честь Джорджа Буля (George Boole), английского математика, который разработал алгебраическую систему, позволяющую формулировать и решать логические задачи. В программировании переменные, представляющие значения `true` и `false`, известны как *булевские* (Boolean), таким образом `_Bool` в языке C является именем типа для упомянутых переменных. Переменная `_Bool` может принимать только значения 1 (истина) и 0 (ложь). Если вы попытаетесь присвоить ненулевое числовое значение переменной типа `_Bool`, эта переменная получит значение 1, которое отражает тот факт, что C трактует любое ненулевое значение как `true`.

В программе, показанной в листинге 6.9, исправлено условие проверки, использованное в программе из листинга 6.8, и переменная `status` типа `int` заменена переменной `input_is_good` типа `_Bool`. Присвоение булевским переменным имен, из которых ясно, какое значение эта переменная принимает, превратилось в установившуюся практику.

Листинг 6.9. Программа `boolean.c`

```
// boolean.c -- использование переменной типа _Bool
#include <stdio.h>
int main(void)
{
   long num;
   long sum = 0L;
   _Bool input_is_good;
```

```
    printf("Введите целое число для последующего суммирования ");
    printf("(или q для завершения программы): ");
    input_is_good = (scanf("%ld", &num) == 1);
    while (input_is_good)
    {
        sum = sum + num;
        printf("Введите следующее целое число (или q для завершения программы): ");
        input_is_good = (scanf("%ld", &num) == 1);
    }
    printf("Сумма введенных целых чисел равна %ld.\n", sum);
    return 0;
}
```

Обратите внимание на то, как программа присваивает результат сравнения переменной:

```
input_is_good = (scanf("%ld", &num) == 1);
```

Это присваивание имеет смысл, поскольку операция == возвращает либо 1, либо 0. По сути дела, круглые скобки, заключающие в себе выражение ==, не нужны, поскольку операция == имеет более высокий приоритет, чем операция =; тем не менее, они способны повысить удобочитаемость кода. Заметьте также, насколько изменение имени переменной делает проверку цикла более понятной:

```
while (input_is_good)
```

Стандарт C99 предлагает для использования заголовочный файл `stdbool.h`. В этом заголовочном файле `bool` определяется как альтернативное имя для типа `_Bool`, а также `true` и `false` как символические константы со значениями 1 и 0. Включение этого заголовочного файла позволяет писать код, совместимыq с программами на языке C++, в котором `bool`, `true` и `false` представляют собой ключевые слова.

Если ваша система не поддерживает тип `_Bool`, можете заменить `_Bool` типом `int`, и этот пример будет работать так же.

Приоритеты операций отношения

Приоритет операций отношения ниже приоритета арифметических операций, + и -, но выше, чем у операции присваивания. Это значит, что, например,

```
x > y + 2
```

означает то же самое, что и

```
x > (y + 2)
```

Это также значит, что

```
x = y > 2
```

эквивалентно

```
x = (y > 2)
```

Иначе говоря, переменной `x` присваивается 1, если `y` больше 2, и 0 — в противном случае; переменной `x` не присваивается значение `y`.

Операции отношения имеют больший приоритет, чем операции присваивания. Поэтому

```
x_bigger = x > y;
```

означает

```
x_bigger = (x > y);
```

Операции отношения по своему приоритету делятся на две группы.

- Группа с высоким приоритетом: < <= > >=
- Группа с низким приоритетом: == !=

Подобно большей части других операций, операции отношения выполняется слева направо. В связи с этим конструкция

```
ex != wye == zee
```

эквивалентна

```
(ex != wye) == zee
```

Сначала C проверяет, имеет ли место неравенство значений переменных `ex` и `wye`. Затем полученное значение, которое может быть равно `1` или `0` ("истина" или "ложь"), сравнивается со значением `zee`. Мы не намерены использовать конструкции подобного рода, но считаем своим долгом подчеркнуть, что они возможны.

В табл. 6.2 перечислены приоритеты операций, изученных к данному моменту. В справочном разделе II (приложение Б) приведены все операции, упорядоченные в порядке убывания приоритетов.

Таблица 6.2. Приоритет операций

Операции (в порядке убывания приоритетов)	*Ассоциативность*
`()`	Слева направо
`- ++ -- sizeof (`*тип*`)` (все унарные)	Справа налево
`* / %`	Слева направо
`+ -`	Слева направо
`< > <= >=`	Слева направо
`== !=`	Слева направо
`=`	Справа налево

Сводка: оператор `while`

Ключевое слово:

`while`

Комментарии общего характера:

Оператор `while` определяет операции, которые повторяются до тех пор, пока проверяемое выражение не станет ложным или равным нулю. Оператор `while` представляет собой цикл с предусловием; это значит, что решение еще одной итерации цикла принимается перед очередным проходом цикла. Следовательно, вполне возможно, что не

будет выполнен ни один проход цикла. Операторная часть цикла может быть либо простым, либо составным оператором.

Форма записи:

```
while (выражение)
   оператор
```

Часть *оператор* повторяется до тех пор, пока *выражение* не станет ложным или равным 0.

Примеры:

```
while (n++ < 100)
   printf(" %d %d\n",n, 2 * n + 1);    /* одиночный оператор */

while (fargo < 1000)
{                                      /* составной оператор */
   fargo = fargo + step;
   step = 2 * step;
}
```

Сводка: операции отношения и условные выражения

Операции отношения:

Каждая операция отношения сравнивает значение в ее левой части со значением в ее правой части:

<	меньше
<=	меньше или равно
==	равно
>=	больше или равно
>	больше
!=	не равно

Условные выражения:

Простое условное выражение состоит из знака операции отношения и операндов, расположенных слева и справа от нее. Если выражение истинно, то значение условного выражения равно 1. Если отношение ложно, то значение условного выражения равно 0.

Примеры:

`5 > 2`	истинно и принимает значение 1.
`(2 + a) == a`	ложно и принимает значение 0.

Неопределенные циклы и циклы со счетчиком

Некоторые из примеров цикла while представляют собой образцы так называемых неопределенных циклов. Это означает, что вы заранее не можете знать, сколько раз цикл будет выполнен, прежде чем выражение станет ложным. Например, в программе, представленной в листинге 6.1, использован интерактивный цикл для суммирования целых чисел, и вы не знаете, сколько чисел будет введено. В то же время примеры представляют собой *циклы со счетчиком*. Эти циклы выполняют заранее известное количество итераций.

В листинге 6.10 приводится пример оператора цикла while со счетчиком.

Листинг 6.10. Программа sweetie1.c

```
// sweetie1.c -- цикл со счетчиком
#include <stdio.h>
int main(void)
{
   const int NUMBER = 22;
   int count = 1;                              // инициализация
   while (count <= NUMBER)                     // проверка
   {
      printf("Будьте моим Валентином!\n");     // действие
      count++;                                 // обновление значения count
   }
   return 0
}
```

И хотя форма, использованная в листинге 6.10, работает прекрасно, тем не менее, это не самый лучший выбор в подобной ситуации, поскольку действия, определяющие цикл, не собраны воедино. Рассмотрим этот вопрос более подробно.

Чтобы организовать цикл, который должен быть повторен заданное количество раз, необходимо выполнить следующие три действия:

1. Инициализировать счетчик.
2. Сравнить показание счетчика с некоторой ограниченной величиной.
3. Обновить значение счетчика после каждого прохода цикла.

Условие цикла while обеспечивает выполнение сравнения. Операция инкремента приводит к обновлению значения счетчика. В программе из листинга 6.10 инкрементирование счетчика выполняется по завершении итерации. Такой вариант исключает возможность случайного увеличения значения счетчика. Следовательно, желательно совместить действия по проверке и обновлению значения счетчика в одном выражении, воспользовавшись конструкцией count++ <= NUMBER, однако инициализация счетчика все еще проводится за пределами цикла, при этом сохраняется вероятность того, что мы забудем инициализировать счетчик. Опыт нас учит: то, что может случиться, в конечном итоге когда-нибудь случается, поэтому мы должны хорошо изучить управляющий оператор, лишенный этих проблем.

Цикл for

Цикл for собирает все три указанных выше действия (инициализация, проверка и обновление) в одном месте. Воспользовавшись циклом for, вы сможете заменить предыдущую программу кодом, показанным в листинге 6.11.

Листинг 6.11. Программа sweetie2.c

```
// sweetie2.c -- цикл со счетчиком, использующий for
#include <stdio.h>
int main(void)
```

```
{
   const int NUMBER = 22;
   int count;

   for (count = 1; count <= NUMBER; count++)
      printf("Будьте моим Валентином!\n");

   return 0;
}
```

В круглых скобках, следующих за ключевым словом for, содержатся три выражения, отделенные друг от друга двумя точками с запятой. В первом выражении выполняется инициализация. Это делается только один раз, когда цикл for начинает выполняться. Второе выражение — проверяемое условие, оно вычисляется перед каждым возможным выполнением цикла. Когда выражение получает значение false (а именно, когда значение счетчика count становится больше величины NUMBER), выполнение цикла прекращается. Третье выражение, осуществляющее изменение или обновление, вычисляется в конце каждой итерации. Программа из листинга 6.10 использует его для увеличения значения переменной count, однако, этим его использование не ограничивается. Оператор for завершается выполнением простого или составного оператора, следующего за ключевым словом for. Каждое из трех выражений является полным, так что любой побочный эффект в управляющем выражении, такой как увеличение значения той или иной переменной, происходит до вычисления другого выражения. На рис. 6.3 показана структура цикла for.

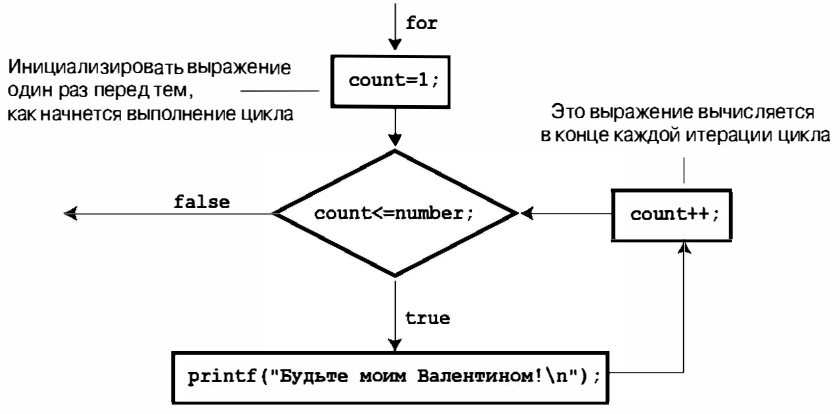


Рис. 6.3. *Структура цикла* for

В качестве еще одного примера программа, представленная в листинге 6.12, использует цикл for для печати таблицы кубов целых чисел.

Листинг 6.12. Программа for_cube.c

```
/* for_cube.c -- использование цикла for для построения таблицы кубов целых
чисел */
#include <stdio.h>
int main(void)
{
```

```
   int num;

   printf("    n    n в кубе\n");
   for (num = 1; num <= 6; num++)
      printf("%5d %5d\n", num, num*num*num);

   return 0;
}
```

Программа в листинге 6.12 печатает целые числа от 1 до и их кубы.

```
    n    n в кубе
    1      1
    2      8
    3     27
    4     64
    5    125
    6    216
```

Первая строка цикла for сразу предоставляет всю информацию о параметрах цикла: начальное значение переменной num, конечное значение переменной num и величину, на которую num увеличивается при каждом проходе цикла.

Использование цикла for с целью повышения гибкости

Хотя цикл for во многом похож на цикл DO в языке FORTRAN, на цикл FOR в языке Pascal и на цикл FOR...NEXT в языке BASIC, он обладает гораздо большей гибкостью, чем любой из выше перечисленных циклов. Эта гибкость обусловлена возможностями использования трех выражений в спецификации цикла for. В приведенных выше примерах первое выражение служило для инициализации счетчика, следующее выражение — для задания граничных значений счетчика и третье выражение — для увеличения значения счетчика на 1. Когда оператор for языка C используется подобным образом, он во многом аналогичен упомянутым выше операторам. В то же время он обладает многими другими возможностями, укажем девять из них:

- Вы можете воспользоваться операцией декремента для отсчета цикла в порядке убывания, а не в порядке возрастания значений счетчика:

```
/* Программа for_down.c */
#include <stdio.h>
int main(void)
{
   int secs;
   for (secs = 5; secs > 0; secs--)
      printf("%d секунд!\n", secs);
   printf("Есть зажигание!\n");
   return 0;
}
```

Ниже показан результат выполнения этой программы:

```
5 секунд!
4 секунд!
```

```
3 секунд!
2 секунд!
1 секунд!
Есть зажигание!
```

- При желании вы можете считать двойками, десятками и так далее:

```
/* Программа for_13s.c */
#include <stdio.h>
int main(void)
{
   int n;                    /* кратность счета 13 */
   for (n = 2; n < 60; n = n + 13)
      printf("%d \n", n);
   return 0;
}
```

Значение переменной n на каждой итерации увеличивается на 13, на печать выводятся следующие выходные данные:

```
2
15
28
41
54
```

- Можно считать по символам, а не по числам:

```
/* Программа for_char.c */
#include <stdio.h>
int main(void)
{
   char ch;
   for (ch = 'a'; ch <= 'z'; ch++)
      printf("Значение ASCII для %c равно %d.\n", ch, ch);
   return 0;
}
```

Фрагмент выходных данных имеет вид:

```
Значение ASCII для a равно 97.
Значение ASCII для b равно 98.
...
Значение ASCII для x равно 120.
Значение ASCII для y равно 121.
Значение ASCII для z равно 122.
```

Эта программа работает, так как символы хранятся в памяти в виде целых чисел, так что итерации во всех случаях используют целые числа.

- Можно выполнять проверку некоторого произвольного условия, а не только количества итераций. В программе for_cube вы можете заменить

```
for (num = 1; num <= 6; num++)
```

на

```
for (num = 1; num*num*num <= 216; num++)
```

Вы можете воспользоваться этим проверяемым условием, если больше озабочены тем, чтобы величина куба того или иного целого числа не превосходила некоторого конкретного значения, а не подсчетом числа итераций.

- Вы можете сделать так, чтобы некоторая величина возрастала не в арифметической, а в геометрической прогрессии, то есть, вместо того, чтобы на каждом этапе цикла прибавлять фиксированную величину, вы можете умножать на фиксированное значение:

```
/* Программа for_geo.c */
#include <stdio.h>
int main(void)
{
   double debt;
   for (debt = 100.0; debt < 150.0; debt = debt * 1.1)
      printf("Теперь ваш долг равен $%.2f.\n", debt);
   return 0;
}
```

В этом фрагменте программы значение переменной `debt` умножается на `1.1` на каждом проходе цикла, что эквивалентно его увеличению каждый раз на 10%. Выходные данные этой программы имеют вид:

```
Теперь ваш долг равен $100.00.
Теперь ваш долг равен $110.00.
Теперь ваш долг равен $121.00.
Теперь ваш долг равен $133.10.
Теперь ваш долг равен $146.41.
```

- В качестве третьего выражения можно использовать любое корректно составленное выражение. Какое значение вы бы не ввели, оно будет обновляться на каждой итерации.

```
/* Программа for_wild.c */
#include <stdio.h>
int main(void)
{
   int x;
   int y = 55;
   for (x = 1; y <= 75; y = (++x * 5) + 50)
      printf("%10d %10d\n", x, y);
   return 0;
}
```

В результате выполнения этого цикла на печать выводятся значения переменной `x` и алгебраического выражения `++x * 5 + 50`. Выходные данные имеют следующий вид:

```
1    55
2    60
3    65
4    70
5    75
```

Обратите внимание на то, что проверяются значения y, а не x. В каждом из трех выражений, управляющих работой цикла for, могут использоваться различные переменные. (Следует также отметить, что хотя этот пример и составлен правильно, он не может служить образом хорошего стиля программирования. Программа только выиграла бы в наглядности, если бы мы не смешивали процесс обновления значений с алгебраическими вычислениями.)

- Вы даже можете опустить одно или несколько выражений (но не забудьте при этом проставить точки с запятыми). Не забудьте также включить в цикл тот или иной оператор, который в конечном итоге приведет к завершению цикла.

```
/* Программа for_none.c */
#include <stdio.h>
int main(void)
{
   int ans, n;
   ans = 2;
   for (n = 3; ans <= 25; )
      ans = ans * n;
   printf("n = %d; ans = %d.\n", n, ans);
   return 0;
}
```

В результате выполнения программы получаем следующий результат:

```
n = 3; ans = 54.
```

При выполнении этого цикла значение переменной n остается равным 3. Значение переменной ans вначале равно 2, затем оно увеличивается до 6 и 18 и в конечном итоге становится равным 54. (Значение 18 меньше 25, следовательно, в цикле for выполняется еще одна итерация, а 18 умножается 3, что дает в конечном итоге 54.) Между прочим, пустое управляющее выражение в средней части спецификации цикла рассматривается как истинное, следовательно, приводимый ниже цикл будет выполняться бесконечно:

```
for (; ; )
   printf("Я хочу, чтобы что-то было сделано.\n");
```

- Первое выражение не требует инициализации переменной. Это может быть одна из разновидностей оператора printf(). Необходимо только помнить, что первое выражение вычисляется или выполняется всего лишь один раз, до того как начнут выполняться другие части цикла.

```
/* Программа for_show.c */
#include <stdio.h>
int main(void)
{
   int num = 0;
   for (printf("Продолжайте вводить числа!\n"); num != 6; )
      scanf("%d", &num);
   printf("Вот то число, которое было нужно!\n");
   return 0;
}
```

В этом фрагменте первое число выводится на печать всего лишь один раз, далее он продолжает принимать числа до тех пор, пока вы не введете число 6:

```
Продолжайте вводить числа!
3
5
8
6
Вот то число, которое было нужно!
```

- Параметры выражений, входящих в спецификацию цикла, могут быть изменены с помощью действий, выполняемых внутри тела цикла. Например, предположим, что цикл определен следующим образом:

  ```
  for (n = 1; n < 10000; n = n + delta)
  ```

  И если после нескольких итераций ваша программа придет к выводу, что значение `delta` слишком мало или слишком велико, оператор `if` (см. главу 7) внутри цикла может изменить величину `delta`. В интерактивной программе пользователь может изменить значение `delta` в процессе выполнения цикла. Этот вид регулировки таит в себе определенную опасность, например, если переменной `delta` присвоить значение `0`, то это приведет вас (равно как и цикл) в никуда.

Короче говоря, свобода, которая предоставлена вам при выборе выражений, управляющих работой цикла `for`, позволяет этому циклу делать гораздо больше, чем просто выполнять фиксированное число операций. Эти возможности цикла `for` могут быть еще больше расширены благодаря использованию операций, которые мы вскоре обсудим.

Сводка: оператор `for`

Ключевое слово: `for`

Комментарии общего характера:

Оператор `for` использует три выражения, управляющие работой цикла, которые отделяются друг от друга точками с запятой. Выражение `инициализация` вычисляется только один раз, до того, как будет выполнен хотя бы один из операторов цикла. Затем вычисляется выражение `проверка`, и если оно истинно (или не равно нулю), то один раз выполняется тело цикла. Далее вычисляется выражение `обновление`, после чего снова вычисляется выражение `проверка`. Оператор `for` представляет собой цикл с предусловием — решение, выполнять в очередной раз цикл или нет, принимается перед проходом цикла. В силу этого обстоятельства существует вероятность того, что тело цикла не будет выполнено ни разу. Операторная часть формы может быть как простым, так и составным оператором.

Форма:

```
for (инициализация; проверка; обновление)
   оператор
```

Цикл повторяется до тех пор, пока выражение *проверка* не получит значение `false` или не станет нулем.

Пример:

```
for (n = 0; n < 10; n++)
   printf(" %d %d\n", n, 2 * n + 1);
```

Дополнительные операции присваивания: +=, -=, *=, /=, %=

В языке C имеется несколько операций присваивания. Важнейшей из них, несомненно, является операция =, которая просто присваивает значение выражения, стоящего справа от знака операции, переменной, стоящей слева от этого знака. Другие операции присваивания обновляют значения переменных. В записи каждой такой операции имя переменной стоит слева от знака операции, а выражение — справа от знака. Переменной присваивается новое значение, равное ее старому значению, скорректированному на величину выражения, стоящего справа. Точный результат корректировки зависит от операции. Например:

```
scores += 20    то же, что и    scores = scores + 20
dimes -= 2      то же, что и    dimes = dimes - 2
bunnies *= 2    то же, что и    bunnies = bunnies * 2
time /= 2.73    то же, что и    time = time / 2.73
reduce %= 3     то же, что и    reduce = reduce % 3
```

В представленном выше перечне использовались простые числа, однако вполне можно применять и более сложные выражения:

```
x *= 3 * y + 12 то же, что и    x = x * (3 * y + 12)
```

Операции присваивания, которые мы только что обсудили, обладают такими же приоритетами, что и операция =, то есть, меньшими, чем приоритет операции + или *. Низкий приоритет этих операций наглядно демонстрирует последний пример, в котором 12 прибавляется к 3 * y до того, как результат сложения будет умножен на x.

Вовсе не обязательно пользоваться всеми этими формами. В то же время они отличаются большей компактностью и позволяют получать более эффективный машинный код, чем более громоздкие формы записи. Сочетания различных операций присваивания особенно полезны в тех случаях, когда вы пытаетесь "втиснуть" какие-то сложные выражения в спецификацию цикла for.

Операция запятой

Операция запятой повышает гибкость цикла for, позволяя включить в его спецификацию сразу несколько инициализирующих или корректирующих выражений. Например, в листинге 6.13 представлена программа, которая выводит на печать тарифы почтового обслуживания первого класса. (На момент написания этой книги тарифы составляли 37 центов за первую унцию и 23 цента за каждую последующую унцию пересылаемого груза.)

Листинг 6.13. Программа postage.c

```
// postage.c -- тарифы почтового обслуживания первого класса
#include <stdio.h>
int main(void)
{
   const int FIRST_OZ = 37;
```

```
    const int NEXT_OZ = 23;
    int ounces, cost;

    printf(" унции  тариф\n");
    for (ounces=1, cost=FIRST_OZ; ounces <= 16; ounces++,
          cost += NEXT_OZ)
       printf("%5d $%4.2f\n", ounces, cost/100.0);

    return 0;
}
```

Первые пять строк выходных данных этой программы имеют вид:

```
унции  тариф
   1   $0.37
   2   $0.60
   3   $0.83
   4   $1.06
```

В этой программе операция запятой используется в инициализирующем и обновляющем выражениях. Ее наличие в первом выражении приводит к инициализации переменных `ounces` и `cost`. Ее второе появление приводит к тому, что значение `ounces` увеличивается на 1, а значение `cost` увеличивается на 23 (значение константы `NEXT_OZ`) на каждой итерации. Все эти вычисления выполняются в спецификации цикла `for` (см. рис 6.4).

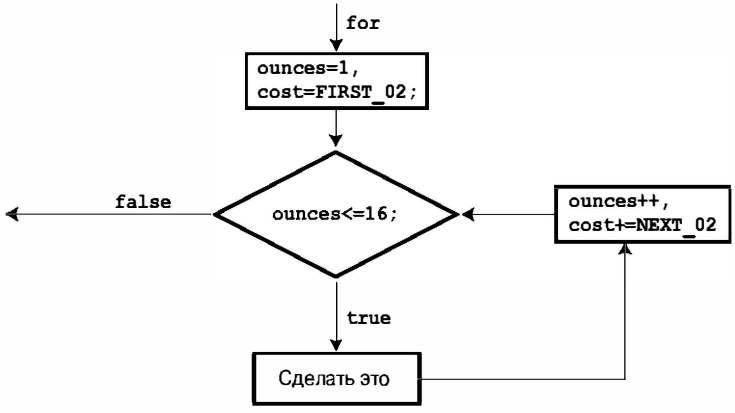


Рис. 6.4. *Операция запятой и цикл* `for`

Применение операции запятой не ограничивается только циклами `for`, однако, именно здесь она чаще всего используется. Эта операция имеет еще два характерных свойства. Во-первых, она гарантирует, что выражения, которые она отделяет одно от другого, вычисляются в порядке слева направо. (Другими словами, запятая является точкой последовательности, это значит, что все побочные эффекты слева от запятой проявятся до того, как программа перейдет вправо от запятой.) Отсюда следует, что переменная `ounces` инициализируется раньше переменной `cost`. В рассматриваемом примере порядок не имеет значения, однако он важен в тех случаях, когда выражение для `cost` содержит переменную `ounces`.

Рассмотрим, например, такое выражение:

```
ounces++, cost = ounces * FIRST_OZ
```

В данном случае значение переменной ounces увеличивается на 1, после чего новое значение ounces используется во втором подвыражении. Запятая, будучи точкой последовательности, гарантирует, что побочные эффекты левого подвыражения проявятся перед тем, как будет вычислено правое подвыражение.

Во-вторых, значение всего выражения, содержащего операцию запятой, является значением выражения в правой части. Результат оператора

```
x = (y = 3, (z = ++y + 2) + 5);
```

состоит в том, что сначала 3 присваивается переменной y, после чего значение y увеличивается до 4, а затем 2 прибавляется к 4, получающееся при этом значение 6 присваивается переменной z, далее 5 прибавляется к z, и в завершение значение переменной z, равное к этому моменту 11, присваивается переменной x. Объяснение, зачем все это надо делать, не входит в задачу данной книги. С другой стороны, предположим, что вы по неосторожности использовали запятую в записи числа:

```
houseprice = 249,500;
```

Это уже не синтаксическая ошибка. Наоборот, язык C интерпретирует правую часть как выражение с запятой, при этом houseprice = 249 представляет собой левое подвыражение, а 500 рассматривается как правое подвыражение. Следовательно, значение всего выражения с запятой — это выражение из правой части, а левый подоператор присваивает значение 249 переменной houseprice. Следовательно, результат этой операции совпадает с результатом выполнения следующего программного кода:

```
houseprice = 249;
500;
```

Вспомните, что любое выражение становится оператором, если ему в конец добавить точку с запятой; следовательно, 500; является оператором, который не производит никаких действий.

С другой стороны, оператор

```
houseprice = (249,500);
```

присваивает переменной houseprice значение 500.

Запятая применяется также в качестве разделителя, так что запятые в выражении

```
char ch, date;
```

и в операторе

```
printf("%d %d\n", chimps, chumps);
```

представляют собой разделители, но не операции запятой.

Сводка: новые операции

Операции присваивания:

Каждая из приведенных ниже операций обновляет переменную, стоящую слева от знака операции, значением, стоящим справа от знака:

`+=` Прибавляет величину справа от знака операции к значению слева от знака.

`-=` Вычитает величину справа от знака операции из значения слева от знака.

`*=` Умножает значение переменной слева от знака операции на величину справа от знака.

`/=` Делит значение переменной слева от знака операции на величину справа от знака.

`%=` Возвращает остаток от деления значения переменной слева от знака операции на величину справа от знака.

Пример:

```
rabbits *= 1.6;
```

это то же самое, что и

```
rabbits = rabbits * 1.6;
```

Эти комбинированные операции присваивания имеют такой же низкий приоритет, что и традиционная операция присваивания, то есть меньший, чем приоритет арифметических операций. В силу этого обстоятельства, следующий оператор

```
contents *= old_rate + 1.2;
```

дает тот же результат, что и приведенный ниже оператор:

```
contents = contents * (old_rate + 1.2);
```

Операция запятой:

Операция запятой связывает два выражения в одно и гарантирует, что выражение, находящееся слева от знака операции, вычисляется первым. Обычно она используется для того, чтобы включить как можно больше информации в управляющее выражение цикла `for`. Значение всего выражения есть значение выражения справа от знака операции.

Пример:

```
for (step = 2, fargo = 0; fargo < 1000; step *= 2)
   fargo += step;
```

Греческий философ Зенон и цикл `for`

Посмотрим, как с помощью цикла `for` и операции запятой можно разрешить древний парадокс. Греческий философ Зенон утверждал, что стрела никогда не поразит цели. Сначала, говорил он, стрела пролетает половину пути до цели. Затем она пролетает половину оставшегося пути, затем она пролетит половину того пути, что остался, и так до бесконечности. Поскольку весь путь стрелы разбит на бесконечное количество частей, утверждал Зенон, стреле потребуется бесконечно большой промежуток времени для достижения конца пути. Однако мы сомневаемся в том, что Зенон добровольно согласился бы стать живой мишенью, чтобы доказать свою правоту.

Применим количественный подход и предположим, что стреле потребуется одна секунда, чтобы пролететь первую половину пути, затем ей потребуется 1/2 секунды, чтобы пролететь половину пути, который остался, еще 1/4 секунды, чтобы преодолеть половину пути, который остался после этого, и так далее. Полное время пролета стрелы можно представить в виде следующей бесконечной последовательности:

```
1 + 1/2 + 1/4 + 1/8 + 1/16 + ...
```

Короткая программа, показанная в листинге 6.14, вычисляет сумму нескольких первых элементов этой последовательности.

Листинг 6.14. Программа zeno.c

```
/* zeno.c -- сумма последовательности */
#include <stdio.h>

int main(void)
{
   int t_ct;           // счетчик элементов последовательности
   double time, x;
   int limit;

   printf("Введите необходимое количество элементов последовательности: ");
   scanf("%d", &limit);
   for (time=0, x=1, t_ct=1; t_ct <= limit; t_ct++, x *= 2.0)
   {
      time += 1.0/x;
      printf("Время = %f, когда количество элементов = %d.\n", time, t_ct);
   }

   return 0;
}
```

Результат выполнения программы, в ходе которого суммируются первые 15 элементов последовательности, выглядит следующим образом:

```
Введите необходимое количество элементов последовательности: 15
Время = 1.000000, когда количество элементов = 1.
Время = 1.500000, когда количество элементов = 2.
Время = 1.750000, когда количество элементов = 3.
Время = 1.875000, когда количество элементов = 4.
Время = 1.937500, когда количество элементов = 5.
Время = 1.968750, когда количество элементов = 6.
Время = 1.984375, когда количество элементов = 7.
Время = 1.992188, когда количество элементов = 8.
Время = 1.996094, когда количество элементов = 9.
Время = 1.998047, когда количество элементов = 10.
Время = 1.999023, когда количество элементов = 11.
Время = 1.999512, когда количество элементов = 12.
Время = 1.999756, когда количество элементов = 13.
Время = 1.999878, когда количество элементов = 14.
Время = 1.999939, когда количество элементов = 15.
```

Легко заметить, что хотя мы и прибавляем все новые элементы, общая сумма, по-видимому, не превысит некоторой величины. И в самом деле, математики доказали, что сумма этой последовательности стремится к 2.0 по мере того, как число просуммированных элементов последовательности стремится к бесконечности, на что указывают и результаты выполнения этой программы. Рассмотрим следующие математические выкладки.

Предположим, что S представляет собой сумму последовательности:

```
S = 1 + 1/2 + 1/4 + 1/8 + ...
```

Здесь многоточие означает "и так далее". Разделив S на 2, получаем

```
S/2 = 1/2 + 1/4 + 1/8 + 1/16 + ...
```

Вычитание второго выражения из первого дает

```
S - S/2 = 1 +1/2 -1/2 + 1/4 -1/4 +...
```

За исключением первого элемента, равного 1, все остальные значения образуют пары, в которых одно значение положительное, а равное ему второе значение отрицательное, так что эти элементы уничтожают друг друга, и в результате получаем

```
S/2 = 1.
```

И, наконец, умножение обеих сторон на 2 дает

```
S = 2.
```

Отсюда можно сделать полезный вывод: прежде чем пускаться в трудоемкие вычисления, проверьте, не нашли ли математики более простого способа получить тот же результат.

Что можно сказать о самой программе? Она показывает, что вы можете в выражениях использовать более одной операции запятой. В ней вы инициализировали переменные time, x и count. После того, как вы определили условия выполнения цикла, программа оказалась очень короткой.

Цикл с постусловием: do while

Циклы while и for представляют собой циклы с предусловиями. Проверяемые условия вычисляются *перед* каждой итерацией цикла, таким образом, существует вероятность, что операторы, заложенные в цикл, никогда не будут выполнены. В C имеется также цикл, в котором проверка условия осуществляется на выходе из каждой итерации цикла, благодаря чему гарантируется выполнение операторной части цикла, по меньшей мере, один раз. Эта разновидность цикла называется циклом do while. Листинг 6.15 являет собой пример такого цикла.

Листинг 6.15. Программа do_while.c

```
/* do_while.c -- цикл с постусловием */
#include <stdio.h>

int main(void)
{
   const int secret_code = 13;
   int code_entered;

   do
   {
      printf("Чтобы войти в клуб лечения шпиономании,\n");
      printf("пожалуйста, введите секретный код: ");
      scanf("%d", &code_entered);
   } while (code_entered != secret_code);
   printf("Поздравляем! Вы уже здоровы!\n");

   return 0;
}
```

Программа в листинге 6.15 читает входные значения до тех пор, пока пользователь не введет число 13. Ниже показан результат выполнения этой программы:

```
Чтобы войти в клуб лечения шпиономании,
пожалуйста, введите секретный код: 12
Чтобы войти в клуб лечения шпиономании,
пожалуйста, введите секретный код: 14
Чтобы войти в клуб лечения шпиономании,
пожалуйста, введите секретный код: 13
Поздравляем! Вы уже здоровы!
```

Эквивалентная программа, в которой используется цикл while, будет несколько длиннее, как можно видеть в листинге 6.16.

Листинг 6.16. Программа entry.c

```
/* entry.c -- цикл с постусловием */
#include <stdio.h>
int main(void)
{
   const int secret_code = 13;
   int code_entered;

   printf("Чтобы войти в клуб лечения шпиономании,\n");
   printf("пожалуйста, введите секретный код: ");
   scanf("%d", &code_entered);
   while (code_entered != secret_code)
   {
      printf("Чтобы войти в клуб лечения шпиономании,\n");
      printf("пожалуйста, введите секретный код: ");
      scanf("%d", &code_entered);
   }
   printf("Поздравляем! Вы уже здоровы!\n");

   return 0;
}
```

Общая форма цикла do while имеет вид:

```
do
   оператор
while  ( выражение );
```

Оператор может быть простым и составным. Обратите внимание на то, что цикл do while считается оператором и требует, чтобы после него стояла точка с запятой, как показано на рис. 6.5.

Цикл do while выполняется, по меньшей мере, один раз, поскольку проверка условия производится только после того как тело цикла выполнено. С другой стороны, может случиться так, что циклы for и while не будут выполнены ни разу, поскольку проверка условия цикла производится перед выполнением операторной части цикла. Цикл do while следует использовать только в тех случаях, когда требуется, по меньшей мере, одна итерация.

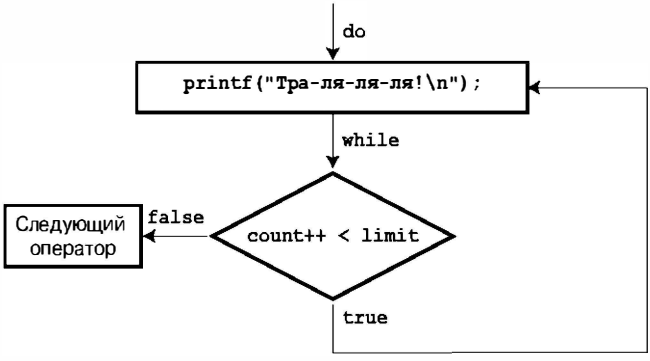


Рис. 6.5. *Структура цикла* `do while`

Например, программа проверки пароля наряду с приведенными ниже строками псевдокода должна содержать и такой цикл:

```
do
{
   приглашение ввести пароль
   чтение данных, введенных пользователем
} while (ввод не совпадает с паролем);
```

Избегайте применения структур `do while`, которые имеют тип, демонстрируемый следующим псевдокодом:

```
do
{
   запрос пользователю, намерен ли он продолжать
   некоторые умные действия
} while (ответом является 'да');
```

В данном случае, после того как пользователь ответит "нет", умные действия со стороны пользователя непременно последуют, поскольку проверка выполняется слишком поздно.

Сводка: оператор `do while`

Ключевые слова:

`do while`

Комментарии общего характера:

Оператор `do while` создает цикл, который повторяется до тех пор, пока проверяемое *выражение* не станет ложным или равным нулю. Оператор `do while` является циклом с постусловием, то есть решение о необходимость выполнения операторной части цикла еще раз, принимается после очередного прохождения цикла. Поэтому цикл будет выполнен, по меньшей мере, один раз. Часть *оператор* формы может быть либо простым, либо составным оператором.

Форма:

```
do
   оператор
while (выражение);
```

Часть *оператор* повторяется до тех пор, пока *выражение* не станет ложным или не получит нулевое значение.

Пример:

```
do
   scanf("%d", &number);
while (number != 20);
```

Какой цикл выбрать?

Когда вы приходите к выводу, что без цикла никак не обойтись, перед вами встает вопрос, каким циклом следует воспользоваться? Во-первых, определитесь с тем, какому типу условия должен отвечать ваш цикл — предусловию или постусловию. В большинстве случаев выбирается цикл с предусловием. Существует целый ряд причин, в силу которых квалифицированные программисты обычно отдают предпочтение циклам с предусловием. Во-вторых, программа становится более понятной, если условие цикла находится в начале цикла. И, наконец, во многих случаях нужно полностью пропустить цикл, если условие на входе в цикл не выполняется.

Допустим, что вам нужен цикл с предусловием. Должен ли это быть цикл `for` или цикл `while`? Частично это дело вкуса, поскольку то, что вы можете сделать с помощью одного типа цикла, вы можете сделать и с помощью другого. Чтобы сделать цикл `for` подобным циклу `while`, вы можете опустить первое и третье выражения. Например:

```
for ( ;проверка; )
```

то же самое, что и

```
while (проверка)
```

Чтобы придать циклу `while` такой же вид, как цикл `for`, выполните перед началом цикла инициализацию и включите операторы, осуществляющие обновления значений. Например:

```
инициализация;
while (проверка)
{
   тело-цикла;
   обновление-значений;
}
```

то же самое, что и

```
for (инициализация; проверка; обновление)
   тело-цикла;
```

С точки зрения преобладающего стиля, цикл `for` подходит больше в тех случаях, когда цикл предусматривает инициализацию и обновление переменной, а цикл `while` предпочтительнее, когда этого делать не нужно. Цикл `while` целесообразно использовать в следующих условиях:

```
while (scanf("%ld", &num) == 1)
```

Цикл `for` представляется более естественным выбором, когда в теле цикла производится вычисление значения индекса:

```
for (count = 1; count <= 100; count++)
```

Вложенные циклы

Вложенный цикл — это цикл внутри другого цикла. Очень часто вложенные циклы используются для упорядоченного отображения данных в форме строк и столбцов. Один цикл может взять на себя обработку, скажем, всех столбцов в строке, в то время как второй цикл занимается обработкой строк. В листинге 6.17 представлен простой пример.

Листинг 6.17. Программа `rows1.c`

```
/* rows1.c -- использование вложенных циклов */
#include <stdio.h>
#define ROWS 6
#define CHARS 10
int main(void)
{
   int row;
   char ch;
   for (row = 0; row < ROWS; row++)                /* строка 10 */
   {
      for (ch = 'A'; ch < ('A' + CHARS); ch++)     /* строка 12 */
         printf("%c", ch);
      printf("\n");
   }
   return 0;
}
```

В результате выполнения этой программы получены следующие результаты:

```
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
```

Анализ программы

Цикл `for`, начинающийся в строке 10, называется *внешним*, а цикл, который берет начало в строке 12, называется *внутренним*. Внешний цикл начинается при значении `0` переменной `row` и завершается, когда ее значение становится равным `6`. В силу этого обстоятельства внешний цикл проходит через шесть итераций, при этом переменная `row` последовательно получает значения от `0` до `5`. Первый оператор в каждой итерации является внутренним для цикла `for`. Этот цикл выполняет 10 итераций, печатая символы от `A` до `J` в одной строке. Вторым оператором внешнего цикла является `printf("\n");`. Этот оператор начинает новую строку, это значит, что в следующий раз, когда начинает работу внутренний цикл, выходные данные будут печататься с новой строки.

Обратите внимание, что в условиях вложенного цикла внутренний цикл проходит весь диапазон итераций для каждой итерации внешнего цикла. В последнем примере внутренний цикл печатает 10 символов в строке, а внешний цикл печатает 6 таких строк.

Вариации вложенных циклов

В предыдущем примере внутренний цикл выполнял одни и те же действия для каждой итерации внешнего цикла. Вы можете сделать так, чтобы внутренний цикл изменял свое поведение в зависимости от внешнего цикла.

Например, программа в листинге 6.18 является незначительной модификацией предыдущей программы, она выбирает начальный символ последовательности символов, распечатываемых внутренним циклом в зависимости от номера итерации внешнего цикла. К тому же в ней присутствует комментарий нового стиля и `const` вместо `#define`, чтобы можно было удобно работать в условиях обоих подходов.

Листинг 6.18. Программа `rows2.c`

```
// rows2.c -- использование зависимых вложенных циклов
#include <stdio.h>
int main(void)
{
   const int ROWS = 6;
   const int CHARS = 6;
   int row;
   char ch;

   for (row = 0; row < ROWS; row++)
   {
      for (ch = ('A' + row); ch < ('A' + CHARS); ch++)
         printf("%c", ch);
      printf("\n");
   }

   return 0;
}
```

На этот раз выходные данные принимают следующий вид:

```
ABCDEF
BCDEF
CDEF
DEF
EF
F
```

Поскольку значение переменной `row` добавляется к символу `'A'` во время каждой итерации внешнего цикла, переменная `ch` инициализируется в каждой строке символом, следующим в алфавите за символом, который использовался для предыдущей инициализации. Проверяемое условие, однако, не меняется, так что каждая новая строка заканчивается символом `F`. В силу этого обстоятельства, в каждой строке выводится на один символ меньше, чем в предыдущей.

Введение в массивы

Массивы играют важную роль во многих программах. Они позволяют хранить некоторое количество элементов однотипной информации в удобной форме. Массивам полностью посвящена глава 10, но поскольку для работы с массивами очень часто используются циклы, имеет смысл ознакомиться с их характерными особенностями.

Массив — это некоторая совокупность значений одного и того же типа, например, 10 значений типа `char` или 15 значений типа `int`, хранящихся в памяти в виде непрерывной последовательности. Массив имеет свое собственное имя, а доступ к отдельным его *элементам* осуществляется с помощью целочисленного индекса. Например, объявление

```
float debts[20];
```

уведомляет о том, что `debts` является массивом, состоящим из 20 элементов, каждый из которых может содержать в себе значение типа `float`. Первый элемент массива получает имя `debts[0]`, второй элемент — `debts[1]` и так далее, вплоть до `debts[19]`. Обратите внимание, что нумерация элементов массива начинается с 0, а не с 1. Каждому элементу данного массива может быть присвоено значение типа `float`. Например, допустимы операторы следующего вида:

```
debts[5] = 32.54;
debts[6] = 1.2e+21;
```

По сути дела, вы можете использовать тот или иной элемент массива так же, как и вы использовали бы переменную того же типа. Например, вы можете читать значение в конкретный элемент массива:

```
scanf("%f", &debts[4]); // читать значение в пятый элемент массива
```

Один из источников возможных ошибок состоит в том, что язык C для повышения скорости вычислений не проверяет, правильно ли выбран индекс элемента массива. Например, каждый из приведенных ниже операторов является ошибочным:

```
debts[20] = 88.32;      // в данном массиве такого элемента нет
debts[33] = 828.12;     // в данном массиве такого элемента нет
```

В то же время компилятор не обнаруживает ошибок подобного рода. В ходе выполнения программы такие операторы размещают данные в ячейках памяти, которые, возможно, используются другими данными, они способны запортить выходные данные программы и даже вызвать ее аварийное завершение.

Массивы могут быть однородными последовательностями данных любого типа:

```
int nannies[22];    /* массив для хранения 22 целых чисел */
char actors[26];    /* массив для хранения 26 символов */
long big[500];      /* массив для хранения 500 целых чисел типа long */
```

Например, ранее в книге речь шла о строках, представляющих собой специальный случай данных, которые можно размещать в массивах типа `char`. (В общем случае массив типа `char` — это такой массив, элементы которого имеют тип `char`.) Содержимое массива тип `char` образуют строку в том случае, когда этот массив содержит пустой символ `\0`, который обозначает конец строки (рис. 6.6).

Рис. 6.6. *Символьные массивы и строки*

Числа, используемые для идентификации элементов массива, называются *указателями*, *индексами* или *смещениями*. Индексами могут быть целые числа, и, как было сказано выше, индексы начинаются с 0. Элементы массива хранятся в памяти, непосредственно примыкая друг к другу, как показано на рис. 6.7.

Рис. 6.7. *Массивы* `char` *и* `int` *в памяти*

Использование цикла `for` при работе с массивами

Массивы очень широко используются в программах. В листинге 6.19 представлен один из простейших примеров применения этой структуры данных. Программа считывает результаты 10 игр в гольф, которые позже подвергаются дальнейшей обработке. Используя массив, вы избегаете необходимости объявлять 10 переменных под различными именами, по одной переменной для результата каждой игры. Опять-таки, вы можете воспользоваться циклом `for` для считывания входных данных. Программа предназначена для подсчета общей суммы очков, их среднего значения и гандикапа, который представляет собой разность между средним и стандартным числом очков, или пар.

Листинг 6.19. Программа `scores_in.c`

```
// scores_in.c -- использование циклов для обработки массивов
#include <stdio.h>
#define SIZE 10
#define PAR 72
```

```
int main(void)
{
   int index, score[SIZE];
   int sum = 0;
   float average;
   printf("Введите %d результатов игры в гольф:\n", SIZE);
   for (index = 0; index < SIZE; index++)
      scanf("%d", &score[index]);  // прочитать 10 результатов игры в гольф
   printf("Введены следующие результаты:\n");
   for (index = 0; index < SIZE; index++)
      printf("%5d", score[index]); // проверить введенную информацию
   printf("\n");
   for (index = 0; index < SIZE; index++)
      sum += score[index];         // сложить результаты
   average = (float) sum / SIZE;   // распределение во времени
   printf("Окончательная сумма очков = %d, среднее значение = %.2f\n",
          sum, average);
   printf("Полученный гандикап равен %.0f.\n", average - PAR);
   return 0;
}
```

Сначала мы посмотрим, как работает программа из листинга 6.19, а после этого прокомментируем ее действия. Ниже показаны выходные данные:

```
Введите 10 результатов игры в гольф:
102  98  112  108  105  103
99  101  96  102  100
Введены следующие результаты:
  102  98  112  108  105  103  99  101  96  102
Окончательная сумма очков = 1026, среднее значение  = 102.60
Полученный гандикап равен 31.
```

В общем, программа работает, а теперь рассмотрим некоторые ее особенности. Во-первых, обратите внимание на тот факт, что, хотя вы и напечатали 11 чисел, считаны только 10 из них, так как цикл, выполняющий чтение, вводит только 10 значений. Поскольку функция `scanf()` игнорирует пробелы, вы можете ввести все 10 чисел в одной строке, поместить каждое число в отдельную строку, либо, как в рассматриваемом случае, использовать смесь символов новой строки и пробелов, чтобы отделить входные значения одно от другого. (Так как входные данные запоминаются в буфере, эти числа передаются в программу только после нажатия клавиши <Enter>.)

Далее, работать с массивами и циклами гораздо удобнее, чем использовать 10 отдельных операторов `scanf()` и 10 операторов `printf()` для того, чтобы прочитать 10 результатов игры. Цикл `for` предлагает простой способ использования индексов массивов. Обратите внимание, что программа манипулирует элементами массива типа `int` так же, как и переменными типа `int`. Чтобы прочитать переменную `fue` типа `int`, вы должны прибегнуть к помощи функции `scanf("%d", &fue)`. Программа из листинга 6.19 производит считывание элемента `score[index]` типа `int`, следовательно, она использует выражение `scanf("%d", &score[index])`.

Этот пример является иллюстрацией нескольких моментов, характеризующих стиль программирования. Во-первых, удачной является идея использования директивы #define для создания именованной константы (SIZE), описывающий размер массива. Вы используете эту константу при описании массива и при выборе пределов циклов. Если вам в дальнейшем потребуется расширить программу до 20 счетов, достаточно просто переопределить константу SIZE, установив ее равной 20. При этом отпадает необходимость менять каждую часть программы, в которой используется размер массива. Стандарт C99 позволяет использовать значения типа const для определения размеров массива, в то время как в стандарте C90 это невозможно, зато директива #define разрешена и в том, и в другом стандартах.

Во-вторых, конструкция

```
for (index = 0; index < SIZE; index++)
```

удобна для обработки массива размером SIZE. Очень важно задать размеры массива. Первый элемент массива имеет индекс 0, поэтому цикл начинается с того, что устанавливается значение переменной index равным 0. Поскольку отсчет индексов начинается с 0, индекс последнего элемента принимает значение SIZE - 1. То есть, десятый элемент массива — это score[9]. Проверяемое условие index < SIZE учитывает это обстоятельство, благодаря чему последним значением переменной index, использованным в цикле, является SIZE - 1.

В-третьих, хороший тон в программировании требует от программы проверять значения, которые были только что введены (эхо-контроль). Это позволяет обрести уверенность в том, что программа выполняет обработку именно тех данных, которые вы ввели.

И, наконец, обратите внимание на то, что в программе, представленной в листинге 6.19, используются три отдельных цикла for. У вас могут возникнуть сомнения, нужно ли столько циклов на самом деле. Нельзя ли совместить хотя бы некоторые операции в одном цикле? Ответом будет "да", то есть вы можете некоторые из операций выполнить в одном цикле. Это сделало бы программу более компактной. Однако этому мешает принцип *модульности*. Идея, которая стоит за этим термином, состоит в том, что программа должна быть разбита на отдельные фрагменты, при этом каждый фрагмент должен решать конкретную задачу. Это облегчает чтение программы. Но что, возможно, еще важнее — модульность облегчает решение задачи обновления или модификации программы, если ее фрагменты взаимно не переплетаются. Когда вы узнаете больше о функциях, вы сможете превратить каждый такой фрагмент в отдельную функцию, увеличивая тем самым степень модульности конечной программы.

Пример цикла, использующего возвращаемое значение функции

Последний пример в этой главе использует функцию, которая вычисляет результат возведения чисел в заданную целую степень. (Для решения более серьезных задач по обработке чисел в библиотеке math.h предусмотрена более мощная функция pow(), которая позволяет возводить вещественные числа в степень, также представленную вещественным числом.) Тремя основными задачами, которые решаются в этом примере, являются: разработка алгоритма вычисления ответа, представление этого алго-

ритма в виде функции, которая возвращает ответ, и разработка удобного метода тестирования этой функции.

Сначала рассмотрим алгоритм. Упростим задачу, ограничившись вычислением только положительных целых степеней. В таком случае, если вы захотите возвести n в степень p, вы должны умножить n само на себя p раз. Эта задача легко решается с помощью цикла for. Вы можете присвоить переменной pow значение 1, а затем многократно умножать ее на n:

```
for(i = 1; i <= p; i++)
   pow *= n;
```

Напомним, что в результате выполнения операции *= левая часть выражения умножается на правую. После выполнения первой операции цикла значение переменной pow равно 1, умноженное на n, то есть n. По завершении второго цикла значение переменной pow равно ее предыдущему значению (n), умноженному на n, или n в квадрате, и так далее. В этом контексте цикл for вполне обоснован, поскольку цикл выполняется заранее известное количество раз (после того, как станет известным p).

Теперь, когда разработан алгоритм, вы должны решить, какой тип данных необходимо использовать. Показатель степени p, будучи целым числом, должен иметь тип int. Чтобы обеспечить достаточно широкий диапазон значений переменной n и ее степеней, для переменных n и pow выбирается тип double.

Далее, нужно подумать, как написать функцию. Вы должны передать функции два значения, а функция должна вернуть одно значение. Чтобы передать функции необходимую информацию, вы можете воспользоваться двумя аргументами, один из которых имеет тип double, а другой — тип int; эти аргументы определяют, какое число возводится в какую степень. Как устроить, чтобы функция возвращала значение в вызывающую программу? Чтобы написать функцию с возвращаемым значением, выполните следующие действия:

1. При определении функции установите тип значения, которое она возвращает.
2. Используйте ключевое слово return, чтобы указать возвращаемое значение.

Например, вы можете поступить так:

```
double power(double n, int p)     // возвращает значение типа double
{
   double pow = 1;
   int i;
   for (i = 1; i <= p; i++)
      pow *= n;
   return pow;                    // возвращает значение переменной pow
}
```

Чтобы объявить тип функции, укажите тип перед именем функции точно так же, как это делается при объявлении переменной. Ключевое слово return заставляет функцию вернуть следующее за ним значение в вызывающую функцию. В рассматриваемом примере вы возвращаете значение переменной, вы также можете возвращать значения выражений. Например, приведенная ниже конструкция является допустимой:

```
return 2 * x + b;
```

Функция вычисляет значение выражения и возвращает его. В вызывающей функции возвращаемое значение может быть присвоено другой переменной, оно может быть использовано как значение выражения, как аргумент другой функции, например, `printf("%f", power(6.28, 3))` или может быть вообще проигнорировано.

Давайте воспользуемся такой функцией в программе. Чтобы протестировать функцию, желательно иметь возможность передать этой функции несколько значений и посмотреть, как она на них прореагирует. Для этого потребуется организовать цикл ввода. Вполне естественно выбрать для этой цели цикл `while`. Вы можете воспользоваться функцией `scanf()`, чтобы прочитать одновременно два значения. Если функция `scanf()` успешно прочтет два значения, она возвратит значение 2, следовательно, вы можете управлять выполнением цикла, сравнивая значение, которое возвращает функция `scanf()`, со значением 2. Еще одна деталь: чтобы воспользоваться функцией `power()` в вашей программе, ее следует объявить подобно тому, как объявляются переменные, используемые в программе. В листинге 6.20 показана результирующая программа.

Листинг 6.20. Программа `power.c`

```
// power.c -- возведение чисел в целую степень
#include <stdio.h>
double power(double n, int p);   // прототип ANSI

int main(void)
{
   double x, xpow;
   int exp;

   printf("Введите число и положительную целую степень,");
   printf(" в которую\nчисло будет возведено. Для завершения программы");
   printf(" введите q.\n");

   while (scanf("%lf%d", &x, &exp) == 2)
   {
      xpow = power(x,exp);      // вызов функции
      printf("%.3g в степени %d равно %.5g\n", x, exp, xpow);
      printf("Введите следующую пару чисел или q для завершения.\n");
   }
   printf("Надеемся, вас удовлетворило качество программы — до свидания!\n");

   return 0;
}

double power(double n, int p)    // определение функции
{
   double pow = 1;
   int i;

   for (i = 1; i <= p; i++)
      pow *= n;
   return pow;                   // возврат значения переменной pow
}
```

Ниже показаны результаты выполнения программы:

```
Введите число и положительную целую степень, в которую
число будет возведено. Для завершения программы введите q.
1.2 12
1.2 в степени 12 равно 8.9161
Введите следующую пару чисел или q для завершения.
2
16
2 в степени 16 равно 65536
Введите следующую пару чисел или q для завершения.
q
Надеемся, вас удовлетворило качество программы — до свидания!
```

Анализ программы

Программа `main()` представляет собой пример *драйвера*, то есть короткой программы, предназначенной для тестирования конкретной функции.

Цикл `while` является обобщением формы, которой мы пользовались раньше. Ввод чисел `1.2 12` заставляет функцию `scanf()` успешно читать эти два значения и вернуть значение 2, после чего цикл продолжает выполняться. Поскольку `scanf()` игнорирует пробелы, входные данные могут быть представлены в нескольких строках, как показывают выходные данные демонстрационного примера, однако ввод символа `q` вызывает возврат значение `0`, поскольку символ `q` не может быть считан при наличии спецификатора `%lf`. Это обстоятельство заставляет функцию `scanf()` вернуть `0` и тем самым прекратить выполнение цикла. Аналогично, ввод значения `2.8 q` вызывает появление возвращаемого значения, равного 1, которое также приводит к завершению цикла.

Теперь рассмотрим все, что связано с использованием функции в программе. Функция `power()` появляется в данной программе три раза. Первый раз она встречается в следующей конструкции:

```
double power(double n, int p);   // прототип ANSI
```

Этот оператор описывает, или *объявляет*, что в программе будет использоваться функция с именем `power()`. Первое ключевое слово `double` показывает, что функция `power()` возвращает значение типа `double`. Компилятору нужно знать, какое значение возвращает функция `power()`, чтобы определить, сколько байтов данных следует ожидать на входе и как их следует интерпретировать; именно по этой причине и следует объявлять функцию. Объявления `double n, int p`, заключенные в круглые скобки, означают, что функции `power()` требуются два аргумента. Первый из них должен иметь тип `double`, а второй — тип `int`.

Второй раз эта функция появляется в следующем операторе:

```
xpow = power(x,exp);       // вызов функции
```

На этот раз программа вызывает функцию и передает ей два значения. Сама функция вычисляет значение `x` в степени `exp` и возвращает результат вызывающей программе, в которой возвращаемое значение присваивается переменной `xpow`.

Третий раз рассматриваемая функция встречается в заголовке определения функции:

```
double power(double n, int p)     // определение функции
```

В данном случае функция power() принимает два параметра, один типа double и второй типа int, представленные значениями переменных n и p. Обратите внимание, что после конструкции power() не стоит точка с запятой, когда она появляется в определении функции, зато точка с запятой следует за ней в объявлении функции. За заголовком расположен программный код, который и определяет, что должна делать функция power().

Напоминаем, что эта функция использует цикл for для вычисления значения n в степени p, а затем присваивает это значение переменной pow. Следующая строка делает это значение pow возвращаемым значением функции:

```
return pow;                       // возврат значения переменной pow
```

Использование функций с возвращаемыми значениями

Объявление функции, вызов функции, определение функции, использующей ключевое слово return — это основные действия при определении и использовании функций с возвращаемыми значениями.

На этом этапе у вас может возникнуть ряд вопросов. Например, если от вас требуется объявить функции до того, как вы воспользуетесь их возвращаемыми значениями, тогда почему можно пользоваться возвращаемыми значениями функции scanf() без объявления самой функции scanf()? Почему нужно объявлять функцию power() отдельно, если в ее определении сказано, что она имеет тип double?

Рассмотрим сначала ответ на второй вопрос. Компилятор должен знать, какой тип имеет функция power(), когда он впервые сталкивается с ней в программе. В этот момент компилятор пока не вышел на определение функции power(), поэтому он еще не знает, что, как сказано в определении, типом возвращаемого функцией значения является double. Чтобы помочь компилятору, вы упреждаете неизбежное появление у компилятора соответствующих вопросов путем использования *предварительного* (или *упреждающего*) *объявления*. Такого рода объявления уведомляют компилятор о том, что где-то в программе будет объявлена функция power() и что она возвращает тип double. Если вы поместите в файле определение функции power() раньше функции main(), вы можете опустить предварительное объявление, поскольку компилятор уже будет знать все, что ему нужно о функции power(), прежде чем он попадет на функцию main().

Однако, такой стиль в языке C не является стандартным. В силу того, что main() обычно представляет собой базовую структуру всей программы, лучше, чтобы main() была представлена первой. Кроме того, функции часто размещаются в отдельных файлах, и в этих случаях их предварительные объявления просто необходимы.

Далее, почему вы не объявили функцию scanf()? А ведь вы ее фактически объявили. В заголовочном файле stdio.h содержатся объявления функций scanf(), printf() и других функций ввода-вывода. В объявлении функции scanf() указано, что она возвращает значение типа int.

Ключевые понятия

Цикл представляет собой мощное средство программирования. При организации цикла вы должны обращать особое внимание на три следующих аспекта:

- Четко определять условия прекращения цикла.
- Убедиться в том, что значения, используемые в проверке условия цикла, инициализированы перед первым их использованием.
- Убедиться в том, что цикл выполняет конкретные действия для обновления проверяемого условия на каждой итерации.

Язык C вычисляет числовое значение проверяемого условия. Результат, равный `0`, рассматривается как ложное значение, любое другое числовое значение трактуется как истинное. Выражения, использующие операции отношения, часто выступают в качестве проверяемых условий, они несколько специфичны. Результатом вычисления выражения отношения является `1`, если оно истинно, и `0`, если оно ложно, что согласуется со значениями, допустимыми для нового типа `_Bool`.

Массив представляет собой совокупность расположенных рядом ячеек памяти с величинами одного и того же типа. Вы не должны забывать, что нумерация элементов массива начинается с `0`, и в силу этого обстоятельства последний элемент массива имеет индекс на единицу меньший, чем количество элементов массива. C не выполняет проверку на правильность использования значений индексов, следовательно, эта обязанность целиком ложится на ваши плечи.

Использование функций состоит из трех отдельных этапов:

1. Объявление функции с указанием ее прототипа.
2. Обращение к функции из программы посредством ее вызова.
3. Определение функции.

Прототип позволяет компилятору проверять, правильно ли вы используете функцию, а определение функции показывает, как эта функция должна работать. Прототип и определение функции — это примеры современного стиля программирования, когда программный элемент делится на интерфейс и реализацию. Интерфейс описывает, как используется то или иное средство, и именно для этой цели и предназначен прототип функции, а реализация детально расписывает конкретные действия программного элемента, что входит в обязанности определения функции.

Резюме

Основной темой настоящей главы было обсуждение возможностей управления ходом выполнения программы. Язык C предлагает различные виды помощи в структуризации ваших программ. Построение циклов с предусловиями производится с помощью операторов `while` и `for`. Операторы `for` особенно удобны для написания циклов, выполняющих инициализацию и обновление значений переменных. Операция запятой позволяет выполнять инициализацию и обновление сразу нескольких переменных в цикле `for`. Для тех редких случаев, когда требуется цикл с постусловием, C предлагает оператор `do while`.

Типичная конструкция цикла while имеет следующий вид:

```
получить первое значение
while (значение удовлетворяет проверяемому условию)
{
   обработать значение
   получить следующее значение
}
```

Цикл for, выполняющий те же действия, имеет следующий вид:

```
for (получить первое значение; значение удовлетворяет проверяемому
условию; получить следующее значение)
   обработать значение
```

Во всех этих циклах используется условие проверки для определения, нужно ли выполнять еще одну итерацию цикла. Другими словами, цикл продолжается, если проверяемое выражение дает в результате ненулевое значение, в противном случае цикл завершается.

Довольно часто проверяемым условием является выражение отношения, которое представляет собой выражение, построенное на базе некоторой операции отношения. Такое выражение получает значение 1, если отношение истинно, и 0 — во всех остальных случаях. Переменные типа _Bool, введенные в употребление стандартом C99, могут принимать только значения 1 или 0, соответствующие true или false.

В дополнение к операциям отношения в этой главе были рассмотрены несколько арифметических операций присваивания языка C, таких как += и *=. Эти операции модифицируют значения операнда слева от знака операции путем выполнения над ним соответствующей арифметической операции.

Следующим предметом обсуждений были массивы. Массивы объявляются путем использования квадратных скобок, в которых указано количество элементов массива. Первый элемент произвольного массива получает номер 0; второй — номер 1 и так далее.

Например, объявление

```
double hippos[20];
```

создает массив из 20 элементов, а отдельные элементы массива получают имена в диапазоне от hippos[0] до hippos[19] включительно. Манипулировать индексами, используемыми для нумерации элементов массива, удобно с помощью циклов.

И, наконец, в этой главе было показано, как создавать и использовать функции с возвращаемыми значениями.

Вопросы для самоконтроля

1. Найдите значения переменной quack в каждой строке.

```
int quack = 2;
quack += 5;
quack *= 10;
quack -= 6;
quack /= 8;
quack %= 3;
```

2. При условии, что переменная value имеет тип int, определите, какие выходные данные будут получены в результате выполнения следующего цикла:

```
for ( value = 36; value > 0; value /= 21 )
   printf("%3d", value);
```

Какие при этом могут возникнуть проблемы, если переменная value будет иметь тип double вместо int?

3. Запишите выражение для каждого из следующих проверяемых условий:
   а. x больше 5.
   б. Функция scanf() предпринимает неудачную попытку прочитать одно значение типа double (с именем x).
   в. x имеет значение 5.
4. Запишите выражение для каждого из следующих проверяемых условий:
   а. функция scanf() успешно читает одно целое число.
   б. x не равно 5.
   в. x равно 20 или больше.
5. Приведенная ниже программа далека от идеала. Какие ошибки вы можете в ней найти?

```
#include <stdio.h>
int main(void)
{                                   /* строка 3 */
   int i, j, list(10);              /* строка 4 */
   for (i = 1, i <= 10, i++)        /* строка 6 */
   {                                /* строка 7 */
      list[i] = 2*i + 3;            /* строка 8 */
      for (j = 1, j > = i, j++)     /* строка 9 */
         printf(" %d", list[j]);    /* строка 10 */
      printf("\n");                 /* строка 11 */
   }                                /* строка 12 */
```

6. Воспользуйтесь вложенными циклами, чтобы создать программу, которая выводит на печать следующую фигуру:

```
$$$$$$$$
$$$$$$$$
$$$$$$$$
$$$$$$$$
```

7. Что выведет на печать каждая из следующих программ?
   а.

```
#include <stdio.h>
int main(void)
{
   int i = 0;
   while (++i < 4)
     printf("Здравствуйте! ");
   do
     printf("До свидания! ");
   while (i++ < 8);
   return 0;
}
```

б.
```
#include <stdio.h>
int main(void)
{
   int i;
   char ch;
   for (i = 0, ch = 'A'; i < 4; i++, ch += 2 * i)
      printf("%c", ch);
   return 0;
}
```

8. Если на вход каждой из приведенных ниже программ подается фраза "Go west, young man!", какие выходные данные они выведут на печать? (Восклицательный знак ! следует сразу за символом пробела в последовательности ASCII.)

а.
```
#include <stdio.h>
int main(void)
{
   char ch;
   scanf("%c", &ch);
   while ( ch != 'g'  )
   {
      printf("%c", ch);
      scanf("%c", &ch);
   }
   return 0;
}
```

б.
```
#include <stdio.h>
int main(void)
{
   char ch;
   scanf("%c", &ch);
   while ( ch != 'g' )
   {
      printf("%c", ++ch);
      scanf("%c", &ch);
   }
   return 0;
}
```

в.
```
#include <stdio.h>
int main(void)
{
   char ch;
   do {
      scanf("%c", &ch);
      printf("%c", ch);
   } while ( ch != 'g' );
   return 0;
}
```

г.
```
#include <stdio.h>
int main(void)
{
   char ch;
   scanf("%c", &ch);
   for ( ch = '$'; ch != 'g'; scanf("%c", &ch) )
      putchar(ch);
   return 0;
}
```

9. Что выведет на печать следующая программа?

```
#include <stdio.h>
int main(void)
   {
   int n, m;
   n = 30;
   while (++n <= 33)
      printf("%d|",n);
   n = 30;
   do
      printf("%d|",n);
   while (++n <= 33);
   printf("\n***\n");
   for (n = 1; n*n < 200; n += 4)
      printf("%d\n", n);
   printf("\n***\n");
   for (n = 2, m = 6; n < m; n *= 2, m+= 2)
      printf("%d %d\n", n, m);
   printf("\n***\n");
   for (n = 5; n > 0; n--)
   {
      for (m = 0; m <= n; m++)
         printf("=");
      printf("\n");
   }
   return 0;
}
```

10. Рассмотрим следующее объявление:

```
double mint[10];
```

а. Какое имя присвоено массиву?

б. Сколько элементов в массиве?

в. Какие виды значений могут храниться в каждом элементе массива?

г. Какие из примеров использования функции `scanf()` применительно к этому массиву правильны?

i. `scanf("%lf", mint[2])`

ii. `scanf("%lf", &mint[2])`

iii. `scanf("%lf", &mint)`

11. Мистер Но предпочитает иметь дело с четными числами, поэтому он написал программу, которая создает массив и заполняет его четными числами 2, 4, 6, 8 и так далее. Есть ли ошибки в этой программе, а если есть, то что это за ошибки?

```
#include <stdio.h>
#define SIZE 8
int main(void)
{
   int by_twos[SIZE];
   int index;

   for (index = 1; index <= SIZE; index++)
      by_twos[index] = 2 * index;

   for (index = 1; index <= SIZE; index++)
      printf("%d ", by_twos);
   printf("\n");

   return 0;
}
```

12. Вы хотите написать программу, которая возвращает значения типа `long`. Что должно включать описание функции в этом случае?

13. Напишите функцию, которая принимает аргумент типа `int` и возвращает квадрат этого значения как тип `long`.

14. Что напечатает следующая программа?

```
#include <stdio.h>
int main(void)
{
   int k;

   for(k = 1, printf("%d: Здравствуйте!\n", k); printf("k = %d\n",k),
      k*k < 26; k+=2, printf("Теперь k равно %d\n", k) )
         printf("k равно %d на итерации\n",k);

   return 0;
}
```

Упражнения по программированию

1. Напишите программу, которая создает массив из 26 элементов и помещает в него 26 букв нижнего регистра. Заставьте ее вывести содержимое массива.

2. Воспользуйтесь вложенными циклами, чтобы написать программу, которая выводит на печать следующую фигуру:

```
$
$$
$$$
$$$$
$$$$$
```

3. Воспользуйтесь вложенными циклами, чтобы написать программу, которая выводит на печать такую фигуру:

```
F
FE
FED
FEDC
FEDCB
FEDCBA
```

Примечание: если система не использует ASCII или некоторые другие кодировки, которые кодируют буквы в числовом порядке, то для инициализации символьного массива буквами алфавита вы можете воспользоваться следующим оператором:

```
char lets[26] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Затем вы можете использовать индексы массива для выбора конкретных букв, например, `lets[0]` равно `'A'` и так далее.

4. Напишите программу, которая запрашивает у пользователя ввод буквы верхнего регистра. Воспользуйтесь вложенными циклами, чтобы написать программу, которая выводит на печать фигуру в виде пирамиды, подобную изображенной ниже:

```
    A
   ABA
  ABCBA
 ABCDCDA
ABCDEDCBA
```

Эта фигура должна разворачиваться в зависимости от того, какой символ был введен. Например, представленная выше фигура должна быть получена в результате ввода символа E. Совет: для обработки строк воспользуйтесь внешним циклом. Три внутренних цикла используйте для работы со строкой: один — для манипуляции пробелами, второй — для печати букв в порядке возрастания и еще одни — для печати букв в порядке убывания. Если ваша система не использует кодировку ASCII или подобную, в которой буквы представлены в строгом числовом порядке, см. примечание в упражнении 3.

5. Напишите программу, печатающую таблицу, в каждой строке которой представлено целое число, его квадрат и его куб. Запросите у пользователя верхний и нижний пределы таблицы. Используйте цикл `for`.

6. Напишите программу, которая читает слово в символьный массив, а затем печатает это слово в обратном порядке. Совет: воспользуйтесь функцией `strlen()` (глава 4) для вычисления индекса последнего символа массива.

7. Напишите программу, которая запрашивает два числа с плавающей запятой и печатает значение их разности, деленной на их произведение. Программа обрабатывает пары вводимых чисел до тех пор, пока пользователь не введет нечисловое значение.

8. Модифицируйте упражнение 7 таким образом, чтобы программа использовала функцию для возврата результатов вычислений.

9. Напишите программу, которая запрашивает ввод верхнего и нижнего пределов последовательности целых чисел, вычисляет сумму всех квадратов целых чисел, начиная с квадрата нижнего целочисленного предела и заканчивая квадратом верхнего целочисленного предела, после чего отображает результат на экране. Далее программа запрашивает ввод следующих предельных значений и отображает ответ, пока пользователь не введет значение верхнего предела, который меньше или равен нижнему пределу. Результаты выполнения программы могут выглядеть следующим образом:

```
Введите верхний и нижний целочисленные пределы: 5 9
Сумма квадратов целых чисел от 25 до 81 равна 255
Введите следующую комбинацию пределов: 3 25
Сумма квадратов целых чисел от 9 до 625 равна 5520
Введите следующую комбинацию пределов: 5 5
Работа завершена
```

10. Напишите программу, которая читает восемь целых чисел в массив, а затем выводит их в обратном порядке.

11. Рассмотрим две следующих бесконечных последовательности:

```
1.0 + 1.0/2.0 + 1.0/3.0 + 1.0/4.0 + ...
1.0 - 1.0/2.0 + 1.0/3.0 - 1.0/4.0 + ...
```

Напишите программу, которая подсчитывает текущие суммы этих двух последовательностей до тех пор, пока не будет обработано заданное количество элементов последовательностей. Это количество пользователь вводит в программу в интерактивном режиме. Программа отображает значения текущих сумм 20 элементов, 100 элементов, 500 элементов. Сходятся ли эти последовательности к какому-либо значению? Совет: `-1`, умноженная сама на себя нечетное число раз, дает в результате `-1`, а четное число раз — `1`.

12. Напишите программу, которая создает восьмиэлементный массив значений типа `int` и помещает в него элементы восьми первых степеней числа 2, а затем выводит полученные значения на печать. Используйте цикл `for` для вычисления элементов массива, и для разнообразия воспользуйтесь циклом `do while` для отображения значений.

13. Напишите программу, которая создает два восьмиэлементных массива значений типа `double` и использует цикл для ввода значений восьми элементов первого массива. Программа должна накапливать в элементах второго массива суммы первого массива с нарастающим итогом. Например, четвертый элемент второго массива должен быть равен сумме первых четырех элементов первого массива, а пятый элемент второго массива — сумме пяти первых элементов первого массива. (Это можно сделать с помощью вложенных циклов, однако если учесть тот факт, что пятый элемент второго массива равен четвертому элементу второго массива плюс пятый элемент первого массива, можно избежать вложенных циклов и использовать для решения задачи единственный цикл.) В завершение воспользуйтесь циклом для отображения на экране содержимого обоих массивов, при этом первый массив должен отображаться в первой строке, а каждый элемент второго массива должен помещаться непосредственно под соответствующим элементом первого массива.

14. Напишите программу, которая читает строку входных данных, а затем печатает эту строку в обратном порядке. Вы можете запоминать входные данные в массиве значений типа `char`; предполагается, что строка состоит не более чем из 255 символов. Напоминаем о том, что вы можете воспользоваться функцией `scanf()` со спецификатором `%c`, чтобы выполнять посимвольное считывание с устройства ввода, а также о том, что после каждого нажатия клавиши <Enter> генерируется символ новой строки (`\n`).
15. Дафна делает вклад в сумме $100 и ежегодной процентной ставкой (простой процент) 10%. (То есть, ежегодный прирост вклада составляет 10% от первоначальной суммы.) Дейдра вкладывает $100 с ежегодной процентной ставкой 5%, но процент сложный. (Это значит, что ежегодное увеличение вклада достигает 5% от текущего баланса, включая предыдущий прирост вклада.) Напишите программу, которая вычисляет, сколько нужно лет, чтобы сумма на счету Дейдры превзошла сумму на счету Дафны. Выведите также размеры обоих вкладов на тот момент.
16. Чаки Лаки выиграл миллион долларов, которые он кладет на счет со ставкой 8% годовых. В последний день каждого года Чаки снимает со счета по 100 тысяч долларов. Напишите программу, которая может вычислить, сколько лет пройдет до того, как на счету Чаки не останется денег.

ГЛАВА 7

Управляющие операторы: ветвление и безусловные переходы

В этой главе:

- Ключевые слова: `if`, `else`, `switch`, `continue`, `break`, `case`, `default`, `goto`
- Операции: `&& || ?:`
- Функции: `getchar()`, `putchar()`, семейство `ctype.h`
- Использование операторов `if` и `if else` и вложение их друг в друга
- Использование логических операторов для комбинирования выражений отношения в более сложные выражения проверки
- Условные операторы
- Оператор `switch`
- Операторы `break`, `continue` и `goto`
- Использование функций символьного ввода-вывода: `getchar()` и `putchar()`
- Семейство функций анализа символов, предоставляемых заголовочным файлом `ctype.h`

По мере обретения уверенности в обращении с операторами языка C, вы, скорее всего, захотите решать все более сложные задачи, и тогда вам понадобятся методы организации и управления проектами. В языке C имеются необходимые инструментальные средства. Вы уже научились пользоваться циклами для выполнения многократно повторяющихся действий. В этой главе мы будем изучать структуры ветвления, такие как `if` и `switch`, которые позволяют программе базировать свои действия условиями, которые она проверяет. Кроме того, вы получите начальные сведения о логических операциях C, которые позволяют проверять более одного отношения в условных выражениях операторов `while` или `if`, а также ознакомитесь с операторами безусловного перехода, с помощью которых можно изменять естественный ход выполнения программы. В завершение этой главы вы получите в кратких формулировках всю основную информацию, необходимую для разработки программы, поведение которой отвечает всем предъявляемым вами требованиям.

Оператор if

Начнем с того, что рассмотрим простой пример использования оператора if, показанный в листинге 7.1. Эта программа считывает список ежедневных минимальных температур (по шкале Цельсия) в специальный список и выдает общее количество записей, а также процент значений ниже точки замерзания воды (то есть ниже нуля по Цельсию). Для ввода этих значений программа использует в цикле функцию scanf(). В процессе каждой итерации она увеличивает значение счетчика на единицу, чтобы отслеживать количество записей. Оператор if распознает случаи, когда температура опускается ниже нуля по Цельсию, и отдельно подсчитывает число таких случаев.

Листинг 7.1. Программа colddays.c

```
// colddays.c -- вычисляет процент случаев, когда температура
// опускается ниже нуля
#include <stdio.h>
int main(void)
{
   const int FREEZING = 0;
   float temperature;
   int cold_days = 0;
   int all_days = 0;

   printf("Введите список дневных температур.\n");
   printf("Используйте шкалу Цельсия; для завершения введите q.\n");
   while (scanf("%f", &temperature) == 1)
   {
      all_days++;
      if (temperature < FREEZING)
         cold_days++;
   }
   if (all_days != 0)
      printf("%d - общее количество дней: %.1f%% с температурой ниже нуля.\n",
         all_days, 100.0 * (float) cold_days / all_days);
   if (all_days == 0)
      printf("Данные не введены!\n");

   return 0;
}
```

Ниже показан результат выполнения этой учебной программы:

```
Введите список дневных температур.
Используйте шкалу Цельсия, для завершения введите q.
12 5 -2.5 0 6 8 -3 -10 5 10 q
10 - общее количество дней: 30.0% дней с температурой ниже нуля.
```

Проверяемое условие цикла while использует значение, возвращаемое функцией scanf(), для прекращения выполнения цикла, когда функция scanf() сталкивается с нечисловым значением. Применяя тип float вместо int для значений температуры, программа получает возможность принимать такие показания температуры, как -2.5 и 8.

В блоке while появляется новый оператор:

```
if (temperature < FREEZING)
   cold_days++;
```

Этот оператор if дает компьютеру команду увеличить значение счетчика cold_days на 1, если только что считанное значение (температуры) меньше нуля. Что произойдет, если значение переменной temperature не меньше нуля? Тогда оператор cold_days++; пропускается, а выполнение цикла while продолжается, и далее читается следующее значение переменной temperature.

В программе еще дважды используется оператор if для управления циклом. Если имеются какие-либо данные, программа печатает результаты. Если данных нет, программа сообщает об этом. (Вскоре вы ознакомитесь с более элегантным способом реализации этой части программы.)

Чтобы избежать целочисленного деления при вычислении процентного отношения, в данном примере выполняется приведение к типу float. На самом деле приведение типов здесь не требуется, поскольку входящее в выражение 100.0 * cold_days / all_days подвыражение 100.0 * cold_days сначала вычисляется, а затем приводится к типу с плавающей запятой в соответствии с правилом автоматического приведения типов. В то же время использование операции приведения типов документирует ваши намерения и служит защитой программы от ошибок в случае неудачных версий компиляторов.

Оператор if называется *оператором ветвления* или *оператором выбора*, поскольку он представляет собой узловой пункт, по достижении которого программа оказывается перед выбором, по какому из двух возможных путей следовать дальше. Обобщенная форма оператора имеет вид:

```
if (выражение)
   оператор
```

Если в результате вычислений *выражение* принимает истинное (ненулевое) значение, *оператор* выполняется. В противном случае он пропускается. Как и в цикле while, *оператор* может быть как одиночным оператором, так и блоком операторов (составным оператором). Его структура во многом похожа на структуру оператора while. Основное различие заключается в том, что в операторе if проверка условия и (возможно) выполнение производится всего лишь один раз, в то время как в цикле while проверка условия и выполнение могут повторяться многократно.

Как правило, *выражение* является выражением отношения, то есть в нем сравниваются две количественных величины, как это имеет место в выражениях x > y и c == 6. Если *выражение* истинно (x больше y, либо c равно 6), оператор выполняется. В противном случае оператор игнорируется. В общем случае можно использовать любое выражение, при этом выражение, принимающее значение 0, дает в результате "ложь".

Операторная часть может быть простым выражением, как, например, в следующем программном коде, либо она может быть составным оператором или блоком, заключенным в фигурные скобки:

```
if (score > big)
   printf("Джекпот!\n");          // простой оператор
```

```
if (joe > ron)
{                                   // сложный оператор
   joecash++;
   printf("Ты проиграл, Ron.\n");
}
```

Обратите внимание, что вся эта структура `if` рассматривается как один оператор, даже если в ней используется составной оператор.

Добавление конструкции `else` к оператору `if`

Простейшая форма оператора `if` предоставляет вам возможность выполнить оператор (возможно, составной) или пропустить его. Язык C предлагает также возможность выбора одного из двух операторов, для чего служит форма `if else`. Воспользуемся формой `if else`, чтобы заменить громоздкие сегменты программы, показанной на листинге 7.1.

```
if (all_days != 0)
   printf("%d - общее количество дней: %.1f%% с температурой ниже нуля.\n",
         all_days, 100.0 * (float) cold_days / all_days);
if (all_days == 0)
   printf("Данные не введены!\n");
```

Если программа обнаруживает, что значение `all_days` не равно `0`, она должна знать, когда значение переменной `all_days` должно стать равным `0` без дальнейшей проверки, и она об этом знает. Располагая формой `if else`, вы можете извлечь пользу из этих сведений, переписав данный фрагмент в следующем виде:

```
if (all_days!= 0)
   printf("%d – общее количество дней: %.1f%% с температурой ниже нуля.\n",
         all_days, 100.0 * (float) cold_days / all_days);
else
   printf("Данные не введены!\n");
```

Производится только одна проверка. Если проверяемое выражение оператора `if` принимает значение `true`, данные о температуре выводятся на печать. Если оно ложно, печатается предупреждающее сообщение.

Обратите внимание на общую форму оператора `if else`:

```
if (выражение)
   оператор1
else
   оператор2
```

Если *выражение* истинно (не равно нулю), выполняется *оператор1*. Если *выражение* ложно или равно нулю, выполняется один оператор, следующий за `else`. Операторы могут быть простыми либо составными. Язык C не требует использования отступов, однако, это стало общепринятой практикой. Отступы позволяют визуально различать операторы, выполнение которые зависят от результатов проверки.

Если вы поместить между `if` и `else` более одного оператора, вы должны воспользоваться фигурными скобками, позволяющими образовать отдельный блок операторов. В приведенной ниже конструкции нарушен синтаксис языка C, поскольку компилятор ожидает обнаружить только один оператор между `if` и `else`:

```
if (x > 0)
   printf("Инкремент значения x:\n");
   x++;
else                    // приводит к возникновению ошибки
   printf("x <= 0 \n");
```

Компилятор трактует оператор `printf()` как часть оператора `if`, оператор `x++;` он рассматривает как отдельный оператор, а не как составную часть оператора `if`. Поэтому он считает, что `else` не принадлежит оператору `if`, что является ошибкой. Вместо этой конструкции оператора `if` воспользуйтесь следующей формой:

```
if (x > 0)
{
   printf("Инкремент значения x:\n");
   x++;
}
else
   printf("x <= 0 \n");
```

Оператор `if` предоставляет возможность выбора — выполнять или не выполнять какое-то одно конкретное действие. Оператор `if else` предлагает на выбор возможность выполнять одно из двух действий. На рис. 7.1 сравниваются эти два оператора.

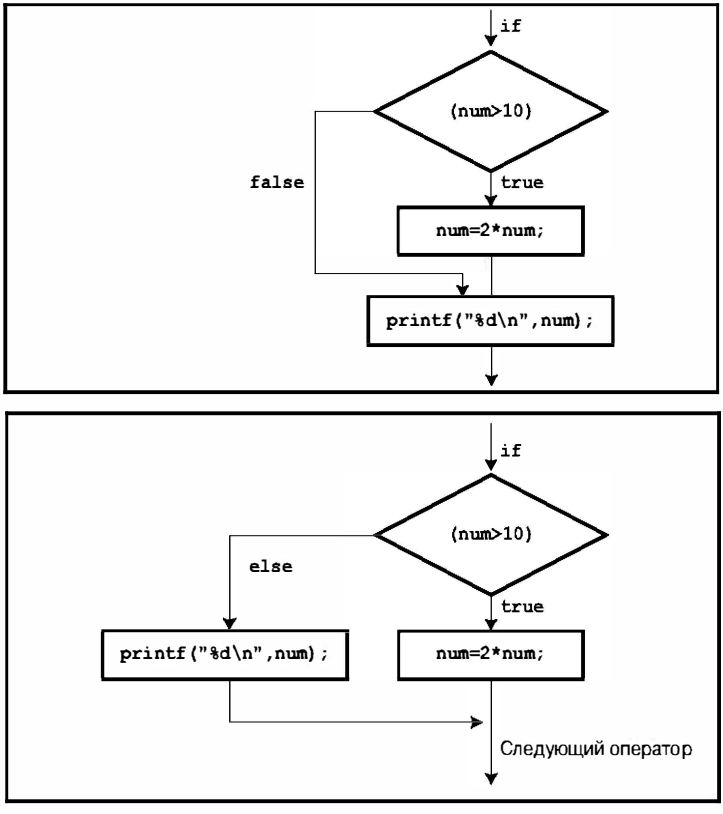


Рис. 7.1. *Операторы `if` и `if else`*

Еще один пример: знакомство с функциями getchar() и putchar()

В большинстве рассмотренных выше примеров использовались числовые входные данные. Чтобы попрактиковаться с данными других типов, рассмотрим пример, ориентированный на обработку символов. Вы уже знаете, как используются функции `scanf()` и `printf()` со спецификатором `%c`, обеспечивающие чтение и вывод символов, однако в этом примере вам придется иметь дело с двумя другими функциями языка C, специально предназначенными для ввода-вывода символов: `getchar()` и `putchar()`.

Функция `getchar()` не имеет аргументов, она возвращает очередной символ из входного потока. Например, показанный ниже оператор читает следующий входной символ и присваивает его значение переменной `ch`:

```
ch = getchar();
```

Выполнение этого оператора дает тот же результат, что и выполнение оператора

```
scanf("%c", &ch);
```

Функция `putchar()` выводит на печать свой аргумент. Например, следующий оператор выводит в виде символа значение, присвоенное переменной `ch`:

```
putchar(ch);
```

Этот оператор выдает тот же результат, что и оператор:

```
printf("%c", ch);
```

Поскольку эти функции предназначены только для работы с символами, они выполняются быстрее и характеризуются большей компактностью, чем универсальные функции `scanf()` и `printf()`. Кроме того, обратите внимание, что для них не нужны спецификаторы формата; это объясняется тем, что они предназначены для работы только с символами. Определения этих функций обычно содержатся в заголовочном файле `stdio.h`. (Следует отметить, что они скорее являются макросами препроцессора, а не функциями в полном смысле этого слова; о функционально-подобных макросах речь пойдет в главе 16.)

Теперь рассмотрим, как работают эти функции, на примере программы, которая повторяет строку входных символов, но при этом заменяет каждый отличный от пробела символ на следующий за ним символ в последовательности кодов ASCII. Пробелы в выходной последовательности сохраняются. Вы можете сформулировать требуемый выходной результат следующим образом: “Если символ является пробелом, он выводится на печать, в противном случае печатается символ, следующий за ним в последовательности кодов ASCII”.

Соответствующий программный код представлен в листинге 7.2.

Листинг 7.2. Программа `cypher1.c`

```
/* cypher1.c -- вносит изменения во входные данные, сохраняя пробелы */
#include <stdio.h>
#define SPACE ' '          /* кавычка, пробел, кавычка */
int main(void)
{
   char ch;
```

```
   ch = getchar();         /* читать символ */
   while (ch != '\n')      /* если это не символ конца строки */
   {
      if (ch == SPACE)     /* оставить пробел нетронутым */
         putchar(ch);      /* символ не меняется */
      else
         putchar(ch + 1); /* изменить другие символы */
      ch = getchar();      /* взять следующий символ */
   }
   putchar(ch);            /* печатать символ новой строки */
   return 0;
}
```

В результате выполнения этой программы получаем следующий результат:

```
CALL ME HAL.
DBMM NF IBM/
```

Сравним этот цикл с циклом из листинга 7.1. С целью определить момент прекращения выполнения цикла программа в листинге 7.1 использует бит состояния, возвращаемый функцией `scanf()`, а не значение элемента ввода. Программа из листинга 7.2, однако, использует значение самого элемента входных данных для определения, когда нужно прекратить выполнение цикла. Это обстоятельство определяет небольшое различие структур рассматриваемых циклов, когда в одном случае оператор считывания предшествует циклу, а в другом случае оператор считывания находится в конце цикла. В то же время гибкий синтаксис языка С позволяет эмулировать программу листинга 7.1 за счет совмещения операций считывания и проверки условия в одном выражении. То есть, вы можете заменить цикл вида

```
ch = getchar();         /* читать символ */
while (ch != '\n')      /* если это не конец строки */
{
   ...                  /* обработать символ */
   ch = getchar();      /* взять следующий символ */
}
```

на цикл, который выглядит следующим образом:

```
while ((ch = getchar()) != '\n')
{
                        /* обработать символ */
}
```

Интерес вызывает приведенная ниже строка:

```
while ((ch = getchar()) != '\n')
```

Она демонстрирует стиль программирования, характерный для программирования на языке С: сочетание двух действий в одном выражении. Механизм свободного форматирования языка С можно использовать для того, чтобы легче было определить назначение отдельных компонентов строки:

```
while (
   (ch = getchar())                  // присвоить значение переменной ch
                        != '\n')     // сравнить ch с \n
```

Действиями являются операция присваивания конкретного значения переменной ch и сравнение этого значения с символом новой строки. Круглые скобки, охватывающие конструкцию ch = getchar(), делают ее левым операндом операции !=. Чтобы вычислить это выражение, сначала вызывается функция getchar(), после чего возвращаемое ею значение присваивается переменной ch. Поскольку значением выражения присваивания является значение переменной в левой части операции, значение всей операции ch = getchar() есть именно новое значение переменной ch. Поэтому, после того, как значение ch считано, проверяемое условие сводится к отношению ch != '\n' (то есть, к утверждению, что значение переменной ch *неравно* символу новой строки).

Языковая конструкция подобного рода очень часто применяется в программировании на C, и вы должны быть с ней знакомы. Вы должны также твердо знать, как пользоваться круглыми скобками, чтобы правильно объединять подвыражения в группы.

Ни одну пару скобок опустить нельзя, все скобки необходимы. Предположим, что вы по ошибке использовали следующее выражение:

```
while (ch = getchar() != '\n')
```

Операция != имеет более высокий приоритет, чем =, следовательно, первым вычисляется выражение getchar() != '\n'. Поскольку это условное выражение, оно принимает значение 1 или 0. Затем это значение присваивается переменной ch. Отсутствие скобок означает, что переменной ch присвоено значение 0 или 1, а не возвращаемое значение функции getchar(); но это не то, что нам нужно.

Оператор

```
putchar(ch + 1); /* изменить другие символы */
```

служит еще одной иллюстрацией того, что символы хранятся как целые числа. В выражении ch + 1 тип переменной ch расширяется до int с тем, чтобы можно выполнить вычисления, а полученный результат передается в функцию putchar(), которая принимает аргумент типа int, но при этом использует только завершающий байт, чтобы определить, какой символ необходимо вывести на экран.

Семейство символьных функций ctype.h

Обратите внимание, что из выходных данных программы, представленной в листинге 7.2, следует, что точка преобразуется в косую черту; это объясняется тем, что в классификации ASCII код символа косой черты на единицу больше, чем код символа точки. Однако если основная цель программы состоит в том, чтобы выполнять преобразование только букв, было бы неплохо оставлять неизменными все символы, отличные от букв, а не только пробелы. Логические операции, которые обсуждаются ниже, дают нам в распоряжение средства, с помощью которых можно проверить, не является ли символ пробелом, запятой и так далее, но если перечислять все возможные варианты, то это будет довольно громоздкая процедура. К счастью, в ANSI C имеется стандартный набор функций для анализа символов; заголовочный файл ctype.h содержит соответствующие прототипы. Эти функции принимают символ в качестве аргумента и возвращают ненулевое значение (true), если символ принадлежит некоторой конкретной категории, и ноль (false) в противном случае.

Например, функция isalpha() возвращает ненулевое значение, если ее аргументом является буква. Листинг 7.3 становится обобщением листинга 7.2 благодаря использованию этой функции; она также содержит укороченную структуру цикла, который мы только что рассмотрели.

Листинг 7.3 Программа cypher2.c

```
// cypher2.c -- меняет символы входных данных, оставляя неизменными
символы,
// не являющиеся буквами
#include <stdio.h>
#include <ctype.h>          // для функции isalpha()
int main(void)
{
   char ch;

   while ((ch = getchar()) != '\n')
   {
      if (isalpha(ch))      // если это буква,
         putchar(ch + 1);   // изменить ее
      else                  // в противном случае
         putchar(ch);       // вывести символ таким, каким он есть
   }
   putchar(ch);             // печатать символ новой строки

   return 0;
}
```

Ниже показаны результаты выполнения программы; обратите внимание, что строчные и прописные буквы изменяются, а пробелы и знаки препинания — нет:

```
Look! It's a programmer!
Mppl! Ju't b qsphsbnnfs!
```

В таблицах 7.1 и 7.2 представлен перечень нескольких функций, использование которых становится возможным в результате включения в программу заголовочного файла ctype.h. В некоторых записях упоминается локализация, под которой понимаются средства языка C, позволяющие учитывать региональные настройки и модифицирующие или расширяющие базовое использование C. (Например, во многих странах используется запятая вместо десятичной точки при записи дробных частей, конкретные местные условия могут указать, что язык C использует запятую в выходных данных с плавающей запятой, отображая, скажем, 123.45 как 123,45.) Обратите внимание, что функции отображения не изменяют исходный аргумент, а вместо этого возвращают модифицированное значение. То есть,

```
tolower(ch);          // не вызывает изменений ch
```

не меняет значение переменной ch.

Чтобы изменить ch, выполните следующее:

```
ch = tolower(ch);     // преобразовать ch к нижнему регистру
```

Таблица 7.1. Функции проверки символьных значений из заголовочного файла `ctype.h`

Имя функции	*Истинно, если аргумент*
`isalnum()`	Алфавитно-цифровой (буквенный или цифровой).
`isalpha()`	Алфавитный.
`isblank()`	Стандартный пробельный символ (пробел, горизонтальная табуляция или символ новой строки) или любой дополнительный местный символ, описанный подобным способом.
`iscntrl()`	Управляющий символ, такой как <Ctrl+B>.
`isdigit()`	Цифра.
`isgraph()`	Любой печатный символ, отличный от пробела.
`islower()`	Символ нижнего регистра.
`isprint()`	Печатный символ.
`ispunct()`	Знак пунктуации (любой печатный символ, отличный от пробела и алфавитно-цифрового символа).
`isspace()`	Пробельный символ (пробел, символы новой строки, перевода страницы, возврата каретки, вертикальной табуляции, горизонтальной табуляции и, возможно, другие локально определенные символы).
`isupper()`	Символ верхнего регистра.
`isxdigit()`	Символ шестнадцатеричной цифры.

Таблица 7.2. Функции отображения символов из файла `ctype.h`

Имя функции	*Действие*
`tolower()`	Если аргумент является символом верхнего регистра, эта функция возвращает его версию для нижнего регистра; иначе возвращает исходное значение аргумента.
`toupper()`	Если аргумент является символ нижнего регистра, эта функция возвращает его версию для верхнего регистра; иначе возвращает исходное значение аргумента.

Множественный выбор `else if`

Жизнь часто предоставляет вам выбор из более чем двух вариантов. Можно расширить структуру `if else` путем включения конструкции `else if`, чтобы быть готовым к ситуациям подобного рода. Рассмотрим конкретный пример. Компании, предоставляющие коммунальные услуги, часто выставляют тарифы за использование электроэнергии, которые зависят от количества электроэнергии, потребленного клиентом. Ниже приводятся данные о тарифах на оплату потребленной электроэнергии в киловатт-часах (кВт/ч) одной из таких компаний:

```
Первые 360 кВт/ч:       $0.11439 за 1 кВт/ч
Следующие 320 кВт/ч:    $0.13290 за 1 кВт/ч
Свыше 680 кВт/ч:        $0.14022 за 1 кВт/ч
```

Если вы намерены вести учет расхода электроэнергии, вам, возможно, понадобится программа, подсчитывающая стоимость потребленной электроэнергии. Программа, представленная в листинге 7.4, может стать первым шагом в этом направлении.

Листинг 7.4. Программа `electric.c`

```
/* electric.c -- подсчитывает сумму для счета за электроэнергию */
#include <stdio.h>
#define RATE1 0.12589          /* тариф за первые 360 кВт/ч */
#define RATE2 0.17901          /* тариф за следующие 320 кВт/ч */
#define RATE3 0.20971          /* тариф, когда расход превышает 680 кВт/ч */
#define BREAK1 360.0           /* первая точка разрыва тарифов */
#define BREAK2 680.0           /* вторая точка разрыва тарифов */
#define BASE1 (RATE1 * BREAK1) /* стоимость 360 кВт/ч */
#define BASE2 (BASE1 + (RATE2 * (BREAK2 - BREAK1)))
                               /* стоимость 680 кВт/ч */
int main(void)
{
   double kwh;                 /* израсходованные киловатт-часы */
   double bill;                /* сумма к оплате */

   printf("Введите количество израсходованной электроэнергии в кВт/ч.\n");
   scanf("%lf", &kwh);         /* %lf для типа double */
   if (kwh <= BREAK1)
      bill = RATE1 * kwh;
   else if (kwh <= BREAK2) /* количество кВт/ч в промежутке от 360 до 680 */
      bill = BASE1 + (RATE2 * (kwh - BREAK1));
   else                    /* количество кВт/ч превышает 680 */
      bill = BASE2 + (RATE3 * (kwh - BREAK2));
   printf("Сумма к оплате за %.1f кВт/ч составляет $%1.2f.\n", kwh, bill);

   return 0;
}
```

Ниже показаны выходные данные этой программы:

```
Введите количество израсходованной электроэнергии в кВт/ч.
580
Сумма к оплате за 580.0 кВт/ч составляет $84.70.
```

В программе из листинга 7.4 для представления тарифов применяются символьные константы, при этом для удобства они собраны в одном месте. Если электрическая компания меняет свои тарифы (что вполне возможно), то поскольку они определены в одном месте, задача их обновления существенно облегчается. В этом листинге также используются символьные обозначения значений расхода, в которых тарифы меняют свои значения (так называемые точки разрыва). Их тоже время от времени меняют. Константы `BASE1` и `BASE2` выражены через тарифы и точки разрыва. Далее, если тарифы и точки разрыва претерпевают изменения, значения `BASE1` и `BASE2` обновляются автоматически. Здесь полезно напомнить, что препроцессор не выполняет вычислений. Там, где программе появляется константа `BASE1`, она заменяется произведением `0.12589 * 360.0`. Можете не беспокоиться, компилятор вычислит числовое значение этого выражения (`45.3204`), так что в окончательной редакции программы используется уже число `45.3204`, а соответствующие вычисления не выполняются.

Ход выполнения этой программы достаточно прост. Она выбирает одну из трех формул в зависимости от количества кВт/ч израсходованной электроэнергии. Поток управления этой программы показан на рис. 7.2. Особое внимание следует обратить на то, что программа может выйти на первый `else`, если значение переменной `kwh` равно или больше `360`. По этой причине строка `else if (kwh <= BREAK2)` фактически эквивалентна требованию, чтобы значение `kwh` находилось в пределах от `360` до `680`, как указано в комментарии к программе. Аналогично, завершающий `else` может быть достигнут, только когда значение `kwh` превысит `680`. И, наконец, обратите внимание на то, что константы `BASE1` и `BASE2` представляют собой общую стоимость, соответственно, первых 360 и 680 киловатт-часов. Поэтому требуется только прибавить дополнительную плату за количество потребленной энергии, превышающее эти величины.

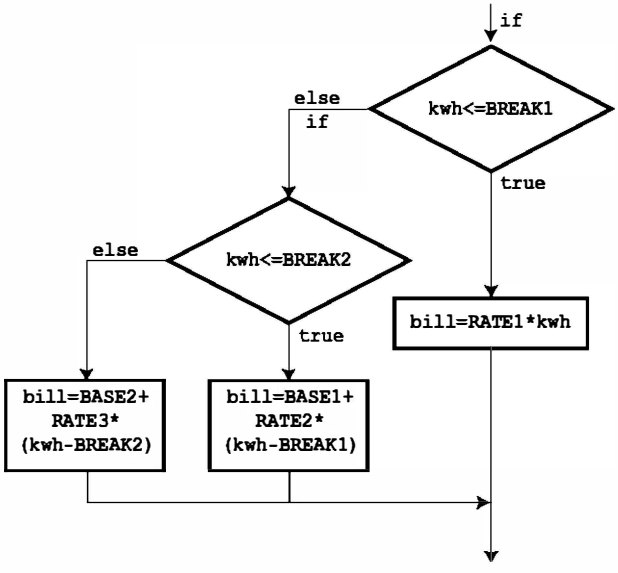


Рис. 7.2. *Поток управления программы* `electric.c` *из листинга 7.4.*

Фактически конструкция `else if` является видоизмененным способом задания условного оператора, с которым вы ознакомились раньше. Например, ядро рассматриваемой программы представляет собой другую форму записи следующей последовательности операторов:

```
if (kwh <=BREAK1)
   bill = RATE1 * kwh;
else
   if (kwh <=BREAK2)
      bill = BASE1 + RATE2 * (kwh - BREAK1);
   else
      bill = BASE2 + RATE3 * (kwh - BREAK2);
```

То есть программа состоит из оператора `if else`, часть `else` которого представляет собой другой оператор `if else`. Про второй оператор `if else` говорят, что он *вложен* в первый. Напомним, что вся структура `if else` считается одним оператором.

Вот почему не обязательно заключать вложенную конструкцию `if else` в фигурные скобки. В то же время использование скобок проясняет назначение этого конкретного формата.

Эти две формы практически эквивалентны. Единственное различие между ними заключается в использовании пробелов и символов новой строки, в то же время эти различия компилятором игнорируются. Тем не менее, первая форма предпочтительнее, поскольку из нее сразу видно, что необходимо делать выбор одной из трех возможностей. Кроме того, она облегчает просмотр программы и понимание ее семантики. Сохраняйте все отступы для вложенных форм, так как они могут понадобиться, например, когда у вас возникнет необходимость проверки двух разных величин. Примером могла бы служить 10%-ная дополнительная плата за потребление энергии свыше 680 киловатт-часов в летние месяцы.

В одном операторе можно использовать столько операторов `else if`, сколько вам нужно (разумеется, в пределах возможностей компилятора), как показывает следующий фрагмент:

```
if (score < 1000)
   bonus = 0;
else if (score < 1500)
   bonus = 1;
else if (score < 2000)
   bonus = 2;
else if (score < 2500)
   bonus = 4;
else
   bonus = 6;
```

(Этот фрагмент может быть взят из игровой программы, в которой переменная `bonus` представляет собой количество дополнительных "фотонных бомб" или "питательных таблеток", которые получает игрок, чтобы продолжить игру в следующем раунде.)

Что касается возможностей компилятора, то стандарт C99 требует, чтобы компилятор поддерживал не менее 127 уровней вложения.

Объединение `else` и `if` в пары

Когда в программе присутствует множество операторов `if` и `else`, как удается компилятору разобраться, какой `if` какому `else` соответствует? В качестве примера рассмотрим следующий фрагмент программы:

```
if (number > 6)
   if (number < 12)
      printf("Вы закончили игру!\n");
else
   printf("К сожалению, вы потеряли право хода!\n");
```

В каком случае фраза `К сожалению, вы потеряли право хода!` будет напечатана? Когда значение переменной `number` меньше или равно `6`, или когда значение `number` больше `12`? Другими словами, чему соответствует `else`, первому `if` или второму?

Правильный ответ такой: `else` относится ко второму `if`. То есть, вы получите следующие ответы:

Число	*Результат*
5	Нет ответа
10	Вы закончили игру!
15	К сожалению, вы потеряли право хода!

Согласно существующему правилу, `else` соответствует ближайшему `if`, кроме тех случаев, когда с помощью фигурных скобок задается другой порядок (рис. 7.3).

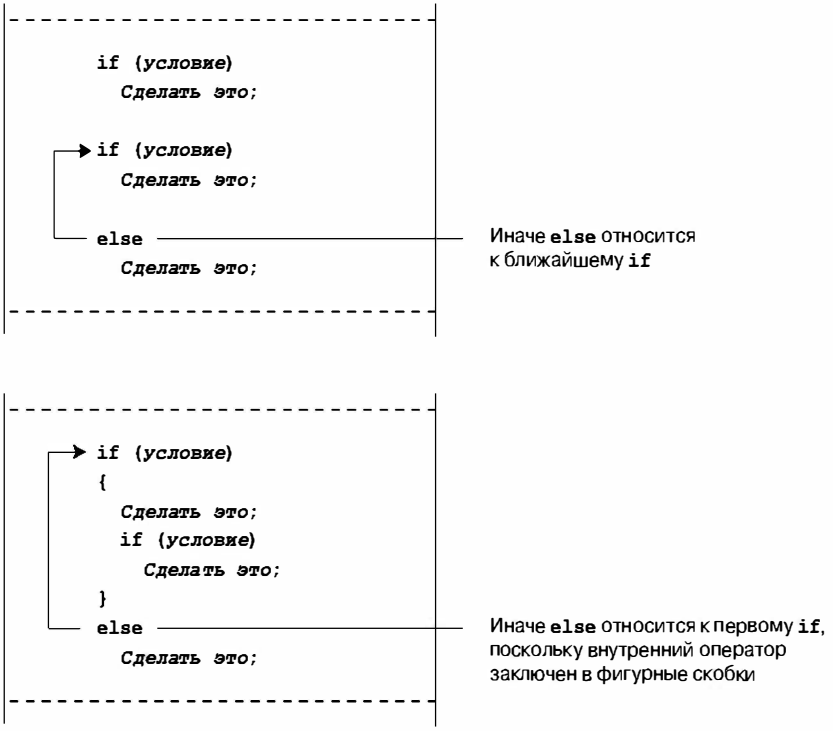


Рис. 7.3. *Правило объединения `if` и `else` в пары*

Мы сознательно расставили отступы таким образом, как будто `else` соответствует первому `if`, при этом еще раз обращаем ваше внимание на то, что компилятор игнорирует отступы. Если вы на самом деле хотите, чтобы `else` относился к первому `if`, вы могли бы записать этот фрагмент следующим образом:

```
if (number > 6)
{
   if (number < 12)
      printf("Вы закончили игру!\n");
}
else
   printf("К сожалению, вы потеряли право хода!\n");
```

Теперь вы получите следующие ответы:

Число	*Результат*
5	К сожалению, вы потеряли право хода!
10	Вы закончили игру!
15	Нет ответа

Большее число вложений операторов `if`

Вы уже имели возможность убедиться в том, что последовательность `if...else if...else` является одной из форм вложенного оператора `if`, который производит выбор из некоторого набора вариантов. Другой вид вложенного оператора `if` используется в тех случаях, когда сделанный конкретный выбор приводит к дополнительному варианту. Например, программа может использовать оператор `if else` для выбора между мужчинами и женщинами. Каждая ветвь в рамках оператора `if else` может содержать другой оператор `if else`, чтобы, скажем, провести различие между группами лиц с разным годовым доходом.

Применим эту форму вложенного оператора `if` для решения следующей задачи: для заданного целого числа распечатать все целые числа, на которые заданное число делится без остатка, а если таких делителей нет, вывести сообщение о том, что заданное число является простым.

Такая задача требует решения некоторых принципиальных вопросов, прежде чем приступать к написанию собственно кода. Прежде всего, требуется разработать общую схему программы. Для удобства программа должна использовать цикл для ввода числа, подлежащего исследованию на предмет его делителей. Благодаря циклу нет необходимости запускать программу всякий раз, когда вы хотите проверить новое число. Ниже представлена модель такого цикла:

```
приглашение пользователю на ввод числа
пока функция scanf() возвращает значение 1
   выполнить анализ числа и сообщить результаты
   приглашение пользователю на ввод числа
```

Вспомните, что с помощью функции `scanf()` в условии проверки цикла программа пытается прочитать число и выполнить проверку с тем, чтобы определить, необходимо ли прекратить работу цикла.

Далее потребуется выработать план определения делителей. Возможно, наиболее очевидным подходом является нечто в этом роде:

```
for (div = 2; div < num; div++)
   if (num % div == 0)
      printf("%d делится на %d\n", num, div);
```

В этом цикле проверяются все числа в промежутке между 2 и `num` для нахождения тех из них, которые делят значение `num` без остатка. К сожалению, такой подход требует больших затрат машинного времени. Можно найти гораздо более экономное решение. Рассмотрим, например, процедуру поиска делителей числа 144. Выясняется, что 144 % 2 равно 0, что означает, что 144 делится на 2 без остатка. Если вы теперь выполните обычную операцию деления 144 на 2, вы получите 72, это число также является делителем числа 144, следовательно, в случае успешной проверки `num % div`, вы получите сразу два делителя вместо одного.

Однако главное достоинство такого подхода состоит в том, что изменяются пределы при проверке условия конца цикла. Чтобы понять, как работает этот подход, исследуйте пары делителей, полученных в процессе выполнения цикла: 2,72, 3,48, 4,36, 6,24, 8,18, 9,16, 12,12, 16,9, 18,8 и так далее.

Так вот в чем дело! После пары 12,12 в последовательности встречаются те же делители (в обратном порядке), которые уже были найдены. Вместо того чтобы продолжать цикл до 143, вы можете завершить его сразу после того, как достигли 12. Это делает ненужным выполнение большого количества итераций!

Обобщая это открытие, вы убеждаетесь, что для проверки, достиг ли цикл значения, равного квадратному корню из num, а не значения num. Для таких чисел, как 9, выигрыш не слишком велик, но для чисел порядка 10000 и более он огромен. При этом вместо того, чтобы извлекать квадратные корни, вы можете сформулировать проверяемое условие следующим образом:

```
for (div = 2; (div * div) <= num; div++)
   if (num % div == 0)
      printf("%d делится на %d и %d.\n",
                  num, div, num / div);
```

Если переменная num принимает значение 144, цикл выполняется вплоть до div = 12. Если num принимает значение 145, цикл выполняется вплоть до div = 13.

Такая проверка предпочтительнее проверки с вычислением квадратных корней, по меньшей мере, по двум причинам. Во-первых, компьютер выполняет умножение целых чисел гораздо быстрее, чем извлечение квадратного корня. Во-вторых, формально мы еще не знакомы с функцией вычисления квадратного корня.

Нам надо найти подход к решению еще двух проблем, и только после этого мы будем готовы приступить к написанию кода. Первая из них: что делать, если проверяемое число представляет собой точный квадрат? Сообщение о том, что число 144 делится на два числа 12 и 12 несколько искусственно, однако можно предусмотреть вложенный оператор if, в котором проверить, равно ли div значению num / div. Если это так, программа должна распечатать один делитель вместо двух.

```
for (div = 2; (div * div) <= num; div++)
{
   if (num % div == 0)
   {
      if (div * div != num)
         printf("%d делится на %d и %d.\n",
            num, div, num / div);
      else
         printf("%d делится на %d.\n", num, div);
   }
}
```

На заметку!

С технической точки зрения оператор if else рассматривается как отдельный оператор, следовательно, заключать его в фигурные скобки не требуется. Внешний оператор if также представляет собой отдельный оператор, поэтому скобки для него не нужны. Тем не менее, когда операторы становятся слишком длинными, скобки помогают понять, что происходит, они также служат защитой, если вы в дальнейшем добавите еще один оператор в if или в цикл.

С другой стороны, как вы можете установить, что число простое? Если значение переменной num является простым, то поток управления программы никогда не попадет внутрь оператора if. Чтобы решить эту проблему, можно присвоить некоторой переменной конкретное значение, скажем, 1, за пределами цикла и присвоить ей значение 0 внутри оператора if. Затем, после того как цикл будет завершен, можно проверить, сохранила ли эта переменная значение 1. Если сохранила, то управление ни разу не передавалось этому оператору if, и в силу этого обстоятельства число является простым. Часто такая переменная называется *флагом*.

Обычно в языке C для флагов используется тип int, в то же время новый тип _Bool наилучшим образом подходит для этой цели. Более того, включив в программу заголовочный файл stdbool.h, вы можете воспользоваться ключевым словом bool, вместо _Bool для обозначения этого типа и использовать идентификаторы true и false вместо, соответственно, 1 и 0. В программе, представленной в листинге 7.5, реализованы все упомянутые идеи. Чтобы расширить область применения, программа использует тип long вместо int. (Если ваша система поддерживает тип _Bool, вы можете выбрать тип int для переменной isPrime и применять 1 и 0 вместо true и false.)

Листинг 7.5. Программа `divisors.c`

```
// divisors.c -- вложенные операторы if отображают на экране делители
// заданного числа
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
   unsigned long num;          // анализируемое число
   unsigned long div;          // возможные делители
   bool isPrime;               // флаг простого числа
   printf("Введите целое число для анализа; ");
   printf("для завершения введите q.\n");
   while (scanf("%lu", &num) == 1)
   {
      for (div = 2, isPrime= true; (div * div) <= num; div++)
      {
         if (num % div == 0)
         {
            if ((div * div) != num)
            printf("%lu делится на %lu и %lu.\n",
               num, div, num / div);
            else
               printf("%lu делится на %lu.\n",
                  num, div);
            isPrime= false;   // число не является простым
         }
      }
      if (isPrime)
         printf("%lu является простым числом.\n", num);
      printf("Введите следующее число для анализа; ");
      printf("для завершения введите q.\n");
   }
   printf("Всего хорошего.\n");
   return 0;
}
```

Обратите внимание на то, что программа использует операцию запятой в управляющем выражении цикла for, чтобы предоставить вам возможность инициализации переменной isPrime значением true при каждом вводе нового числа.

Ниже показан пример выполнения этой программы:

```
Введите целое число для анализа; для завершения введите q.
36
36 делится на 2 и 18.
36 делится на 3 и 12.
36 делится на 4 и 9.
36 делится на 6.
Введите следующее число для анализа; для завершения введите q.
149
149 является простым числом.
Введите следующее число для анализа; для завершения введите q.
30077
30077 делится на 19 и 1583.
Введите следующее число для анализа; для завершения введите q.
q
```

Данная программа рассматривает 1 как простое число, которое в техническом аспекте таковым не является. Логические операции, обсуждаемые в следующем разделе, позволят вам исключить 1 из списка простых чисел.

Сводка: использование операторов if для принятия решений

Ключевые слова:

if, else

Комментарии общего характера:

В каждой из приводимых ниже форм *оператором* может быть либо простой, либо составной оператор. Истинным является выражение, принимающее ненулевое значение.

Форма 1:

```
if (выражение)
   оператор
```

оператор выполняется, когда *выражение* принимает истинное значение.

Форма 2:

```
if (выражение)
   оператор1
else
   оператор2
```

Если *выражение* истинно, выполняется *оператор1*, в противном случае — *оператор2*.

Форма 3:

```
if (выражение1)
   оператор1
else if (выражение2)
   оператор2
 else
   оператор3
```

Если *выражение1* истинно, выполняется *оператор1*. Если *выражение1* ложно, но *выражение2* истинно, выполняется *оператор2*. Если оба выражения ложны, выполняется *оператор3*.

Пример:

```
if (legs == 4)
   printf("По всей видимости, это лошадь.\n");
else if (legs > 4)
   printf("Это не лошадь.\n");
else               /* случай, когда ног меньше 4 */
{
   legs++;
   printf("Теперь у нее на одну ногу больше.\n");
}
```

Давайте будем логичными

Как вы уже имели возможность убедиться, в операторах `if` и `while` в качестве условий проверки часто используются условные выражения. Время от времени вы находите полезным применять сочетание двух или большего количества выражений. Например, предположим, что вы хотите иметь программу, которая подсчитывает количество символов, отличных от одиночных и двойных кавычек, во входном предложении. Для этой цели вы можете использовать логические операции и символ точки (`.`) для обозначения конца предложения. В листинге 7.6 показана короткая программа, решающая эту задачу.

Листинг 7.6. Программа `chcount.c`

```
// chcount.c -- использование логического оператора AND
#include <stdio.h>
#define PERIOD '.'
int main(void)
{
   int ch;
   int charcount = 0;
   while ((ch = getchar()) != PERIOD)
   {
      if (ch != '"' && ch != '\'')
         charcount++;
   }
   printf("В данном предложении содержатся %d символов, отличных от
кавычек.\n", charcount);
   return 0;
}
```

Так выглядит пример выполнения этой программы:

```
Я не читал бестселлер "Я ничего не смыслю в программировании".
В данном предложении содержатся 60 символов, отличных от кавычек.
```

Выполнение программы начинается со считывания символа и проверки, не является ли этот символ точкой, поскольку точка обозначает конец предложения. Далее в программе появляется нечто новое — логическая операция "И", представляемая с помощью `&&`. Вы можете истолковать оператор `if` следующим образом: если символ не является двойной кавычкой И не является одиночной кавычкой, увеличить значение переменной `charcount` на 1.

Чтобы выражение было истинным, необходимо, чтобы истинными были оба условия. Логические операции имеют более низкий приоритет, чем условные операции, так что нет необходимости в использовании дополнительных круглых скобок для формирования подвыражений.

В языке C имеются три логических операции:

Операция	*Значение*
`&&`	И
`||`	Или
`!`	Не

Предположим, что `exp1` и `exp2` — два простых условных выражения, такие как, например, `cat > rat` и `debt == 1000`. Тогда вы можете утверждать:

- `exp1 && exp2` истинно тогда и только тогда, когда оба выражения `exp1` и `exp2` истинны.
- `exp1 || exp2` истинно, когда одно из выражений `exp1` и `exp2` истинно либо оба выражения истинны.
- `!exp1` истинно, если `exp1` ложно, и ложно, если `exp1` истинно.

Рассмотрим несколько конкретных примеров:

`5 > 2 && 4 > 7` ложно, поскольку истинно только одно из подвыражений.

`5 > 2 || 4 > 7` истинно, ибо, по меньшей мере, одно из подвыражений истинно.

`!(4 > 7)` истинно, поскольку 4 не больше 7.

Последнее выражение фактически эквивалентно следующему выражению:

```
4 <= 7
```

Если вы не знакомы с логическими операциями или недостаточно преуспели в их применении, не забывайте простой истины:

```
(практика && время) == совершенство
```

Альтернативное представление: заголовочный файл `iso646.h`

Язык C разрабатывался в США на системах, использующих клавиатуры, построенные по стандартам, действующим в США. Однако в других странах мира не во всех клавиатурах присутствуют символы, принятые в США. В силу этого обстоятельства стандарт C99 вводит альтернативные формы написания логических операций. Они определяются в заголовочном файле `iso646.h`. Если вы включили этот файл в свою программу, то можете использовать `and` вместо `&&`, `or` вместо `||` и `not` вместо `!`.

Например, фрагмент

```
if (ch != '"' && ch != '\'')
   charcount++;
```

можно переписать следующим образом:

```
if (ch != '"' and ch != '\'')
   charcount++;
```

Вы можете сделать свой выбор с помощью таблицы 7.3; представления нетрудно запомнить. По сути дела у вас может возникнуть вопрос, почему в языке C не используется новая терминология. Ответ, скорее всего, заключается в том, что в историческом плане язык C всегда пытался обходиться минимальным количеством ключевых слов. В справочном разделе VII (приложение Б) приводится список альтернативных форм записи некоторых операций, с которыми вам еще не приходилось сталкиваться.

Таблица 7.3. Альтернативное представление логических операций

Традиционная форма	*Предлагаемая заголовочным файлом* `iso646.h`
`&&`	`and`
`||`	`or`
`!`	`Not`

Приоритеты операций

Операция ! имеет очень высокий уровень приоритета — выше, чем операция умножения, — который равен приоритету операции инкремента; выше его только приоритет круглых скобок. Операция && имеет более высокий приоритет, чем ||, и обе они по приоритету уступают условным операциям, но превосходят операцию присваивания. В связи с этим выражение

```
a > b && b > c || b > d
```

интерпретируется следующим образом:

```
((a > b) && (b > c)) || (b > d)
```

Иначе говоря, значение b находится между a и c или b больше d.

Многие программисты используют, как и во второй версии, скобки даже там, где в них нет необходимости. В этом случае смысл ясен, даже если читатель не очень хорошо помнит приоритеты конкретных логических операций.

Порядок вычисления выражений

Помимо случаев, когда две операции совместно используют один и тот же операнд, язык C в общем случае не устанавливает, какие части сложного выражения вычисляются первыми. Например, в приведенном ниже выражении подвыражение 5 + 3 может быть вычислено раньше, чем будет вычислено подвыражение 9 + 6, но может быть вычислено и после:

```
apples = (5 + 3) * (9 + 6);
```

Такая неоднозначность оставлена в C с таким расчетом, чтобы разработчики компилятора могли сделать наиболее оптимальный выбор для конкретной системы. Единственным исключением из этого правила является порядок выполнения логических операций (или отсутствие такого правила). C устанавливает порядок вычисления логических выражений слева направо. Операции && и || — это точки последовательности, так что все побочные эффекты проявятся до того, как программа перейдет от одного операнда к другому. Более того, компилятор действует таким образом, что как только будет найден элемент, из-за которого все выражение становится ложным, вычисления прекращаются. Эти свойства компилятора позволяют использовать конструкции, подобные следующим:

```
while ((c = getchar()) != ' ' && c != '\n')
```

Эта конструкция образует цикл, который считывает символы до первого пробела или до символа новой строки. Первое подвыражение присваивает значение переменной c, которая затем используется во втором подвыражении. Если порядок вычислений не установлен, компьютер может сделать попытку вычислить значение второго подвыражения, прежде чем выяснит, какое значение имеет переменная c.

Рассмотрим еще один пример:

```
if (number != 0 && 12/number == 2)
   printf("Значение переменной number равно 5 или 6.\n");
```

Если переменная number имеет значение 0, то первое подвыражение ложно, и вычисление условного выражения дальше не продолжается. Это позволяет компьютеру избежать ошибки деления на ноль. Убедившись, что значением переменной number является 0, он переходит к вычислению следующего условия.

И, наконец, рассмотрим такой пример:

```
while ( x++ < 10 && x + y < 20)
```

Тот факт, что операция && представляет собой точку последовательности, служит гарантией того, что значение x будет увеличено на 1, прежде чем будет вычислено выражение справа.

Сводка: логические операции и выражения

Логические операции:

В логических операциях в качестве операндов выступают условные выражения. Операция ! выполняется над одним операндом. Остальные логические операции выполняются над двумя операндами, один из них расположен слева от знака операции, другой — справа.

Операция	*Значение*
&&	И
||	ИЛИ
!	НЕ

Логические выражения:

Логическое произведение *выражение1* && *выражение2* истинно тогда и только тогда, когда оба выражение истинны; *выражение1* || *выражение2* истинно, если одно или оба выражения истинны. !*выражение* истинно, если выражение ложно, и наоборот.

Порядок вычисления:

Логические выражения вычисляются слева направо. Вычисление прекращается, как только обнаруживается подвыражение, которое делает ложным все выражение.

Примеры:

`6 > 2 && 3 == 3`	Истинно
`! (6 > 2 && 3 == 3)`	Ложно
`x != 0 && (20 / x) < 5`	Второе выражение вычисляется только при условии, что `x` имеет ненулевое значение

Диапазон значений

Вы можете воспользоваться операцией `&&` для проверки на вхождение в диапазон значений. Например, чтобы проверить, находится ли значение переменной `score` в диапазоне от `90` до `100`, вы можете предусмотреть следующий оператор:

```
if (range >= 90 && range <= 100)
   printf("Неплохой результат!\n");
```

Следует раз и навсегда отказаться от стереотипов математических обозначений, как показано в следующем примере:

```
if (90 <= range <= 100)   // Не поступайте так!
   printf("Неплохой результат!\n");
```

Проблема заключается в том, что в этот программный код вкралась семантическая, а не синтаксическая ошибка, поэтому компилятор не способен ее обнаружить (хотя он может выдать какое-то предупреждающее сообщение). Поскольку для выполнения операции `<=` принят порядок слева направо, проверяемое выражение интерпретируется следующим образом:

```
(90 <= range) <= 100
```

Подвыражение `90 <= range` получает либо значение `1` (истина), либо `0` (ложь). И то и другое значение меньше `100`, поэтому все выражение всегда истинно, независимо от значения `range`. Следовательно, для проверки на вхождение в диапазон следует пользоваться операцией `&&`.

Во многих программах проверка на вхождение в диапазон используется с целью определить, скажем, представлен ли конкретный символ в нижнем регистре. Например, предположим, что переменная `ch` имеет тип `char`:

```
if (ch >= 'a' && ch <= 'z')
   printf("Это символ нижнего регистра.\n");
```

Представленный метод подходит для таких символьных кодов, как ASCII, в этом случае последовательность кодов представляет собой последовательность возрастающих целых чисел. Однако, этот метод неприменим к некоторым другим видам кодировки символов, в том числе к EBCDIC. Менее зависимым от выбора вида кодировки является использование функции `islower()` из заголовочного файла `ctype.h` (см. табл. 7.1):

```
if (islower(ch))
   printf("Это символ нижнего регистра.\n");
```

Функция islower() работает независимо от используемой кодировки символов. (Однако в некоторых устарелых реализациях семейство функций ctype.h отсутствует.)

Программа подсчета слов

Теперь у нас есть все необходимые инструментальные средства для программы подсчета слов (то есть программы, которая читает входной текст и сообщает количество содержащихся в нем слов). Вы можете с тем же успехом подсчитать число символов и число строк. Сначала определимся с тем, что должно быть включено в эту программу.

Прежде всего, эта программа должна выполнять посимвольный ввод, при этом тем или иным образом знать, когда следует остановиться. Во-вторых, она должна быть способна распознавать такие элементы, как символы, строки и слова. Представление этой программы в псевдокоде имеет вид:

```
читать символ
пока ввод не закончен,
   инкрементировать счетчик символов
   если считана строка, инкрементировать счетчик строк
   если считано слово, инкрементировать счетчик слов
   читать следующий символ
```

Вот как выглядит модель цикла ввода символов:

```
while ((ch = getchar()) != STOP)
{

}
```

В данном случае STOP представляет собой некоторое значение переменной ch, сигнализирующее о конце ввода. В рассмотренных ранее примерах мы использовали для этой цели символ новой строки и точку, однако ни тот ни другой символ не подходит для универсальной программы подсчета слов. На текущий момент выберем символ (такой как, например, |), который не встречаются в тексте. В главе 8 будет показано более удачное решение, которое позволит использовать программу как для работы с текстовыми файлами, так и с данными, которые вводятся с клавиатуры.

Теперь рассмотрим тело цикла. Поскольку для ввода программа использует функцию getchar(), она может вести подсчет символов, инкрементируя счетчик при каждом проходе цикла. Чтобы определить количество строк, программа может подсчитать количество символов в новой строке. Если программа сталкивается с символом новой строки, она должна увеличить значение счетчика строк на 1. Потребуется дать ответ еще на один вопрос: что делать, если символ STOP встретился в середине строки. Нужно ли считать это строкой или нет? Один из ответов гласит: это нужно рассматривать как частичную строку, то есть строку, в которой содержатся различные символы, но нет символа конца строки. В этом случае вы можете сохранять значение предыдущего считанного символа. Если считанный символ, предшествующий символу STOP, не является символом новой строки, то вы имеете дело с неполной строкой.

Наиболее сложной является та часть программы, которая распознает слова. Во-первых, необходимо дать определение, что понимается под словом. Выберем сравнительно простой подход и определим слово как последовательность символов, которая

не содержит символов форматирования текста (пробелов, символов табуляции и символов новой строки). В этом смысле “glymxck” и “r2d2” также являются словами. Слово начинается в том случае, когда программа в первый раз встречает символ, отличный от пробельных символов, и кончается, когда появляется следующий пробельный символ. Ниже показан пример простейшего тестового выражения, обеспечивающего обнаружения символов, отличных от пробельных:

```
c != ' ' && c != '\n' && c != '\t'
/* истинно, если c не является пробельным символом */
```

Самое простое проверяемое выражение, обеспечивающее выявление пробельных символов, имеет следующий вид:

```
c == ' ' || c == '\n' || c == '\t'
/* истинно, если c является пробельным символом */
```

Однако проще воспользоваться функцией `isspace()` из заголовочного файла `ctype.h`, которая возвращает значение `true`, если аргумент представляет собой пробельный символ. Следовательно, функция `isspace(c)` возвращает значение `true`, если `c` — пробельный символ, и `!isspace(c)` будет истинным, если `c` таковым не является.

Чтобы определить, входит ли тот или иной символ в конкретное слово, вы должны установить флаг (назовем его `inword`) в `1`, когда считывается первый символ слова. В этой точке вы можете также инкрементировать значение счетчика слов. Далее, пока значение флага `inword` равно `1` (или `true`), последующие пробельные символы не означают начало следующего слова. При появлении следующего символа, не являющегося пробельным, вы должны сбросить флаг в `0` (или `false`), а программа будет готова к выявлению следующего слова. Все сказанное выше можно представить в виде псевдокода:

```
если c не есть пробел и inword есть false,
   установить флаг inword равным true и подсчитать слово
if c есть пробел и  inword есть true
   установить флаг inword равным false
```

При таком подходе флаг `inword` устанавливается в `1` (`true`) в начале каждого слова и в `0` (`false`) в конце каждого слова. Подсчет слов производится только в те моменты, когда установка влага меняется с `0` на `1`. Если в вашей системе реализован тип `_Bool`, вы можете включить заголовочный файл `stdbool.h` и использовать ключевое слово `bool` для обозначения типа флага `inword`, принимающего значения `true` и `false`. В противном случае используйте тип `int`, принимающий значения `1` и `0`.

Если вы работаете с булевской переменной, проще всего использовать значение самой этой переменной в качестве проверяемого условия. В этом случае употребляйте

```
if (inword)
```

вместо

```
if (inword == true)
```

а также

```
if (!inword)
```

вместо

```
if (inword == false)
```

Основная идея состоит в том, что выражение inword == true принимает значение true, если флаг inword уже имеет значение true, и false, если inword принимает значение false, отсюда следует, что вы можете использовать флаг inword в качестве проверяемого условия. Аналогично, !inword принимает те же значения, что и выражение inword == false (not true есть false, а not false — true).

В программе, показанной в листинге 7.7, описанные идеи (распознавание строк, распознавание частичных строк и распознавание слов) реализованы.

Листинг 7.7. Программа wordcnt.c

```
// wordcnt.c -- производит подсчет символов, слов, строк
#include <stdio.h>
#include <ctype.h>    // файл, содержащий функцию isspace()
#include <stdbool.h>  // ключевые слова bool, true, false
#define STOP '|'
int main(void)
{
   char c;                // считанный символ
   char prev;             // предыдущий считанный символ
   long n_chars = 0L;     // счетчик символов
   int n_lines = 0;       // счетчик строк
   int n_words = 0;       // количество слов
   int p_lines = 0;       // количество частичных слов
   bool inword = false;   // == true если с внутри слова

   printf("Введите текст для анализа (| для завершения):\n");
   prev = '\n';           // используется для распознавания полноценных строк

   while ((c = getchar()) != STOP)
   {
      n_chars++;          // подсчет символов
      if (c == '\n')
         n_lines++;       // подсчет строк
      if (!isspace(c) && !inword)
      {
         inword = true;   // начало нового слова
         n_words++;       // подсчет слов
      }
      if (isspace(c) && inword)
         inword = false;  // достигнут конец слова
      prev = c;           // сохраняет символьное значение
   }

   if (prev != '\n')
      p_lines = 1;

   printf("количество символов = %ld, количество слов = %d, количество строк = %d, ",
      n_chars, n_words, n_lines);
   printf("количество частичных строк = %d\n", p_lines);

   return 0;
}
```

Вот результат выполнения этой программы:

```
Введите текст для анализа (| для завершения):
Reason is a
powerful servant but
an inadequate master.
|
количество символов = 55, количество слов = 9, количество строк = 3,
количество частичных строк = 0
```

Данная программа использует логические операции для перевода псевдокода в язык C. Например, псевдокод `если с есть пробел и inword принимает значение false` транслируется в следующее выражение:

```
if (!isspace(c) && !inword)
```

Обратите внимание на то, что `!inword` эквивалентно выражению `inword == false`. Все выражение проверки легче понять, чем выражение для проверки каждого пробельного символа по отдельности:

```
if (c != ' ' && c != '\n' && c != '\t' && !inword)
```

Обе эти формы означают: "если `c` не есть пробел и если вы не внутри слова". Если оба условия выполняются, это означает, что вы начинаете новое слово, а значение переменной `n_words` увеличивается на 1. Если вы находитесь в середине слова, то выполняется первое условие, в то же время значение флага `inword` будет `true`, а счетчик `n_words` не увеличивается. Как только вы выйдете на следующий пробельный символ, `inword` снова получает значение `false`. Проверьте, правильно ли работает программа в ситуации, когда между двумя словами следуют несколько пробелов подряд. В главе 8 будет показано, какие изменения потребуется внести в эту программу, чтобы она могла считывать слова из файла.

Условная операция: ?:

Язык C предлагает сокращенный способ представления одной из форм оператора `if else`. Он называется условным выражением и использует условную операцию `?:`. Эта операция состоит из двух частей и работает с тремя операндами. Вспомните, что операция с одним операндом называется унарной, а с двумя операндами — бинарной. Соблюдая эту традицию, назовем операции с тремя операндами тернарными, а рассматриваемая условная операция является в C единственным примером операций такой категории. Примером этой операции может служить получение абсолютного значения числа:

```
x = (y < 0) ? -y : y;
```

Все, что находится между знаком = и точкой с запятой, представляет собой условное выражение. Смысл этого оператора можно выразить следующим образом: "если `y` меньше нуля, то `x = -y`, в противном случае `x = y`". В терминах оператора `if else` это можно выразить следующим образом:

```
if (y < 0)
   x = -y;
else
   x = y;
```

В общем виде условное выражение можно записать так:

выражение1 ? *выражение2* : *выражение3*

Если *выражение1* дает в результате `true` (ненулевое значение), то все условное выражение принимает то же значение, что и *выражение2*. Если *выражение1* равно `false` (ноль), все условное выражение получает то же значение, что и *выражение3*.

Вы можете использовать условное выражение в ситуации, когда переменной нужно присвоить одно из двух возможных значений. Типичным примером может служить присваивание конкретной переменной наибольшего из двух значений:

```
max = (a > b) ? a : b;
```

Этот оператор присваивает переменной `max` значение `a`, если оно больше `b`, и `b` в противном случае.

Обычно оператор `if else` делает то же самое, что и условная операция. Однако версия условной операции отличается компактностью и в зависимости от типа компилятора может обеспечить генерацию более компактного кода.

В качестве примера рассмотрим программу, показанную в листинге 7.8. Эта программа вычисляет, сколько банок краски необходимо для того, чтобы покрасить заданное число квадратных футов поверхности. Основной алгоритм достаточно прост: разделить число квадратных футов, которые нужно покрасить, на число квадратных футов, которые можно покрасить содержимым одной банки. Однако предположим, что ответом будет 1.7 банки. В магазине вам не продадут такого количества краски, там краска продается целыми банками, так что вам придется приобрести две банки. Следовательно, программа должна округлить ответ до следующего целого числа. С этой целью программа использует условную операцию, эта же операция применяется для печати слова “банка” в соответствующем числе и падеже.

Листинг 7.8. Программа `paint.c`

```
/* paint.c -- использование условной операции */
#include <stdio.h>
#define COVERAGE 200 /* число квадратных футов на одну банку краски */
int main(void)
{
   int sq_feet;
   int cans;

   printf("Введите число квадратных футов, которые необходимо покрасить:\n");

   while (scanf("%d", &sq_feet) == 1)
   {
      cans = sq_feet / COVERAGE;
      cans += ((sq_feet % COVERAGE == 0)) ? 0 : 1;

      printf("Для этого потребуется %d %s краски.\n", cans,
         cans == 1 ? "банка" : "банки");
      printf("Введите следующее значение (или q для завершения):\n");
   }

   return 0;
}
```

Ниже показаны результаты выполнения этой программы:

```
Введите число квадратных футов, которые необходимо покрасить:
200
Для этого потребуется 1 банка краски.
Введите следующее значение (или q для завершения):
215
Для этого потребуется 2 банки краски.
Введите следующее значение (или q для завершения):
q
```

Поскольку в программе используется тип `int`, дробная часть результата от деления отбрасывается, то есть, 215/200 дает 1. Следовательно, количество банок округляется до ближайшего меньшего целого. Если `sq_feet % COVERAGE` равно `0`, то `COVERAGE` делит `sq_feet` без остатка, и значение `cans` остается без изменений. В противном случае появляется остаток, и значение `cans` увеличивается на 1. Это достигается с помощью следующего оператора:

```
cans += ((sq_feet % COVERAGE == 0)) ? 0 : 1;
```

Он прибавляет значение выражения, стоящее справа от знака +=, к значению `cans`. Выражение справа есть условное выражение, принимающее значение `0` или `1` в зависимости от того, делится ли `sq_feet` на `COVERAGE` без остатка.

Окончательное значение аргумента функции `printf()` также есть условное выражение:

```
cans == 1 ? "банка" : "банки");
```

Если значение переменной `cans` равно 1, употребляется строка `"банка"`, в противном случае — строка `"банки"`. Это показывает, что в условной операции могут использоваться строки в качестве второго и третьего операндов.

Сводка: условная операция

Условная операция:

`?:`

Комментарии общего характера:

Эта операция принимает три операнда, каждый из которых является выражением:

выражение1 `?` *выражение2* `:` *выражение3*

Значение всего выражения равно значению *выражение2*, если *выражение1* принимает значение `true`. В противном случае оно равно значению *выражение3*.

Примеры:

`(5 > 3) ? 1 : 2`	получает значение 1
`(3 > 5) ? 1 : 2`	получает значение 2
`(a > b) ? a : b`	получает значение большего из значений `a` или `b`

Дополнительные средства организации цикла: continue и break

Как правило, после того как управление передается телу цикла, программа выполняет все операторы тела цикла, прежде чем будет осуществлена очередная проверка конца цикла. Операторы continue и break дают возможность пропустить часть цикла и даже прекратить его выполнение в зависимости от результатов проверок, выполняемых в теле цикла.

Оператор continue

Этот оператор может использоваться во всех трех формах циклов. Когда ему передается управление, остальная часть цикла пропускается и начинается новая итерация. Если оператор continue содержится во вложенной структуре, его действия распространяются только на самую внутреннюю структуру, содержащую этот оператор. Попробуем воспользоваться оператором continue в короткой программе, показанной в листинге 7.9.

Листинг 7.9. Программа skippart.c

```
/* skippart.c -- использует оператор continue, чтобы пропустить часть цикла */
#include <stdio.h>
int main(void)
{
   const float MIN = 0.0f;
   const float MAX = 100.0f;

   float score;
   float total = 0.0f;
   int n = 0;
   float min = MAX;
   float max = MIN;

   printf("Введите результат первой игры (или q для завершения): ");
   while (scanf("%f", &score) == 1)
   {
      if (score < MIN || score > MAX)
      {
         printf("%0.1f - недопустимое значение. Повторите попытку: ", score);
         continue;
      }
      printf("Воспринято %0.1f:\n", score);
      min = (score < min)? score: min;
      max = (score > max)? score: max;
      total += score;
      n++;
      printf("Введите результат следующей игры (или q для завершения): ");
   }
   if (n > 0)
   {
      printf("Среднее значение %d результатов равно %0.1f.\n", n, total / n);
```

```
      printf("Минимальное = %0.1f, максимальное = %0.1f\n", min, max);
   }
   else
      printf("Допустимые результаты игр не введены.\n");
   return 0;
}
```

В программе из листинга 7.9 в цикле while считываются входные данные до тех пор, пока не будет введено нечисловое значение. Оператор if, включенный в этот цикл, выводит на экран недопустимые результаты. Если вы, скажем, вводите число 188, программа сообщает, что это недопустимое значение. Затем оператор continue заставляет программу пропустить остальную часть цикла, которая предназначена для обработки допустимых входных значений. Вместо того чтобы продолжать текущую итерацию, программа инициирует новую итерацию, считывая очередное входное значение.

Обратите внимание на то, что существует два способа избежать использования оператора continue. Один из таких способов заключается в том, чтобы опустить оператор continue и представить оставшуюся часть итерации как блок else:

```
if (score < 0 || score > 100)
   /* оператор printf() */
else
{
   /* операторы */
}
```

С другой стороны, вы можете воспользоваться следующим форматом:

```
if (score >= 0 && score <= 100)
{
   /* операторы */
}
```

Преимущество применения оператора continue в рассматриваемом случае связано с тем, что вы можете отказаться от одного уровня отступов в главной группе операторов. Более компактная форма повышает удобочитаемость программы, когда операторы громоздки и обладают большой глубиной вложения.

Наряду с этим оператор continue может использоваться в качестве заполнителя. Например, следующий цикл считывает и передает в программу входную последовательность символов, пока не встретит символ конца строки (включительно):

```
while (getchar() != '\n')
   ;
```

Такой метод удобен, когда программа уже прочитала некоторые входные символы из конкретной строки, и ей необходимо пропустить оставшиеся символы текущей строки и перейти в начало следующей строки. Проблема заключается в том, что одиночный символ точки с запятой обнаружить достаточно трудно. Программный код становится более удобочитаемым, если вы воспользуетесь оператором continue:

```
while (getchar() != '\n')
continue;
```

Не следует использовать оператор continue, если он усложняет, а не упрощает код. В качестве примера рассмотрим следующий фрагмент:

```
while ((ch = getchar() ) != '\n')
{
   if (ch == '\t')
      continue;
   putchar(ch);
}
```

В этом цикле пропускаются символы табуляции, а сам цикл завершается только в случае, когда встречается символ новой строки. Такой цикл можно записать в более компактной форме:

```
while ((ch = getchar()) != '\n')
   if (ch != '\t')
      putchar(ch);
```

Довольно часто, как, впрочем, и в данном случае, изменение порядка проверки в операторе if устраняет необходимость применения continue.

Как уже было показано, оператор continue обеспечивает игнорирование оставшейся части тела цикла. В какой точке возобновляется выполнение цикла? Если речь идет о циклах while и do while, то следующим действием, выполняемым сразу после continue, будет оценка условия выполнения цикла. Рассмотрим, например, следующий цикл:

```
count = 0;
while (count < 10)
{
   ch = getchar();
   if (ch == '\n')
      continue;
   putchar(ch);
   count++;
}
```

Цикл считывает 10 символов (исключая символы новой строки, поскольку оператор count++; пропускается, когда значение ch соответствует символу новой строки) и осуществляет их эхо-вывод на экран, исключая символ новой строки. Когда выполняется оператор continue, следующим вычисляется выражение проверки окончания цикла.

Для цикла for следующим действием является вычисление значения корректирующего выражения, за которым следует вычисление проверка конца цикла. Для примера рассмотрим такой цикл:

```
for (count = 0; count < 10; count++)
{
   ch = getchar();
   if (ch == '\n')
      continue;
   putchar(ch);
}
```

Когда в этом случае выполняется оператор continue, сначала инкрементируется значение count, которое затем сравнивается с 10. Таким образом, поведение этого цикла несколько отличается от поведения цикла while из рассмотренного выше примера. Как и раньше, на экране отображаются только символы, отличные от символа новой строки. Но на этот раз счетчик count учитывает символы новой строки, поэтому цикл считывает ровно 10 символов, включая символ новой строки.

Оператор break

Оператор break, включенный в цикл, заставляет программу прервать выполнение цикла, выйти из него и перейти к выполнению следующей стадии программы. В программе, представленной в листинге 7.9, замена оператора continue оператором break вызывает выход программы из цикла, когда на входе появляется, скажем, число 188, но не пропуск следующих за ним операторов цикла и переход к выполнению следующей итерации. На рис. 7.4 сравниваются операторы break и continue. Если оператор break содержится внутри вложенных циклов, его действие распространяется только на самый внутренний цикл, где он присутствует.

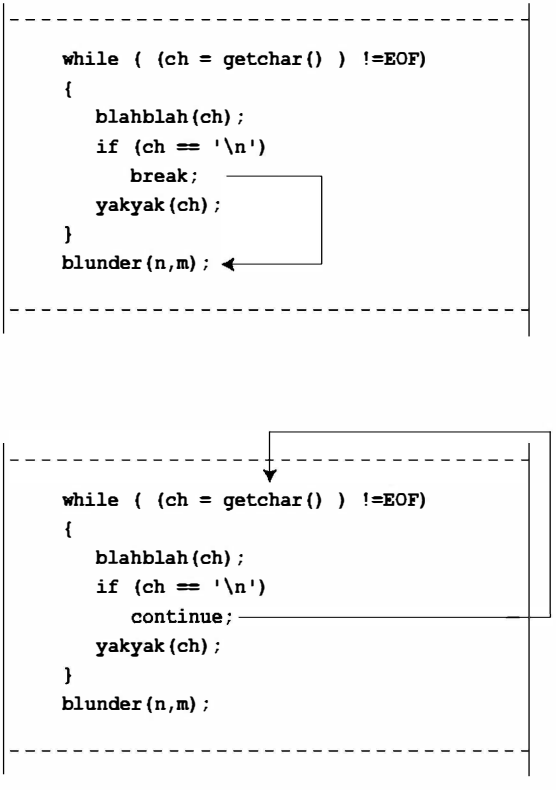


Рис. 7.4. *Сравнение операторов* `break` *и* `continue`

Иногда оператор `break` используется для выхода из цикла, когда для такого выхода имеются две разные причины. Программа, показанная в листинге 7.10, содержит цикл для вычисления площади прямоугольника. Цикл прекращается, когда вы вводите нечисловое значение в качестве длины или ширины прямоугольника.

Листинг 7.10. Программа `break.c`

```
/* break.c -- использует оператор break для выхода из цикла */
#include <stdio.h>
int main(void)
{
   float length, width;

   printf("Введите длину прямоугольника:\n");
   while (scanf("%f", &length) == 1)
   {
      printf("Длина = %0.2f:\n", length);
      printf("Введите ширину прямоугольника:\n");
      if (scanf("%f", &width) != 1)
         break;
      printf("Ширина = %0.2f:\n", width);
      printf("Площадь = %0.2f:\n", length * width);
      printf("Введите длину прямоугольника:\n");
   }
   printf("Программа завершена.\n");

   return 0;
}
```

Вы могли бы организовать цикл следующим образом:

```
while (scanf("%f %f", &length, &width) == 2)
```

В то же время использование оператора `break` существенно упрощает эхо-вывод вводимых значений. Как и в случае с `continue`, оператора `break` следует избегать, если он усложняет код программы:

```
while ((ch = getchar()) != '\n')
{
   if (ch == '\t')
      break;
   putchar(ch);
}
```

Логика программы становится прозрачнее, если обе проверки выполняются в одном и том же месте:

```
while ((ch = getchar() ) != '\n' && ch != '\t')
   putchar(ch);
```

Оператор `break` представляет собой полезное дополнение оператора `switch`, к изучению которого мы вскоре перейдем.

Оператор `break` передает управление оператору, который непосредственно следует за циклом, и в отличие от `continue`, включенного в цикл `for`; в этом случае пропускается та часть, в которой производится обновление цикла. Оператор `break`, вклю-

ченный во вложенный цикл, выводит программу только из этого внутреннего цикла, чтобы выйти из внешнего цикла, требуется еще один оператор `break`:

```
int p, q;
scanf("%d", &p);
while ( p > 0)
{
   printf("%d\n", p);
   scanf("%d", &q);
   while( q > 0)
   {
      printf("%d\n",p*q);
      if (q > 100)
         break;          // оператор break во внутреннем цикле
      scanf("%d", &q);
   }
   if (q > 100)
      break;             // оператор break во внешнем цикле
   scanf("%d", &p);
}
```

Множественный выбор: операторы `switch` и `break`

Условная операция и конструкция `if else` существенно облегчают написание программ, в которых производится выбор из двух возможных вариантов. Однако в некоторых случаях должен делаться выбор одного конкретного варианта из нескольких возможностей. Вы можете реализовать это с помощью конструкции `if else if...else`. В то же время во многих случаях может оказаться гораздо сподручнее воспользоваться оператором `switch` языка C. Программа в листинге 7.11 представляет собой пример применения оператора `switch`. Эта программа считывает букву и отвечает тем, что выводит на печать название животного, которое начинается с этой буквы.

Листинг 7.11. Программа `animals.c`

```
/* animals.c -- использование оператора switch */
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   char ch;

   printf("Дайте мне букву алфавита, и я укажу вам ");
   printf("название животного,\nначинающееся с этой буквы.\n");
   printf("Введите букву или # для завершения.\n");
   while ((ch = getchar()) != '#')
   {
      if('\n' == ch)
         continue;
      if (islower(ch))        /* только строчные буквы */
         switch (ch)
```

```
        {
           case 'а' :
              printf("аргали, дикий горный азиатский баран\n");
              break;
           case 'б' :
              printf("бабирусса, дикая малайская свинья\n");
              break;
           case 'к' :
              printf("коати, носуха обыкновенная\n");
              break;
           case 'в' :
              printf("выхухоль, водоплавающее существо\n");
              break;
           case 'е' :
              printf("ехидна, игольчатый муравьед\n");
              break;
           case 'р' :
              printf("рыболов, светло-коричневая куница\n");
              break;
           default :
              printf("Это трудная задача!\n");
        } /* конец оператора выбора */
     else
        printf("Распознаются только строчные буквы.\n");
     while (getchar() != '\n')
        continue;         /* пропустить оставшуюся часть входной строки */
     printf("Введите следующую букву или # для завершения.\n");
  }                       /* конец цикла while */
  printf("Программа завершена.\n");
  return 0;
}
```

Мы немного расслабились и предусмотрели варианты далеко не для всех букв, но можно было вполне продолжить в том же духе. Сначала рассмотрим пример выполнения этой программы, а затем проанализируем ее структуру:

```
Дайте мне букву алфавита, и я укажу вам название животного,
начинающееся с этой буквы.
Введите букву или # для завершения.
а [enter]
аргали, дикий горный азиатский баран
Введите следующую букву или # для завершения.
ваш [enter]
выхухоль, водоплавающее существо
Введите следующую букву или # для завершения.
Ф [enter]
Это трудная задача!
Введите следующую букву или # для завершения.
Е [enter]
Распознаются только строчные буквы.
Введите следующую букву или # для завершения.
# [enter]
Программа завершена.
```

Две основных особенности программы состоят в использовании оператора switch и в том, как она манипулирует входными данными. Для начала рассмотрим, как работает оператор switch.

Использование оператора switch

Сначала вычисляется выражение в круглых скобках, следующее за словом switch. В этом случае оно получает выражение, которое переменная ch получила в результате последнего ввода. Затем программа просматривает список меток (в рассматриваемом случае это case 'а' :, case 'б' : и так далее), пока не найдет подходящее значение. Программа переходит на эту строку. А что произойдет, если она не найдет соответствующей метки? Если в операторе имеется строка, помеченная как default :, программа переходит на эту строку. Если такой строки нет, программа переходит к выполнению оператора, следующего непосредственно за switch.

Как в подобной ситуации ведет себя оператор break? Он заставляет программу выйти из оператора switch и перейти к оператору, непосредственно следующему за switch (рис. 7.5). Если бы не было оператора break, выполнялся бы каждый оператор в промежутке от оператора, помеченного совпадающей с входным символом меткой, до конца оператора switch. Например, если вы удалите из программы все операторы break, а затем запустите ее и введете букву в, будет иметь место следующий диалог:

```
Дайте мне букву алфавита, и я укажу вам название животного,
начинающееся с этой буквы.
Введите букву или # для завершения.
в [enter]
выхухоль, водоплавающее существо
ехидна, игольчатый муравьед
рыболов, светло-коричневая куница
Это трудная задача!
Введите следующую букву или # для завершения.
# [enter]
Программа завершена.
```

Были выполнены все операторы, начиная с case 'в' : и до конца оператора switch.

Между прочим, оператор break работает и с циклами и с оператором выбора, в то время как оператор continue работает исключительно с циклами. Однако continue может использоваться как часть оператора switch, если switch присутствует в цикле. В такой ситуации, как, впрочем, и в других циклах, оператор continue заставляет программу пропустить остальные операторы цикла, в том числе и другие части оператора switch.

Если вы знакомы с языком программирования Pascal, то вы можете отметить большое сходство оператора switch с оператором case языка Pascal. Основное различие между ними заключается в том, что оператор switch требует использования оператора break, если вы хотите, чтобы обработке подвергались только помеченные операторы. Наряду с этим, вы не можете использовать диапазон так, как это можно делать в case.

Заключенное в круглые скобки проверочное выражение оператора switch должно иметь целочисленный тип (включая и тип char). Метки операторов также должны

быть константами целочисленного типа (включая тип char) или выражениями целочисленного типа (выражениями, содержащими только целочисленные константы). Переменные в качестве меток операторов case использовать нельзя. В общем случае структура оператора switch имеет следующий вид:

```
switch (целочисленное выражение)
{
   case константа1:
      операторы          ←не обязательные
   case константа2:
      операторы          ←не обязательные
   default :             ←не обязательно
      операторы          ←не обязательные
}
```

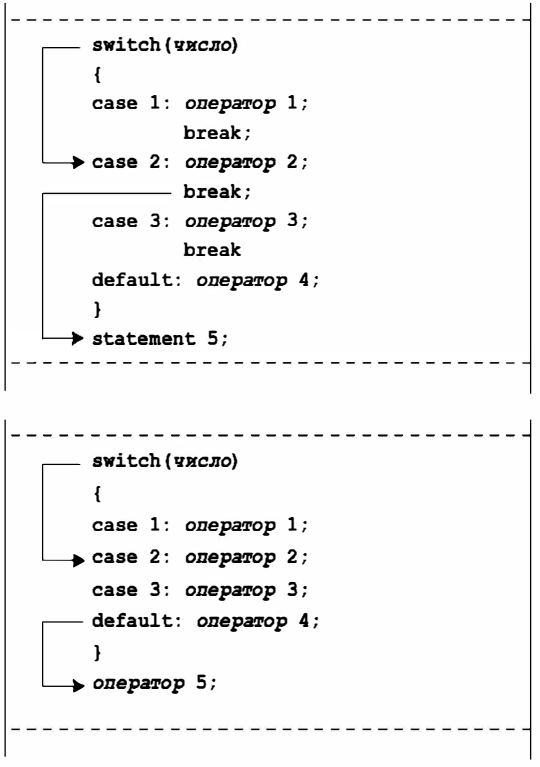


Рис. 7.5. *Поток управления программы в операторах* `switch` *с и без операторов* `break`

Считывание только первого символа строки

Еще одно новое свойство, присущее программе animals.c, связано с тем, как она считывает входные данные. Как вы, должно быть, уже заметили во время выполнения этой программы, когда была введена последовательность символов вап, обработался только первый символ. Часто такое поведение интерактивных программ, принимаю-

щих односимвольные запросы, предпочтительнее любых других. Такое поведение реализует приведенный ниже программный код:

```
while (getchar() != '\n')
   continue;       /* пропустить остальную часть входной строки */
```

Этот цикл читает входную последовательность символов, включая символ новой строки, который генерируется нажатием клавиши <Enter>. Обратите внимание, что значение, возвращаемое функцией `getchar()`, не присваивается переменной `ch`, следовательно, программа читает символы и просто их отбрасывает за ненадобностью. Так как последним отброшенным символом будет символ новой строки, то следующий считываемый символ принадлежит уже новой строке. Функция `getchar()` читает его и присваивает его значение переменной `ch` во внешнем цикле `while`.

Предположим, что пользователь начинает работу с программой, нажав <Enter>, так что первый прочитанный программой символ является символом новой строки. Следующий далее программный код учитывает такую возможность:

```
if (ch == '\n')
   continue;
```

Множественные метки

Вы можете использовать несколько меток оператора `case` для данного оператора `switch`, как показано в листинге 7.12.

Листинг 7.12. Программа `vowels.c`

```
/* vowels.c -- использование множества меток */
#include <stdio.h>
int main(void)
{
   char ch;
   int a_ct, e_ct, i_ct, o_ct, u_ct;
   a_ct = e_ct = i_ct = o_ct = u_ct = 0;
   printf("Введите текст или # для завершения программы.\n");
   while ((ch = getchar()) != '#')
   {
      switch (ch)
      {
         case 'a' :
         case 'A' : a_ct++;
                    break;
         case 'e' :
         case 'E' : e_ct++;
                    break;
         case 'i' :
         case 'I' : i_ct++;
                    break;
         case 'o' :
         case 'O' : o_ct++;
                    break;
         case 'u' :
         case 'U' : u_ct++;
                    break;
```

```
        default :  break;
      }                    /* конец оператора switch */
   }                      /* конец цикла while */
   printf("Количество гласных: A E I O U\n");
   printf("                    %4d %4d %4d %4d %4d\n",
      a_ct, e_ct, i_ct, o_ct, u_ct);
   return 0;
}
```

Если ch равен, скажем, букве i, оператор switch переходит в точку, помеченную как case 'i' :. Поскольку нет оператора break, связанного с этой меткой, поток управления программы переходит к следующему оператору, а таковым в рассматриваемом случае является оператор i_ct++;. Если значение ch равно I, поток управления программы выходит непосредственно на этот оператор. По сути дела, обе метки ссылаются на один и тот же оператор.

Строго говоря, оператор break для case 'U' не нужен, так как при его отсутствии поток управления программы выходит на следующий оператор в рассматриваемом операторе switch, каковым является break для случая по умолчанию. Следовательно, оператор break для case 'U' можно было опустить, тем самым сократив программный код. Однако с другой стороны, если позже могут быть добавлены другие случаи, то наличие оператора break там, где он должен быть, всегда будет напоминать о том, что его надо добавлять.

Ниже показан пример выполнения этой программы:

```
Введите текст или # для завершения программы.
I see under the overseer.#
Количество гласных:   A   E   I   O   U
                      0   7   1   1   1
```

В рассматриваемом случае вы можете отказаться от использования множественных меток, прибегнув к помощи функции toupper() из семейства ctype.h (см. табл. 7.2), чтобы перед проверкой условия преобразовать все строчные буквы в прописные:

```
while ((ch = getchar()) != '#')
{
 ch = toupper(ch);
 switch (ch)
 {
   case 'A' : a_ct++;
              break;
   case 'E' : e_ct++;
              break;
   case 'I' : i_ct++;
              break;
   case 'O' : o_ct++;
              break;
   case 'U' : u_ct++;
              break;
   default :  break;
 }                    /* конец оператора switch */
}                     /* конец оператора while */
```

Либо, если вы хотите оставить значение ch неизменным, воспользуйтесь этой функцией следующим образом:

```
switch(toupper(ch))
```

Сводка: множественный выбор с помощью оператора switch

Ключевое слово:

switch

Комментарии общего характера:

Поток управления программы передается на метку оператора case, которая по значению совпадает со значением *выражения*. Затем поток управления проходит через все оставшиеся операторы, пока не выйдет на оператор break. Как *выражение*, так и метки операторов case должны быть целочисленными значениями (включая тип char), а сами метки должны быть константами и выражениями, образованными исключительно из констант. Если ни одна из меток операторов case не совпадает со значением выражения, управление передается оператору с меткой default, если таковой имеется. В противном случае управление переходит к оператору, следующему непосредственно за switch.

Форма:

```
switch (выражение)
{
   case метка1 : оператор1    /* используйте break, чтобы пропустить
                                 остальные операторы */
   case метка2 : оператор2
   default : оператор3
}
```

В операторе выбора могут быть два помеченных оператора, а наличие case с меткой default необязательно.

Пример:

```
switch (choice)
   {
   case 1 :
   case 2 : printf("Мерзопакостная погода!\n"); break;
   case 3 : printf("Не так уж и плохо на дворе!\n");
   case 4 : printf("Хороший день!\n"); break;
   default : printf("Желаем хорошего отдыха.\n");
   }
```

Если переменная choice принимает целое значение 1 или 2, печатается первое сообщение, если 3 — второе и третье сообщения. (Поток управления переходит на следующий оператор, поскольку после case 3 нет оператора break.) Если она принимает значение 4, то печатается третье сообщение. Другие значения приводят к выводу только последнего сообщения.

Операторы switch и if else

Когда следует использовать оператор switch, а когда if else? Часто у вас просто нет выбора. Вы не можете использовать оператор switch, если ваш выбор основан на оценке переменной или выражения типа float. Возникают определенные трудности

при использовании оператора switch, если переменная проверяется на принадлежность к тому или иному диапазону. Легко написать следующее выражение:

```
if (integer < 1000 && integer > 2)
```

К сожалению, применение оператора switch требует назначения меток case для каждого целого числа в промежутке от 3 до 999. В то же время, если вы имеете возможность воспользоваться оператором switch, программа будет выполняться немного быстрее и станет более компактной.

Оператор goto

Оператор goto, одно важнейших средств ранних версий языков программирования BASIC и FORTRAN, реализован также и в C. В то же время язык C, в отличие от упомянутых языков, вполне может обойтись и без этого оператора. Керниган и Ритчи отзываются об операторе goto как о потенциальном источнике ошибок и советуют использовать его как можно реже, а еще лучше вообще отказаться от его применения. Далее мы объясним, почему нам не следует прибегать к его помощи.

Оператор goto состоит из двух частей: ключевого слова goto и имени метки. Именование меток производится по тем же правилам, которые соблюдаются при именовании переменных, как в показанном ниже примере:

```
goto part2;
```

Чтобы этот оператор работал правильно, в программе должен присутствовать другой оператор с присвоенной ему меткой part2. В этом случае этому второму оператору предшествует метка, за которой идет двоеточие:

```
part2: printf("Уточненный анализ:\n");
```

Избегайте использования оператора goto

По сути дела, в программе на языке C вы вполне можете обойтись и оператора goto, однако если ранее вы имели дело с языками FORTRAN или BASIC, оба из которых требуют использования этого оператора, у вас, по-видимому, выработались навыки программирования, предполагающие применение оператора goto. Чтобы помочь вам преодолеть эту привычку, рассмотрим несколько типовых ситуаций, в которых используются операторы goto, а затем покажем, как эти проблемы решаются в языке C другими средствами.

- Использование операторов goto в ситуации, когда в операторе if требуется выполнить сразу несколько операторов:

```
if (size > 12)
   goto a;
goto b;
a: cost = cost * 1.05;
   flag = 2;
b: bill = cost * flag;
```

  В ранних версиях языков программирования BASIC и FORTRAN к оператору if относился только один оператор, непосредственно следующий за условием.

Никаких средств для образования блоков и составных операторов в этих языках не было. Мы перевели этот пример на язык C. Стандартный для языка C подход с использованием составных операторов или блоков существенно облегчает восприятие смысла программы:

```
if (size > 12)
{
   cost = cost * 1.05;
   flag = 2;
}
bill = cost * flag;
```

- Выбор одного из двух возможных вариантов:

```
if (ibex > 14)
   goto a;
sheds = 2;
goto b;
a: sheds= 3;
b: help = 2 * sheds;
```

Наличие в языке C структуры `if else` позволяет сформулировать такой выбор более наглядно:

```
if (ibex > 14)
   sheds = 3;
else
   sheds = 2;
help = 2 * sheds;
```

На самом деле, в более поздних версиях BASIC и FORTRAN появилась конструкция `else`.

- Организация бесконечного цикла:

```
readin: scanf("%d", &score);
if (score < 0)
   goto stage2;
множество операторов;
goto readin;
stage2: дополнительная обработка;
```

Благодаря использованию цикла `while`, этот фрагмент программы приобретает следующий вид:

```
scanf("%d", &score);
while (score <= 0)
{
   множество операторов;
   scanf("%d", &score);
}
дополнительная обработка;
```

- Пропуск операторов до конца тела цикла и начало следующей итерации. Используйте для этой цели оператор `continue`.

- Выход из цикла. По сути, операторы `break` и `continue` представляют собой специализированные формы оператора `goto`. Преимущество их использования заключается в том, что их названия говорят, для чего они предназначены, и в том, что можно не опасаться, что метки будут расставлены не там, где надо, поскольку они вообще не имеют дела с метками.
- Беспорядочные переходы из одних частей программы в другие.

И все-таки имеет место случай, когда опытные программисты, имеющие солидный опыт работы на языке C, допускают употребление оператора `goto`: выход из вложенного набора циклов при возникновении ошибки (одиночный оператор `break` обеспечивает выход только из самого внутреннего цикла):

```
while (funct > 0)
   {
   for (i = 1, i <= 100; i++)
     {
     for (j = 1; j <= 50; j++)
         {
         совокупность операторов;
         if (ошибка)
            goto help;
         операторы;
         }
      некоторое число операторов;
      }
   еще некоторое число операторов;
   }
дальнейшие операторы;
help : код устранения ошибки;
```

Как легко убедиться на других примерах, альтернативные формы представления программ более удобны для восприятия, чем формы, в которых используются операторы `goto`. Эти различия становятся еще более заметными, когда имеет место сочетание нескольких ситуаций подобного рода. Какие операторы `goto` успешно используются в сочетании с операторами `if else`, какие из них управляют циклами, а какие появились в программе только потому, что вы столкнулись с затруднениями и не нашли другого выхода? Используя операторы там где надо и где не надо, вы загоняете в лабиринт поток управления программы. Если вы не знаете, как правильно применять операторы `goto`, не тратьте сил на ее его изучение и вообще откажитесь от его использования. Если вы привыкли пользоваться этим оператором, постарайтесь отвыкнуть. По иронии судьбы, язык C, который вовсе не нуждается в операторе `goto`, лучше всех языков программирования приспособлен для его применения, поскольку в качестве меток можно выбирать смысловые имена, а не числа.

Сводка: переходы в программах

Ключевые слова:

`break`, `continue`, `goto`

Комментарии общего характера:

Выполнение каждого из трех этих операторов вызывает прерывание естественного потока управления программы, которое проявляется в скачкообразной передаче управления из одного места программы в другое.

Оператор `break`:

Оператор `break` может использоваться с любой из трех форм цикла, а также внутри оператора `switch`. Он заставляет программу пропустить все дальнейшие операторы или конструкцию `switch`, содержащую его, и перейти к команде, следующей за циклом или за оператором `switch`.

Пример:

```
switch (number)
{
   case 4: printf("Это лучший выбор.\n");
           break;
   case 5: printf("Это хороший выбор.\n");
           break;
   default: printf("Это плохой выбор.\n");
}
```

Оператор `continue`:

Оператор `continue` может использоваться с любой из трех форм циклов, но не в операторе `switch`. Его выполнение приводит к тому, что следующие за ним в теле цикла операторы пропускаются. В циклах `while` и `for` после этого начинается новая итерация. В цикле `do while` проверяется условие выхода из цикла, а затем, при необходимости, стартует следующая итерация цикла.

Пример:

```
while ((ch = getchar()) != '\n')
{
   if (ch == ' ')
      continue;
   putchar(ch);
   chcount++;
}
```

Этот фрагмент осуществляет эхо-печать символов, отличных от пробела, и ведет их подсчет.

Оператор `goto`:

Оператор `goto` вызывает передачу управления в программе оператору, помеченному меткой, указанной в `goto`. Двоеточие используется для отделения помеченного оператора от его метки. Имена меток подчиняются тем же правилам, что и имена переменных. Помеченный оператор может находиться до или после оператора `goto`.

Форма:

```
goto метка;
.
.
.
метка : оператор
```

Пример:

```
top : ch = getchar();
  .
  .
  .
if (ch != 'y')
  goto top;
```

Ключевые понятия

Один из аспектов интеллекта состоит в способности выбирать реакцию в зависимости от ситуации. В силу этого обстоятельства операторы выбора представляют собой фундамент для разработки программ с элементами интеллекта. В языке C выбор реализуется с помощью операторов `if`, `if else` и `switch`, а также условной операции (`?:`).

Операторы `if` и `if else` используют условие проверки для определения того, какие операторы должны быть выполнены. Любое ненулевое значение рассматривается как истинное значение, в то время как ноль — как ложное. Обычно в проверках используются условные выражения, в которых сравниваются два значения, и логические выражения, в которых посредством логических операций создаются сложные выражения.

Один из основных принципов, которые всегда следует иметь в виду, гласит: если вы хотите проверить два условия, вы должны использовать соответствующую логическую операцию и два завершенных выражения проверки. Например, две следующие попытки ошибочны:

```
if (a < x < z)            //неправильно — отсутствует логическая операция
...
if (ch != 'q' && != 'Q') //неправильно — отсутствует завершенное выражение
...
```

Напомним, что правильный способ заключается в соединении двух условных выражение в одно с помощью логической операции:

```
if (a < x && x < z)          // для объединения двух выражений
                             // используется операция &&
...
if (ch != 'q' && ch != 'Q') // для объединения двух выражений
                             // используется операция &&
```

Управляющие операторы, представленные в двух последних главах, позволяют составлять программы, которые обладают большими возможностями и способны решать более сложные задачи, нежели те, которые рассматривались в начальных главах книги. Чтобы убедиться в этом, достаточно сравнить текущие примеры программ с программами из предшествующих глав.

Резюме

В данной главе было рассмотрено несколько тем, и сейчас проведем их краткий обзор. Оператор `if` использует условие проверки для определения, когда программа должна выполнить какой-либо отдельный оператор или блок операторов. Выполнение происходит в том случае, когда проверочное выражение получает ненулевое значение, и не происходит, если это значение равно нулю. Оператор `if else` позволяет производить выбор из двух возможностей. Если проверяемое значение не равно нулю, выполняется оператор, предшествующий `else`. Если проверяемое значение является нулевым, выполняется оператор, следующий за `else`. Если воспользоваться другим оператором `if`, который непосредственно следует за `else`, можно построить структуру, которая производит выбор из некоторой последовательности возможных вариантов.

Часто в качестве выражения проверки применяется *условное выражение*, то есть выражение, построенное с использованием одной условной операции, такой как, например, < или ==. С помощью логических операций языка C можно создавать различные комбинации условных выражений, представляющие более сложные выражения.

Условная операция (`? :`) позволяет создавать выражения, которые во многих случаях оказываются более компактной альтернативой операторам `if else`.

Семейство символьных функций `ctype.h`, в состав которого входят, например, `isspace()` и `isalpha()`, предлагает удобные инструментальные средства для построения выражений проверки на базе классификации символов.

Оператор `switch` предоставляет возможность выбора из заданной последовательности операторов, помеченных целочисленными значениями. Если целое значение условия проверки, следующего непосредственно за ключевым словом оператора `switch`, совпадает с одной из меток, управление передается оператору, с которым связана эта метка. Управление проходит через операторы, следующие за помеченным оператором, пока не встретится оператор `break`.

И, наконец, операторы `break`, `continue` и `goto` — это операторы безусловного перехода, которые заставляют программу изменить естественный ход процесса и передать управление в другое место. Оператор `break` вынуждает программу перейти к выполнению оператора, который непосредственно следует за концом цикла или оператора `switch`, содержащего `break`. Оператор `continue` заставляет программу пропустить выполнение операторов, следующих за ним в теле цикла, и начать новую итерацию.

Вопросы для самоконтроля

1. Определите, какие выражения истинны, а какие ложны.
   а. `100 > 3 && 'a'>'c'`
   б. `100 > 3 || 'a'>'c'`
   в. `!(100>3)`
2. Напишите выражения, формулирующие следующие условия:
   а. Значение `number` равно или больше 90, но меньше 100.
   б. Значение переменной `ch` не является символом `q` или `k`.
   в. Значение переменной `number` находится в промежутке между 1 и 9 (включая концевые значения), но не равно 5.
   г. Значение `number` не попадает в диапазон от 1 до 9.
3. В приведенной ниже программе используются чрезмерно сложные выражения, в ней также присутствуют явные ошибки. Упростите программу и устраните ошибки.

```
#include <stdio.h>
int main(void)                                          /* 1 */
{                                                       /* 2 */
int weight, height;          /* вес в фунтах, рост в дюймах */
                                                        /* 4 */
   scanf("%d , weight, height);                         /* 5 */
   if (weight < 100 && height > 64)                     /* 6 */
```

```
      if (height >= 72)                                  /* 7 */
         printf("Ваш вес слишком мал для вашего роста.\n");
      else if (height < 72 && > 64)                      /* 9 */
         printf("Ваш вес мал для вашего роста.\n");      /* 10 */
   else if (weight > 300 && ! (weight <= 300)            /* 11 */
          && height < 48)                                /* 12 */
      if (!(height >= 48) )                              /* 13 */
         printf("Ваш рост мал для вашего веса.\n");
   else                                                  /* 15 */
      printf("У вас идеальный вес.\n");                  /* 16 */
                                                         /* 17 */
   return 0;
}
```

4. Каковы числовые значения каждого из следующих выражений?

   а. `5 > 2`

   б. `3 + 4 > 2 && 3 < 2`

   в. `x >= y || y > x`

   г. `d = 5 + ( 6 > 2 )`

   д. `'X' > 'T' ? 10 : 5`

   е. `x > y ? y > x : x > y`

5. Что напечатает следующая программа?

```
#include <stdio.h>
int main(void)
{
   int num;
   for (num = 1; num <= 11; num++)
   {
      if (num % 3 == 0)
         putchar('$');
      else
         putchar('*');
         putchar('#');
      putchar('%');
   }
   putchar('\n');
   return 0;
}
```

6. Что напечатает следующая программа?

```
#include <stdio.h>
int main(void)
{
   int i = 0;
   while ( i < 3) {
      switch(i++) {
         case 0 : printf("fat ");
         case 1 : printf("hat ");
         case 2 : printf("cat ");
         default: printf("Oh no!");
      }
      putchar('\n');
   }
   return 0;
}
```

7. Какие ошибки допущены в следующей программе?

```
#include <stdio.h>
int main(void)
{
   char ch;
   int lc = 0; /* подсчет символов нижнего регистра
   int uc = 0; /* подсчет символов верхнего регистра
   int oc = 0; /* подсчет всех остальных символов
   while ((ch = getchar()) != '#')
   {
      if ('a' <= ch >= 'z')
         lc++;
      else if (!(ch < 'A') || !(ch > 'Z')
         uc++;
      oc++;
   }
   printf(%d lowercase, %d uppercase, %d other, lc, uc, oc);
   return 0;
}
```

8. Что напечатает следующая программа?

```
/* retire.c */
#include <stdio.h>
int main(void)
{
   int age = 20;
   while (age++ <= 65)
   {
      if (( age % 20) == 0) /* Делится ли возраст на 20? */
        printf("Вам %d лет. Получите заслуженную надбавку к зарплате.\n",
              age);
      if (age = 65)
        printf("Вам are %d лет. Получите ваши золотые часы.\n", age);
   }
   return 0;
}
```

9. Что напечатает в приведенная ниже программа в ответ на следующий ввод?

```
q
c
g
b
#include <stdio.h>
int main(void)
{
   char ch;
   while ((ch = getchar()) != '#')
   {
      if (ch == '\n')
         continue;
      printf("Шаг 1\n");
```

```
        if (ch == 'c')
           continue; q
        else if (ch == 'b')
           break;
        else if (ch == 'g')
           goto laststep;
        printf("Шаг 2\n");
     laststep: printf("Шаг 3\n");
     }
     printf("Готово\n");
     return 0;
}
```

10. Перепишите программу из пункта 9 таким образом, чтобы она не использовала операторы `continue` и `goto`, но при этом не изменила поведение.

Упражнения по программированию

1. Напишите программу, которая читает входные символы до тех пор, пока не встретится символ #, а затем отображает количество считанных пробелов, количество символов новой строки и количество всех других символов.
2. Напишите программу, которая читает входные символы до тех пор, пока не встретится символ #. Программа должна печатать каждый символ и его ASCII-код в десятичном представлении. Распечатайте выходные данные по восемь пар символ — код в строке. Совет: используйте счетчик символов и операцию деления по модулю (`%`) для печати символа новой строки для каждых восьми итераций цикла.
3. Напишите программу, которая читает целые числа до тех пор, пока при вводе не встретится `0`. Как только ввод прекратится, программа должна сообщить общее количество четных чисел во входных данных (за исключением `0`), среднее количество четных чисел, общее количество нечетных чисел во входных данных и среднее значение нечетных чисел.
4. Используя операторы `if else`, напишите программу, которая читает входную последовательность символов, пока не встретится символ #, заменяет каждую точку на восклицательный знак, каждый восклицательный знак в исходном входном потоке — на два восклицательных знака, и сообщает в конце о количестве выполненных замен.
5. Выполнить упражнение 3 с использованием оператора `switch`.
6. Напишите программу, которая читает входные символы до тех пор, пока не встретится символ #, и сообщает, сколько раз во входной последовательности символов встретилось сочетание `ei`.

На заметку!

Эта программа должна "помнить" предыдущий символ, равно как и текущий символ. Проверьте ее на входной последовательности символов "Receive your eieio award".

7. Напишите программу, которая запрашивает ввод количества часов, отработанных за неделю, а затем выводит на печать заработную плату без вычетов, сумму налогов и зарплату после вычетов. Примите во внимание следующие условия:
   а. Базовый почасовый тариф = $10.00/час
   б. Переработка (при превышении 40 часов в неделю) = в полтора раза
   в. Налоговая ставка: 15% с первых $300
      20% со следующих $150
      5% с остальной суммы

   Воспользуйтесь константами #define, и пусть вас не беспокоит тот факт, что рассматриваемый пример не соответствует действующему налоговому законодательству.

8. В условие (а) упражнения 7 внесите такие изменения, чтобы программа предлагала меню для выбора ставки заработной платы. С этой целью используйте оператор switch. Начало выполнения программы может выглядеть приблизительно так:

```
*********************************************************************
Введите число, соответствующее предпочитаемой ставке заработной платы
или действию:
1) $8.75/час                                    2) $9.33/час
3) $10.00/час                                   4) $11.20/час
5) выход
*********************************************************************
```

   Если выбраны варианты 1–4, программа должна потребовать ввода отработанных за неделю часов. Программа должна выполнять цикл до тех пор, пока не будет введена цифра 5. Если будет введена цифра, отличная от цифр в диапазоне от 1 до 5, программа должна напомнить пользователю, каким должен быть правильный ввод, после чего начать следующий цикл ввода. Воспользуйтесь константами #define для представления различных налоговых ставок и ставок заработной платы.

9. Напишите программу, которая в качестве входных данных принимает целое число, а затем выводит на экран все простые числа, которые меньше или равны введенному числу.

10. В 1988 году шкала федеральных налоговых ставок Соединенных Штатов за последнее время была одной из самых простых. Она разбита на четыре категории, каждая из которых содержит две ставки. Ниже приведены самые общие данные (суммы в долларах представляют собой налогооблагаемый доход):

Категория	*Налог*
Одинокий	15% с первых $17 850 плюс 28% при превышении этой суммы
Глава семьи	15% с первых $23 900 плюс 28% при превышении этой суммы
Состоит в браке, совместное ведение хозяйства	15% с первых $29 750 плюс 28% при превышении этой суммы
Состоит в браке, раздельное ведение хозяйства	15% с первых $14 875 плюс 28% при превышении этой суммы

Например, одинокий работник, получающий налогооблагаемую заработную плату $20 000, платит налоги в сумме 0.15 × $17 850 + 0.28 × ($20 000 – $17 850). Напишите программу, которая предоставляет пользователю возможность выбрать категорию и налогооблагаемый доход, после чего вычисляет сумму налогов. Воспользуйтесь циклом с тем, чтобы пользователь мог рассмотреть различные варианты налогообложения.

11. Компания ABC Mail Order Grocery, торгующая бакалейными товарами по заказам, поступающим по электронной почте, продает артишоки по цене $1.25 за фунт, свеклу по $0.65 за фунт и морковь по $0.89 за фунт. Она предоставляет 5-процентную скидку на заказы на сумму $100 без учета затрат на транспортировку. Она назначает тариф в сумме $3.50 за доставку и обработку заказа весом в 5 фунтов и ниже, $10.00 за обработку и доставку заказа весом от 5 до 20 фунтов и $8.00 плюс $0.10 за каждый фунт для при заказе с весом, превышающем 20 фунтов. Напишите программу, использующую оператор `switch` в цикле, котором в ответ на ввод символа `a` пользователю предоставляется возможность указать вес заказываемых артишоков в фунтах, в ответ на ввод `b` — вес заказываемой свеклы в фунтах, в ответ на ввод `c` — вес заказываемой моркови в фунтах и в ответ на ввод `q` — завершить процесс формирования заказа. Затем программа вычисляет общие издержки, скидку, если таковая имеет место, расходы на доставку и окончательную сумму заказа. Затем программа должна отобразить на экране всю информацию о покупке: стоимость фунта товара, количество заказанных фунтов, стоимость данного заказа в расчете на каждый овощ, общую стоимость заказа, скидку (если есть), стоимость доставки и итоговую сумму заказа с учетом всех факторов.

ГЛАВА 8

Символьный ввод–вывод и верификация ввода

В этой главе:

- Дальнейшее изучение вопросов ввода–вывода и различия между буферизованным и небуферизованным вводом данных
- Моделирование условия конца файла с клавиатуры
- Использование перенаправления для установки соединения программы с файлами
- Создание дружественных пользовательских интерфейсов

В вычислительном мире термины *ввод* и *вывод* употребляются в нескольких смыслах. Мы говорим об устройствах ввода и вывода, таких как клавиатуры, дисковые накопители и лазерные принтеры. Мы говорим о данных, которые вводятся в систему и выводятся из системы. Мы употребляем эти слова в отношении функций, которые выполняют ввод и вывод. Основное внимание в этой главе уделяется именно функциям ввода-вывода.

Функции ввода-вывода транспортируют информацию в вашу программу и из нее; примерами могут служить такие функции, как `printf()`, `scanf()`, `getchar()` и `putchar()`. Вам уже приходилось встречаться с этими функциями в предыдущих главах, а в этой главе вы изучите их концептуальные основы. Наряду с этим вы узнаете, как можно улучшить пользовательский интерфейс программ.

Первоначально функции ввода-вывода не входили в определение языка C. Их разработка была оставлена за реализациями. На практике моделью для этих функций стала реализация языка C на базе операционной системы Unix. Библиотека функций языка ANSI C, признавая всю важность функций, разработанных в прошлом, содержит большое количество функций ввода-вывода, ориентированных на работу под управлением Unix, в том числе и некоторые из тех, которые мы использовали выше. Поскольку эти стандартные функции должны работать в широком диапазоне компьютерных сред, они редко пользуются преимуществами, характерными для конкретных систем. В силу этого обстоятельства многие поставщики вариантов языка C предлагают дополнительные функции ввода-вывода, которые используют эти особые свойства, такие как, например, порты ввода-вывода микропроцессоров Intel или стандартные

программы ПЗУ Macintosh. Другие функции или семейства функций включаются в конкретные операционные системы, которые поддерживают, например, специальные графические интерфейсы наподобие интерфейсов операционных систем Windows и Macintosh OS. Эти специализированные, нестандартные функции предоставляют возможность писать программы, которые используют конкретные компьютеры с большей эффективностью. К сожалению, они зачастую не могут использоваться в средах других компьютерных систем. Соответственно, мы сосредоточим основное внимание на стандартных функциях ввода-вывода, которые доступны на всех ваших системах, поскольку они позволяют создавать переносимые программы, которые легко можно переносить с одной системы на другую. Они также повышают уровень универсальности программ, позволяя использовать файлы для ввода и вывода.

Одна важная задача, с которой сталкиваются многие программы, связана с проверкой допустимости входных данных, или верификацией данных, то есть в проверке, соответствуют ли данные, вводимые пользователями, ожиданиям программы. В данной главе рассматриваются некоторые из проблем, связанные с проверкой допустимости данных, и предлагаются методы ее решения.

Односимвольные функции ввода-вывода: getchar() и putchar()

Как можно было убедиться во время изучения материала главы 7, функции `getchar()` и `putchar()` выполняют посимвольный ввод и вывод. Такой метод решения задачи ввода-вывода может показаться вам нерациональным и неразумным. В конце концов, вы легко можете считывать группы символов, состоящих из более чем одного символа, однако этот метод соответствует возможностям компьютера. Более того, такой подход служит основой большей части программ, в задачу которых входит обработка текста, иначе говоря, обычных слов. Чтобы вспомнить, как работают эти функции, рассмотрим очень простой пример, представленный в листинге 8.1. Эта программа принимает символ с клавиатуры и отображает его на экране. Такой процесс называется *эхо-повтором ввода*. В ней используется цикл `while`, который завершается в случае ввода символа #.

Листинг 8.1. Программа `echo.c`

```
/* echo.c -- повторяет вводимые символы */
#include <stdio.h>
int main(void)
{
   char ch;
   while ((ch = getchar()) != '#')
      putchar(ch);
   return 0;
}
```

Язык ANSI C ассоциирует заголовочный файл `stdio.h` с использованием функций `getchar()` и `putchar()`, и мы включили этот файл в программу именно по этой причине. (Как правило, функции `getchar()` и `putchar()` не являются функциями в пол-

ном смысле этого слова, они определяются посредством макросов препроцессора; этой теме будет посвящена глава 16.) При выполнении этой программы возможен примерно такой диалог:

```
Здравствуйте. Я хотел бы[enter]
Здравствуйте. Я хотел бы
приобрести #3 сетки картофеля.[enter]
приобрести
```

Ознакомившись с тем, как работает эта программа, вы, очевидно, захотите узнать, почему нужно ввести с клавиатуры всю строку, прежде чем входные символы будут отображены на экране. Вы, возможно, также захотите знать, нет ли лучшего способа завершить ввод. Использование специального символа, такого как #, для завершения ввода, не позволяет указывать этот символ в тексте. Чтобы ответить на эти вопросы, рассмотрим, каким образом программы на языке C обрабатывают ввод с клавиатуры. В частности, выясним, что такое буферизация, и ознакомимся с понятием стандартного входного файла.

Буферы

Когда вы выполняете приведенную выше программу на некоторых системах, текст, который вводится с клавиатуры, немедленно отображается на экране. Выполняя эту программу, вы получите нечто, подобное следующему результату:

```
ЗЗддррааввсстввууййттее..  ЯЯ  ххоотеелл  ббы[enter]
ппрриоообрреесстии  #
```

Подобное поведение не является типичным. В большинстве систем ничего не произойдет, если не нажать клавишу <Enter>, как и в первом примере. Немедленный эхо-вывод символов входных данных на экране представляет собой пример *небуферизованного* ,или *прямого* ввода, означающего, что символ, который вы ввели с клавиатуры, немедленно становится доступным для ожидающей его программы. С другой стороны, задержанный эхо-вывод характеризует *буферизованный* ввод, когда введенные вами символы накапливаются и хранятся во временной области, называемой *буфером*. Нажатие клавиши <Enter> приводит к тому, что введенный блок символов становятся доступными для программы. На рис. 8.1 сравниваются эти два вида ввода.

Для чего нужны буферы? Во-первых, пересылка нескольких символов в виде блока требует меньше времени, чем посимвольная пересылка. Во-вторых, в случае опечатки можно выполнить необходимые исправления с клавиатуры. Когда вы в конечном итоге нажмете клавишу <Enter>, вы сможете передать откорректированную версию текста.

С другой стороны, небуферизованный ввод может подходить для некоторых интерактивных программ. Например, в играх вы хотите, чтобы действие команды исполнялось, как только нажимается соответствующая клавиша. В результате как буферизованный, так и небуферивованный ввод находят широкое применение.

Буферизация реализуется в двух видах: *полностью буферизованный ввод-вывод* и *построчно буферизованный ввод-вывод*. В рамках полностью буферизованного ввода буфер очищается (содержимое посылается по месту назначения) в момент его заполнения. Этот вид буферизации обычно осуществляется при вводе файла. Размер буфера зависит от системы, но обычно их размеры составляют 512 или 4096 байтов. При использовании построчно буферизованного ввода-вывода посылка его содержимого произ-

водится всякий раз, когда обнаруживается символ новой строки. Ввод с клавиатуры представляет собой пример обычного построчно буферизованного ввода, когда при нажатии клавиши <Enter> буфер очищается.

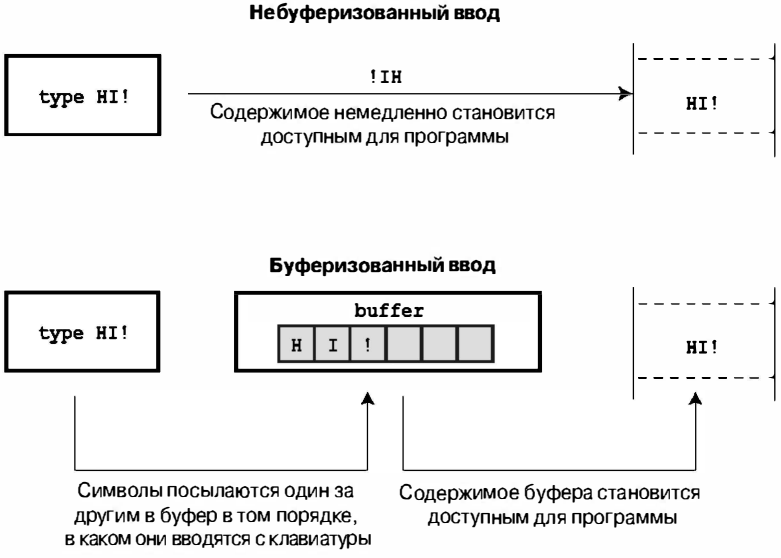


Рис. 8.1. *Буферизованный и небуферизованный ввод*

С каким типом ввода вы будете иметь дело — буферизованным или небуферизованным? Язык ANSI C требует, чтобы ввод был буферизованный, в то же время K&R оставляет право выбора за разработчиками компилятора. Тип ввода, используемый в системе, можно определить, запустив на выполнение программу `echo.c` и проанализировав ее поведение.

Причина выбора языком ANSI C буферизованного ввода в качестве стандарта заключается в том, что некоторые компьютерные системы не могут работать с небуферизованным вводом. Если конкретный компьютер допускает небуферизованный ввод, то вполне вероятно, что компилятор C разрешает небуферизованный ввод в качестве дополнительной и необязательной возможности. Например, многие компиляторы для персональных компьютеров, совместимых с IBM PC, предоставляют специальное семейство функций, поддерживаемых заголовочным файлом `conio.h` и предназначенных для обеспечения небуфферизованного ввода-вывода. К числу этих функций относится функция `getche()`, используемая для эхо-вывода символов (*Ввод с эхо-выводом* означает, что вводимый символ отображается на экране, а *ввод без эхо-вывода* — что значения нажатых клавиш на экране не отображаются.) В операционной системе Unix используется другой подход, поскольку буферизацией управляет сама система Unix. Что касается Unix, то в ней применяется функция `ioctl()` (входит в состав библиотеки Unix, но не является частью стандарта C), которая устанавливает нужный тип ввода, и функция `getchar()`, которая ведет себя соответствующим образом. В языке ANSI C функции `setbuf()` и `setvbuf()` (см. главу 13) обеспечивают некоторый контроль над буферизацией, но характерные для ряда систем ограничения снижают их эффективность.

Короче говоря, в стандарте ANSI не существует способа задания небуферизованного ввода; соответствующие средства зависят от компьютерной системы. В этой книге мы будем полагать, что вы используете буферизованный ввод, в связи с чем приносим свои извинения приверженцам небуферизованного ввода.

Завершение ввода с клавиатуры

При вводе символа # выполнение программы `echo.c` останавливается, что удобно до тех пор, пока этот символ исключен из обычных входных данных. Однако, как вы уже видели, символ # может встречаться при обычном вводе. В идеальном случае хотелось бы иметь символ завершения ввода, который в обычном тексте не появляется. Такой символ не может неожиданно появиться в середине некоторого входного текста, в результате чего программа останавливается там, где это не ожидается. Язык C предоставляет такие возможности, однако, чтобы воспользоваться ими, вы должны знать, как C работает с файлами.

Файлы, потоки и ввод данных с клавиатуры

Файл — это область памяти, в которой хранится информация. Обычно файл размещается на некотором носителе постоянной памяти, например, гибком и жестком магнитном диске или магнитной ленте. Важность файлов для компьютерных систем несомненна. Например, программы, написанные на C, хранятся в файлах, программы, используемые для компиляции ваших программ, также хранятся в файлах. Последний пример указывает на то, что некоторые программы испытывают потребность доступа к конкретным файлам. При компиляции программы, сохраненной в файле `echo.c`, компилятор открывает этот файл и читает его содержимое. По окончании компиляции он закрывает этот файл. Другие программы, такие как текстовые процессоры, не только открывают, читают и закрывают файлы, они также выполняют в них запись.

C, будучи мощным, гибким языком программирования с множеством достоинств, обладает многими библиотечными функциями, предназначенными для открывания, чтения, записи и закрытия файлов. На одном уровне он может работать с файлами, используя с этой целью основные инструментальные средства манипулирования файлами, которые имеет операционная система хоста. Эти инструментальные средства получили название *низкоуровневого ввода-вывода*. В связи с тем, что между компьютерными системами существуют очень важные различия, невозможно создать стандартную библиотеку универсальных функций низкоуровневого ввода-вывода, к тому же стандарт ANSI C и не пытается делать это; в то же время язык C работает с файлами и на другом уровне, получившем название *стандартного пакета ввода-вывода*. При этом предполагается создание стандартной модели и стандартного набора функций ввода-вывода, предназначенных для работы с файлами. На этом более высоком уровне различия между системами нивелируются специальными реализациями языка C, обеспечивающими единый интерфейс, с которыми вы работаете.

О каких различиях идет речь? Разные системы, например, сохраняют файлы разными способами. Некоторые из них хранят содержимое файла в одном месте, а информацию о нем — в другом месте. Другие системы встраивают описание файла в сам файл. При работе с текстами многие системы используют одиночный символ новой строки для обозначения конца строки. Иногда для представления конца строки при-

меняется комбинация символов возврата каретки и перевода строки. Есть системы, измеряющие размер файлов до ближайшего байта, другие же измеряют файлы блоками байтов.

Использование стандартного пакета ввода-вывода позволяет смягчить проблемы, порождаемые такими различиями. Благодаря этому пакету, вы можете выполнить проверку на наличие символа новой строки, воспользовавшись конструкцией `(ch == '\n')`. Если система фактически использует комбинацию символов возврата каретки и перевода строки, то функция ввода-вывода осуществляет автоматическую трансляцию в прямом и обратном направлениях между двумя этими представлениями.

По идее, программа на C имеет дело непосредственно с потоком, а не с файлом. *Поток* представляет собой идеализированный поток данных, на который фактически отображаются входные и выходные данные. Это означает, что различные виды входных данных с отличающимися свойствами представлены в виде потоков, свойства которых в значительной степени унифицированы. Процесс открытия файла в этом случае становится процессом ассоциирования конкретного потока с файлом, а операции чтения и записи осуществляются через этот поток.

В главе 13 представлено более подробное обсуждение файлов. В данной главе мы просто отметим, что C рассматривает устройства ввода и вывода как обычные файлы, размещенные на запоминающих устройствах. В частности, клавиатура и устройства отображения рассматриваются как файлы, которые автоматически открываются каждой программой на C.

Ввод данных с клавиатуры представлен потоком с именем `stdin`, а данные, выводимые на экран (или на телетайп или другое устройство вывода), представлены потоком с именем `stdout`. Функции `getchar()`, `putchar()`, `printf()` и `scanf()` — элементы стандартного пакета ввода-вывода, и они работают с этими двумя потоками.

Одним из следствий рассмотренных выше подробностей есть то, что вы можете использовать одну и ту же технологию как при вводе данных с клавиатуры, так и при работе с файлами. Например, программе, считывающей файл, необходим способ обнаружения конца файла, чтобы знать, где закончить считывание. В силу этого обстоятельства функции ввода в языке C укомплектованы встроенным средством обнаружения конца файла. Поскольку данные, вводимые с клавиатуры, рассматриваются как файл, то вы также должны иметь возможность употребить это средство обнаружения конца файла для завершения ввода данных с клавиатуры. Рассмотрим, как все это делается сначала на примере работы с файлами.

Конец файла

В операционных системах возникает потребность в тех или иных способах, отмечающих, где начинается и где кончается файл. Один из таких методов обнаружения конца файла требует помещения в файл специального символа, отмечающего его конец. Этот метод когда-то использовался, например, в текстовых файлах операционных систем CP/M, IBM-DOS и MS-DOS. В настоящее время эти операционные системы могут прибегать к помощи встроенного символа <Ctrl+Z> для отмечания конца файла. Было время, когда этот маркер был единственным средством, которое эти операционные системы употребляли с этой целью, однако сейчас доступны и другие варианты, например, отслеживание размеров файла. Таким образом, в современных системах в текст может быть, а может и не быть встроен символ <Ctrl+Z>, но если он

встроен, то операционная система рассматривает его как маркер конца файла. Рисунок 8.2 служит иллюстрацией такого подхода.

Второй подход заключается в том, что операционные системы хранят информацию о размере файла. Если файл содержит 3000 байтов, и программа прочитала 3000 байтов, значит, она достигла конца файла. Операционная система MS-DOS и ей подобные применяют этот подход для двоичных файлов, поскольку метод позволяет хранить все символы, включая и <Ctrl+Z>. Более поздние версии системы DOS также используют этот подход при работе с текстовыми файлами. Система Unix применяет этот подход в отношении всех файлов.

Язык C управляет всем этим разнообразием методов с помощью функции `getchar()`, которая возвращает специальное значение при достижении конца файла независимо от того, как на самом деле операционная система обнаружила этот конец файла. Это специальное значение получило имя `EOF` ("end of file" – "конец файла"). Следовательно, значением, которое возвращает функция `getchar()`, когда обнаруживает конец файла, является `EOF`. Функция `scanf()` также возвращает `EOF` при обнаружении конца файла. Обычно `EOF` определяется в файле `stdio.h` следующим образом:

```
#define EOF (-1)
```

Почему используется значение `-1`? Обычно функция `getchar()` возвращает значение в диапазоне от `0` до `127`, поскольку таковыми являются значения, соответствующие стандартному набору символов, в то же время она может возвратить значение в пределах от `0` до `255`, если система воспринимает расширенный набор символов. В любом случае значение `-1` не соответствует ни одному из символов, следовательно, оно может использоваться в качестве признака конца файла.

Некоторые системы могут определять значение `EOF` как величину, отличную от `-1`, но его определение всегда отличается от возвращаемого значения, генерируемого допустимым входным символом. Если вы включаете в программу файл `stdio.h` и используете символ `EOF`, вам не стоит беспокоиться относительно его числового значения. При этом необходимо отметить, что `EOF` представляет значение, которое сообщает о том, что обнаружен конец файла: это не есть символ, обнаруженный в файле.

Все это хорошо, но как вы можете использовать маркер `EOF` в программе? Сравните значение, возвращаемое функцией `getchar()` с `EOF`. Если они отличаются друг от друга, это означает, что вы еще не достигли конца файла.

Текст:

В половине двенадцатого с северо-запада,
со стороны деревни Чмаровки, в Старгород
вошел молодой человек лет двадцати восьми.

Текст в файле:

```
В половине двенадцатого с северо-запада,\n со стороны деревни Чмаровки, в Старгород\n вошел молодой человек лет двадцати восьми.\n^Z
```

Рис. 8.2. *Файл и маркер конца файла*

Другими словами, вы можете использовать выражение наподобие приведенного ниже:

```
while ((ch = getchar()) != EOF)
```

А что происходит, когда вы вводите не файл, а данные с клавиатуры? Большая часть систем (но далеко не все) обладают возможностью имитировать условия конца файла при вводе с клавиатуры. Зная об этом, вы можете переписать программу echo базового ввода и эхо-вывода, как показано в листинге 8.2.

Листинг 8.2. Программа echo_eof.c

```
/* echo_eof.c -- повторяет ввод до момента достижения конца файла */
#include <stdio.h>
int main(void)
{
   int ch;
   while ((ch = getchar()) != EOF)
      putchar(ch);
   return 0;
}
```

Обратите внимание на следующие моменты:

- Нет необходимости отлавливать EOF, поскольку заголовочный файл stdio.h берет на себя эту функцию.
- Вам не нужно беспокоиться о фактическом значении маркера EOF, поскольку оператор #define в файле stdio.h позволяет использовать символьное представление маркера EOF. Вам не нужно писать программный код, которые требуют знания конкретного значения EOF. Переменная ch поменяла тип с char на int, поскольку переменные типа char могут быть представлены целыми числами без знака в диапазоне от 0 до 255, в то же время EOF может иметь числовое значение -1. Такое значение недопустимо для переменной типа char без знака, но вполне допустимо для типа int. К счастью, функция getchar() сама имеет тип int, следовательно, она может читать символ EOF. Реализации, которые используют тип char со знаком, могут обойтись объявлением переменной ch типа char, но гораздо лучше воспользоваться более общей формой.
- Тот факт, что переменная ch есть целое число, никак не отражается на функции putchar(). На печать она выводит его символьный эквивалент.
- Используя эту программу применительно к вводу с клавиатуры, вам каким-то образом нужно ввести символ EOF. Разумеется, вы не можете просто ввести буквы *E O F*, в то же время вы не можете напечатать -1. (Ввод с клавиатуры -1 представляет собой ввод двух символов: дефиса и цифры 1.) Вместо этого, вы должны найти то, что требует ваша система. В большинстве систем на базе Unix, например, нажатие клавиш <Ctrl+D> в начале строки вызывает передачу сигнала конца файла. Многие микрокомпьютерные системы распознают комбинацию клавиш <Ctrl+Z> в начале строки как сигнал конца файла, некоторые системы интерпретируют комбинацию <Ctrl+Z> в любом месте строки как сигнал конца файла.

Ниже показан пример применения буферизованного ввода в программе echo_eof.c под управлением Unix:

```
У него не было даже пальто.
У него не было даже пальто.
В город молодой человек вошел в зеленом в талию костюме.
В город молодой человек вошел в зеленом в талию костюме.
      И. Ильф, Е. Петров
      И. Ильф, Е. Петров
[Ctrl+D]
```

Каждый раз, когда вы нажимаете клавишу <Enter>, символы, хранящиеся в буфере, подвергаются обработке, а копия строки выводится на печать. Это продолжается до тех пор, пока вы не смоделируете конец файла. На персональном компьютере для этой цели вы можете нажать комбинацию <Ctrl+Z>.

Остановимся на минутку и задумаемся о возможностях программы echo_eof.c. Она воспроизводит на экране любые входные данные, какие вы ей предоставите. Предположим, что вы каким-то образом предоставили в ее распоряжение файл. Затем она выводит на экран содержимое этого файла и останавливается, когда достигает конца файла, обнаружив EOF. Предположим, что вместо этого вы нашли способ направить выходные данные этой программы в файл. Затем вы можете ввести данные с клавиатуры и использовать программу echo_eof.c, чтобы сохранить в файле то, что вы ввели с клавиатуры. Предположим, что вы можете сделать и то и другое одновременно: направить входные данные из одного файла в программу echo_eof.c и послать выходные данные в другой файл. После этого вы можете воспользоваться программой echo_eof.c для копирования файлов. Эта небольшая программа может применяться для просмотра содержимого файлов, для создания новых файлов и для снятия копий существующих файлов, что довольно неплохо для такой скромной программы! Ее основой является умение управлять потоками входных и выходных данных, но это тема последующих разделов.

Эмуляция EOF и графические интерфейсы

Идея эмуляции символа EOF возникла в среде командной строки, использующей текстовый интерфейс. В среде такого рода пользователь взаимодействует с программой посредством нажатия клавиш, а операционная система генерирует сигнал EOF. Некоторые практические методы не очень хорошо переносятся в среды графических интерфейсов, подобные Windows и Macintosh, в которых реализованы более сложные пользовательские интерфейсы, включающие перемещение мыши и щелчки на кнопках. Поведение программы, сталкивающейся с эмуляцией EOF, зависит от компилятора и типа проекта. Например, одновременное нажатие клавиш <Ctrl+Z>, в зависимости от конкретных установок, может завершить ввод данных, а может и завершить выполнение всей программы.

Перенаправление и файлы

Ввод и вывод выполняется с использованием функций, данных и устройств. Рассмотрим, например, программу, `echo_eof.c`. В ней используется функция `getchar()`. Входным устройством (по предположению) является клавиатура, а поток входных данных состоит из отдельных символов. Предположим, что вы хотите сохранить ту же входную функцию и тот же тип данных, но хотите изменить источник, из которого программа черпает данные. При этом возникает вполне резонный вопрос: "Как программа узнает, откуда нужно получить входные данные?"

По умолчанию программа на C, использующая стандартный пакет ввода-вывода, рассматривает свое устройство ввода-вывода как источник входных данных. Это поток входных данных, идентифицированный выше как `stdin`. Его можно рассматривать как обычный способ считывания данных в компьютер. Им могут быть такие старомодные устройства, как магнитная лента, перфокарты или (далее мы будем подразумевать именно этот вариант) ваша клавиатура либо же современная технология наподобие голосового ввода. Однако в современных компьютерах это настраиваемое инструментальное средство, и вы можете обнаружить входные данные в различных средах и на различных носителях. В частности, вы можете потребовать от программы извлекать входные данные из файла, а не вводить их с клавиатуры.

Существуют два способа заставить программу работать с файлами. Один из них заключается в прямом использовании специальных функций, которые открывают, закрывают, читают, записывают файлы и тому подобное. Изучение этого метода мы оставим до главы 13. Второй способ предполагает использование программы, предназначенной для работы с клавиатурой и экраном, но при этом входные и выходные данные *перенаправляются* в различные каналы, например, в файл и из файла. Другими словами, вы переадресуете поток `stdin` в файл. Программа `getchar()` продолжает получать данные из потока, ее совершенно не интересует, откуда поток получает данные. Такой подход (перенаправление) в некоторых аспектах является более ограниченным, чем первый подход, однако им гораздо проще пользоваться, при этом он позволяет ознакомиться с распространенными методами обработки файлов.

Одна из главных проблем перенаправления состоит в том, что она связана с операционной системой, но не с языком C. Однако многие среды языка C, включая операционные системы Unix, Linux и MS-DOS (версии 2.0 и более поздние), поддерживают перенаправление, а некоторые реализации языка C моделируют свойство перенаправления в системах, где оно отсутствует. Мы рассмотрим перенаправление в средах Unix, Linux и DOS.

Перенаправление в Unix, Linux и DOS

Операционные системы Unix, Linux и текущие версии DOS позволяют выполнять перенаправление ввода и вывода. Перенаправление ввода дает вашей программе возможность использовать для ввода файл вместо клавиатуры, а перенаправление вывода — применять для вывода файл вместо экрана.

Перенаправление ввода

Предположим, что вы откомпилировали программу `echo_eof.c` и поместили исполняемую версию в файл, получивший имя `echo_eof` (или `echo_eof.exe` в системах DOS). Чтобы выполнить программу, введите имя файла:

```
echo_eof
```

Эта программа выполняется так, как описано выше, получая входные данные с клавиатуры. Теперь предположим, что вы хотите использовать эту программу применительно к текстовому файлу с именем `words`. *Текстовый файл* — это файл, содержащий текст, иначе говоря, файл, в котором данные хранятся в виде символов, воспринимаемых человеком. Например, это может быть реферат или программа на языке C. Файл, содержащий инструкции на машинном языке, например, файл с исполняемой версией программы, не относится к категории текстовых. Поскольку программа работает с символами, она должна использоваться с текстовыми файлами. В этом случае вместо ранее указанной команды потребуется ввести следующую команду:

```
echo_eof < words
```

Здесь символ < является операцией перенаправления в операционных системах Unix и Linux (а также в DOS). При этом устанавливается связь между файлом `words` и потоком `stdin`, которая обеспечивает перекачку содержимого файла в программу `echo_eof`.

Программа `echo_eof` сама по себе не знает (или не желает знать), что входные данные поступают из файла, а не с клавиатуры. Все, что она знает, это как поступает в нее поток символов, благодаря чему она осуществляет их посимвольное считывание и отображение, пока не будет достигнут конец файла. Поскольку язык C рассматривает файлы и устройства ввода-вывода как эквивалентные категории, то файл теперь становится *устройством ввода-вывода*. Попробуйте им воспользоваться!

Дополнительные сведения о перенаправлении

При работе в системах Unix, Linux и DOS пробелы с обеих сторон знака < не обязательны. Некоторые системы, такие как AmigaDOS, поддерживают перенаправление, но не допускают употребления пробелов между символом перенаправления и именем файла.

Ниже приводится пример выходных данных программы `echo_eof`, примененной к конкретному текстовому файлу, при этом знак `$` есть одно из стандартных приглашений операционных системы Unix и Linux. В операционной системе DOS, скорее всего, приглашение будет выглядеть как `A>` или `C>`.

```
$ echo_eof < words
Пешеходов надо любить.
Пешеходы составляют большую часть человечества.
Мало того — лучшую его часть.
Пешеходы создали мир.
$
```

Итак, перейдем к сути дела.

Перенаправление вывода

Теперь предположим, что вы хотите, чтобы программа echo_eof пересылала ваши входные данные с клавиатуры в файл с именем mywords. В этом случае потребуется ввести приведенную ниже команду и начать ввод с клавиатуры:

```
echo_eof > mywords
```

Знак > представляет операцию перенаправления. Он приводит к созданию нового файла с именем mywords, который вы можете использовать в своих целях, а затем переадресовать выходные данные программы echo_eof (то есть копии символов, введенные с клавиатуры) в этот файл. Перенаправление переназначает поток stdout с устройства отображения (ваш экран) в файл mywords. Если файл с именем mywords уже существует, обычно он удаляется и заменяется новым содержимым. (Многие операционные системы, однако, предлагают возможности защиты существующих файлов, объявляя их файлами только для чтения.) Все, что появляется на вашем экране — это те буквы, которые вы вводите с клавиатуры, а их копии поступают в файл. Чтобы завершить программу, нажмите комбинацию клавиш <Ctrl+D> (Unix) или <Ctrl+Z> (DOS) в начале строки. Проверьте это. Если не можете придумать, что вводить с клавиатуры, повторите приведенный ниже пример. В нем мы присутствует приглашение $ системы Unix. Напоминаем, что ввод каждой строки заканчивается нажатием клавиши <Enter>, в результате чего содержимое буфера передается в программу.

```
$ echo_eof > mywords
У вас не должно быть никаких проблем при запоминании, что делает
тот или иной оператор перенаправления. Запомните только, что оператор
указывает направление потока информации. Представьте себе, что
это воронка.
[Ctrl+D]
$
```

После того, как <Ctrl+D> или <Ctrl+Z> будут обработаны, программа завершает работу, и приглашение операционной системы возвращается на экран. Была ли выполнена программа? Команда ls системы Unix или команда dir операционной системы DOS, которые выводят на экран список имен файлов, должны подтвердить существование файла mywords. Вы можете также воспользоваться командой cat в Unix и Linux или type в DOS для проверки содержимого либо снова запустить программу echo_eof, на этот раз перенаправив файл в программу:

```
$ echo_eof < mywords
```

Комбинированное перенаправление

Теперь предположим, что вы хотите создать копию файла mywords и присвоить ей имя savewords. Для этого воспользуйтесь следующей командой:

```
echo_eof < mywords > savewords
```

и дело сделано. Можно также выдать и приведенную ниже команду, поскольку порядок выполнения операций перенаправления не имеет значения:

```
echo_eof > savewords < mywords
```

Соблюдайте осторожность — не используйте один и тот же файл как для ввода, так и для вывода в рамках одной и той же команды.

```
echo_eof < mywords > mywords....<--НЕПРАВИЛЬНО
```

Это объясняется тем, что команда `> mywords` усекает исходный файл `mywords` до нулевой длины, прежде чем он будет использован в качестве входного файла.

Подводя итоги, можно отметить, что существуют правила, регламентирующие использование указанных выше двух операций перенаправления (< и >) в операционных системах Unix, Linux и DOS:

- Операция перенаправления ассоциирует *исполняемую* программу (включая стандартные команды операционных систем) с файлом данных. Она не может быть использована для соединения одного файла данных с другим, а также для соединения одной программы с другой.
- Входные данные можно брать только из одного файла, но не из нескольких, это справедливо и в отношении выходных данных.
- Обычно пробелы между именами и операциями не обязательны, бывают эпизодические исключения, когда используются специальные символы, имеющие особый смысл для командных процессоров Unix, Linux или DOS. Можно было бы, например, применить команду `echo_eof < words`.

Выше вы ознакомились с примерами правильно составленных команд. Ниже показано несколько примеров неправильно сформулированных команд, в которых `addup` и `count` являются исполняемыми программами, а `fish` и `beets` — текстовыми файлами:

```
fish > beets          ← Нарушается первое правило
addup < count         ← Нарушается первое правило
addup < fish < beets  ← Нарушается второе правило
count > beets fish    ← Нарушается второе правило
```

В операционных системах Unix, Linux и DOS реализована также операция >>, которая предоставляет возможность добавлять данные в конец существующего файла, а также операция канала (`|`), позволяющая подключать вывод одной программы к вводу другой программы. За дополнительной информацией по всем этим операциям обращайтесь к книгам, посвященным операционной системе Unix.

Комментарии

Перенаправление позволяет использовать с файлами программы, предназначенные для обработки ввода с клавиатуры. В этих условиях программа должна осуществлять проверку на предмет конца файла. Например, в главе 7 рассматривалась программа, подсчитывающая количество слов для появления первого символа `|`. Поменяйте тип `char` переменной `ch` на тип `int` и замените `'|'` на `EOF` в проверочном выражении цикла, и вы сможете пользоваться этой программой для подсчета слов в текстовых файлах.

Перенаправление — это концепция командной строки, поскольку вы задаете ее путем ввода с клавиатуры специальных символов в командной строке. Если вы не используете среду командной строки, у вас все еще остается возможность воспользоваться этой технологией.

Во-первых, в некоторых интегрированных средах имеются пункты меню, позволяющие выполнить перенаправление.

Во-вторых, в средах Windows вы можете открыть окно DOS и запустить исполняемый файл из командной строки. По умолчанию система Microsoft Visual C++ 7.1 помещает исполняемый файл в подкаталог с именем `Debug`. Имя файла будет иметь то же базовое имя, что и имя проекта и расширение `.exe`.

В случае отладчика Codewarrior воспользуйтесь режимом контрольного приложения Win 32 (Win 32 Console App), который присвоит исполняемому файлу имя `Cproj Debug.exe` по умолчанию (где `Cproj` будет заменено именем вашего проекта), и поместите его в папку проекта.

Если перенаправление не работает, можете попытаться заставить программу открыть файл напрямую. Примером может служить программа, показанная в листинге 8.3 и сопровождаемая минимальными пояснениями. Более подробную информацию можно найти в главе 13.

Листинг 8.3. Программа `file_eof.c`

```
// file_eof.c -- открыть файл и отобразить его
#include <stdio.h>
#include <stdlib.h>            // для функции exit()
int main()
{
   int ch;
   FILE * fp;
   char fname[50];             // для запоминания имени файла
   printf("Введите имя файла: ");
   scanf("%s", fname);
   fp = fopen(fname, "r");     // открыть файл для чтения
   if (fp == NULL)             // попытка завершилась неудачей
   {
      printf("Не удается открыть файл. Программа завершена.\n");
      exit(1);                 // выйти из программы
   }
// функция getc(fp) получает символ из открытого файла
   while ((ch = getc(fp)) != EOF)
      putchar(ch);
   fclose(fp);                 // закрыть файл
   return 0;
}
```

Сводка: как перенаправить ввод и вывод

В большинстве систем C вы можете использовать перенаправление либо для всех программ с помощью операционной системы, либо только для программ на C, благодаря возможностям компилятора языка C. Пусть `prog` является именем исполняемой программы и пусть `file1` и `file2` — имена файлов.

Перенаправление вывода в файл: `>prog >file1`

Перенаправление ввода из файла: `<`

```
prog <file2
```

Комбинированное перенаправление:

```
prog <file2 >file1
prog >file1 <file2
```

Обе формы используют `file2` для ввода и `file1` для вывода.

Пробелы:

Некоторые системы требуют наличия пробела слева от знака операции перенаправления и не требуют пробела справа от знака. Другие системы (например, Unix) допускают наличие пробелов с обеих сторон, либо запрещают эти пробелы.

Создание дружественного пользовательского интерфейса

Большинству из нас доводилось писать программы, которыми неудобно было пользоваться. К счастью, язык C предлагает инструментальные средства, способные превратить ввод в более предсказуемый и приятный процесс. К сожалению, изучение этих инструментальных средств на первых порах порождает новые проблемы. Цель данного раздела заключается в том, чтобы провести вас через некоторые из этих проблем к получению более дружественного пользовательского интерфейса, который облегчает задачу ввода интерактивных данных и минимизирует эффект от ошибочного ввода данных.

Работа с буферизованным вводом

Буферизованный ввод часто удобен для пользователя, поскольку обеспечивает возможность редактирования входных данных до передачи их в программу, однако для программиста символьный ввод служит источником дополнительных забот. Проблема, как вы могли убедиться во время изучения приведенных выше примеров, заключается в том, что буферизованный ввод требует нажатия клавиши <Enter> для передачи введенной информации. Это действие пересылает также символ новой строки, который программа должна обработать. Теперь проведем исследования этой и других проблем с помощью программы отгадывания чисел. Вы выбираете число, а компьютер пытается его отгадать. При этом используется достаточно сложный метод, однако основное внимание мы уделим вводу-выводу, а не собственно алгоритму. Начальная версия такой программы показана в листинге 8.4.

Листинг 8.4. Программа `guess.c`

```
/* guess.c -- неэффективное и ошибочное отгадывание числа */
#include <stdio.h>
int main(void)
{
   int guess = 1;

   printf("Выберите целое число в промежутке от 1 до 100. Я попробую отгадать ");
   printf("его.\nНажмите клавишу y, если моя догадка верна и ");
   printf("\n клавишу n в противном случае.\n");
   printf("Вашим числом является %d?\n", guess);
   while (getchar() != 'y')            /* получить ответ, сравнить с y */
      printf("Итак, это будет %d?\n", ++guess);
```

```
   printf("Я знал, что у меня получится!\n");
   return 0;
}
```

Приводим результат выполнения этой учебной программы:

```
Выберите целое число в промежутке от 1 до 100. Я попробую отгадать его.
Нажмите клавишу y, если моя догадка верна и
клавишу n в противном случае.
Вашим числом является 1?
n
Вашим числом является 2?
Вашим числом является 3?
n
Вашим числом является 4?
Вашим числом является 5?
y
Я знал, что у меня получится!
```

Вопреки ожиданиям амбициозного алгоритма, реализованного в программе, мы выбрали небольшое число. Обратите внимание на то, что программа выполняет два предположения каждый раз, когда вы вводите n. Программа читает ответ n и рассматривает его как отрицание того, что было загадано число 1, и при этом считывает символ новой строки как отрицание того факта, что было загадано число 2.

Одно из решений предусматривает использование цикла while для игнорирования остальных символов входной строки, в том числе и символа новой строки. В результате ввод, например, no ("нет") или no way ("ни в коем случае"), можно также интерпретировать аналогично n. Версия программы, представленная в листинге 8.4, интерпретирует no как два ввода. Ниже показан пример цикла, в котором эта проблема решена:

```
while (getchar() != 'y') /* получить ответ, сравнить с y */
{
   printf("Вашим числом является %d?\n", ++guess);
   while (getchar() != '\n')
      continue;          /* пропустить оставшуюся часть входной строки*/
}
```

При использовании этого цикла получается следующий диалог:

```
Выберите целое число в промежутке от 1 до 100. Я попробую отгадать его.
Нажмите клавишу y, если моя догадка верна и
клавишу n в противном случае.
Вашим числом является 1?
n
Вашим числом является 2?
no
Вашим числом является 3?
no sir
Вашим числом является 4?
forget it
Вашим числом является 5?
y
Я знал, что у меня получится!
```

Проблема с символом новой строки решена. В то же время, как борцу за чистоту нравов в программировании, вам вряд ли понравится, что f будет трактоваться так же, как n. Чтобы устранить этот дефект, вы можете воспользоваться оператором if, чтобы отфильтровать другие ответы. Во-первых, добавьте переменную типа char для запоминания ответа:

```
char response;
```

Затем внесите изменения в цикл, чтобы он приобрел следующий вид:

```
while ((response = getchar()) != 'y')  /* получить ответ */
{
   if (response == 'n')
      printf("Вашим числом является %d?\n", ++guess);
   else
      printf("К сожалению, я понимаю только y или n.\n");
   while (getchar() != '\n')
      continue;                  /* пропустить оставшуюся часть строки */
}
```

Теперь ответ программы имеет следующий вид:

```
Выберите целое число в промежутке от 1 до 100. Я попробую отгадать его.
Нажмите клавишу y, если моя догадка верна и
клавишу n в противном случае.
Вашим числом является 1?
n
Вашим числом является 2?
no
Вашим числом является 3?
no sir
Вашим числом является 4?
forget it
К сожалению, я понимаю только y или n.
Вашим числом является 5?
y
Я знал, что у меня получится!
```

Когда вы пишете интерактивные программы, вы должны предусмотреть случаи, когда пользователи могут нарушать инструкции. В таких ситуациях вы должны построить свою программу таким образом, чтобы она помогала пользователям исправлять свои ошибки.

Она уведомляет их о том, где они допустили ошибку, и предоставляет им дополнительный шанс.

Разумеется, вы должны предоставить пользователю четкие инструкции, однако независимо от их четкости и полноты, обязательно найдется кто-то, кто поймет их неправильно, а потом вас же обвинит в том, что вы составили непонятные инструкции.

Смешивание числового и символьного ввода

Предположим, что ваша программа требует ввода символьных данных с помощью функции getchar() и числовых данных с помощью функции scanf(). Каждая из этих функций по отдельности добросовестно выполняет свою задачу, однако их сочетание

работает некорректно. Это объясняется тем, что функция getchar() читает каждый символ, в том числе пробелы, символы табуляции и новой строки, в то время как scanf() при считывании чисел пропускает пробелы, символы табуляции и новой строки.

Чтобы продемонстрировать проблемы, которые при этом возникают, в листинге 8.5 представлена программа, которая считывает символ и два числа в качестве ввода. Затем она печатает символ, используя при этом номер строки и столбца, указанные во входных данных.

Листинг 8.5. Прорамма showchar1.c

```
/* showchar1.c — программа с большой проблемой, связанной с вводом-выводом */
#include <stdio.h>
void display(char cr, int lines, int width);
int main(void)
{
   int ch;                    /* символ, выводимый на печать */
   int rows, cols;            /* количество строк и столбцов */

   printf("Введите символ и два целых числа:\n");
   while ((ch = getchar()) != '\n')
   {
      scanf("%d %d", &rows, &cols);
      display(ch, rows, cols);
      printf("Введите еще один символ и два целых числа;\n");
      printf("введите символ новой строки для завершения программы.\n");
   }
   printf("Программа завершена.\n");

   return 0;
}

void display(char cr, int lines, int width)
{
   int row, col;

   for (row = 1; row <= lines; row++)
   {
      for (col = 1; col <= width; col++)
         putchar(cr);
      putchar('\n');          /* закончить строку и начать новую */
   }
}
```

Обратите внимание на то, что программа читает символ как тип int, чтобы обеспечить проверку на EOF. В то же время она передает символ как тип char в функцию display(). Поскольку тип char меньше, чем тип int, некоторые компиляторы выдают предупреждающие сообщения о возможных ошибках при преобразовании типов. В данном случае, вы можете игнорировать это предупреждение. Программа написана таким образом, что функция main() получает данные, а функция display() выполняет печать. Рассмотрим результат выполнения программы, что позволит определить, в чем суть проблемы:

```
Введите символ и два целых числа:
c 2 3
ccc
ccc
Введите еще один символ и два целых числа;
введите символ новой строки для завершения программы.
Программа завершена.
```

Сначала программа работает хорошо. Вы вводите `c 2 3`, и она печатает два ряда по три символа *c*, как и ожидалось. Затем программа обращается к вам с предложением ввести следующий набор данных и завершает работу, прежде чем вы сможете ответить! Что случилось? И опять все дело в символе новой строки, на этот раз он следует непосредственно за цифрой 3 в первой строке ввода. Функция `scanf()` оставляет его во входной очереди. В отличие от `scanf()`, функция `getchar()` не пропускает символов новой строки, поэтому такой символ читается функцией `getchar()` на следующей итерации цикла, прежде чем вы получите возможность ввести что-либо еще. Это значение присваивается переменной `ch`, а если `ch` получает символ новой строки в качестве своего значения, удовлетворяется условие выхода из цикла.

Чтобы устранить эту проблему, программа должна пропускать любые символы новой строки или пробелы между последним числом, набранным в одном цикле ввода, и символом, идущим первым в следующей строке. Кроме того, было бы неплохо, если выполнение программы можно было бы прекратить на стадии выполнения функции `scanf()` в дополнение к проверке функции `getchar()`. Все это реализовано в следующей версии программы, текст которой приведен в листинге 8.6.

Листинг 8.6. Программа `showchar2.c`

```
/* showchar2.c -- печатает символы в строках и столбцах */
#include <stdio.h>
void display(char cr, int lines, int width);
int main(void)
{
   int ch;                    /* символ, выводимый на печать */
   int rows, cols;            /* количество строк и столбцов */

   printf("Введите символ и два целых числа:\n");

   while ((ch = getchar()) != '\n')
   {
      if (scanf("%d %d",&rows, &cols) != 2)
         break;

      display(ch, rows, cols);

      while (getchar() != '\n')
         continue;

      printf("Введите еще один символ и два целых числа;\n");
      printf("введите символ новой строки для завершения программы.\n");
   }
   printf("Программа завершена.\n");

   return 0;
}
```

```
void display(char cr, int lines, int width)
{
   int row, col;
   for (row = 1; row <= lines; row++)
   {
      for (col = 1; col <= width; col++)
         putchar(cr);
      putchar('\n');           /* закончить строку и начать новую */
   }
}
```

Оператор while заставляет программу пропускать все символы, следующие за вводом scanf(), включая символ новой строки. Это подготавливает цикл для чтения первого символа в начале следующей строки. Это означает, что вы можете вводить данные без каких-либо ограничений:

```
Введите символ и два целых числа:
c 1 2
cc
Введите еще один символ и два целых числа;
введите символ новой строки для завершения программы.
! 3 6
!!!!!!
!!!!!!
!!!!!!
Введите еще один символ и два целых числа;
введите символ новой строки завершения программы.

Программа завершена.
```

Используя оператор if совместно с оператором break, вы завершаете выполнение программы, если значение, возвращаемое функцией scanf() не равно 2. Это имеет место в тех случаях, когда одно или оба входных значения не являются целыми числами или если встретился символ конца файла.

Проверка допустимости ввода

На практике пользователи программ не всегда соблюдают требования инструкций, и вполне возможно расхождение между тем, что программа ожидает, и тем, что фактически получает. В этих условиях могут возникнуть проблемы с выполнением программы. В то же время, осознавая возможность возникновения ошибок, вы можете включить в свою программу дополнительный код, помогающий избежать этих ошибок.

Предположим, например, что вы разрабатываете программу, которая приглашает пользователя ввести неотрицательное целое число. Другим видом ошибки является ввод значений, недопустимых для конкретной задачи, которую решает программа.

Предположим, например, что у вас имеется цикл, выполняющий обработку неотрицательных чисел. Одним из видов ошибок, которые может совершить пользователь в этом случае, является ввод отрицательного числа. Вы можете воспользоваться условным выражением и провести соответствующую проверку:

```
int n;
scanf("%d", &n);      // получить первое значение
while (n >= 0)        // обнаружить значение, выходящее за пределы диапазона
{
   // обработать n
   scanf("%d", &n);   // получить следующее значение
}
```

Еще одна потенциальная ошибка состоит в том, что при вводе пользователь может выбрать неправильный тип входного значения, например, ввести символ q. Один из способов обнаружения такого вида ошибок предусматривает проверку значений, возвращаемых функцией scanf(). Эта функция, как вы знаете, возвращает количество элементов, которые были успешно прочитаны, поэтому выражение

```
scanf("%d", &n) == 1
```

принимает истинное значение, только когда пользователь вводит целое число. Это требует внесения в код следующего изменения:

```
int n;
while (scanf("%d", &n) == 1 && n >= 0)
{
   // обработка n
}
```

Другими словами, условие цикла while звучит так: "если вводится целое значение и это целое значение больше нуля". Последняя версия программы прерывает ввод, когда пользователь вводит значение неправильного типа. В то же время вы можете сделать программу несколько более дружественной по отношению к пользователю и предоставить ему возможность исправить ошибку при вводе и ввести правильное значение. В этом случае вы, прежде всего, должны удалить те входные данные, которые стали источником проблем. Если функция scanf() не смогла прочитать ввод, она оставляет его во входной очереди. В этом случае тот факт, что ввод представляет собой поток символов, благоприятствует успеху дела, поскольку вы можете воспользоваться функцией getchar() для посимвольного чтения входных данных. Вы можете реализовать все эти идеи в рамках функции, подобной представленной ниже:

```
int get_int(void)
{
   int input;
   char ch;
   while (scanf("%d", &input) != 1)
   {
      while ((ch = getchar()) != '\n')
         putchar(ch);           // освобождение от неправильного символа
      printf(" не является целочисленным.\nПожалуйста, введите ");
      printf("целое число, такое как 25, -178 или 3: ");
   }
   return input;
}
```

Эта функция предпринимает попытку прочитать значение типа int в переменную input. Если ей это не удастся, функция входит в тело внешнего цикла while.

Затем внутренний цикл while выполняет посимвольное чтение неправильного ввода. Обратите внимание, что эта функция предпочитает удалить все, что осталось во входной строке. Другими возможными вариантами остаются удаление следующего символа или слова. Затем функция приглашает пользователя осуществить еще одну попытку ввода. Внешний цикл продолжает выполняться до тех пор, пока пользователь не завершит успешно ввод целого числа и функция scanf() вернет значение 1.

После того, как пользователь устранит все препятствия для ввода целых чисел, программа может выполнить просмотр введенных данных на предмет их допустимости. Рассмотрим пример, по условиям которого требуется, чтобы пользователь ввел значения верхнего и нижнего пределов, определяющих диапазон допустимых значений. В этом случае вы, возможно, захотите, чтобы программа проверяла, чтобы первое значение не было больше второго (обычно при задании диапазонов полагают, что первое значение меньше второго). Может быть, придется проверить, что обе величины находятся в приемлемых пределах. Например, поиск в архиве, возможно, не следует проводить, если значения даты, которые принимают соответствующие переменные, меньше 1958 или больше 2005. Эту проверку также должна выполнять специальная функция.

Ниже предлагается одна из возможностей, рассматриваемая далее функция предполагает, что в программу был включен заголовочный файл stdbool.h. Если в вашей системе не используется тип _Bool, вы можете воспользоваться типом int вместо bool, 1 вместо true и 0 вместо false. Обратите внимание, что эта функция возвращает значение true, если ввод был выполнен неправильно, откуда, впрочем, и следует ее название bad_limits():

```
bool bad_limits(int begin, int end, int low, int high)
{
   bool not_good = false;
   if (begin > end)
   {
      printf("%d не меньше чем %d.\n", begin, end);
      not_good = true;
   }
   if (begin < low || end < low)
   {
      printf("Значения должны быть равными %d или больше.\n", low);
      not_good = true;
   }
   if (begin > high || end > high)
   {
      printf("Значения должны быть равными %d или меньше.\n", high);
      not_good = true;
   }
      return not_good;
}
```

Эти две функции используются в программе, показанной в листинге 8.7, для того, чтобы снабжать целыми числами арифметическую функцию, которая вычисляет сумму квадратов всех целых чисел из заданного диапазона. Программа ограничивает верхние и нижние границы диапазона, соответственно, значениями 1000 и –1000.

Листинг 8.7. Программа `checking.c`

```
/* checking.c -- проверка допустимости ввода */
#include <stdio.h>
#include <stdbool.h>
// проверка, является ли вводимое значение целым числом
int get_int(void);
// проверка, являются ли границы диапазона допустимыми
bool bad_limits(int begin, int end, int low, int high);
// вычисление суммы квадратов целых чисел
// от a до b
double sum_squares(int a, int b);
int main(void)
{
   const int MIN = -1000;     // нижняя граница диапазона
   const int MAX = +1000;     // верхняя граница диапазона
   int start;                 // начало диапазона
   int stop;                  // конец диапазона
   double answer;

   printf("Эта программа вычисляет сумму квадратов "
         "целых чисел в заданном диапазоне.\nНижняя граница не должна "
         "быть меньше -1000, \na верхняя - "
         "больше +1000.\nВведите значения границ диапазона (введите 0 для "
         "обеих границ для завершения программы):\nнижняя граница: ");
   start = get_int();
   printf("верхняя граница: ");
   stop = get_int();
   while (start !=0 || stop != 0)
   {
      if (bad_limits(start, stop, MIN, MAX))
         printf("Пожалуйста, повторите попытку.\n");
      else
      {
         answer = sum_squares(start, stop);
         printf("Сумма квадратов целых чисел ");
         printf("от %d до %d равна %g\n", start, stop, answer);
      }
      printf("Введите значения границ диапазона (введите 0 для "
      "обеих границ для завершения программы):\n");
      printf("нижняя граница: ");
      start = get_int();
      printf("верхняя граница: ");
      stop = get_int();
   }
   printf("Программа завершена.\n");
   return 0;
}
int get_int(void)
{
   int input;
   char ch;
```

```
   while (scanf("%d", &input) != 1)
   {
      while ((ch = getchar()) != '\n')
         putchar(ch);        // удаление неправильных входных данных
      printf(" не является целочисленным.\nПожалуйста, введите ");
      printf("целое число, такое как 25, -178 или 3: ");
   }
   return input;
}
double sum_squares(int a, int b)
{
   double total = 0;
   int i;
   for (i = a; i <= b; i++)
      total += i * i;
   return total;
}
bool bad_limits(int begin, int end, int low, int high)
{
   bool not_good = false;
   if (begin > end)
   {
      printf("%d не меньше %d.\n", begin, end);
      not_good = true;
   }
   if (begin < low || end < low)
   {
      printf("Значения должны быть равными %d или больше.\n", low);
      not_good = true;
   }
   if (begin > high || end > high)
   {
      printf("Значения должны быть равными %d или меньше.\n", high);
      not_good = true;
   }
   return not_good;
}
```

Ниже показан пример выполнения этой программы:

```
Эта программа вычисляет сумму квадратов целых чисел в заданном
диапазоне.
Нижняя граница не должна быть меньше -1000,
а верхняя - больше +1000.
Введите значения границ диапазона (введите 0 для обеих границ для
завершения программы):
нижняя граница: low
low не является целочисленным.
Пожалуйста, введите целое число, такое как 25, -178 или 3: 3
верхняя граница: a big number
a big number не является целочисленным.
Пожалуйста, введите целое число, такое как 25, -178 или 3: 12
```

```
Сумма квадратов целых чисел от 3 до 12 равна 645
Введите значения границ диапазона (введите 0 для обеих границ для
завершения программы):
нижняя граница: 80
верхняя граница: 10
80 не меньше 10.
Пожалуйста, повторите попытку.
Введите значения границ диапазона (введите 0 для обеих границ для
завершения программы):
нижняя граница: 0
верхняя граница: 0
Программа завершена.
```

Анализ программы

Вычислительное ядро (функция `sum_squares()`) программы `checking.c` занимает немного места, в то же время поддержка проверки допустимости ввода использует его более интенсивно, чем в примерах, приведенных выше. Рассмотрим некоторые его элементы, обратив внимание в первую очередь на общую структуру программы.

Мы придерживались модульного подхода, используя отдельные функции (модули) для проверки допустимости ввода и для управления отображением данных. Чем больше программа, тем важнее использование модульного программирования.

Функция `main()` управляет потоком, распределяя задачи на выполнение другими функциями. Она использует фукнцию `get_int()`, чтобы получать значения, цикл `while` для их обработки, функцию `badlimits()` для проверки допустимости входных данных и функцию `sum_squares()`, чтобы производит фактические вычисления:

```
start = get_int();
printf("верхняя граница: ");
stop = get_int();
while (start !=0 || stop != 0)
{
   if (bad_limits(start, stop, MIN, MAX))
      printf("Пожалуйста, повторите попытку.\n");
   else
   {
      answer = sum_squares(start, stop);
      printf("Сумма квадратов целых чисел ");
      printf("от %d до %d равна %g\n", start, stop, answer);
   }
   printf("Введите значения границ диапазона (введите 0 для "
   "обеих границ для завершения программы):\n");
   printf("нижняя граница: ");
   start = get_int();
   printf("верхняя граница: ");
   stop = get_int();
}
```

Поток ввода и числа

При написании программного кода, который выполняет обработку неправильного ввода, такого как в листинге 8.7, вы должны иметь четкое представление от том, как осуществляется ввод в языке C. Рассмотрим следующий ввод:

```
is  28 12.4
```

Мы воспринимаем этот ввод как строку символов, за которой идет целое число, а за ним следует значение с плавающей запятой. В то же время программа на C воспринимает эту последовательность как поток байтов. Первый байт есть символьный код буквы `i`, второй байт — символьный код буквы `s`, третий байт — символьный код пробела, четвертый байт — символьный код цифры 2 и так далее. Следовательно, если функция `get_int()` столкнется с этой строкой, представленный ниже код читает и отвергает всю строку, включая числа, которые в строке являются такими же символами, как и остальные:

```
while ((ch = getchar()) != '\n')
   putchar(ch);          // удаление неправильного ввода
```

И хотя входной поток состоит из символов, функция `scanf()` может преобразовать их в числовые значения, если вы попросите ее сделать это. Например, рассмотрим следующий ввод:

```
42
```

Если вы используете функцию `scanf()` со спецификатором `%c`, она читает только символ `4` и запоминает его в переменной типа `char`. Если вы указываете спецификатор `%s`, она читает два символа, символ `4` и символ `2`, и сохраняет их в строке символов. Если вы используете спецификатор `%d`, функция `scanf()` читает эти же два символа, однако затем далее она посчитает, что целочисленное значение, соответствующее этим символам, есть $4 \times 10 + 2$, или 42. Затем она запоминает целочисленное двоичное представление этого значения в переменной типа `int`. Если вы задаете спецификатор `%f`, функция `scanf()` читает два символа, выполняет вычисления, показывающие, что они соответствуют числовому значению 42, выражает это значение во внутреннем представлении в виде значения с плавающей запятой и сохраняет этот результат в переменной `float`.

Короче говоря, ввод состоит из символов, в то же время функция `scanf()` может преобразовать входные данные в целое значение или значение с плавающей запятой. Прибегая к помощи таких спецификаторов, как `%d` или `%f`, мы можем ограничить количество приемлемых типов при вводе, в то же время функции `getchar()` и `scanf()`, использующие спецификатор `%c`, могут принимать любые символы.

Просмотр меню

Многие компьютерные программы используют меню как часть пользовательского интерфейса. Меню делают программу более удобной, в то же время они ставят определенные проблемы перед программистом. Посмотрим, в чем они состоят.

Меню предлагают пользователю ответы на выбор. Ниже показан гипотетический пример:

```
Введите любую букву на ваш выбор:
с. совет            з. звонок
п. подсчет          в. выход
```

В идеальном случае пользователь вводит одну из представленных в примере букв, и программа функционирует в соответствии с этим выбором. Будучи программистом, вы хотите, чтобы этот процесс протекал как можно более гладко. Первая цель состоит в том, чтобы программа работала гладко, когда пользователь придерживается инструкций по ее эксплуатации. Вторая цель предполагает устойчивую работу программы и в тех случаях, когда пользователь нарушает эти инструкции. Как нетрудно догадаться, второй цели достичь гораздо труднее, чем первой, поскольку трудно предусмотреть все возможные нарушения, с которыми может столкнуться программа на протяжении всего срока службы.

Задачи

Будем конкретными и рассмотрим задачи, которые должна решать программа с реализованным меню. Прежде всего, необходим ответ пользователя, получив который, программа выберет совокупность действий, соответствующих этому ответу. Наряду с этим, программа должна обеспечить способ возврата меню в состояние, пригодное для последующего выбора вариантов. Оператор выбора языка C представляет собой естественный механизм выбора действий, поскольку каждый выбор пользователя должен соответствовать конкретной метке `case`. Вы можете воспользоваться оператором `while`, чтобы обеспечить многократный доступ. С помощью псевдокода можно описать этот процесс следующим образом:

```
сделать выбор
пока не выбрана буква 'в'
   переключиться на нужный вариант и выполнить его
   сделать следующий выбор
```

Обеспечение устойчивого выполнения программ

Задача устойчивого (без сбоев) выполнения программы (корректное функционирование программы при обработке безошибочного ввода и при обработке входных данных, содержащих ошибки) выходит на передний план, когда вы принимаете решение, каким образом реализовать представленный выше псевдокод. Например, одно из действий, которое вы можете предпринять — это исключить на стадии "сделать выбор" неправильные отклики с тем, чтобы оператору `switch` передавались только правильные отклики. Это предполагает представление процесса ввода в виде функции, которая может возвращать только правильные отклики. Комбинация этого с циклом `while` и оператором `switch` порождает следующую структуру программы:

```
#include <stdio.h>
char get_choice(void);
void count(void);
int main(void)
{
   int choice;
```

```
while ( (choice = get_choice()) != 'в')
   {
   switch (choice)
   {
      case 'с' : printf("Покупайте дешево, продавайте дорого.\n");
                 break;
      case 'з' : putchar('\a'); /* ANSI */
                 break;
      case 'п' : count();
                 break;
      default :  printf("Программная ошибка!\n");
                 break;
   }
   }
   return 0;
}
```

Функция `get_choice()` определена таким образом, что она может возвращать только значения `'с'`, `'з'`, `'п'` и `'в'`. Вы используете ее примерно так же, как и функцию `getchar()` — она получает конкретное значение и сравнивает его с символом завершения программы (в рассматриваемом случае это `'в'`). Мы сохранили выбор из меню на достаточно простом уроне с тем, чтобы уделить основное внимание структуре программы; далее мы рассмотрим функцию `count()`. Вариант `default` удобно использовать во время отладки. Если функция `get_choice()` не сможет ограничить свои возвращаемые значения до заранее заданных, вариант `default` дает вам понять, что происходит нечто, вызывающее опасения.

Функция `get_choice()`

В рассматриваемом случае одна из возможных структур этой функции имеет в псевдокоде следующий вид:

```
показать возможные варианты
получить отклик
пока отклик неприемлем
   приглашение на ввод дальнейших откликов
   получить отклик
```

Ниже показана простая, и в то же время неудобная реализация:

```
char get_choice(void)
{
   int ch;
   printf("Введите букву выбранного варианта:\n");
   printf("с. совет           з. звонок\n");
   printf("п. подсчет         в. выход\n");
   ch = getchar();
   while ( ch != 'с' && ch != 'з' && ch != 'п' && ch != 'в')
   {
      printf("Выберите с, з, п или в.\n");
      ch = getchar();
   }
   return ch;
}
```

Проблема заключается в том, что в условиях буферизованного ввода каждый символ новой строки, порожденный нажатием <Enter>, рассматривается как ошибочный отклик. Чтобы сделать программный интерфейс устойчивым, данная функция должна пропускать символы новой строки.

Существует несколько способов сделать это. Один из них предусматривает замену функции `getchar()` новой функцией с именем `get_first()`, которая считывает первый символ строки и игнорирует все остальные. Преимущество такого метода состоит в том, что он трактует введенную строку, например, `спа`, как просто символ `с`, но не рассматривает ее как правильный вариант `с`, за которым следует еще один правильный вариант в виде буквы `п`, означающей `подсчет`. С учетом всего этого, функцию ввода можно переписать в следующем виде:

```
char get_choice(void)
{
   int ch;

   printf("Введите букву выбранного варианта:\n");
   printf("с. совет           з. звонок\n");
   printf("п. подсчет         в. выход\n");
   ch = get_first();
   while ( ch != 'с' && ch != 'з' && ch != 'п' && ch != 'в')
   {
      printf("Выберите с, з, п или в.\n");
      ch = get_first();
   }
   return ch;
}

char get_first(void)
{
   int ch;

   ch = getchar();          /* считывание следующего символа */
   while (getchar() != '\n')
      continue;             /* пропустить остальную часть строки */
   return ch;
}
```

Смешивание символьного и числового ввода

Создание меню является еще одной иллюстрацией того, какие проблемы порождает смешивание ввода символов и чисел. Предположим, например, что функция `count()` (выбор `п`) имеет следующий вид:

```
void count(void)
{
   int n,i;
   printf("Считать до какого предела? Введите целое число:\n");
   scanf("%d", &n);
   for (i = 1; i <= n; i++)
      printf("%d\n", i);
}
```

Если в ответ вы введете **3**, функция `scanf()` прочтет `3` и оставит символ новой строки в качестве следующего символа во входной очереди. Результат следующего вызова `get_choice()` будет состоять в том, что функция `get_first()` вернет этот символ новой строки, что приведет к нежелательному поведению. Один из способов уладить эту проблему заключается в том, чтобы переписать функцию `get_first()` таким образом, чтобы она возвращала следующий непробельный символ, а не любой следующий встреченный ею символ. Мы оставим эту задачу в качестве упражнения для самостоятельного выполнения. Второй подход заставить саму функцию `count()` следить за порядком и удалять символы новой строки. Именно этот подход применяется в следующем примере:

```
void count(void)
{
   int n,i;
   printf("Считать до какого предела? Введите целое число:\n");
   n = get_int();
   for (i = 1; i <= n; i++)
      printf("%d\n", i);
   while ( getchar() != '\n')
      continue;
}
```

Эта функция использует также функцию `get_int()` из листинга 8.7; вспомните, что она выполняет проверку допустимости ввода и предоставляет пользователю возможность повторного ввода данных. В листинге 8.8 показан окончательный вариант программы, в которой используется меню.

Листинг 8.8. Программа `menuette.c`

```
/* menuette.c -- технология меню */
#include <stdio.h>
char get_choice(void);
char get_first(void);
int get_int(void);
void count(void);
int main(void)
{
   int choice;
   void count(void);
   while ( (choice = get_choice()) != 'в')
   {
      switch (choice)
      {
         case 'с' : printf("Покупайте дешево, продавайте дорого.\n");
                    break;
         case 'з' : putchar('\a'); /* ANSI */
                    break;
         case 'п' : count();
                    break;
         default :  printf("Программная ошибка!\n");
                    break;
      }
   }
```

```
   printf("Всего хорошего.\n");
   return 0;
}
void count(void)
{
   int n,i;
   printf("Считать до какого предела? Введите целое число:\n");
   n = get_int();
   for (i = 1; i <= n; i++)
      printf("%d\n", i);
   while ( getchar() != '\n')
      continue;
}
char get_choice(void)
{
   int ch;
   printf("Введите букву выбранного варианта:\n");
   printf("с. совет           з. звонок\n");
   printf("п. подсчет         в. выход\n");
   ch = get_first();
   while ( ch != 'с' && ch != 'з' && ch != 'п' && ch != 'в')
   {
      printf("Выберите с, з, п или в.\n");
      ch = get_first();
   }
   return ch;
}
char get_first(void)
{
   int ch;
   ch = getchar();
   while (getchar() != '\n')
      continue;
   return ch;
}
int get_int(void)
{
   int input;
   char ch;
   while (scanf("%d", &input) != 1)
   {
      while ((ch = getchar()) != '\n')
         putchar(ch);               // удалить неправильный вывод
      printf(" не является целочисленным.\nПожалуйста, введите ");
      printf("целое число, такое как 25, -178 или 3: ");
   }
   return input;
}
```

Вот как выглядит пример выполнения этой программы:

```
Введите букву выбранного варианта:
с. совет            з. звонок
п. подсчет          в. выход
с
Покупайте дешево, продавайте дорого.
Введите букву выбранного варианта:
с. совет            з. звонок
п. подсчет          в. выход
подсчет
Считать до какого предела? Введите целое число:
два
два не является целочисленным.
Пожалуйста, введите целое число, такое как 25, -178 или 3: 5
1
2
3
4
5
Введите букву выбранного варианта:
с. совет            з. звонок
п. подсчет          в. выход
а
Выберите с, з, п или в.
в
Всего хорошего.
```

Иногда довольно-таки трудно добиться того, чтобы интерфейс, использующий меню, работал без сбоев, то есть настолько гладко, насколько это возможно, однако после разработки жизнеспособного подхода вы сможете применять его во множестве ситуаций. Следует также обратить внимание на то, как каждая функция, столкнувшись с необходимостью выполнить слегка усложненную задачу, передает эту задачу другой функции, тем самым повышая уровень модульности программы.

Ключевые понятия

Программы на языке C рассматривают входные данные как поток байтов. Функция `getchar()` интерпретирует каждый байт как символьный код. Функция `scanf()` воспринимает ввод аналогично, в то же время, с помощью спецификаторов преобразования она может перевести символьный ввод в числовое значение. Многие операционные системы предлагают механизм перенаправления, которые позволяет вам сменить клавиатуру на файл при вводе и направлять выходные данные в файл.

Часто программы ожидают входные данные, представленные в специальной форме. Вы можете существенно повысить надежность программы и сделать ее более дружественной по отношению к пользователям, выявляя ошибки, которые может допустить пользователь при вводе, и снабжая программу средствами, способными справиться с этими ошибками.

В случае небольшой программы проверка допустимости ввода может оказаться наиболее интенсивно используемой частью программы. Она предлагает несколько ва-

риантов. Например, если пользователь вводит неправильную информацию, вы можете завершить программу, предоставить пользователю фиксированное количество попыток для исправления входных данных либо предложить неограниченное число таких попыток.

Резюме

Многие программы используют функцию `getchar()` для посимвольного ввода входных данных. Обычно системы используют *построчно буферизованный ввод* в том смысле, что входные данные передаются в программу всякий раз, когда нажимается клавиша <Enter>. Нажатие клавиши <Enter> генерирует символ новой строки, и этому явлению необходимо уделять внимание при разработке программ. Стандарт ANSI C требует применения буферизованного ввода.

Язык C располагает семейством функций, получившим название стандартного пакета ввода-вывода, который позволяет применять унифицированный подход при работе с различными формами файлов в различных системах. Функции `getchar()` и `scanf()` принадлежат этому семейству. Обе функции возвращают значение `EOF` (определение этого знака содержится в заголовочном файле `stdio.h`), когда обнаруживают конец файла. Системы Unix обеспечивают возможность моделировать условие конца файла с клавиатуры, для этого нужно нажать <Ctrl+D> в начале каждой строки; системы DOS используют для этой цели клавиши <Ctrl+Z>.

Во многих операционных системах, в том числе в Unix и DOS, реализован механизм *перенаправления*, который позволяет использовать для входных и выходных данных файлы вместо клавиатуры и экрана. Программы, которые читают ввод до тех пор, пока не встретится `EOF`, могут использоваться как при вводе данных с клавиатуры с эмулированными сигналами конца файла, так и при перенаправлении ввода в файлы.

Использование вызовов функции `scanf()` при обращениях к функции `getchar()` порождает проблемы в тех случаях, когда функция `scanf()` оставляет символы новой строки во входных данных, перед тем как вызывать функцию `getchar()`. Тем не менее, сознавая важность таких проблем, вы можете разрабатывать свои программы с их учетом.

При написании собственной программы тщательно планируйте пользовательский интерфейс. Постарайтесь предусмотреть все виды ошибок, которые могут совершить пользователи, чтобы создавать такие программы, которые могли бы их исправлять.

Вопросы для самоконтроля

1. Выражение `putchar(getchar())` является допустимым; что оно означает? Допустимо ли также и выражение `getchar(putchar())`?
2. Какие действия выполняют следующие операторы?
   а. `putchar('H');`
   б. `putchar('\007');`
   в. `putchar('\n');`
   г. `putchar('\b');`

3. Предположим, что имеется исполняемая программа с именем count, которая подсчитывает количество символов во входных данных. Придумайте команду для среды командной строки, которая использует программу count для подсчета количества символов в essay и для запоминания результата в файле с именем essayct.
4. Пусть заданы программа и файлы, описанные в пункте 3, какие из приведенных ниже команд являются допустимыми?
   а. `essayct <essay`
   б. `count essay`
   в. `essay >count`
5. Что такое EOF?
6. Какими являются выходные данные каждого из следующих ниже фрагментов в условиях указанных ниже входных данных (предполагается, что переменная ch имеет тип int и ввод буферизованный)?
   а. Задан следующий ввод:
   If you quit, I will.[enter]
   Фрагмент программы имеет вид:
```
while ((ch = getchar()) != 'i')
   putchar(ch);
```
   б. Задан следующий ввод:
   Harhar[enter]
   Фрагмент программы имеет вид:
```
while ((ch = getchar()) != '\n')
{
   putchar(ch++);
   putchar(++ch);
}
```
7. Какой подход применяет язык C к разным компьютерным системам с разными соглашениями относительно файлов и символов новой строки?
8. С какой потенциальной проблемой вы столкнетесь при смешивании символьного ввода и ввода чисел в системах с буферизованным вводом?

Упражнения по программированию

Некоторые из описанных ниже программ требуют, чтобы ввод прекращался в результате появления символа EOF. Если в используемой вами операционной системе процедура перенаправления неудобна или вообще невозможна, воспользуйтесь какой-то другой проверкой для прекращения ввода, такой как, например, считывание символа &.

1. Напишите программу, которая подсчитывает количество символов при их вводе до достижения конца файла.
2. Напишите программу, которая воспринимает входные данные как поток символов и читает их до тех пор, пока не встретит символ EOF. Заставьте программу

распечатывать каждый входной символ и его десятичное значение. Обратите внимание на то, что в последовательности ASCII символу пробела предшествуют непечатаемые символы. Примените к ним специальную обработку. Если непечатаемым символом является символ новой строки или символ табуляции, печатайте, соответственно, `\n` или `\t`. В противном случае, воспользуйтесь для обозначения символами управления. Например, ASCII 1 — это <Ctrl+A>, который может отображаться как `^A`. Обратите внимание, что ASCII-значение для символа `A` представляет собой значение <Ctrl+A> плюс 64. Аналогичное отношение выполняется и для других непечатаемых символов. Печатайте по 10 пар в строке, но начинайте печать с новой строки всякий раз, когда встречается символ новой строки.

3. Напишите программу, которая считывает входные данные как поток символов, пока не встретит символ `EOF`. Сделайте так, чтобы программа отдельно сообщала о количестве букв верхнего регистра и количестве букв нижнего регистра. Можете предположить, что числовые значения букв нижнего регистра образуют непрерывную последовательность, это же вы можете предположить и в отношении букв верхнего регистра. Либо можете воспользоваться функциями из библиотеки `ctype.h`, выполняющими соответствующую классификацию символов, в этом случае уровень переносимости программы будет увеличен.
4. Напишите программу, которая считывает входные данные как поток символов, пока не встретит символ `EOF`. Сделайте так, чтобы программа отдельно сообщала среднее количество букв на слово. Пробелы не должны трактоваться как буквы слова. Фактически, знаки препинания также можно подсчитать, но в данном конкретном случае этого пока делать не следует. (Если вас заинтересовала эта проблема, попробуйте воспользоваться функцией `ispunct()` из семейства `ctype.h`.)
5. Внесите в программу угадывания чисел, представленную в листинге 8.4, такие изменения, которые реализуют более интеллектуальную стратегию. Например, пусть программа сначала предложит число 50, и спросит, отклонился ли этот вариант от задуманного числа в большую сторону, в меньшую сторону или же число угадано. Если, скажем, предположение меньше задуманного числа, следующая попытка угадать число производится в диапазоне от 50 до 100, то есть 75. Если же предположение больше задуманного числа, следующая попытка производится в середине диапазона чисел от 75 до 50 и так далее. Используя стратегию *бинарного поиска*, программа быстро находит правильный ответ, конечно, если пользователь не вводит ее в заблуждение.
6. Внесите изменения в функцию `get_first()`, представленную в листинге 8.8, с таким расчетом, чтобы она возвращала первый встреченный ею непробельный символ. Проверьте ее в какой-нибудь простой программе.
7. Видоизмените упражнение 8 из главы 7 таким образом, чтобы варианты меню были помечены буквами, а не номерами.
8. Напишите программу, которая выводит на экран меню, предлагая выбрать арифметическую операцию сложения, вычитания, умножения или деления. Получив выбор, программа приглашает ввести два числа, а затем выполняет заказанную арифметическую операцию. Программа должна принимать только пред-

ложенный выбор из меню. Она должна использовать тип `float` для чисел и предоставлять пользователю возможность делать повторные попытки, если с первого раза ему не удается ввести число. В случае вычитания программа должна предложить пользователю ввести новое значение, если в качестве второго операнда вычитания был выбран `0`. Выполнение такой программы должно иметь следующий вид:

```
Выберите желаемую операцию:
с. сложение             в. вычитание
у. умножение            д. деление
к. выход из программы
с
Введите первое число: 22.4
Введите второе число: один
Один не является числом.
Пожалуйста, введите число, такое как 2.5, -1.78E8 или 3: 1
22.4 + 1 = 23.4
Выберите желаемую операцию:
с. сложение             в. вычитание
у. умножение            д. деление
к. выход из программы
д
Введите первое число: 18.4
Введите второе число: 0
Введите число, отличное от 0: 0.2
18.4 / 0.2 = 92
Выберите желаемую операцию:
с. сложение             в. вычитание
у. умножение            д. деление
к. выход из программы
к
Всего хорошего.
```

ГЛАВА 9

Функции

В этой главе:

- Ключевые слова: `return`
- Операции: * (унарная), & (унарная)
- Функции и их определение
- Использование аргументов и возвращаемых значений
- Использование переменных типа указатель в качестве аргументов функций
- Типы функций
- Прототипы ANSI C
- Рекурсия

Какова организация вашей программы? Принципы построения программ на языке C рассматривают функции как строительные блоки. Ваши программы широко используют стандартную библиотеку C для работы с такими функциями, как `printf()`, `scanf()`, `getchar()`, `putchar()` и `strlen()`. Теперь вы созрели для более активных действий – для создания собственных функций. В ранних главах этой книги были заложены основы этого процесса, а в этой главе вся информация будет систематизирована.

Обзор функций

Прежде всего, что такое функция? *Функция* представляет собой самодостаточную единицу программного кода, разработанную для решения конкретной задачи. Функция в языке C играет ту же роль, какую играют функции, подпрограммы и процедуры в других языках программирования, хотя в деталях эти роли могут быть различными. В результате выполнения некоторых функций происходит то или иное событие. Например, в результате выполнения функции `printf()` на вашем экране появляются конкретные данные. Другие функции возвращают значения для их последующего использования в программе. Например, функция `strlen()` сообщает программе длину заданной строки. В общем случае функция может одновременно выполнять действия и возвращать значения.

Почему вы должны пользоваться функциями? Во-первых, они снимают с вас обременительную обязанность многократного повторения в программе одних и тех же кодовых последовательностей. Если в программе приходится решать одну и ту же задачу несколько раз, вам достаточно написать соответствующую функцию всего лишь один раз. Программа использует эту функцию там, где необходимо, а вы можете использовать одну и

ту же функцию в нескольких программах, ведь ранее вы вызывали функцию putchar() в нескольких программах, не так ли? Даже в тех случаях, когда задача в программе решается всего лишь один раз, использование функции целесообразно, поскольку при этом увеличивается уровень модульности, благодаря чему программа становится более понятной при чтении, к тому в нее легче вносить изменения и исправления.

Предположим, например, что вы хотите написать программу, которая выполняет следующие действия:

- Считывает список чисел.
- Сортирует эти числа.
- Вычисляет среднее значение.
- Вычерчивает гистограмму.

С этой целью вы можете воспользоваться следующей программой:

```
#include <stdio.h>
#define SIZE 50
int main(void)
{
   float list[SIZE];

   readlist(list, SIZE);
   sort(list, SIZE);
   average(list, SIZE);
   bargraph(list, SIZE);
   return 0;
}
```

Разумеется, вам также придется написать четыре функции readlist(), sort(), average() и bargraph(), но это уже детали. Описательные имена функций позволяют определить, что делает программа и как она организована. Далее вы можете автономно работать с каждой функцией, и если вы придадите функциям более общий характер, то сможете использовать их в других программах.

Многие программисты предпочитают думать о функции как о "черном ящике", представленном в терминах информации, которая поступает на вход этого ящика (ввод), и значением или действием, которое он производит (вывод). Вас не должно интересовать, что происходит внутри черного ящика, если, конечно, вы не программист, занимающийся разработкой этой функции. Например, когда вы пользуетесь функцией printf(), вы знаете, что ей нужно передать управляющую строку и, возможно, некоторые аргументы. Вы также знаете, какой выход должна генерировать функция printf(). Вам также вовсе не нужно знать программный код реализации printf(). Такой подход к использованию функций позволяет сосредоточить все усилия на создании общей структуры программы и не отвлекаться на отдельные детали. Тщательно продумайте, что должна выполнять функция и какое место она занимает в программе, прежде чем приступать к написанию ее программного кода.

Что вы должны знать о функциях? Прежде всего, вы должны знать, как их правильно определять, как их вызывать для последующего использования и как наладить их взаимодействие. Чтобы освежить эти моменты в памяти, начнем с рассмотрения очень простого примера, а затем будем добавлять в него все новые возможности, пока не получим полное представление о функциях.

Создание и использование простой функции

Нашей первой скромной целью является создание функции, которая печатает 40 звездочек в строке. Чтобы придать этой функции конкретный смысл, мы включим ее в программу, которая печатает простой заголовок письма. В листинге 9.1 программа показана полностью. Она состоит из функций `main()` и `starbar()`.

Листинг 9.1. Программа `lethead1.c`

```
/* lethead1.c */
#include <stdio.h>
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40
void starbar(void);           /* прототип функции */
int main(void)
{
   starbar();
   printf("%s\n", NAME);
   printf("%s\n", ADDRESS);
   printf("%s\n", PLACE);
   starbar();                 /* использование функции */
   return 0;
}
void starbar(void)            /* определение функции */
{
   int count;
   for (count = 1; count <= WIDTH; count++)
      putchar('*');
   putchar('\n');
}
```

Вывод программы выглядит следующим образом:

```
****************************************
GIGATHINK, INC.
101 Megabuck Plaza
Megapolis, CA 94904
****************************************
```

Анализ программы

Следует отметить несколько важных особенностей этой программы:

- Она использует идентификатор `starbar` в трех контекстах: в *прототипе функции*, который сообщает компилятору о том, какого рода функцией является `starbar()`, в *вызове функции*, благодаря которому происходит выполнение функции, и в *определении функции*, которое описывает все, что делает функция.

- Подобно переменным, функции имеют типы. Любая программа, которая использует функцию, должна объявить тип этой функции, прежде чем она ею воспользуется. В силу этого обстоятельства прототип, отвечающий требованиям стандарта ANSI C, предшествует объявлению функции `main()`:

  ```
  void starbar(void);
  ```

  Круглые скобки указывают, что `starbar` является именем функции. Первое ключевое слово `void` — это тип функции; тип `void` указывает, что данная функция не возвращает значения. Второе слово `void` (которое заключено в круглые скобки) показывает, что у функции нет аргументов. Точка с запятой означает, что вы объявляете функцию, а не определяете ее. То есть эта строка извещает о том, что программа использует функцию типа `void` под именем `starbar()`, и что компилятор должен искать ее определение в другом месте программы. В случае компиляторов, которые не распознают прототипов ANSI C, просто объявите тип функции следующим образом:

  ```
  void starbar();
  ```

  Следует отметить, что некоторые очень старые компиляторы не распознают тип `void`. В этом случае используйте тип `int` для функций, которые не возвращают значения.

- Программа помещает прототип функции `starbar()` перед функцией `main()`; вместо этого она может быть помещена внутри функции `main()` в том месте, в каком находятся объявления любых других переменных. Годится любой из этих способов.

- Программа *вызывает* функцию (обращается к функции) `starbar()` из функции `main()`, используя ее имя, за которым следуют круглые скобки и точка с запятой, тем самым, создавая оператор:

  ```
  starbar();
  ```

  Это форма вызова функции типа `void`. Каждый раз, когда компьютер выходит на оператор `starbar();`, он ищет функцию `starbar()` и выполняет содержащиеся в ней команды. Завершив выполнение команд функции `starbar()`, компьютер возвращается к следующей строке *вызывающей функции*, в рассматриваемом случае это `main()` (рис. 9.1).

- Для определения функции `starbar()` программа выбирает ту же форму, что и для определения функции `main()`. Оно начинается с типа, имени и круглых скобок. Далее следует открывающая фигурная скобка, объявление используемых переменных, определение операторов функции и закрывающая фигурная скобка (рис. 9.2). Обратите внимание, что за именем функции `starbar()` не следует точка с запятой. Отсутствие точки с запятой говорит компилятору о том, что в данном случае вы объявляете функцию `starbar()`, но не вызываете ее и не создаете ее прототип.

- Программа включает функции `starbar()` и `main()` в один и тот же файл. В то же время вы можете использовать два отдельных файла. Вариант с одним файлом легче компилировать. Вариант с двумя файлами упрощает использование одной и той же функции в различных программах. Если вы храните функцию в

отдельном файле, вы должны поместить в этот же файл необходимые директивы `#define` и `#include`. Вариант с двумя или большим числом файлов мы рассмотрим позднее. А пока мы поместим все функции в один файл. Закрывающая скобка функции `main()` указывает компилятору, где эта функция оканчивается, а следующий за ней заголовок функции `starbar()` уведомляет компилятор о том, что `starbar()` является функцией.

- Переменная `count` в функции `starbar()` является *локальной*. Это означает, что она известна только функции `starbar()`. Вы можете использовать имя `count` в других функциях, включая `main()`, и в этом случае конфликта удается избежать. Просто в программе применяются отдельные независимые друг от друга переменные, получившие одно и то же имя.

Если представить себе функцию `starbar()` как черный ящик, то его действие заключается в том, чтобы печатать строку звездочек. Она вообще не выполняет ввод, поскольку ей не нужна никакая информация от вызывающей функции.

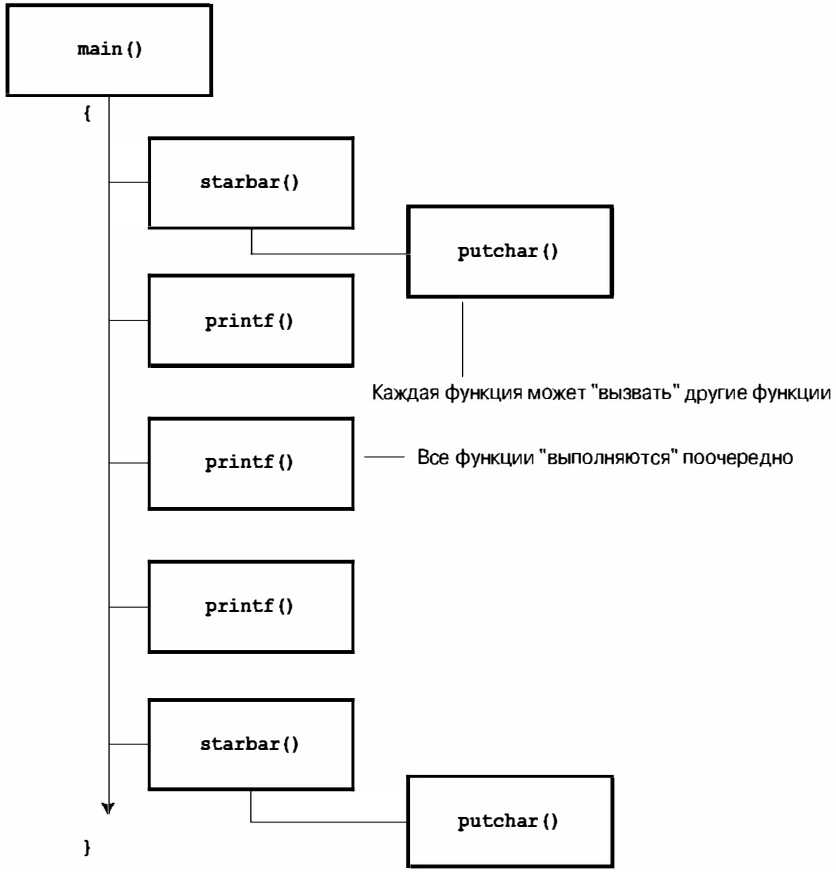


Рис. 9.1. *Поток управления программы* `lethead1.c` *(листинг 9.1)*

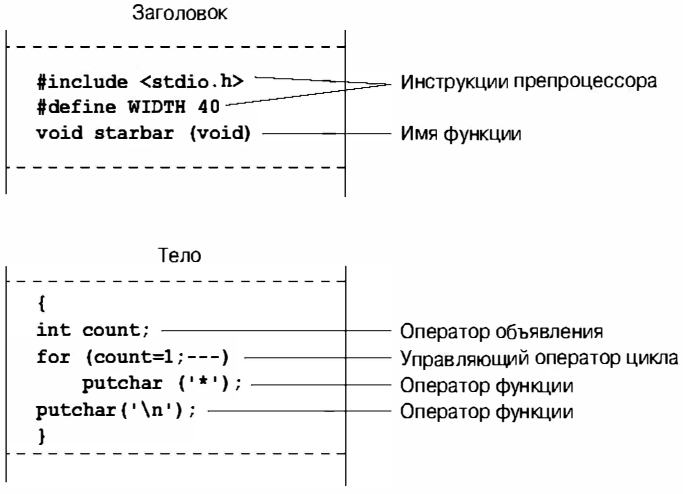


Рис. 9.2. *Структура простой функции*

Она не предоставляет (то есть не *возвращает*) никакой информации в функцию `main()`, следовательно, функция `starbar()` не имеет возвращаемого значения. Короче говоря, функция `starbar()` не нуждается ни в какой связи с вызывающей функцией.

А теперь создадим функцию, для которой подобный обмен данным необходим.

Аргументы функции

Показанный выше заголовок письма выглядел намного лучше, если бы текст располагался по центру. Вы можете поместить текст в центре, поместив требуемое количество ведущих пробелов перед тем, как начать печать текст. Такое поведение аналогично функции `starbar()`, которая печатала заданное число звездочек, но в данный момент вы хотите заданное число пробелов. Вместо того чтобы создавать отдельные функции для каждой задачи, мы, соблюдая принципы языка C, напишем одну, более универсальную функцию, которая решает обе эти задачи. Мы назовем эту новую функцию `show_n_char()` (имя означает, что конкретный символ отображается *n* раз). Единственное изменение заключается в том, что вместо использования встроенных значений отображаемого символа и количества повторений функция `show_n_char()` будет использовать аргументы для получения соответствующих значений.

А теперь займемся более конкретными делами. Предположим, что доступное пространство имеет ширину в 40 символов. Строка из звездочек содержит 40 символов, расположенных вплотную друг к другу, а обращение к функции `show_n_char('*', 40)` должна печатать эту строку точно так же, как это делала функция `starbar()` раньше. Что можно сказать о пробелах, используемых для центрирования строки `GIGATHINK, INC`? Строка `GIGATHINK, INC.` имеет ширину, содержащую 15 пробелов, поэтому в первой версии программы за заголовком следовало 25. Чтобы поместить строку в центр, нужно включить в начало строки 12 пробелов, в результате получим 12 пробелов с одной стороны фразы и 13 пробелов с другой стороны. Следовательно, сначала необходимо выполнить такой вызов функции:

```
show_n_char(' ', 12)
```

Если не учитывать наличие аргумента, то это обращение к функции show_n_char() во многом похоже на вызов функции starbar(). Одно из различий заключается в том, что функция show_n_char() не добавляет символа новой строки, как это делает функция starbar(), так как, скорее всего, в эту строку потребуется добавить текст. В листинге 9.2 показан модифицированный вариант рассматриваемой программы. Чтобы продемонстрировать, как работают аргументы, программа использует различные их формы.

Листинг 9.2. Программа `lethead2.c`

```
/* lethead2.c */
#include <stdio.h>
#include <string.h>              /* для функции strlen() */
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40
#define SPACE ' '

void show_n_char(char ch, int num);

int main(void)
{
   int spaces;

   show_n_char('*', WIDTH);     /* использование констант
                                   в качестве аргументов */
   putchar('\n');
   show_n_char(SPACE, 12);      /* использование констант
                                   в качестве аргументов */
   printf("%s\n", NAME);
   spaces = (WIDTH - strlen(ADDRESS)) / 2;
                                /* Пусть программа вычислит, */
                                /* сколько пробелов пропустить */
   show_n_char(SPACE, spaces);  /* использование переменной
                                   в качестве аргумента */
   printf("%s\n", ADDRESS);
   show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
                                /* выражение в качестве аргумента */
   printf("%s\n", PLACE);
   show_n_char('*', WIDTH);
   putchar('\n');

   return 0;
}

/* определение функции show_n_char()*/
void show_n_char(char ch, int num)
{
   int count;

   for (count = 1; count <= num; count++)
      putchar(ch);
}
```

Ниже показаны результаты выполнения программы:

```
**************************************
            GIGATHINK, INC.
          101 Megabuck Plaza
         Megapolis, CA 94904
**************************************
```

А теперь рассмотрим, как можно построить функцию, которая принимает аргументы. После этого вы узнаете, как эта функция используется.

Определение функции с аргументами: формальные параметры

Определение функции начинается со следующего заголовка функции, соответствующего стандарту ANSI C:

```
void show_n_char(char ch, int num)
```

Эта строка сообщает компилятору о том, что функция `show_n_char()` использует два аргумента под именами `ch` и `num`, а также о том, что переменная `ch` имеет тип `char`, а переменная `num` — тип `int`. Обе переменных `ch` и `num` называются *формальными аргументами* или, как на них ссылаются сейчас, *формальными параметрами*. Подобно переменным, определенным внутри функции, формальные параметры представляют собой локальные переменные, действующие в рамках только данной функции. Это означает, что вы можете не беспокоиться, если дубликаты имен переменных будут использованы и в других функциях. Этим переменным назначаются конкретные значения при каждом обращении к функции.

Обратите внимание, что форма ANSI C требует, чтобы каждой переменной предшествовал ее тип. То есть, в отличие от случая с обычными объявлениями, вы не можете использовать список переменных одного типа:

```
void dibs(int x, y, z)              /* неправильный заголовок функции */
void dubs(int x, int y, int z)      /* правильный заголовок функции */
```

Стандарт ANSI C признает также форму, которая применялась до появления формы ANSI, однако характеризует ее как устаревшую и выходящую из употребления:

```
void show_n_char(ch, num)
char ch;
int num;
```

В рассматриваемом случае в круглые скобки заключен список имен аргументов, но их типы объявляются позже. Обратите внимание на то, что аргументы объявляются перед фигурной скобкой, которая обозначает начало тела функции, но обычные локальные переменные объявляются после фигурной скобки. Такая форма позволяет использовать списки имен переменных, разделенных запятыми, если эти переменные имеют один и тот же тип, как показано в примерах ниже:

```
void dibs(x, y, z)
int x, y, z;                    /* правильно */
```

Назначение стандарта ANSI состоит в том, чтобы постепенно прекратить использование форм, употреблявшихся до его появления. Эти формы следует знать, чтобы

можно было понимать старые программы, но в то же время в новых программах вы должны использовать современные формы.

И хотя функция show_n_char() принимает значения от функции main(), она ничего не возвращает. Поэтому функция show_n_char() имеет тип void. Теперь посмотрим, как используется эта функция.

Создание прототипа функции с аргументами

Мы применяем прототип ANSI для объявления функции перед тем, как она будет использоваться:

```
void show_n_char(char ch, int num);
```

Когда функция принимает аргументы, ее прототип задает их количество и типы, при этом употребляются списки по типам, в которых имена переменных отделяются друг от друга запятыми. При желании вы можете опустить имена переменных в прототипах:

```
void show_n_char(char, int);
```

На самом деле использование имен в прототипах функции не приводит к созданию переменных. Оно просто подчеркивает тот факт, что char в прототипе означает переменную типа char и так далее.

Снова напомним, что стандарт ANSI C также признает старые формы объявления функции, в которой список аргументов не употребляется:

```
void show_n_char();
```

Эта форма фактически выпадает из указанного выше стандарта. Если бы даже это было не так, все равно формат прототипа намного лучше, в чем вы вскоре убедитесь сами. Основная причина того, что вы должны знать эту форму, заключается в том, что вам приходится с ними сталкиваться в старых программах.

Вызов функции с аргументом: фактические аргументы

Значения переменным ch и num назначаются с помощью *фактических аргументов* в вызове функции. Рассмотрим первый случай использования функции show_n_char():

```
show_n_char(SPACE, 12);
```

Фактическими аргументами являются символ пробела и число 12. Эти значение присваиваются соответствующим формальным параметрам функции show_n_char(), то есть переменным ch и num. Короче говоря, формальный параметр есть переменная в вызываемой функции, а фактическим аргументом является конкретное значение, присваиваемое переменной функции в момент вызова функции. Как показывает пример, фактическим аргументом может быть константа, переменная и даже более сложное выражение. Независимо от того, что представляет собой фактический аргумент, он вычисляется, и результат вычислений копируется в соответствующий формальный параметр функции. Например, рассмотрим последний случай использования функции show_n_char():

```
show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
```

Вычисление длинного выражения, образующего второй актуальный аргумент, дает в результате 10. Значение 10 присваивается переменной num. Функция не знает, да и не пытается узнать, откуда происходит это число, от константы, переменной или более общего выражения. Еще раз подчеркнем, что фактический аргумент представляет собой конкретное значение, присваиваемое переменной, известной как формальный параметр (рис. 9.3). Поскольку вызываемая функция работает с данными, скопированными из вызывающей функции, исходные данные в вызывающей функции защищены от любых манипуляций, которым вызываемая функция подвергает их копии.

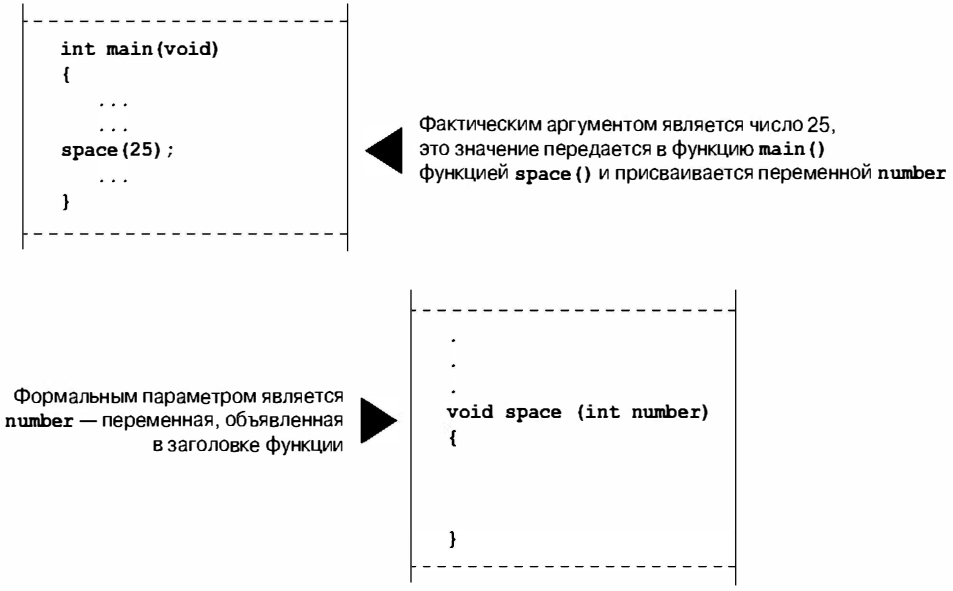


Рис. 9.3. *Формальные параметры и фактические аргументы*

Фактические аргументы и формальные параметры

Фактический аргумент — это выражение, которое появляется в круглых скобках в вызове функции. Формальный параметр — переменная, объявленная в заголовке объявления функции. Когда выполняется вызов функции, переменные, объявленные как формальные параметры, создаются и инициализируются значениями, полученными при вычислении фактических аргументов. В листинге 9.2 '*' и WIDTH являются фактическими аргументами, когда функция show_n_char() вызывалась первый раз, а SPACE и 11 — фактическими аргументами второго вызова этой функции. В определении функции рассматриваемой функции ch и num — это формальные параметры.

Представление функции в виде черного ящика

В представлении функции show_n_char() в виде черного ящика входом является символ, подлежащий отображению, и количество пробелов, которые нужно пропустить. Результатом является печать заданного символа указанное число раз. Такой ввод привязан к функции посредством аргументов.

Этой информации вполне достаточно, чтобы продемонстрировать, как эта функция используется внутри `main()`. Кроме того, она вполне может служить в качестве спецификации для написания функции.

Тот факт, что `ch`, `num` и `count` — локальные переменные, не видимые за пределами функции `show_n_char()`, хорошо вписывается в концепцию черного ящика. Если бы вы использовали переменные с теми же именами в функции `main()`, то это были бы другие, независимые переменные. То есть, если в функции `main()` была бы переменная `count`, то изменение ее значения не привело бы к изменению значения `count` в функции `show_n_char()` и наоборот. Все, что происходит внутри черного ящика, недоступно вызывающей функции.

Возврат значения функцией с помощью оператора `return`

Вы уже видели, как передается информация из вызываемой функции в вызывающую. Чтобы передать информацию в противоположном направлении используется возвращаемое значение. Чтобы напомнить, как это работает, создадим функцию, которая возвращает меньший из двух аргументов. Дадим этой функций имя `imin()`, поскольку она предназначена для манипулирования значениями тип `int`. Кроме того, напишем простую функцию `main()`, единственная цель которой заключается в проверке работоспособности функции `imin()`. Программу, разработанная для тестирования функций таким способом, иногда называют *драйвером*. Драйвер берет функцию для проверки. Если функция проходит проверку успешно, ее можно использовать в более ответственной программе. В листинге 9.3 показан драйвер и функция выбора минимального значения.

Листинг 9.3. Программа `lesser.c`

```
/* lesser.c -- из двух зол она выбирает меньшую */
#include <stdio.h>
int imin(int, int);
int main(void)
{
   int evil1, evil2;

   printf("Введите два целых числа (или q для завершения программы):\n");
   while (scanf("%d %d", &evil1, &evil2) == 2)
   {
      printf("Меньшим из двух чисел %d и %d является %d.\n",
         evil1, evil2, imin(evil1,evil2));
      printf("Введите два целых числа (или q для завершения программы):\n");
   }
   printf("Программа завершена.\n");

   return 0;
}

int imin(int n,int m)
{
   int min;
```

```
   if (n < m)
      min = n;
   else
      min = m;
   return min;
}
```

Ниже показан пример выполнения этой программы:

```
Введите два целых числа (или q для завершения программы):
509 333
Меньшим из двух чисел 509 и 333 является 333.
Введите два целых числа (или q для завершения программы):
-9393 6
Меньшим из двух чисел -9393 и 6 является -9393.
Введите два целых числа (или q для завершения программы):
q
Программа завершена.
```

Ключевое слово `return` обеспечивает то, что следующее за ним выражение становится возвращаемым значением функции. В рассматриваемом случае функция возвращает значение, присвоенное переменной `min`. Поскольку `min` имеет тип `int`, следовательно, функция `imin()` также имеет этот тип.

Переменная `min` — это приватная переменная, принадлежащая функции `imin()`, однако значение `min` передается обратно в вызывающую функцию с помощью `return`. Смысл оператора, подобного приведенному ниже, заключается в том, чтобы присвоить значение `min` переменной `lesser`:

```
lesser = imin(n,m);
```

Нельзя ли было вместо этого написать следующие команды?

```
imin(n,m);
lesser = min;
```

Нет, нельзя, поскольку вызывающая функция ничего не знает о существовании переменной `min`. Вспомните, что переменные функции `imin()` являются локальными по отношению к `imin()`. При вызове функции `imin(evil1,evil2)` производится копирование одного набора данных в другой.

Возвращаемое значение не только может быть присвоено той или иной переменной, оно может также быть использовано как часть выражения. Например, можно поступить следующим образом:

```
answer = 2 * imin(z, zstar) + 25;
printf("%d\n", imin(-32 + answer, LIMIT));
```

Возвращаемое значение может быть представлено любым выражением, а не только переменной. Например, вы можете сократить размеры программы:

```
/* функция, определяющая минимальное значение, вторая версия */
imin(int n,int m)
{
   return (n < m) ? n : m;
}
```

В результате вычислений данное условное выражение принимает меньшее из значений n или m и оно же возвращается вызывающей функции. Если вы предпочитаете для ясности или для соблюдения стиля заключить возвращаемое значение в круглые скобки, можете это сделать, хотя круглые скобки в этом случае не нужны.

А что случится, если функция возвращает тип, отличный от объявленного?

```
int what_if(int n)
{
   double z = 100.0 / (double) n;
   return z;                    // что произойдет?
}
```

В этом случае фактическое значение есть то, что вы получите, если присвоите указанное возвращаемое значение переменной, имеющей объявленный тип возвращаемого значения. Следовательно, в рассматриваемом примере конечный итог будет тем же, как если бы вы присвоили значение z переменной типа int, а затем возвратили бы это значение. Например, предположим, что функция вызвана следующим образом:

```
result = what_if(64);
```

Тогда переменной z присваивается значение 1.5625. Однако оператор return возвращает значение 1.

Использование оператора return дает другой эффект. Он завершает функцию и возвращает управление следующему оператору вызывающей функции. Это происходит даже в тех случаях, когда оператор return в функции не последний. Поэтому вы можете написать функцию imin() так, как показано ниже:

```
/* функция, определяющая минимальное значение, третья версия */
imin(int n,int m)
{
   if (n < m)
      return n;
   else
      return m;
}
```

Многие, хотя и не все, программисты-практики, работающие с языком C, считают, что оператор return желательно использовать только один раз и только в конце функции, поскольку так легче отслеживать прохождение потока управления через функцию. В то же время не будет большим грехом применять несколько операторов return в функции, даже такой короткой, как рассматриваемая. В любом случае, для пользователя все три версии — это одно и то же, ибо все они принимают один и тот же ввод и генерируют один и тот же вывод. Они отличаются только внутренним устройством. Даже приведенная далее версия дает тот же результат:

```
/* функция, определяющая минимальное значение, четвертая версия */
imin(int n, int m)
{
   if (n < m)
      return n;
   else
     return m;
   printf("Профессор Флеппард — напыщенное ничтожество.\n");
}
```

Операторы return расставлены таким образом, что управление никогда не попадет на оператор printf(). Профессор Флеппард может пользоваться откомпилированной версией этой функции в своей программе, так и не узнав истинного к нему отношения со стороны студентов, изучающих программирование. Вы можете употребить следующий оператор:

```
return;
```

Он вызывает прекращение выполнения функции и возвращает управление вызывающей функции. Поскольку за оператором return нет никаких выражений, то никакое значение не возвращается, а эта форма может использоваться только в функциях типа void.

Типы функций

Функции должны объявляться с указанием типов. Функция с возвращаемым значением должна быть объявлена с тем же типом, что и у возвращаемого значения. Функции, которые не возвращают никаких значений, должны иметь тип void. Если функции тип не назначается, в более ранних версиях C предполагается, что такая функция имеет тип int. Это соглашение восходит к тем ранним временам существования языка C, когда большая часть функций, так или иначе, имела тип int. Тем не менее, стандарт C99 уже не поддерживает это неявное предположение в отношении int.

Объявление типа является частью определения функции. Следует иметь в виду, что это положение имеет отношение к возвращаемому значению, но не к аргументам функции. Например, заголовок приведенной ниже функции указывает на то, что вы определяете функцию, которая принимает два аргумента типа int, но при этом возвращает тип double:

```
double klink(int a, int b)
```

Чтобы правильно использовать функцию, программа должна знать тип функции. Один из способов достижения этой цели требует размещения полного определения функции в тексте программы до ее первого применения. Однако такой метод делает программу более трудной для понимания. К тому же функции могут быть частью библиотеки C или храниться в каком-то другом файле. В силу этого обстоятельства вы обычно сообщаете компилятору о наличии функций, объявляя их заранее. Например, функция main() в листинге 9.3 содержит следующие строки:

```
#include <stdio.h>
int imin(int, int);
int main(void)
{
   int evil1, evil2, lesser;
```

Вторая строка устанавливает, что imin является именем функции, возвращающей значение типа int. Теперь компилятор будет знать, как обращаться с функцией imin(), когда она позже появится в программе. Мы поместили предварительные объявления функций за пределами функций, которые их используют. Они также могут быть помещены внутри функции. Например, вы можете представить начало программы lesser.c в таком виде:

```
#include <stdio.h>
int main(void)
{
   int imin(int, int);     /* объявление функции imin()*/
   int evil1, evil2, lesser;
```

В любом случае главное заключается в том, что объявление функции должно появляться до фактического ее использования.

В стандартной библиотеке ANSI C функции сгруппированы в семейства, при этом каждое семейство имеет свой заголовочный файл. Такие заголовочные файлы содержат среди всего прочего и объявления функций из таких семейств. Например, заголовочный файл `stdio.h` содержит объявления функций для стандартных библиотечных функций ввода-вывода, таких как `printf()` и `scanf()`. Заголовочный файл `math.h` содержит объявления множества математических функций, например, объявление

```
double sqrt(double);
```

которое уведомляет компилятор о том, что функция `sqrt()` возвращает значение типа `double`. Не путайте эти объявления с определениями функций. Объявление функции сообщает компилятору о том, что какой тип имеет функция, в то время как программный код приводится в определении функции. Включение заголовочного файла `math.h` уведомляет компьютер о том, что возвращаемым типом функции `sqrt()` является `double`, и что программный код функции `sqrt()` содержится в отдельном файле библиотечных функций.

Прототипирование функций в стандарте ANSI C

Традиционная, применявшаяся еще до появления стандарта ANSI C схема объявления функций обладала тем недостатком, что предусматривала указание типа возвращаемого значения функции, но не типов ее аргументов. Рассмотрим, какие проблемы возникают в тех случаях, когда используется старая форма объявления функций.

Представленное ниже объявление, сделанное в форме, применявшейся до появления стандарта ANSI, уведомляет компилятор о том, что функция `imin()` возвращает значение типа `int`:

```
int imin();
```

В то же время, оно ничего не говорит о количестве и типах аргументов `imin()`. Поэтому, в случае применения функции `imin()` с неправильным количеством или типами аргументов компилятор не выявит ошибку.

Решение проблемы

Рассмотрим несколько примеров использования функции `imax()`, которая во многом напоминает функцию `imin()`. В листинге 9.4 показана программа, которая объявляет функцию `imax()` старым способом, а затем неправильно ее использует.

Листинг 9.4. Программа `misuse.c`

```
/* misuse.c -- неправильное использование функции */
#include <stdio.h>
int imax();                /* объявление в старом стиле */

int main(void)
{
   printf("Наибольшим значением из %d и %d является %d.\n",
          3, 5, imax(3));
   printf("Наибольшим значением из %d и %d является %d.\n",
          3, 5, imax(3.0, 5.0));

   return 0;
}

int imax(n, m)
int n, m;
{
   int max;
   if (n > m)
      max = n;
   else
      max = m;

   return max;
}
```

В первом вызове функции `printf()` опущен аргумент функции `imax()`, а во втором вызове используются аргументы с плавающей запятой вместо целочисленных значений. Несмотря на эти ошибки, компиляция и выполнение программы завершается успешно.

Ниже показан пример выходных данных, полученных при использовании отладчика Metrowerks Codewarrior в среде Development Studio 9:

```
Наибольшим значением из 3 и 5 является 1245120.
Наибольшим значением из 3 и 5 является 1074266112.
```

Компилятор Digital Mars 8.4 генерирует значения `4202837` и `1074266112`. Оба эти компилятора работают хорошо; оба они становятся жертвами неправильного использования программой прототипов функций.

Что происходит в данном случае? Механизмы в различных системах могут быть различными, но вот что происходит в персональных компьютерах IBM или в компьютерах VAX. Вызывающая функция помещает аргументы во временную память, получившую название *стека*, а вызываемая функция читает их оттуда. Оба эти процесса не скоординированы между собой. Вызывающая функция решает, какие типы передавать, исходя из значений фактических аргументов в вызове, а вызываемая функция читает значения на основе своих формальных аргументов. Поэтому вызов `imax(3)` помещает *одно* целое значение в стек. Как только начинается выполнение функции `imax()`, она читает два целых значений из стека. В стек был помещен только один аргумент, следовательно, в качестве следующего значения читается то, что случайно оказалось в стеке в этот момент.

Второй раз, когда в примере используется функция imax(), функции imax() передается значение float. Это означает помещение в стек двух значений double. (Вспомните, что значение типа float повышается до double, когда передается в качестве аргумента.) В нашей системе это два 64-разрядных значения, таким образом, в стек помещаются данные в количестве 128 битов. Когда функция imax() считывает два значения типа int из стека, она читает первые 64 бита из стека, поскольку в нашей системе каждое значение типа int занимает 32 бита. Случаю было угодно, чтобы эти биты соответствовали двум конкретным целочисленным значениям, наибольшим из которых было 1074266112.

Решение стандарта ANSI

Подход к решению задачи несоответствия аргументов, предложенный стандартом ANSI, состоит в том, чтобы разрешить в объявлениях функции также и объявление типов. Результатом становится *прототип функции*, то есть объявление, которое устанавливает тип возвращаемого значения, количество аргументов и типы этих аргументов. Чтобы указать, что функции imax() требуются два аргумента, можете объявить ее со следующими прототипами:

```
int imax(int, int);
int imax(int a, int b);
```

Первая форма использует список типов, разделенных запятыми. Во второй форме к типам добавлены имена переменных. Помните, что имена переменных на самом деле являются фиктивными и не обязательно должны соответствовать именам, используемым в определении функции.

Располагая этой информацией, компилятор может проверить, соответствует ли вызов функции прототипу. Задано ли правильное количество аргументов? Правильно ли выбраны их типы? Если имеет место несовпадение типов и если оба типа представляют собой числа, компилятор преобразует значения фактических аргументов в тот же тип, что и формальные аргументы. Например, imax(3.0, 5.0) становится imax(3, 5). Мы внесли изменения в листинг 9.4, чтобы использовать прототип функции. Результат показан в листинге 9.5.

Листинг 9.5. Программа proto.c

```
/* proto.c -- использует прототипы функции */
#include <stdio.h>
int imax(int, int); /* прототип */
int main(void)
{
   printf("Наибольшим значением из %d и %d является %d.\n",
           3, 5, imax(3));
   printf("Наибольшим значением из %d и %d является %d.\n",
           3, 5, imax(3.0, 5.0));
   return 0;
}

int imax(int n, int m)
{
   int max;
```

```
    if (n > m)
        max = n;
    else
        max = m;
    return max;
}
```

При попытке скомпилировать программу, представленную в листинге 9.5, наш компилятор выдает сообщение об ошибке, утверждающее, что вызов функции `imax()` содержит меньше параметров, чем нужно.

А что можно сказать об ошибках при выборе типов? Чтобы провести их исследование, мы заменили вызов `imax(3)` на `imax(3, 5)` и предприняли очередную попытку компиляции. На этот раз сообщений об ошибках не последовало, и мы запустили программу на выполнение.

Вот как выглядят результаты:

```
Наибольшим значением из 3 и 5 является 5.
Наибольшим значением из 3 и 5 является 5.
```

Как и ожидалось, `3.0` и `5.0` во втором вызове были преобразованы в `3` и `5`, чтобы функция смогла обработать ввод должным образом.

И хотя программа не выдала ни одного сообщения об ошибке, наш компилятор выдал предупреждающее сообщение о том, что тип `double` был преобразован в тип `int` и что при этом возможна потеря данных. Например, вызов

```
imax(3.9, 5.4)
```

становится эквивалентным следующему вызову:

```
imax(3, 5)
```

Различие между сообщением об ошибке и предупреждающим сообщением состоит в том, что ошибка не позволяет выполнить компиляцию, а предупреждение компиляцию допускает. Некоторые компиляторы выполняют приведение типов, не уведомляя вас об этом. Это объясняется тем, что стандарт не требует предупреждения. В то же время многие компиляторы позволяют выбирать уровень предупреждений, который управляет, насколько "красноречивым" будет компилятор, выводящий на экран предупреждение.

Отсутствие аргументов и неопределенные аргументы

Предположим, что вы создали прототип, подобный следующему:

```
void print_name();
```

Компилятор ANSI C полагает, что вы приняли решение заранее построить прототип функции, и не станет проверять аргументы. Чтобы указать, что функция на самом деле не имеет аргументов, в круглые скобки заключается ключевое слово `void`:

```
void print_name(void);
```

Компилятор ANSI C интерпретирует предыдущее выражение следующим образом: функция `print_name()` не имеет аргументов. Затем проверяется, используете ли вы аргументы при вызове этой функции.

Некоторые функции, такие как `printf()` и `scanf()`, принимают переменное количество аргументов. В функции `printf()`, например, в качестве первого аргумента используется строка, в то же время остальные аргументы не фиксированы ни по типу, ни по числу. ANSI C в таких случаях допускает построение частичных прототипов. Например, вы можете использовать такой прототип функции `printf()`:

```
int printf(char *, ...);
```

Этот прототип показывает, что первым аргументом является строка (в главе 11 эта тема рассматривается более подробно) и что функция может принимать другие аргументы, природа которых на текущий момент не уточнена.

Библиотека функций на языке C с помощью заголовочного файла `stdarg.h` предлагает стандартный способ для определения функции с переменным количеством параметров; в главе 16 можно найти соответствующие подробности.

Да здравствуют прототипы

Прототипы являются весомым дополнением к языку. Они предоставляют компилятору возможность выявлять многие ошибки или упущения, которые были допущены при использовании функций. Если они не будут своевременно обнаружены, они станут источником проблем, для решения которых потребуются немалые усилия. Обязательно ли ими пользоваться? Нет, вы можете применять старый способ определения функций (способ, в котором параметры не указываются), но у этого способа нет преимуществ, зато для него характерно множество недостатков.

Существует один способ избежать прототипов, но в то же время сохранить преимущества прототипирования. Назначение прототипа заключается в том, чтобы показать компилятору, как должна использоваться функция, прежде чем компилятор выйдет на первый случай фактического вызова этой функции. Вы достигните того же результата, если поместите полное определение функции раньше, чем она будет использована. В этом случае определение действует как свой собственный прототип. Обычно такой прием применяется в отношении коротких функций:

```
// приведенное ниже описание является одновременно и определением,
// и прототипом
int imax(int a, int b) { return a > b ? a : b; }
int main()
{
...
   z = imax(x, 50);
...
}
```

Рекурсия

Язык программирования C позволяет функции вызывать саму себя. Этот процесс называется *рекурсией*. Иногда рекурсия бывает удобным инструментальным средством, но время от времени она преподносит сюрпризы.

Рекурсию трудно довести до конца, поскольку функция, которая вызывает сама себя, способна делать это до бесконечности, если в программе не предусмотрены специальные средства, включающие проверку условия завершения рекурсии.

Рекурсия часто может использоваться там, где применяется цикл. Иногда более предпочтительным решением является цикл, а иногда — рекурсия. Рекурсивное решение изящно, в то время как решение с использованием циклов обладает большей эффективностью.

Рекурсия в действии

Чтобы посмотреть, что собой представляет рекурсия, обратимся к подходящему примеру. Функция `main()` в листинге 9.6 вызывает функцию `up_and_down()`. Мы назовем это "первым уровнем рекурсии". Затем функция `up_and_down()` вызывает саму себя; назовем это "вторым уровнем рекурсии". Второй уровень вызывает третий уровень рекурсии и так далее. В приведенном ниже примере используются четыре уровня рекурсии. Чтобы получить представление о рекурсии изнутри, рассматриваемая программа не только воспроизводит значение переменной `n`, она также воспроизводит значение `&n`, представляющее собой адрес ячейки памяти, где хранится переменная `n`. (Операция `&`, используемая в этой программе, более подробно рассматривается далее в этой главе. Функция `printf()` для адресов предусматривает спецификатор `%p`.)

Листинг 9.6. Программа `recur.c`

```
/* recur.c -- иллюстрация рекурсии */
#include <stdio.h>
void up_and_down(int);

int main(void)
{
   up_and_down(1);
   return 0;
}

void up_and_down(int n)
{
   printf("Уровень %d: ячейка n %p\n", n, &n);        /* 1 */
   if (n < 4)
      up_and_down(n+1);
   printf("УРОВЕНЬ %d: ячейка n %p\n", n, &n);        /* 2 */
}
```

Выходные данные принимают следующий вид:

```
Уровень 1: ячейка n 0x0012ff48
Уровень 2: ячейка n 0x0012ff3c
Уровень 3: ячейка n 0x0012ff30
Уровень 4: ячейка n 0x0012ff24
УРОВЕНЬ 4: ячейка n 0x0012ff24
УРОВЕНЬ 3: ячейка n 0x0012ff30
УРОВЕНЬ 2: ячейка n 0x0012ff3c
УРОВЕНЬ 1: ячейка n 0x0012ff48
```

Пройдемся по данной программе, чтобы выяснить, как работает рекурсия. Прежде всего, функция `main()` вызывает функцию `up_and_down()` с аргументом `1`. В результате формальный параметр `n` функции `up_and_down()` получает значение `1`, поэтому первый оператор печати выводит `Уровень 1`. Далее, поскольку `n` меньше `4`, функция `up_and_down()` (уровень 1) вызывает функцию `up_and_down()` (уровень 2) с фактическим аргументом `n + 1`, или `2`. Этот вызов приводит к тому, что переменной `n` в вызове на уровне 2 присваивается значение `2`, следовательно, первый оператор печати выводит `Level 2`. Аналогично, в результате вызова следующих двух вызовов печатаются `Level 3` и `Level 4`.

При достижении уровня 4 переменная `n` имеет значение `4`, следовательно, проверка `if` заканчивается неудачей. Вызовы функции `up_and_down()` прекращаются. Вместо этого вызов на уровне 4 передает управление второму оператору печати, который выводит `УРОВЕНЬ 4`, поскольку переменная `n` имеет значение `4`. Затем управление передается оператору `return`. В этой точке заканчивается вызов уровня 4, и управление возвращается функции, которая его вызвала (вызов уровня 3). Последним оператором, выполненным на уровне 3, было обращение к уровню 4 в операторе `if`. По этой причине выполнение уровня 3 возобновляется со следующего оператора, таким оператором является второй оператор печати. При этом печатается `УРОВЕНЬ 3`. По окончании уровня 3 управление передается на уровень 2, на котором печатается `УРОВЕНЬ 2`, и так далее.

Обратите внимание на то, что каждый уровень рекурсии использует свою собственную переменную `n`. Этот факт легко установить, если проанализировать адресные значения. (Разумеется, в общем случае различные системы сообщают различные адреса, возможно, в разных форматах. Одним из важных моментов является то, что адрес в строке `Уровень 1` тот же, что и адрес в строке `УРОВЕНЬ 1` и так далее.)

Если это вам покажется слишком сложным, представьте себе, что вы имеете некоторую цепочку вызовов функций, при этом функция `fun1()` вызывает функцию `fun2()`, функция `fun2()` вызывает `fun3()`, а `fun3()` — `fun4()`. Как только заканчивается выполнение функции `fun4()`, управление передается функции `fun3()`. Как только заканчивается выполнение `fun3()`, управление возвращается функции `fun2()`. А когда завершается выполнение `fun2()`, управление возвращается `fun1()`. Рекурсия работает точно так же, за исключением того факта, что все функции `fun1()`, `fun2()`, `fun3()` и `fun4()` представляют собой одну и ту же функцию.

Основы рекурсии

На первый взгляд рекурсия может показаться очень сложным процессом, поэтому стоит рассмотреть три базовых положения, которые существенно облегчат ее понимание. Во-первых, каждый уровень вызова функции имеет собственные переменные. То есть переменная `n` уровня 1 отлична от переменной `n` уровня 2, следовательно, программа создает четыре разных переменных, каждая из которых имеет имя `n`, и каждая имеет свое собственное значение, отличное от других. Когда в конечном итоге программа возвратится к вызову функции `up_and_down()`, выполненному на первом уровне, исходная переменная `n` все еще будет иметь значение 1 (рис. 9.4).

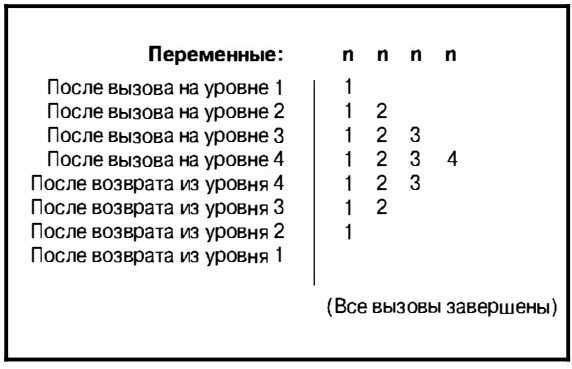


Рис. 9.4. *Переменные рекурсии*

Во-вторых, каждому вызову функции соответствует возврат на предыдущий уровень. Когда поток управления программы достигает оператора `return` в конце последнего уровня рекурсии, управление передается предыдущему уровню рекурсии. Программа не возвращается каждый раз к исходному вызову в функции `main()`. Вместо этого программа должна пройти через каждый уровень рекурсии при возврате с одного уровня функции `up_and_down()` на уровень, на котором функция `up_and_down()` ее вызвала.

В-третьих, операторы рекурсивной функции, которые предшествую рекурсивному вызову, выполняются в том же порядке, в каком были вызваны эти функции. Например, в листинге 9.6 первый оператор печати находится перед рекурсивным вызовом. Он был выполнен четыре раза в порядке следования рекурсивных вызовов: уровень 1, уровень 2, уровень 3 и уровень 4.

В-четвертых, операторы в рекурсивных функциях, которые следуют после рекурсивных вызовов, выполняются в порядке, обратном тому, в каком эти функции вызывались. Например, второй оператор печати располагается после рекурсивного вызова, и выполнялся в следующем порядке: уровень 4, уровень 3, уровень 2, уровень 1. Это свойство рекурсии полезно при написании программ, решающих задачи изменения порядка следования на обратный. Соответствующий пример будет представлен позже.

В-пятых, несмотря на то, что каждый уровень рекурсии имеет собственный набор переменных, сам по себе код не дублируется. Программный код представляет собой последовательность команд, а вызов функции есть команда перехода в начало этой последовательности. Потому рекурсивный вызов возвращает программу в начало этой последовательности команд. Наряду с тем, что рекурсивные вызовы создают новые переменные при каждом вызове, они во многом напоминают цикл. В самом деле, в некоторых случаях рекурсия может быть использована вместо цикла и наоборот.

И, наконец, очень важно, чтобы рекурсивная функция содержала что-то, что могло бы остановить последовательность рекурсивных вызовов. Обычно рекурсивная функция использует проверки `if` или эквивалентные с целью прекратить рекурсию, когда соответствующий параметр функции достигает конкретного значения. Чтобы это работало, необходимо, чтобы каждый вызов использовал различные значения этого параметра. Например, в последнем примере функция `up_and_down(n)` вызывает `up_and_down(n+1)`. В конечном итоге фактический аргумент получает значение `4`, при этом проверка показывает, что условие `if (n < 4)` не выполняется.

Хвостовая рекурсия

В рекурсии простейшей формы рекурсивный вызов расположен в конце функции, непосредственно перед оператором return. Такая рекурсия называется *хвостовой* или *концевой*, поскольку рекурсивный вызов выполняется в конце. Хвостовая рекурсия является простейшей формой рекурсии, поскольку она действует подобно циклу.

Рассмотрим как циклическую, так и рекурсивную версию функции, вычисляющей факториал, при этом будем иметь в виду хвостовую рекурсию. *Факториал* целого числа — это произведение всех целых чисел в промежутке от 1 до заданного числа включительно. Например, факториал 3 (записывается как 3!) равен 1*2*3. При этом 0! полагается равным 1, а факториалы отрицательных чисел не определены. В листинге 9.7 показана одна функция, которая использует цикл для вычисления факториала, и вторая функция, которая для этой цели использует рекурсию.

Листинг 9.7. Программа `factor.c`

```
// factor.c -- использует циклы и рекурсию для вычисления факториала
#include <stdio.h>
long fact(int n);
long rfact(int n);
int main(void)
{
   int num;

   printf("Эта программа вычисляет факториалы.\n");
   printf("Введите значение в диапазоне 0-12 (или q для завершения программы):\n");
   while (scanf("%d", &num) == 1)
   {
      if (num < 0)
         printf("Пожалуйста, не вводите отрицательные числа.\n");
      else if (num > 12)
         printf("Входное значение должно быть меньше 13.\n");
      else
      {
         printf("цикл: факториал %d = %ld\n",
                num, fact(num));
         printf("рекурсия: факториал %d = %ld\n",
                num, rfact(num));
      }
      printf("Введите значение в диапазоне 0-12 (или q для завершения
программы):\n");
   }
   printf("Всего хорошего.\n");
   return 0;
}
long fact(int n)               // функция на базе цикла
{
   long ans;
   for (ans = 1; n > 1; n--)
       ans *= n;
   return ans;
}
```

```
long rfact(int n)                // рекурсивная версия
{
   long ans;
   if (n > 0)
      ans= n * rfact(n-1);
   else
      ans = 1;
   return ans;
}
```

Программа тестового драйвера ограничивает входные данные целыми значениями в диапазоне от 0 до 12. Как показывают вычисления, значение 12! немного меньше полумиллиарда, так что для размещения результата 13! требуется больше памяти, чем может предложить тип `long` в нашей системе. Чтобы вычислять факториалы, превосходящие 12!, необходимо использовать типы больших размеров, такие как `double` или `long long`.

Ниже показаны результаты выполнения демонстрационной программы:

```
Эта программа вычисляет факториалы.
Введите значение в диапазоне 0-12 (или q для завершения программы):
5
цикл: факториал 5 = 120
рекурсия: факториал 5 = 120
Введите значение в диапазоне 0-12 (или q для завершения программы):
10
цикл: факториал 10 = 3628800
рекурсия: факториал 10 = 3628800
Введите значение в диапазоне 0-12 (или q для завершения программы):
q
Всего хорошего.
```

Версия с циклом инициализирует переменную `ans` значением `1`, а затем умножает ее на целые значения в диапазоне от `n` до `2`. В техническом плане необходимо умножать также и на `1`, однако результат от этого не изменится.

Теперь рассмотрим рекурсивную версию. Ключевым фактором является уравнение `n! = n × (n-1)!`. Оно следует из того факта, что `(n-1)!` представляет собой произведение всех положительных целых чисел от `1` до `n-1`. Таким образом, умножение этого результата на `n` дает произведение целых чисел от `1` до `n`. Эта особенность позволяет применить рекурсивный подход. Если соответствующую функцию назвать `rfact()`, то `rfact(n)` есть `n * rfact(n-1)`. Следовательно, вы можете вычислить значение `rfact(n)`, выполнив вызов `rfact(n-1)`, как это делает программа, показанная в листинге 9.7. Разумеется, вы обязательно должны прервать рекурсию в той или иной точке, при этом вы можете сделать это, установив возвращаемое значение равным `1`, когда значением `n` является `0`.

Рекурсивная версия программы в листинге 9.7 позволяет получить тот же результат, что и версия с циклом. Обратите внимание на тот факт, что несмотря на то, что вызов функции `rfact()` не является последней строкой в функции, это последний оператор, выполняемый в условиях, когда `n > 0`, откуда следует, что мы имеем дело с хвостовой рекурсией.

Учитывая тот факт, что при написании кода вы можете использовать как цикл, так и рекурсию, возникает вопрос, какой из этих вариантов выбрать? В большинстве случаев предпочтение отдается циклу. Во-первых, поскольку рекурсивный вызов создает свой собственный набор переменных, вариант с рекурсией использует больше памяти, при этом каждый рекурсивный вызов размещает новый набор переменных в стеке. Во-вторых, рекурсия выполняется медленней, поскольку на каждый вызов функции требуется определенное время. В таком случае, какой смысл уделять этому примеру столько внимания? Дело в том, что хвостовая рекурсия наиболее доступна для понимания, а рекурсия в целом достойна внимательного изучения, так как на практике встречаются случаи, когда с помощью простого цикла задача не решается.

Рекурсия и обратный порядок

Теперь рассмотрим задачу, при решении которой очень удобно воспользоваться рекурсией для изменения порядка на обратный. (Это один из тех случаев, когда реализация рекурсии проще, нежели цикла.) Необходимо решить следующую задачу: написать функцию, которая печатает двоичный эквивалент целого числа. В двоичной системе счисления числа представлены в виде суммы степеней 2. Подобно тому, как в десятичной системе число 234 означает $2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$, число 101 в двоичной системе означает $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$. В двоичных числах используются только цифры 0 и 1.

Вам необходим метод, или *алгоритм*. Как вы можете, скажем, найти двоичный эквивалент числа? Понятно, что нечетные числа должны иметь двоичное представление, оканчивающееся цифрой 1. Четные числа оканчиваются цифрой 0, следовательно, вы можете определить, является ли последняя цифра 1 или 0, вычислив выражение `5 % 2`. Если результатом является 1, то число 5 нечетное, и последней цифрой будет 1. В общем случае, если `n` есть число, то его последняя цифра есть `n % 2`, следовательно, первая цифра, которую вы найдете, будет последней цифрой, которую должны напечатать. Это предполагает использование рекурсивной функции, которая вычисляет выражение `n % 2` до рекурсивного вызова, но которая печатает его результат после рекурсивного вызова. Таким образом, выражение, вычисленное первым, печатается последним.

Чтобы получить следующую цифру, разделите исходное число на 2. Это двоичный эквивалент сдвига десятичной точки на одну позицию влево, благодаря которому вы можете определить следующую двоичную цифру. Если значение четное, следующей двоичной цифрой является 0, а если нечетное — 1. Например, 5/2 равно 2 (целочисленное деление), таким образом, следующая цифра — 0. В результате имеем 01. Продолжим этот процесс. Делим 2 на 2 и в результате получаем 1. Вычисляем 1 % 2 и получаем 1, следовательно, следующей цифрой будет 1. Имеем 101. Когда процесс должен остановиться? Он останавливается в том случае, когда результат деления на 2 меньше 2, поскольку пока он остается равным 2 или больше, для представления числа используются, по меньшей мере, еще одна цифра. Каждое деление на 2 приводит к уменьшению количества цифр на единицу, пока процесс не достигнет конца. (Если есть затруднения в освоении этого алгоритма, перейдите к десятичной аналогии. Остатком от деления 628 на 10 является 8, следовательно, 8 — последняя цифра. Деление на 10 дает 62, а остаток от деления 62 на 10 есть 2, следовательно, таковой является следующая цифра и так далее.) Этот подход реализован в программе, показанной в листинге 9.8.

Листинг 9.8. Программа `binary.c`

```
/* binary.c -- печатает целые числа в двоичной форме */
#include <stdio.h>

void to_binary(unsigned long n);

int main(void)
{
   unsigned long number;
   printf("Введите целое число (или q для завершения программы):\n");
   while (scanf("%ul", &number) == 1)
   {
      printf("Двоичный эквивалент: ");
      to_binary(number);
      putchar('\n');
      printf("Введите целое число (или q для завершения программы):\n");
   }
   printf("Программа завершена.\n");

   return 0;
}

void to_binary(unsigned long n)     /* рекурсивная функция */
{
   int r;

   r = n % 2;
   if (n >= 2)
      to_binary(n / 2);
   putchar('0' + r);

   return;
}
```

В листинге 9.8 выражение `'0' + r` дает в результате символ `'0'`, если `r` равно `0`, и символ `'1'`, если `r` равно `1`. Отсюда следует, что числовой код символа `'1'` на единицу больше, чем символа `'0'`. Как коды ASCII, так и коды EBCDIC удовлетворяют этому условию. В общем случае, вы можете использовать следующий подход:

```
putchar( r ? '1' : '0');
```

Ниже показан пример выполнения этой учебной программы:

```
Введите целое число (или q для завершения программы):
9
Двоичный эквивалент: 1001
Введите целое число (или q для завершения программы):
255
Двоичный эквивалент: 11111111
Введите целое число (или q для завершения программы):
1024
Двоичный эквивалент: 10000000000
Введите целое число (или q для завершения программы):
q
Программа завершена.
```

Можно ли было использовать этот алгоритм для вычисления двоичных представлений числа без применения рекурсии? Можно, однако, поскольку этот алгоритм первой вычисляет последнюю цифру, вы должны где-то сохранить все цифры (например, в массиве), прежде чем отображать результат на экране. В главе 15 будет представлен пример нерекурсивного подхода к решению этой задачи.

Преимущества и недостатки рекурсии

Рекурсия обладает своими преимуществами и недостатками. Одно из преимуществ рекурсии состоит в том, что она предлагает простейшее решение некоторых задач программирования. Один из недостатков рекурсии заключается в том, что некоторые рекурсивные алгоритмы могут довольно-таки быстро исчерпать ресурс памяти компьютера. Наряду с этим рекурсию трудно документировать и поддерживать. Рассмотрим пример, который иллюстрирует как преимущества, так и недостатки рекурсии.

Числа Фибоначчи могут быть определены следующим образом: первое число Фибоначчи есть 1, второе число Фибоначчи также есть 1, а каждое следующее число Фибоначчи есть сумма двух предшествующих чисел. Отсюда следует, что первые несколько чисел последовательности Фибоначчи выглядят так: 1, 1, 2, 3, 5, 8, 13. Числа Фибоначчи пользуются особой любовью в среде математиков, издается даже специальный журнал, посвященный этим числам. Однако в данный момент нас интересует отнюдь не это. Сейчас мы напишем функцию, которая для заданного положительного целого значения `n` возвращает соответствующее число Фибоначчи.

Прежде всего, посмотрим, как работает рекурсия в данном случае: рекурсия задается в виде простого определения. Если мы дадим этой функции имя `Fibonacci()`, то `Fibonacci(n)` должна вернуть значение 1, если `n` равно 1 или 2, и сумму `Fibonacci(n-1)+Fibonacci(n-2)` в любом другом случае:

```
long Fibonacci(int n)
{
   if (n > 2)
      return Fibonacci(n-1) + Fibonacci(n-2);
   else
      return 1;
}
```

Рекурсивная функция в C просто повторяет математическое определение рекурсии. (Чтобы не усложнять проблему, функция не применяется к значениям `n` меньше 1.) Эта функция использует *двойную рекурсию*, то есть функция вызывает себя дважды. Это обстоятельство является источником ее слабости.

Чтобы исследовать эту слабость, предположим, что вы используете вызов `Fibonacci(40)`. Это будет первый уровень рекурсии, и он распределяет переменную с именем `n`. Затем выполняются два вызова функции `Fibonacci()`, создавая еще две переменных с именем `n` на втором уровне рекурсии. Каждый из этих двух вызовов генерирует еще два вызова, которые, в свою очередь, требуют еще четырех переменных с именем `n` на третьем уровне рекурсии, что в итоге составляет семь переменных. На каждом уровне количество переменных удваивается по сравнению с предыдущим уровнем, то есть число переменных возрастает по экспоненте! Как вы могли убедиться на примере с пшеничными зернами в главе 5, экспоненциальное возрастание быстро приводит к огромным значениям. В рассматриваемом случае экспоненциальное

возрастание быстро приводит к тому, что компьютеру потребуются огромные объемы памяти, что, скорее всего, приведет к аварийному завершению программы.

Как мы убедились, это экстремальный пример, в то же время он показывает, что следует соблюдать осторожность при использовании рекурсии, особенно в тех случаях, когда к предъявляются повышенные требования к эффективности программы.

Все функции C созданы на одном и том же уровне

Каждая функция в программе на C находится на одном уровне с остальными функциями. Каждая из них может вызвать любую другую функцию или быть вызванной другими функциями. Эти функции в языке C отличаются от процедур в таких языках программирования, как Pascal и Modula-2, поскольку в них процедуры могут быть вложенными друг в друга. Процедуры в одном вложении не знают, какие процедуры находятся в другом вложении.

Является ли функция `main()` особой? Именно так, она несколько отличается от остальных в том плане, что когда программа составляется из нескольких функций, выполнение этой программы начинается с первого оператора функции `main()`, но этим и ограничивается ее отличие от других функций. Функция `main()` может вызывать сама себя в рамках некоторой рекурсии или быть вызванной из других функций, хотя подобное встречается достаточно редко.

Компиляция программ из двух или большего числа исходных файлов

Простейший метод использования нескольких функций требует их размещения в одном и том же файле. Затем выполняется компиляция этого файла, как если бы он содержал единственную функцию. Другие подходы к решению этой проблемы существенно зависят от конкретной системы, как показано в нескольких следующих разделах.

Unix

В этом случае предполагается, что в системе Unix установлен стандартный для Unix компилятор C — `cc`. Предположим, что `file1.c` и `file2.c` — два файла, содержащие функции языка C. Тогда следующая команда скомпилирует оба файла и создаст исполняемый файл с именем `a.out`:

```
cc file1.c file2.c
```

Кроме того, при этом создаются два объектных файла с именами `file1.o` и `file2.o`. Если позже вы измените содержимое файла `file1.c`, оставив `file2.c` неизменным, вы можете откомпилировать первый файл и объединить его с объектной версией второго файла, с помощью такой команды:

```
cc file1.c file2.o
```

В системе Unix имеется команда `make`, которая автоматизирует управление многофайловыми программами, но эта тема выходит за пределы данной книги.

Linux

В данном случае предполагается, что в системе Linux установлен компилятор gcc языка GNU C. Предположим, что file1.c и file2.c — два файла, содержащие функции языка C. Тогда следующая команда выполнит компиляцию обоих файлов и создаст исполняемый файл с именем a.out:

```
gcc file1.c file2.c
```

Кроме того, при этом создаются два объектных файла file1.o и file2.o. Если позже вы измените содержимое файл file1.c, оставив файл file2.c неизменным, вы можете откомпилировать первый файл и объединить его с объектной версией второго файла, воспользовавшись следующей командой:

```
gcc file1.c file2.o
```

Компиляторы командной строки DOS

Большинство компиляторов командной строки операционной системы DOS работают аналогично команде cc системы Unix. Единственное различие заключается в том, что объектные файлы получают расширение .obj вместо расширения .o. Некоторые компиляторы создают вместо объектных кодов промежуточные файлы с ассемблерным или каким-то другим кодом.

Компиляторы Windows и Macintosh

Компиляторы операционных систем Windows и Macintosh представляют собой *компиляторы, ориентированные на проекты*. *Проект* описывает ресурсы, используемые программой. Эти ресурсы включают ваши файлы исходного кода. Если вы когда-либо работали с такими компиляторами, то, скорее всего, должны были создавать проекты для выполнения однофайловых программ. Что касается многофайловых программ, потребуется найти в меню команду, которая позволяет включить в проект файл исходного кода. Вы должны убедиться в том, что все ваши файлы исходных кодов (то есть файлы с расширением .c) являются частью проекта. Однако будьте осторожны, не помещайте в проект заголовочные файлы (файлы с расширением .h). Идея заключается в том, что проект управляет использованием файлов исходного кода, а директивы #include в файлах исходного кода управляют использованием заголовочных файлов.

Использование заголовочных файлов

Если вы поместили функцию main() в один файл, а определения вашей собственной функции во второй файл, то первому файлу все еще нужны прототипы функций. Вместо того чтобы вводить их с клавиатуры каждый раз, когда вы используете файл с кодом функции, вы можете хранить прототипы функций в одном из заголовочных файлов. Именно эту задачу выполняет стандартная библиотека C, размещая, например, прототипы функций ввода-вывода в файле stdio.h, а математических функций — в файле math.h. Вы можете поступить точно так же в отношении файлов собственных функций.

Кроме того, вы часто будете использовать препроцессор C для определения констант, используемых в программе. Такие определения возможны только для файла, содержащего директивы #define. Если вы поместите функции, используемые в программе, в отдельные файлы, вам также придется обеспечить их доступность, указывая директиву #define для каждого файла. Наиболее прямой путь заключается в повторном вводе директив для каждого файла, но этот процесс требует больших временных затрат и увеличивает вероятность ошибок. Наряду с этим возникает проблема сопровождения: если вы намерены изменить значение директивы #define, то должны помнить, что это нужно сделать для каждого файла. Более предпочтительное решение предусматривает размещение директивы #define в заголовочном файле с последующим использованием директивы #include для каждого файла исходного кода.

Таким образом, хороший тон в программировании рекомендует размещать прототипы функций и объявлять константы в заголовочном файле. Рассмотрим соответствующий пример. Предположим, что вы управляете сетью из четырех отелей. В каждом отеле действуют различные расценки платы за номер, в то же время плата за все номера в одном и том же отеле одинакова. Для постояльцев, которые зарезервировали за собой номера на несколько суток, размер платы за вторые сутки составляет 95% от тарифа за первые сутки, третьи сутки оплачивается в размере 95% от платы за вторые сутки и так далее. (Пусть вас не беспокоит вопрос о целесообразности упомянутой ценовой политики.) Вам нужна программа, которая позволила бы выбрать отель, указать срок проживания в этом отеле в сутках и рассчитать суммарную стоимость пребывания в нем в течение указанного срока. Желательно наличие в программе меню, которое позволило бы вводить данные до тех пор, пока вы не пожелаете выйти.

В листингах 9.9, 9.10 и 9.11 показано, как можно решить эту задачу. Первый листинг содержит функцию main(), которая определяет общую организацию программы. Второй листинг содержит функции поддержки, которые, как мы полагаем, хранятся в отдельном файле. И, наконец, в листинге 9.11 показан заголовочный файл, в котором содержатся определения констант и прототипы функций для всех исходных файлов программы. Напоминаем, что в средах Unix и DOS двойные кавычки в директиве #include "hotels.h" указывают на то, что включаемый файл хранится в текущем рабочем каталоге (обычно в этом каталоге хранится исходный код).

Листинг 9.9. Управляющий модуль usehotel.c

```
/* usehotel.c -- программа определения класса гостиничных номеров */
/* компилируется вместе с листингом 9.10 */
#include <stdio.h>
#include "hotel.h"       /* определяет константы, объявляет функции */

int main(void)
{
   int nights;
   double hotel_rate;
   int code;

   while ((code = menu()) != QUIT)
   {
      switch(code)
      {
      case 1 : hotel_rate = HOTEL1;
```

```
                break;
        case 2 : hotel_rate = HOTEL2;
                break;
        case 3 : hotel_rate = HOTEL3;
                break;
        case 4 : hotel_rate = HOTEL4;
                break;
        default: hotel_rate = 0.0;
                printf("Ошибка!\n");
                break;
        }
        nights = getnights();
        showprice(hotel_rate, nights);
    }
    printf("Благодарим за использование и желаем успехов.");
    return 0;
}
```

Листинг 9.10. Модуль функций поддержки `hotel.c`

```
/* hotel.c -- функции управления отелем */
#include <stdio.h>
#include "hotel.h"
int menu(void)
{
    int code, status;
    printf("\n%s%s\n", STARS, STARS);
    printf("Введите число, соответствующее выбранному отелю:\n");
    printf("1) Fairfield Arms              2) Hotel Olympic\n");
    printf("3) Chertworthy Plaza           4) The Stockton\n");
    printf("5) выход\n");
    printf("%s%s\n", STARS, STARS);
    while ((status = scanf("%d", &code)) != 1 ||
            (code < 1 || code > 5))
    {
        if (status != 1)
            scanf("%*s");
        printf("Пожалуйста, введите целое число от 1 до 5.\n");
    }
    return code;
}
int getnights(void)
{
    int nights;
    printf("На сколько суток вы резервируете номер? ");
    while (scanf("%d", &nights) != 1)
    {
        scanf("%*s");
        printf("Пожалуйста, введите целое число, такое как 2.\n");
    }
    return nights;
}
```

```
void showprice(double rate, int nights)
{
   int n;
   double total = 0.0;
   double factor = 1.0;
   for (n = 1; n <= nights; n++, factor *= DISCOUNT)
      total += rate * factor;
   printf("Общая стоимость составляет $%0.2f.\n", total);
}
```

Листинг 9.11. Заголовочный файл `hotel.h`

```
/* hotel.h -- константы и объявления для программы hotel.c */
#define QUIT 5
#define HOTEL1     80.00
#define HOTEL2    125.00
#define HOTEL3    155.00
#define HOTEL4    200.00
#define DISCOUNT    0.95
#define STARS "**********************************"

// показывает список возможных вариантов
int menu(void);

// возвращает количество суток, на которое резервируется номер
int getnights(void);

// вычисляет стоимость с учетом тарифов и количества суток
// и отображает результат вычислений
void showprice(double rate, int nights);
```

Ниже показан пример выполнения:

```
**********************************************************************
Введите число, соответствующее выбранному отелю:
1) Fairfield Arms            2) Hotel Olympic
3) Chertworthy Plaza         4) The Stockton
5) выход
**********************************************************************
3
На сколько суток вы резервируете номер? 1
Общая стоимость составляет $155.00.

**********************************************************************
Введите число, соответствующее выбранному отелю:
1) Fairfield Arms            2) Hotel Olympic
3) Chertworthy Plaza         4) The Stockton
5) выход
**********************************************************************
4
На сколько суток вы резервируете номер? 3
Общая стоимость составляет $570.50.
```

```
**********************************************************************
Введите число, соответствующее выбранному отелю:
1) Fairfield Arms            2) Hotel Olympic
3) Chertworthy Plaza         4) The Stockton
5) выход
**********************************************************************
5
Благодарим за использование и желаем успехов.
```

Между прочим, для самой этой программы характерны некоторые интересные особенности. В частности, функция `menu()` и `getnights()` пропускают нецифровые данные, и с этой целью они проверяют возвращаемое значение функции `scanf()` и выполняют вызов `scanf("%*s")` с тем, чтобы пропустить следующий пробел. Обратите внимание на то, как приведенный ниже фрагмент функции `menu()` осуществляет проверку на наличие как нецифровых данных, так и числовых данных, выходящих за допустимые пределы:

```
while ((status = scanf("%d", &code)) != 1 ||
       (code < 1 || code > 5))
```

В этом фрагменте кода используется особенность языка C, обеспечивающая вычисление логических выражений слева направо и завершение вычислений в тот момент, когда становится ясно, что выражение принимает ложное значение. В рассматриваемом примере значение переменной проверяется только после того, как подтвердится тот факт, что функции `scanf()` удалось прочитать целочисленное значение.

Назначение отдельных задач отдельным функциям способствует улучшениям программы. При первом проходе функции `menu()` или `getnights()` можно использовать простую функцию `scanf()` без добавленного средства проверки допустимости данных. Затем, после того, как основная версия заработает, можно приступать к работе по совершенствованию каждого модуля.

Поиск адресов: операция &

Одним из наиболее важных понятий языка C (иногда самым трудным для понимания) является *указатель*, представляющий собой адрес, по которому хранится соответствующая переменная. Вы уже видели, что функция `scanf()` использует адреса аргументов. В общем случае любая функция C, которая вносит изменения в значения вызывающей функции без использования возвращаемых значений, использует адреса. Далее мы рассмотрим функции, использующие адреса, и начнем с унарной операции `&`. (В следующей главе мы продолжим исследование указателей и рассмотрим, как они используются.)

Унарная операция `&` выдает адрес, по которому хранится соответствующая переменная. Если `pooh` является именем переменной, то `&pooh` — это адрес переменной. Вы можете представлять адрес как некоторую ячейку в памяти. Предположим, что имеется следующий оператор:

```
pooh = 24;
```

Предположим, что адрес, по которому хранится переменная `pooh`, выглядит как `0B76` (адреса в персональных компьютерах часто задаются в шестнадцатеричной форме).

Тогда оператор

```
printf("%d %p\n", pooh, &pooh);
```

генерирует следующий результат (%p — спецификатор вывода адреса):

```
24 0B76
```

Код в листинге 9.12 использует эту операцию, чтобы узнать, где хранятся переменные с одним и тем же именем, но используемые в различных функциях.

Листинг 9.12. Программа `loccheck.c`

```
/* loccheck.c -- проверка с целью выяснения, где хранятся переменные */
#include <stdio.h>
void mikado(int);           /* объявление функции */
int main(void)
{
   int pooh = 2, bah = 5;   /* переменные, локальные в функции main() */
   printf("В функции main() pooh = %d и &pooh = %p\n",
          pooh, &pooh);
   printf("В функции main() bah = %d и &bah = %p\n",
          bah, &bah);
   mikado(pooh);
   return 0;
}
void mikado(int bah)        /* объявление функции */
{
   int pooh = 10;           /* переменные, локальные в функции mikado() */
   printf("В функции mikado() pooh = %d и &pooh = %p\n", pooh, &pooh);
   printf("В функции mikado() bah = %d и &bah = %p\n", bah, &bah);
}
```

Программа в листинге 9.12 использует формат %p стандарта ANSI C с целью вывода адреса. Для этого небольшого примера наша программа генерирует следующие выходные данные:

```
В функции main() pooh = 2 и &pooh = 0x0012ff48
В функции main() bah = 5 и &bah = 0x0012ff44
В функции mikado() pooh = 10 и &pooh = 0x0012ff34
В функции mikado() bah = 2 и &bah = 0x0012ff40
```

Способ представления адреса с использованием спецификатора %p отличается в различных системах. В то же время многие реализации, такие как используемые в данном примере, отображают адрес в шестнадцатеричной форме.

О чем свидетельствуют эти выходные данные? Во-первых, обе переменных pooh имеют различные адреса. То же можно сказать и о двух переменных bah. Следовательно, как говорилось выше, компьютеры рассматривают их как четыре различных переменных. Во-вторых, вызов функции mikado(pooh) передает значение (2) фактического аргумента (значение pooh из main()) формальному аргументу (значение bah из mikado()). Обратите внимание на то, что было передано только значение. Обе эти переменные (pooh функции main() и bah функции mikado()) сохраняют свою идентичность.

Второй вопрос состоит в том, что это условие выполняется не для всех языков программирования. В языке FORTRAN, например, подпрограмма может затронуть исходную переменную в вызывающей функции. Переменная подпрограммы может иметь другое имя, но тот же адрес. Язык C этого не допускает. Каждая функция использует свои собственные переменные. Такой подход предпочтительнее, поскольку он препятствует непонятным изменениям, производимым некоторыми побочными эффектами вызывающей функции. Тем не менее, он также может породить определенные трудности, как показано в следующем разделе.

Изменение переменных в вызывающей функции

Иногда возникает необходимость изменить значения переменных других функций. Например, распространенная задача алгоритмов сортировки связана с обменом значениями двух переменных. Предположим, что существуют две переменные с именами x и y, и вы хотите, чтобы они обменялись значениями. Простая последовательность операторов

```
x = y;
y = x;
```

не достигает цели, поскольку в тот момент, когда управление будет передано второй строке, исходное значение переменной x уже будет заменено исходным значением переменной y. Потребуется дополнительный оператор для временного запоминания исходного значения x.

```
temp = x;
x = y;
y = temp;
```

Теперь, когда этот метод заработал, вы можете положить его в основу соответствующей функции и подготовить драйвер для ее тестирования. Чтобы выяснить, какая переменная принадлежит функции `main()`, а какая — `interchange()`, в программе, представленной в листинге 9.13, переменные x и y используются в `main()`, а u и v — в `interchange()`.

Листинг 9.13. Программа `swap1.c`

```
/* swap1.c -- первая попытка создания функции обмена значениями */
#include <stdio.h>

void interchange(int u, int v);       /* объявление функции */

int main(void)
{
   int x = 5, y = 10;

   printf("Первоначально x = %d и y = %d.\n", x , y);
   interchange(x, y);
   printf("Теперь x = %d и y = %d.\n", x, y);

   return 0;
}
```

```
void interchange(int u, int v)          /* объявление функции */
{
   int temp;

   temp = u;
   u = v;
   v = temp;
}
```

При выполнении этой программы получены следующие результаты

```
Первоначально x = 5 и y = 10.
Теперь x = 5 и y = 10.
```

Ну и дела! Значения не поменялись! Чтобы выяснить, в чем дело, поместим в функцию `interchange()` операторы печати (см. листинг 9.14).

Листинг 9.14. Программа `swap2.c`

```
/* swap2.c -- пересмотренный вариант программы swap1.c */
#include <stdio.h>

void interchange(int u, int v);

int main(void)
{
   int x = 5, y = 10;

   printf("Первоначально x = %d и y = %d.\n", x , y);
   interchange(x, y);
   printf("Теперь x = %d и y = %d.\n", x, y);

   return 0;
}

void interchange(int u, int v)
{
   int temp;

   printf("Первоначально u = %d и v = %d.\n", u , v);
   temp = u;
   u = v;
   v = temp;
   printf("Теперь u = %d и v = %d.\n", u, v);
}
```

Выполнение этой программы привело к получению следующих выходных данных:

```
Первоначально x = 5 и y = 10.
Первоначально u = 5 и v = 10.
Теперь u = 10 и v = 5.
Теперь x = 5 и y = 10.
```

В функции `interchange()` ошибок нет — она меняет местами значения `u` и `v`. Проблема заключается в передаче результатов в функцию `main()`. Как уже отмечалось выше, функция `interchange()` использует другие переменные из функции `main()`, так что обмен значениями между переменными `u` и `v` никак не сказывается на переменных `x` и `y`!

Может быть, следует воспользоваться оператором return? Итак, вы можете завершить выполнение функции interchange() строкой

```
return(u);
```

а затем завершить вызов в функции main() следующим оператором:

```
x = interchange(x,y);
```

Такая замена приводит к тому, что x получает новое значение, но y при этом остается неизменной. С помощью оператора return вы можете переслать обратно в вызывающую функцию всего лишь одно значение, а вам надо передать два значения. И все-таки их можно передать! Нужно только воспользоваться указателями.

Указатели: первое знакомство

Указатели? А что это такое? По сути дела, *указатель* представляет собой переменную (или, в общем случае, объект данных), значением которой является адрес. Подобно тому, как переменная типа char в качестве значения имеет символ, а переменная типа int — целое число, указатель принимает значение адреса. Указатели имеют многочисленные применения в языке C; в данной главе будет показано, как и почему они используются в качестве параметров функций.

Если конкретной переменной типа указателя присвоить имя ptr, можно будет пользоваться такими операторами, как показанный ниже:

```
ptr = &pooh;        /* присваивает адрес переменной pooh переменной ptr */
```

Мы говорим, что переменная ptr "указывает на" pooh. Различие между ptr и &pooh состоит в том, что ptr является переменной, а &pooh — константой. При необходимости вы можете сделать так, чтобы ptr указывал в любое место памяти:

```
ptr = &bah;         /* переменная ptr указывает на bah вместо pooh */
```

В данном случае значением переменной ptr является адрес переменной bah.

Чтобы создать переменную-указатель, вы должны объявить ее тип. Предположим, что вы хотите объявить переменную ptr с таким расчетом, чтобы она могла хранить адрес значения типа int. Чтобы сделать такое объявление, вы должны использовать новую операцию. Рассмотрим эту операцию.

Операция разыменования: *

Предположим, что вы знаете, что ptr указывает на переменную bah, как показано ниже:

```
ptr = &bah;
```

Теперь вы можете воспользоваться операцией разыменования * (она еще называется операцией *снятия косвенности*), чтобы найти значение, сохраняемое в переменной bah (не путайте эту унарную операцию с бинарной операцией умножения *):

```
val = *ptr; /* определить, на какое значение указывает указатель ptr */
```

Операторы ptr = &bah; и val = *ptr; , взятые вместе, эквивалентны следующему оператору:

```
val = bah;
```

Использование операции адресации и разыменования представляют собой косвенный путь достижения этого результата, отсюда и происходит название "операция разыменования".

Сводка: операции с указателями

Операция адресации:

`&`

Комментарии общего характера:

Знак `&`, за которым следует имя переменной, представляет адрес этой переменной.

Пример:

`&nurse` является адресом переменной `nurse`.

Оператор разыменования:

`*`

Комментарии общего характера:

Если за знаком `*` следует имя или адрес, то он передает значение, которое хранится по указанному адресу.

Пример:

```
nurse = 22;
ptr = &nurse;    /* указатель на переменную nurse */
val = *ptr;      /* присваивает переменной val значение,
                    хранящееся в ячейке ptr */
```

В конечном итоге переменная `val` получает значение `22`.

Объявление указателей

Вы уже знаете, как объявлять переменные типа `int` и других фундаментальных типов. Как объявляются переменные типа указатель? Вы, должно быть, подумали, что эта форма должна иметь вид:

```
pointer ptr;      /* Неправильный способ объявлять указатели */
```

Почему не может? Поскольку недостаточно объявить, что та или иная переменная является указателем. Вы также должны указать вид переменной, на которую указывает указатель. Причина состоит в том, что различные типы переменных требуют для своего размещения различные объемы памяти, а некоторые операции над указателями требуют сведений о размере памяти, отведенной под переменные. Наряду с этим, программа должна знать, какой вид данных хранится по указанному адресу. Типы `long` и `float` могут использовать один и тот же объем данных, однако способы хранения этих чисел существенно различаются. Вот как выполняется объявление указателей:

```
int * pi;         /* pi - указатель на целочисленную переменную */
char * pc;        /* pc - указатель на символьную переменную */
float * pf, * pg; /* pf, pg - указатели на переменные с плавающей запятой*/
```

Спецификация типа задает тип переменной, на которую указывает указатель, а звездочка (`*`) означает, что переменная сама по себе является указателем. Объявление `int * pi;` говорит о том, что значение `*pi` имеет тип `int` (рис. 9.5).

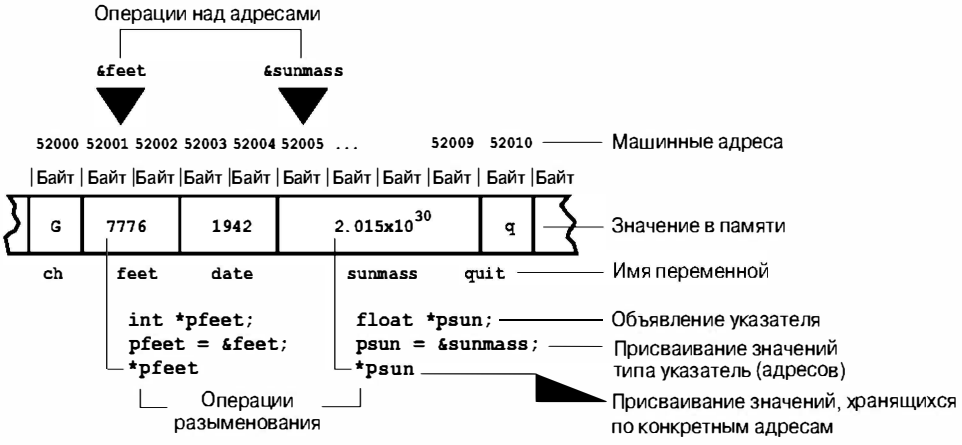


Рис. 9.5. *Объявление и использование указателей*

Пробел между знаком * и именем указателя не обязателен. Часто программисты используют пробел при объявлении и опускают его при разыменовании переменной.

Значение (*pc), на которое указывает указатель pc, имеет тип char. Что можно сказать о самом указателе pc? Мы описываем его как имеющий тип "указателя на значение типа char". Значение pc — это адрес, который имеет внутреннее представление во многих системах в виде целого числа без знака. Однако вы не должны рассматривать указатель как целочисленное значение. Существуют операции, которые можно выполнять над целыми числами, но нельзя над указателями и наоборот. Например, вы можете умножать одно целое число на другое, но вы не можете умножать указатели. Следовательно, указатель и в самом деле представляет собой новый тип, отличный от целочисленного. Таким образом, как уже было указано выше, стандарт ANSI C предусматривает форму %p специально для указателей.

Использование указателей для обмена данными между функциями

Мы лишь слегка коснулись многообразного и удивительного мира указателей, в то же время нас, прежде всего, интересует использование указателей для решения проблемы обмена данными. В листинге 9.15 представлена программа, которая использует указатели с тем, чтобы заставить работать функцию interchange(). Ознакомимся с этой программой, выполним ее и разберемся, как она работает.

Листинг 9.15. Программа *swap3.c*

```
/* swap3.c — использование указателей для обмена значениями переменных */
#include <stdio.h>
void interchange(int * u, int * v);
int main(void)
{
   int x = 5, y = 10;
   printf("Первоначально x = %d и y = %d.\n", x, y);
   interchange(&x, &y);                    /* передача адресов в функцию */
```

```
   printf("Теперь x = %d и y = %d.\n", x, y);
   return 0;
}
void interchange(int * u, int * v)
{
   int temp;
   temp = *u; /*переменная temp получает значение, на которое указывает u*/
   *u = *v;
   *v = temp;
}
```

Будет ли программа из листинга 9.15 работать после сборки?

```
Первоначально x = 5 и y = 10.
Теперь x = 10 и y = 5.
```

Да, она работает.

Теперь посмотрим, как работает программа из листинга 9.15. Во-первых, вызов функции имеет вид:

```
interchange(&x, &y);
```

Вместо передачи *значений* переменных `x` и `y` в функцию передаются их *адреса*. Это означает, что формальные аргументы `u` и `v`, появляющиеся в прототипе и определении функции `interchange()`, используют адреса в качестве своих значений. В силу этого они должны быть объявлены как указатели. Поскольку `x` и `y` — целые числа, `u` и `v` — указатели на целые значения, они должны быть объявлены следующим образом:

```
void interchange (int * u, int * v)
```

Далее, в теле функции содержится объявление

```
int temp;
```

которое отражает потребность во временной памяти. Чтобы сохранить значение `x` в переменной `temp`, воспользуйтесь оператором

```
temp = *u;
```

Напомним, что `u` имеет значение `&x`, так что `u` указывает на `x`. Это означает, что `*u` предоставляет вам значение `x`, именно это и было нужно. Не следует использовать такой оператор

```
temp = u;          /* Неправильно */
```

так как в этом случае переменной `temp` присваивается адрес переменной `x`, а не ее значение, а наша задача состоит в том, чтобы осуществить обмен значениями, а не адресами.

Аналогично, чтобы назначить переменной значение `x`, воспользуйтесь оператором

```
*u = *v;
```

который, по сути дела, дает следующий результат:

```
x = y;
```

Подведем итоги этого примера. Нам нужна была функция, которая меняет значения переменных `x` и `y`. Передавая функции адреса `x` и `y`, мы предоставляем функции `interchange()` доступ к этим переменным. Используя указатели и операцию `*`, эта функция может проверить, какие значения хранятся в этих ячейках, и поменять их.

В прототипе ANSI вы можете опустить имена переменных. Тогда объявление в прототипе принимает следующий вид:

```
void interchange(int *, int *);
```

В общем случае вы можете передать в функцию два вида информации о переменной. Если вызов имеет вид:

```
function1(x);
```

то вы передаете значения переменной x. Если вы вызов имеет вид

```
function2(&x);
```

вы передаете адрес переменной x. Первая форма требует, чтобы определение функции включало формальный аргумент того же типа, что и тип переменной x:

```
int function1(int num)
```

Вторая форма требует, чтобы определение функции включало формальный параметр, который является указателем на правильный тип:

```
int function2(int * ptr)
```

Если функции нужно некоторое значение для соответствующих вычислений или действий, используйте первую форму. Если требуется, чтобы вызываемая функция изменила значения переменных вызывающей функции, воспользуйтесь второй формой. Вы выполняли подобного рода операции с помощью функции `scanf()`. Когда вы хотите прочитать значение той или иной переменной (например, переменной `num`), вы используете вызов `scanf("%d", &num)`. Эта функция читает значение, а затем использует указанный адрес для сохранения значения.

Указатели позволяют обойти тот факт, что переменные функции `interchange()` являются локальными. Они позволяют этой функции изменять то, что хранится внутри `main()`. Пользователи, работающие с языками Pascal и Modula-2, могут обратить внимание на тот факт, что первая форма аналогична параметру-значению в Pascal, а вторая форма подобна (но не идентична) параметру-переменной того же языка. Пользователям, работающим с языком BASIC, все эти построения могут показаться несколько искусственными. Если материал, изложенный в данном разделе, покажется непонятным, не сомневайтесь в том, что после непродолжительной практики использование указателей станет самым обычным делом, простым и удобным (рис. 9.6).

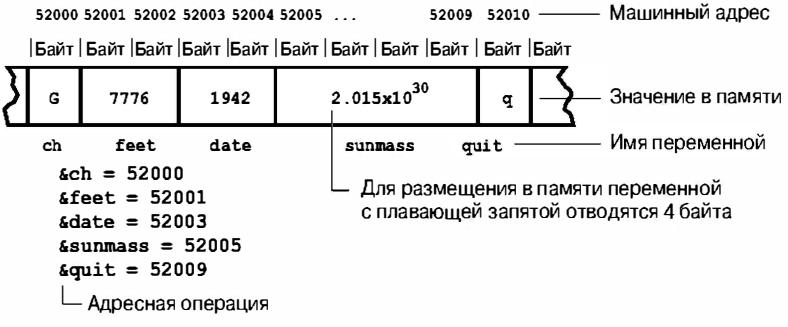


Рис. 9.6. *Имена, адреса и значения в системе байтовой адресации, такой как персональный компьютер IBM PC*

Переменные: имена, адреса и значения

Изучение указателей было прервано на анализе отношений между именами, адресами и значениями переменных. Сейчас изучение будет продолжено.

Когда вы пишете программу, вы считаете, что переменная имеет два атрибута: имя и значение. (Существуют также и другие атрибуты, в том числе тип, но это отдельная тема.) После того, как программа будет откомпилирована и загружена, компьютер воспринимает ту же переменную как сущность, имеющую два аргумента: адрес и значение. Адрес представляет собой компьютерную версию имени.

Во многих языках программирования вопросы адресации решаются компьютером, и к этому процессу программист не имеет доступа. Однако в языке C вы получаете доступ к адресам с помощью операции `&`.

Например, `&barn` — это адрес переменной `barn`.

Вы можете получить значение из имени, для этого достаточно просто воспользоваться именем. Например, `printf("%d\n", barn)` выводит значение переменной `barn`.

Вы можете получить значение переменной по ее адресу, воспользовавшись операцией `*`.

В операторе `pbarn = &barn;` конструкция `*pbarn` — это значение, хранимое по адресу `&barn`.

Короче говоря, обычная переменная в качестве первичной количественной величины использует значение, в то время как адрес через операцию `&` становится производной величиной. Переменная типа указатель делает адрес первичной количественной величиной, а производной величиной с помощью операции `*` становится адрес.

И хотя ради любопытства вы можете распечатывать адреса, операция `&` предназначена в основном не для этого. Гораздо важнее, что использование операций `&`, `*` и указателей позволяет символически манипулировать адресами и их содержимым, как это имеет место в программе `swap3.c` (листинг 9.15).

Сводка: функции

Форма:

Типичное определение функции в стандарте ANSI C имеет следующую форму:

имя (*список объявлений аргументов*)
тело функции

Список объявлений аргументов — это список переменных, в котором имена переменных отделяются друг от друга запятыми. Переменные, отличные от параметров функций, объявляются в теле функции, ограниченном фигурными скобками.

Пример:

```
int diff(int x, int y)          // стандарт ANSI C
{                               // начало тела функции
   int z;                       // объявление локальных переменных
   z = x - y;
   return z;                    // возвращение значений
}                               // конец тела функции
```

Передача значений:

Аргументы используются для передачи значений из вызывающей функции в вызываемую. Если переменные `a` и `b` имеют значения, соответственно, 5 и 2, вызов

```
c = diff(a,b);
```

передает 5 и 2 переменным x и y. Значения 5 и 2 называются *фактическими аргументами*, а переменные x и y функции diff() — *формальными параметрами*. Ключевое слово return передает одно значение из функции в вызывающую функцию. В рассматриваемом примере c получает значение переменной z, которое равно 3. Обычно функция не влияет на переменную вызывающей функции. Чтобы непосредственно менять значения переменных вызывающей функции, используйте указатели в качестве аргументов. Это может оказаться необходимым, если вы хотите возвратить больше одного значения в вызывающую функцию.

Возвращаемый тип функции:

Возвращаемый тип функции указывает тип значения, возвращаемого функцией. Если возвращаемое значение имеет тип, отличный от объявленного возвращаемого типа, это значение преобразуется к объявленному типу.

Пример:

```
int main(void)
{
   double q, x, duff();       /* объявления в вызывающей функции */
   int n;
   ...
   q = duff(x,n);
   ...
}
double duff(u, k)             /* объявления в определении функции */
double u;
int k;
{
   double tor;
   ...
   return tor;                /* возврат значения типа double */
}
```

Ключевые понятия

Если вы хотите успешно и эффективно программировать в языке C, то обязательно должны знать, как работают функции. Очень полезно, и даже необходимо, организовывать крупные программы в виде совокупности нескольких функций. Если вы будете следовать правилу, назначающему одну функцию одной задаче, вашу программу легче будет понять и отладить. Выясните во всех подробностях, как функции обмениваются информацией, то есть убедитесь, что вы понимаете, как работают аргументы и возвращаемые значения функции. Кроме того, убедитесь в том, насколько параметры функции и другие локальные переменные приватны для функции, то есть объявление двух переменных под одним и тем же именем в разных функциях создает две разных переменных. Наряду с этим дна функция не имеет прямого доступа к переменным, объявленным в другой функции. Такой ограниченный доступ позволяет сохранить целостность данных. В то же время, если вам нужно, чтобы одна функция имела доступ к данным другой функции, вы можете использовать аргументы функции с типом указатель.

Резюме

Используйте функции как строительные блоки крупных программ. Каждая функция должна решать единственную, четко определенную задачу. Воспользуйтесь аргументами для передачи значений функции, а для возвращения результата выполнения функции применяйте ключевое слово `return`. Если функция возвращает значение, тип которого не `int`, вы должны описать тип функции в описании этой функции в разделе объявлений вызывающей функции. Если вы хотите, чтобы функция оказывала воздействие на переменные вызывающей функции, используйте адреса и указатели.

Стандарт ANSI C предлагает *прототипирование функций*, мощное усовершенствование языка C, позволяющее компиляторам проверять, правильно ли указано количество и типы аргументов в вызове функции. Функция C способна вызывать сама себя, это свойство называется *рекурсией*. Некоторые задачи программирования можно решать методом рекурсии, но этот метод может оказаться неэффективным с точки зрения использования памяти и времени.

Вопросы для самоконтроля

1. В чем заключается отличие между фактическим аргументом и формальным параметром?
2. Напишите заголовок функции на ANSI C для описанных ниже функций. Обратите внимание, что речь сейчас идет о заголовках, а не о теле функций.
   а. Функция `donut()` принимает аргумент типа `int` и выводит на печать количество нулей, равное значению этого аргумента.
   б. Функция `gear()` принимает два аргумента типа `int` и возвращает значение типа `int`.
   в. Функция `stuff_it()` принимает значение тип `double` и адрес переменной типа `double` и запоминает первое значение в заданной ячейке.
3. Напишите заголовок функции на ANSI C для описанных ниже функций. Обратите внимание на то, что речь сейчас идет о заголовках, а не о теле функций.
   а. Функция `n_to_char()` принимает аргумент типа `int` и возвращает значение типа `char`.
   б. Функция `digits()` принимает значение тип `double` и аргумент типа `int` и возвращает значение типа `int`.
   в. Функция `random()` не имеет аргументов и возвращает значение типа `int`.
4. Создайте функцию, которая возвращает сумму двух целых чисел.
5. Что изменится и изменится ли что-нибудь, если вы потребуете, чтобы функция, описанная в пункте 4, складывала вместо двух целых чисел два числа типа `double`?
6. Создайте функцию с именем `alter()`, которая принимает две переменные `x` и `y` типа `int` и присваивает этим переменным, соответственно, значения их суммы и разности.
7. Нет ли ошибок в следующем определении функции?

```
void salami(num)
{
   int num, count;
   for (count = 1; count <= num; num++)
      printf(" O salami mio!\n");
}
```

8. Напишите функцию, которая возвращает наибольший из трех аргументов.
9. Заданы следующие выходные данные:

```
Выберите один из следующих вариантов:
1) копировать файлы              2) переместить файлы
3) удалить файлы                 4) выйти из программы
Введите номер выбранного варианта:
```

   а. Напишите функцию, которая выводит на экран меню из четырех пронумерованных вариантов выбора и предлагает вам выбрать один из них. (Выходные данные должны иметь показанный выше вид.)

   б. Напишите функцию, которая имеет два аргумента типа `int`: значения верхней и нижней границы. Функция должна читать целое число из входных данных. Если это число выходит за пределы указанных границ, функция снова должна вывести меню на экран (воспользовавшись функцией из пункта а) выше) с целью выдачи повторного приглашения пользователю ввести новое значение. Если будет введено новое значение, попадающее в диапазон, заданный граничными значениями, функция должна возвратить это значение.

   в. Напишите минимальную программу, используя функции из пунктов а) и б) данного вопроса. Под *минимальной* программой мы подразумеваем то, что она по сути дела не должна выполнять действия, объявленные в меню, достаточно предложить пользователю сделать выбор в меню и получить правильный ответ.

Упражнения по программированию

1. Напишите функцию с именем `min(x,y)`, которая возвращает меньшее из двух значений типа `double`. Протестируйте эту функцию с помощью простого драйвера.
2. Напишите функцию с именем `chline(ch,i,j)`, которая печатает требуемый символ в столбцах от `i` до `j`. Протестируйте эту функцию с помощью простого драйвера.
3. Напишите функцию, которая принимает три аргумента: символ и два целых числа. Символ должен быть распечатан в строке. Первое целое значение определяет, сколько раз символ печатается в строке, а второе целое число задает, сколько строк должно быть распечатано. Напишите программу, которая использует эту функцию.
4. Среднее гармоническое значение двух чисел может быть получено, если взять обратные величины этих двух чисел, вычислить их среднее значение и взять его обратную величину. Напишите функцию, которая берет два аргумента типа `double` и возвращает среднее гармоническое значение этих двух чисел.

5. Напишите и протестируйте функцию с именем `larger_of()`, которая заменяет содержимое двух переменных типа `double` большим из двух этих значений. Например, функция `larger_of(x,y)` присвоит обеим переменным `x` и `y` большее из двух этих значений.
6. Напишите программу, которая считывает символы из стандартного устройства ввода до тех пор, пока не встретится символ конца файла. Программа должна сообщить, когда вводимый символ является буквой. Когда вводимый символ есть буква, она сообщает также ее порядковый номер в алфавите. Например, буквы *c* и *C* идут под номером 3. Включите также функцию, которая принимает символ в качестве аргумента и возвращает его номер в алфавите, если этот символ является буквой, и 1 в противном случае.
7. В главе 6 рассматривается функция `power()` (листинг 6.20), которая возвращает результат возведения чисел типа `double` в положительную целую степень. Внесите в эту функцию такие изменения, которые позволили бы использовать ее для возведения чисел в отрицательные степени. Наряду с этим, внесите в эту функцию такие изменения, которые при возведении 0 в любую степень давали бы в результате 0, а при возведении любого числа в степень 0 выдавали бы в качестве результата 1. Воспользуйтесь циклом. Протестируйте эту функцию на примере конкретной программы.
8. Еще раз выполните упражнение 7, но с использованием рекурсивной функции.
9. Раширьте функцию `to_binary()`, представленную в листинге 9.8, до функции `to_base_n()`, которая принимает второй аргумент в диапазоне от 2 до 10. Она должна выводить на печать число, которое является ее первым аргументом, в системе счисления, основание которой задается ее вторым аргументом. Например, вызов `to_base_n(129,8)` выведет на экран 201, эквивалент числа 129 в восьмеричной системе. Протестируйте полученную функцию в составе завершенной программы.
10. Напишите и протестируйте функцию `Fibonacci()`, в которой для вычисления чисел Фибоначчи используется цикл, а не рекурсия.

ГЛАВА 10

Массивы и указатели

В этой главе:

- Ключевое слово: `static`
- Операции: `&` `*` (унарные)
- Создание и инициализация массивов
- Указатели (на основе сведений, которые вам уже известны) и какое отношение они имеют к массивам
- Написание функций, обрабатывающих массивы
- Двумерные массивы

Люди обращаются за помощью к компьютеру при решении таких задач, как подсчет ежемесячных расходов, расчет ежедневного количества осадков, ежеквартальных продаж, еженедельного прироста веса и так далее. Предприятия обращаются к помощи компьютера при составлении платежных ведомостей, при учете материально-производственных запасов и при расчетах с заказчиками. Будучи программистом, вы неизбежно имеете дело с большими объемами соответствующих данных. Довольно часто массивы предлагают наилучшие способы манипулирования такими данными — удобные и эффективные. В главе 6 было введено понятие массива, а в этой главе мы изучим массивы более подробно. В частности, мы рассмотрим функции, выполняющие обработку массивов. Такие функции предоставляют вам возможность распространить преимущества модульного программирования на массивы. Изучая материал этой главы, вы сами можете убедиться в том, насколько тесно связаны между собой массивы и указатели.

Массивы

Вспомните, что *массив* образует некоторая последовательность элементов одного и того же типа данных. Вы используете *объявления* с целью уведомить компилятор о том, что вам необходим массив. *Объявление массива* сообщает компилятору, сколько элементов содержит массив и какой тип имеют его элементы. Располагая такого рода информацией, компилятор может правильно создать массив. Элементы массива могут иметь те же типы, что и обыкновенные переменные. Рассмотрим следующий пример объявления массива:

```
/* несколько объявлений массивов */
int main(void)
{
   float candy[365];      /* массив из 365 значений типа float */
   char code[12];         /* массив из 12 значений типа char */
   int states[50];        /* массив из 50 значений типа int */
   ...
}
```

Квадратные скобки ([]) свидетельствуют о том, что candy и другие такие же структуры данных являются массивами, а число, заключенное в квадратные скобки, задает количество элементов в массиве.

Чтобы получит доступ к элементам массива, вы должны указать отдельный элемент, используя для этой цели его номер, который также называется *индексом*. Нумерация элементов массива начинается с 0. Следовательно, candy[0] — это первый элемент массива candy, а candy[364] — 365-й, он же последний, элемент массива.

Все это нам уже известно; продолжим далее изучение массивов.

Инициализация

Массивы часто используются для хранения данных, необходимых программе. Например, каждый элемент 12-элементный массива может содержать количество дней в соответствующем месяце. В случаях, подобных данному, удобно выполнить инициализацию массива в начале программы. Посмотрим, как это делается. Вы уже знаете, как выполняется инициализация однозначных переменных (иногда они называются *скалярными*) в объявлениях с выражениями наподобие

```
int fix = 1;
float flax = PI * 2;
```

в которых, как можно надеяться, константа была определена ранее. Язык C расширяет инициализацию на массивы с помощью новых синтаксических средств, как показано ниже:

```
int main(void)
{
   int powers[8] = {1,2,4,6,8,16,32,64};   /* только в стандарте ANSI */
   ...
}
```

Нетрудно видеть, что вы инициализируете массив с применением списка элементов, отделенных друг от друга запятыми, заключенного в квадратные скобки. Между запятыми и значениями при желании можно вставлять пробелы. Первому элементу (powers[0]) присваивается значение 1 и так далее. (Если ваш компилятор отказывается выполнять такую форму инициализации, рассматривая ее как синтаксическую ошибку, значит, компилятор был разработан до появления стандарта ANSI. Проблема решается путем помещения перед объявлением массива ключевого слово static. В главе 12 значение этого ключевого слова обсуждается более подробно.) В листинге 10.1 показана короткая программа, которая выводит на печать количество дней каждого месяца.

Листинг 10.1. Программа `day_mon1.c`

```
/* day_mon1.c -- выводит на печать количество дней каждого месяца */
#include <stdio.h>
#define MONTHS 12
int main(void)
{
   int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
   int index;
   for (index = 0; index < MONTHS; index++)
      printf("Месяц %d имеет %2d дней (день).\n", index + 1,
             days[index]);
   return 0;
}
```

Выходные данные имеют следующий вид:

```
Месяц 1 имеет 31 дней (день).
Месяц 2 имеет 28 дней (день).
Месяц 3 имеет 31 дней (день).
Месяц 4 имеет 30 дней (день).
Месяц 5 имеет 31 дней (день).
Месяц 6 имеет 30 дней (день).
Месяц 7 имеет 31 дней (день).
Месяц 8 имеет 31 дней (день).
Месяц 9 имеет 30 дней (день).
Месяц 10 имеет 31 дней (день).
Месяц 11 имеет 30 дней (день).
Месяц 12 имеет 31 дней (день).
```

Не ахти какая программа, однако, она ошибается только один раз в четыре года. Программа инициализирует массив `days[]` с использованием списка значений, отделенных друг от друга запятыми, заключенного в квадратные скобки.

Обратите внимание на то, что в этом примере присутствует символическая константа `MONTHS` для представления размера массива. Это распространенная и рекомендованная практика. Например, если вдруг мир перейдет на 13-месячный календарь, достаточно будет внести соответствующее изменение в оператор `#define`, и не нужно будет выискивать в программе все места, в которых используется размер массива.

Использование констант в массивах

Иногда вам приходится использовать массив, предназначенный только для чтения. То есть программа извлекает значения из массива, но не предпринимает попыток записывать новые значения в этот массив. В таких случаях вы можете и должны использовать ключевое слово `const` во время объявления и инициализации массива. С учетом сказанного выше, в программе, представленной в листинге 10.1, лучше воспользоваться следующей конструкцией:

```
const int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Это позволяет программе рассматривать каждый элемент массива как константу. Как и в случае обычных переменных, вы должны использовать объявление для инициализации данные типа `const`, поскольку если они объявлены как `const`, вы не можете впоследствии присваивать им новые значения. Теперь, когда мы знаем об этом, мы можем использовать константы в последующих примерах.

Что произойдет, если вы не сможете инициализировать массив? Об этом мы узнаем после исследования программы из листинга 10.2.

Листинг 10.2. Программа `no_data.c`

```
/* no_data.c -- неинициализированный массив */
#include <stdio.h>
#define SIZE 4
int main(void)
{
   int no_data[SIZE];      /* неинициализированный массив */
   int i;
   printf("%2s%14s\n",
          "i", "no_data[i]");
   for (i = 0; i < SIZE; i++)
      printf("%2d%14d\n", i, no_data[i]);
   return 0;
}
```

Вот один из примеров выходных данных этой программы (результаты могут меняться):

```
i   no_data[i]
0           16
1      4204937
2      4219854
3   2147348480
```

Элементы массива мало чем отличаются от обычных переменных, и если вы не инициализируете их, они могут принимать произвольные значения. Компилятор использует те значения, которые уже были записаны в соответствующих ячейках памяти, вот почему ваши результаты могут отличаться от представленных выше.

Учет различных классов памяти

Массивы, как и все другие переменные, можно создавать с использованием *классов памяти*. Эта тема рассматривается в главе 12, однако, сейчас вам достаточно знать, что в текущей главе описаны массивы, которые относятся к автоматическому классу памяти. Это означает, что они объявлены внутри функции без употребления ключевого слова `static`. Все переменные и массивы, использовавшиеся до сих пор в этой книге, относятся к автоматическому классу. Мы упомянули здесь классы памяти по той простой причине, что время от времени различные классы памяти проявляют различные свойства, так что вы не должны автоматически переносить все сказанное в этой главе на другие классы памяти. В частности, переменные и массивы некоторых не рассмотренных здесь классов памяти, не будучи инициализированными, получают значение 0.

Количество элементов в списке должно соответствовать размеру массива. Но что будет, если вы ошибетесь при подсчете? Попробуем еще раз выполнить последний пример, как показано в листинге 10.3, включив список, который в два раза короче объявленного размера массива.

Листинг 10.3. Программа somedata.c

```
/* some_data.c -- частично инициализированный массив */
#include <stdio.h>
#define SIZE 4

int main(void)
{
   int some_data[SIZE] = {1492, 1066};
   int i;

   printf("%2s%14s\n",
          "i", "some_data[i]");
   for (i = 0; i < SIZE; i++)
      printf("%2d%14d\n", i, some_data[i]);

   return 0;
}
```

На этот раз выходные данные приобретают следующий вид:

```
i   some_data[i]
0           1492
1           1066
2              0
3              0
```

Нетрудно убедиться в том, что у компилятора проблем не было. Как только значения в списке закончатся, остальные элементы он инициализирует нулями. Иначе говоря, если вы вообще не выполняете инициализацию массива, его элементы, подобно обычным переменным, получают произвольные значения мусора в памяти, в то же время, если вы выполняете частичную инициализацию массива, оставшимся неинициализированным элементам присваиваются нулевые значения.

Но компиляторы не простят вам ошибку, если вы укажете список, в котором значений больше, чем размер объявленного массива. Такая чрезмерная щедрость рассматривается как ошибка. Тем не менее, нет необходимости выставлять себя мишенью для насмешек со стороны собственного компилятора. Вместо этого вы можете предоставить компилятору возможность привести в соответствие размер массива со списком, исключив размер из квадратных скобок (листинг 10.4).

Листинг 10.4. Программа day_mon2.c

```
/* day_mon2.c -- компилятор сам подсчитывает количество элементов */
#include <stdio.h>

int main(void)
{
   const int days[] = {31,28,31,30,31,30,31,31,30,31};
   int index;

   for (index = 0; index < sizeof days / sizeof days[0]; index++)
      printf("Месяц %2d имеет %d дней (день).\n", index +1,
             days[index]);

   return 0;
}
```

В листинге 10.4 необходимо отметить два момента:

- Когда вы освобождаете квадратные скобки с целью инициализации массива, компилятор проводит подсчет элементов списка и устанавливает размер массива равным этому числу.
- Обратите внимание на то, как был скорректирован управляющий оператор цикла `for`. Не имея уверенности в том, что нам удастся выполнить этот подсчет правильно (что вполне объяснимо), мы предоставляем право компьютеру самостоятельно определить размер массива. Операция `sizeof` возвращает размер в байтах объекта, или *типа*, следующего за ним. Таким образом, `sizeof days` — это размер в байтах всего массива, а `sizeof days[0]` — размер в байтах одного элемента этого массива. Выполнив деление размера всего массива на размер одного элемента, мы узнаем, сколько элементов содержится в массиве.

Ниже показан результат выполнения этой программы:

```
Месяц 1 имеет 31 дней (день).
Месяц 2 имеет 28 дней (день).
Месяц 3 имеет 31 дней (день).
Месяц 4 имеет 30 дней (день).
Месяц 5 имеет 31 дней (день).
Месяц 6 имеет 30 дней (день).
Месяц 7 имеет 31 дней (день).
Месяц 8 имеет 31 дней (день).
Месяц 9 имеет 30 дней (день).
Месяц 10 имеет 31 дней (день).
```

Вот как! Вывелось лишь 10 значений, а использованный нами метод, позволяющий программе самостоятельно определить размер массива, не предоставил возможности распечатать массив до конца. Это подчеркивает потенциальный недостаток автоматизированного подсчета: ошибочное количество элементов может оказаться необнаруженным.

Существует еще один более короткий метод инициализации массивов. Однако, поскольку его применение возможно только в отношении символьных строк, мы отложим его рассмотрение до следующей главы.

Выделенные инициализаторы (стандарт C99)

Стандарт C99 добавляет в язык C новую возможность — *выделенные инициализаторы*. Это свойство позволяет выбирать, какие элементы должны быть инициализированы. Предположим, например, что вы хотите инициализировать только последний элемент массива. Полагаясь только на традиционные синтаксические средства инициализации языка C, вы должны также инициализировать и все элементы, предшествующие последнему:

```
int arr[6] = {0,0,0,0,0,212};          // традиционный синтаксис
```

В условиях действия стандарта C99 вы можете использовать индекс в квадратных скобках в списке инициализации при описании некоторого конкретного элемента:

```
int arr[6] = {[5] = 212}; // инициализировать элемент arr[5] значением 212
```

Как и при обычной инициализации, после того, как вы выполните инициализацию, по меньшей мере, одного элемента, инициализированные элементы получают значение 0. В листинге 10.5 представлен более сложный пример.

Листинг 10.5. Программа `designate.c`

```
// designate.c -- использование выделенных инициализаторов
#include <stdio.h>
#define MONTHS 12

int main(void)
{
   int days[MONTHS] = {31,28, [4] = 31,30,31, [1] = 29};
   int i;

   for (i = 0; i < MONTHS; i++)
      printf("%2d %d\n", i + 1, days[i]);

   return 0;
}
```

Ниже приведены выходные данные для случая, когда компилятор поддерживает это свойство стандарта C99:

```
1   31
2   29
3    0
4    0
5   31
6   30
7   31
8    0
9    0
10   0
11   0
12   0
```

Эти выходные данные отражают два важных свойства выделенных инициализаторов. Во-первых, если за выделенным инициализатором следует код с дальнейшими значениями, как, например, в последовательности `[4] = 31,30,31`, при этом приведенные значения используются для инициализации последующих элементов. То есть, после инициализации `days[4]` значением `31` этот код инициализирует элементы `days[5]` и `days[6]`, соответственно, значениями `30` и `31`. Во-вторых, если этот код инициализирует конкретный элемент некоторым значением более одного раза, в действие вступает последняя инициализация. Например, в листинге 10.5, начало списка инициализации инициализирует элемент `days[1]` значением `28`, но это значение будет позже перекрыто выделенным инициализатором `[1] = 29`.

Присваивание значений массивам

После того, как массив будет объявлен, вы можете *присвоить* значения элементам массива, используя для этого *индекс* элемента массива. Например, следующий фрагмент программного кода присваивает (или назначает) массиву четные числовые значения:

```
/* присваивание значений массиву */
#include <stdio.h>
#define SIZE 50
int main(void)
{
   int counter, evens[SIZE];
   for (counter = 0; counter < SIZE; counter++)
      evens[counter] = 2 * counter;
   ...
}
```

Обратите внимание, что в этом коде используется цикл для поэлементного присваивания значений. Язык C не позволяет присваивать один массив в качестве значения другого как элемента данных. Нельзя также использовать форму списка, заключенного в фигурные скобки, нигде, кроме как в операторе инициализации.

В показанном ниже фрагменте программного кода демонстрируются некоторые формы недопустимых методов присваивания значений.

```
/* Недопустимые методы присваивания значений элементам массива */
#define SIZE 5
int main(void)
{
   int oxen[SIZE] = {5,3,2,8};     /* здесь все в порядке */
   int yaks[SIZE];
   yaks = oxen;                    /* недопустимый оператор */
   yaks[SIZE] = oxen[SIZE];        /* неправильно */
   yaks[SIZE] = {5,3,2,8};         /* этот метод не работает */
```

Границы массива

Вы должны убедиться в том, что используемые индексы элементов массива не выходят за допустимые пределы, другими словами, вы должны убедиться в том, что они имеют значения, разрешенные для данного массива. Например, предположим, что вы сделали следующее объявления:

```
int doofi[20];
```

В этом случае вы должны следить за тем, чтобы программа использовала индексы в диапазоне от 0 до 19, поскольку компилятор не станет делать эту проверку за вас.

Рассмотрим программу в листинге 10.6. Она создает массив из четырех элементов, а затем безответственно использует значения индексов в диапазоне от –1 до 6.

Листинг 10.6. Программа bounds.c

```
// bounds.c -- выход за границы массива
#include <stdio.h>
#define SIZE 4
int main(void)
{
   int value1 = 44;
   int arr[SIZE];
   int value2 = 88;
   int i;
```

```
    printf("value1 = %d, value2 = %d\n", value1, value2);
    for (i = -1; i <= SIZE; i++)
        arr[i] = 2 * i + 1;
    for (i = -1; i < 7; i++)
       printf("%2d %d\n", i , arr[i]);
    printf("value1 = %d, value2 = %d\n", value1, value2);
    return 0;
}
```

Компилятор не выполняет проверку правильности использования индексов в массиве. В стандартном языке C результат неправильного использования индекса не определен. Это означает, что когда вы инициируете программу, может показаться, что она работает правильно, вести себя странным образом или аварийно завершиться. Ниже показан пример выходных данных этой программы, откомпилированной с помощью Digital Mars 8.4:

```
value1 = 44, value2 = 88
-1 -1
0  1
1  3
2  5
3  7
4  9
5  5
6  1245120
value1 = -1, value2 = 9
```

Обратите внимание, что компилятор сохранил значение `value2` непосредственно после массива, а значение `value1` — непосредственно перед ним. (Другие компиляторы могут хранить данные в памяти в другом порядке.) В данном случае `arr[-1]` соответствует той же ячейке памяти, что и `value1`, а `arr[4]` — той же ячейке памяти, что и `value2`. В силу этого обстоятельства, использование индексов элементов массива, выходящих за допустимые пределы, приводит к тому, что программа меняет значения других переменных. Другой компилятор может выдать другие результаты, и одним из них может быть аварийное завершение программы.

Вы, должно быть, хотели бы знать, почему язык C допускает такие вещи. Все это является следствием философии языка C, которая предоставляет программистам большую свободу в принятии решений. Отказ от проверки выхода за допустимые пределы ускоряет выполнение программы на C. Компилятор не всегда способен отлавливать все индексные ошибки, поскольку значение индекса может оставаться неопределенным до тех пор, пока не начнется выполнение результирующей программы. В силу этого обстоятельства, чтобы обеспечить безопасность, компилятор должен добавить в программу дополнительный код для проверки каждого индекса во время выполнения, но от этого замедлится само выполнение программы. По этой причине C оставляет за программистом задачу правильного кодирования, и вознаграждает его за это увеличением быстродействия программы. Разумеется, не все программисты заслуживают такого доверия, вот тогда-то и начинаются настоящие проблемы.

Следует всегда помнить одну простую истину — нумерация элементов массива начинается с 0. Нужно также выработать в себе привычку использовать символические константы в объявлениях массивов и других местах, где используется размер массива:

```
#define SIZE 4
int main(void)
{
   int arr[SIZE];
   for (i = 0; i < SIZE; i++)
   ....
```

Это позволит обрести уверенность в том, что вы последовательно используете один и тот же размер массива в программе.

Указание размера массива

До сих пор при объявлении массивов использовались целочисленные константы:

```
#define SIZE 4
int main(void)
{
   int arr[SIZE];          // символическая целочисленная константа
   double lots[144];       // литеральная целочисленная константа
   ...
```

Что еще разрешено программисту? До появления стандарта C99 на этот вопрос можно было ответить, что при объявлении массива вы можете воспользоваться *константным целочисленным выражением*, заключив его в квадратные скобки. Константное выражение — это выражение, сформированное из целочисленных констант. В этом смысле выражение `sizeof` рассматривается как целочисленная константа, в то же время (в отличие от `case` в языке C++) значением типа `const` не является. Кроме того, значение такого выражения должно быть больше 0:

```
int n = 5;
int m = 8;
float a1[5];                 // да
float a2[5*2 + 1];           // да
float a3[sizeof(int) + 1];   // да
float a4[-4];                // нет, размер должен быть > 0
float a5[0];                 // нет, размер должен быть > 0
float a6[2.5];               // нет, размер должен быть целым числом
float a7[(int)2.5];          // да, преобразование типа из float int constant
float a8[n];                 // не разрешалось до появления стандарта C99
float a9[m];                 // не разрешалось до появления стандарта C99
```

Как показывают комментарии, компиляторы языка C, действующие в соответствие с требованиями стандарта C90, не разрешают двух последних объявлений. В то же время стандарт C99 их допускает, благодаря чему создается новый тип массивов, получивший название *массива переменной длины*.

Стандарт C99 вводит в употребление массивы переменной длины главным образом для того, чтобы увеличить возможности языка C в плане применения численных методов. Например, массивы переменной длины значительно облегчают преобразование существующих библиотек программ на языке FORTRAN, выполняющих цифро-

вые расчеты, в программы на языке C. На массивы переменной длины накладываются определенные ограничения, например, вы не можете выполнять инициализацию массива переменной длины в его объявлении. Мы еще вернемся к массивам переменной длины в этой главе несколько позже, после того, как изучим ограничения, которым в языке C подвергается классический массив.

Многомерные массивы

Мисс Темпест Клауд (Tempest Cloud — буквально, "грозовая туча"), специалист-метеоролог, исключительно серьезно относящаяся к своим профессиональным обязанностям, намеревается провести анализ данные о ежемесячных осадках за последние пять лет. Одно из ее первых решений касается выбора подходящей формы представления данных. Один из возможных вариантов состоит в использовании 60 переменных, по одной для каждого элемента данных. (Мы уже обсуждали этот вариант, нам он и сейчас кажется таким же бессмысленным, каким он казался и раньше.) Использование массива, содержащего 60 элементов, намного лучше, в то же время гораздо удобнее держать отдельно все данные, относящиеся к тому или иному конкретному году. Можно использовать пять массивов по 12 элементов, но такая попытка имеет свои недостатки и перерастает в трудно решаемую проблему, если мисс Темпест Клауд решит изучить данные о ежемесячных осадках за период длиной в 50 лет вместо пяти лет. Для этого ей потребуется более подходящая форма представления данных.

Более рациональный подход предусматривает использование массива массивов. Главный массив должен содержать пять элементов, по одному на каждый год. Каждый из этих элементов, в свою очередь, представляет собой 12-элементный массив, по одному элементу на каждый месяц. Такой массив объявляется следующим образом:

```
float rain[5][12];   // массив, состоящий из 5 массивов, каждый из
                     // которых состоит из 12 переменных типа float
```

Один из подходов к состоит в том, что сначала рассматривается внутренняя часть объявления (выделенная полужирным):

```
float rain[5][12];   // rain - это массив из 5 пока еще не известных объектов
```

Это говорит о том, что `rain` является массивом, состоящим из пяти элементов. Но что представляет собой каждый из этих элементов? Теперь рассмотрим другую часть этого объявления (выделенную полужирным):

```
float rain[5][12];   // массив из 12 значений типа float
```

Это говорит о том, что каждый элемент массива имеет тип `float[12]`, то есть каждый из этих пяти элементов массива `rain` сам по себе является массивом из 12 значений типа `float`.

В соответствие с этой логикой элемент `rain[0]`, будучи первым элементом массива `rain`, представляет собой массив из 12 значений типа `float`. Это справедливо и в отношении элементов `rain[1]`, `rain[2]` и так далее. Если `rain[0]` есть массив, его первым элементом будет `rain[0][0]`, вторым элементом — `rain[0][1]` и так далее. Короче говоря, `rain` — это пятиэлементный массив 12-элементных массивов значений типа `float`, `rain[0]` — массив из 12 элементов типа `float`, а `rain[0][0]` — значение типа `float`. Для получения доступа, скажем, к значению, находящемуся в строке 2 и столб-

це 3, используется конструкция `rain[2][3]`. (Напоминаем, что отсчет элементов массива начинается с 0, так что строка с номером 2 будет третьей.)

Вы можете рассматривать этот массив `rain` как двумерный массив, состоящий из пяти строк, при этом в каждой строке имеется 12 столбцов, как показано на рис. 10.1. Изменяя второй индекс, вы продвигаетесь по строке, месяц за месяцем. Изменяя первый индекс, вы перемещаетесь вертикально вдоль столбца, год за годом.

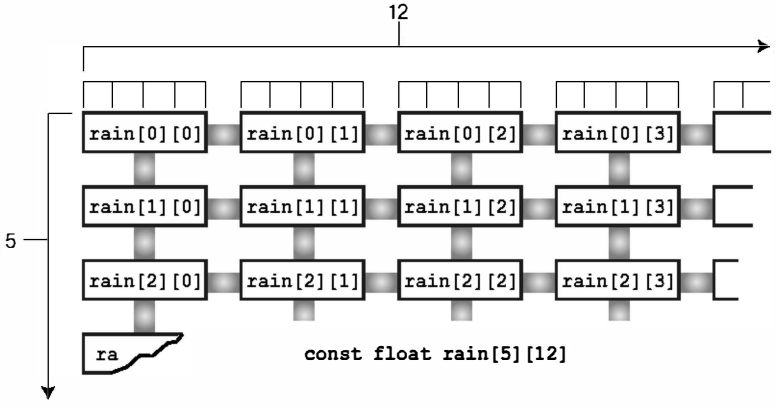


Рис. 10.1. *Двумерный массив*

Двумерное представление — это всего лишь удобный способ просмотра массива с двумя индексами. В памяти компьютера такой массив хранится последовательно, начиная с первого 12-элементного массива, за которым следует второй 12-элементный массив, и так далее.

Воспользуемся таким двумерным массивом в программе обработки погодных данных. Цель этой программы состоит в подсчете осадков за год, в вычислении средних ежегодных осадков и средних ежемесячных осадков. Чтобы вычислить общие осадки за год, нужно просуммировать все данные конкретной строки. Чтобы вычислить осадки за конкретный месяц, потребуется сложить все значения в заданном столбце. Двумерный массив упрощает визуализацию и выполнение этих действий. В листинге 10.7 показана соответствующая программа.

Листинг 10.7. Программа `rain.c`

```
/* rain.c -- вычисляет итоговые данные по годам, ежегодные средние значения
            и ежемесячные средние значения осадков за период в несколько лет*/
#include <stdio.h>
#define MONTHS 12  // количество месяцев в году
#define YEARS 5    // количество лет, в течение которых проводились наблюдения
int main(void)
{
   // инициализация массива данными об осадках за период с 2000 по 2004
   const float rain[YEARS][MONTHS] =
   {
      {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
      {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
      {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
```

```
      {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
      {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
   };
   int year, month;
   float subtot, total;

   printf(" ГОД КОЛИЧЕСТВО ОСАДКОВ (в дюймах)\n");
   for (year = 0, total = 0; year < YEARS; year++)
   {      // для каждого года суммарное количество осадков за каждый месяц
      for (month = 0, subtot = 0; month < MONTHS; month++)
         subtot += rain[year][month];
      printf("%5d %15.1f\n", 2000 + year, subtot);
      total += subtot;           // общая сумма за все годы
   }
   printf("\nСреднегодовое количество осадков составляет %.1f дюймов.\n\n",
          total/YEARS);
   printf("СРЕДНЕМЕСЯЧНОЕ КОЛИЧЕСТВО ОСАДКОВ:\n\n");
   printf(" Янв  Фев  Мар  Апр  Май  Июн  Июл  Авг  Сен  Окт");
   printf(" Ноя  Дек\n");
   for (month = 0; month < MONTHS; month++)
   {      // суммарные осадки по каждому месяцу на протяжении всего периода
      for (year = 0, subtot =0; year < YEARS; year++)
         subtot += rain[year][month];
      printf("%4.1f ", subtot/YEARS);
   }
   printf("\n");

   return 0;
}
```

Ниже показаны выходные данные этой программы:

```
ГОД КОЛИЧЕСТВО ОСАДКОВ (в дюймах)
2000            32.4
2001            37.9
2002            49.8
2003            44.0
2004            32.9
Среднегодовое количество осадков составляет 39.4 дюймов.

MONTHLY AVERAGES:

Янв  Фев  Мар  Апр  Май  Июн  Июл  Авг  Сен  Окт  Ноя  Дек
7.3  7.3  4.9  3.0  2.3  0.6  1.2  0.3  0.5  1.7  3.6  6.7
```

Во время изучения этой программы обратите особое внимание на инициализации и на схему вычислений. При этом инициализация является достаточно сложной процедурой, поэтому сначала рассмотрим более простую часть (вычисления).

Чтобы вычислить итоговую сумму за год, значение `year` остается неизменным, в то время как значение `month` пробегает весь диапазон значений. Это внутренний цикл `for` первой части программы. Затем этот процесс выполняется для следующего значения переменной `year`. Это внешний цикл первой части программы. Структура вложенного цикла вполне естественна при работе с двумерными циклами. Один цикл выполняют обработку первого индекса, а второй цикл обрабатывает второй индекс.

Вторая часть программы имеет аналогичную структуру, но на этот раз значение year меняется во внутреннем цикле, а значение month — во внешнем. Вспомните, что каждый раз, когда внешний цикл выполняет одну итерацию, внутренний цикл пробегает весь диапазон значений. По этой причине, при такой организации цикл пробегает все года, прежде чем изменится месяц. Сначала вы получаете среднегодовое значение осадков для первого месяца, затем для второго и так далее.

Инициализация двумерного массива

В основу инициализации двумерного массива положена методика инициализации одномерного массива. Во-первых, вспомните, что инициализация одномерного массива выполняется следующим образом:

```
sometype ar1[5] = {val1, val2, val3, val4, val5};
```

В рассматриваемом случае значения val1, val2 и так далее соответствуют некоторому типу sometype. Например, если бы sometype был типом int, значение val1 могло бы быть 7, или если бы sometype был бы типом double, значение val1 могло бы быть 11.34. Однако rain — это массив, состоящий из пяти элементов, причем этими элементами являются элементы в виде массивов, каждый из которых содержит 12 значений типа float. Следовательно, что касается массива rain, то в качестве val1 должно быть значение, пригодное для инициализации одномерного массива значений типа float, например, следующий массив:

```
{4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6}
```

То есть, если sometype — массив, состоящий из 12 значений типа double, то значение val1 представляет собой список 12 значений типа double. Следовательно, для инициализации двумерного массива, такого как rain, нам нужен список из пяти таких объектов, отделенных друг от друга запятыми:

```
const float rain[YEARS][MONTHS] =
{
   {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
   {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
   {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
   {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
   {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
};
```

Такая инициализация требует задания пяти списков числовых значений, заключенных в фигурные скобки. Данные, содержащиеся в первой паре фигурных скобок, присваиваются первой строке массива, данные, содержащиеся во второй паре фигурных скобок, — второй строке массива и так далее. Рассмотренные выше правила, касающиеся несоответствия размеров данных и массивов, применимы к каждой строке. Иначе говоря, если внутренняя пара фигурных скобок содержит 10 чисел, только 10 элементов получат соответствующие значения. В этом случае два последних элемента в этой строке инициализируются нулями. Если в списке слишком много чисел, это означает ошибку, и эти числа не будут распределены в следующей строке.

Можно опустить внутренние фигурные скобки и оставить две внешних скобки. Если в этом случае в скобках будет представлено корректное количество элементов, результат будет таким же. Однако, если элементов меньше, чем необходимо, массив заполняется последовательно, строка за строкой, пока не закончатся данные.

После этого оставшиеся элементы инициализируются нулями. На рис. 10.2 показаны оба способа инициализации массива.

Поскольку массив `rain` содержит данные, которые не могут быть изменены, программа использует модификатор `const` в объявлении массивов.

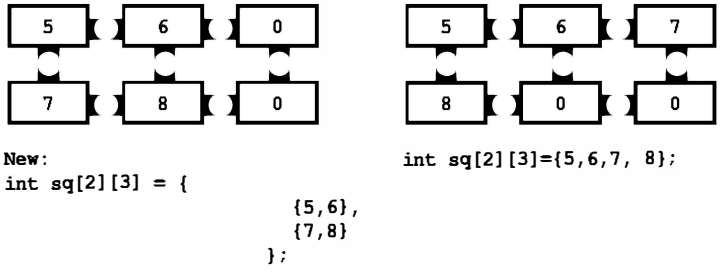


Рис. 10.2. *Два метода инициализации массива*

Массивы с размерностями больше двух

Все, что было сказано о двумерных массивах, можно распространить на трехмерные массивы и на массивы больших размерностей. Вы можете объявить трехмерный массив следующим образом:

```
int box[10][20][30];
```

Вы можете рассматривать одномерный массив как строку данных, двумерный массив — как таблицу данных, а трехмерный массив — как набор таблиц данных. Например, вы можете рассматривать многомерную таблицу как совокупность 10 двумерных массивов (размером 20×30 каждый), помещенных друг поверх друга.

Другим способом восприятия `box` является массив массивов, состоящих из массивов. Иначе говоря, это массив, элементы которого представляют собой 20-элементный массив. Каждый элемент этого 20-элементного массива представляет собой 30-элементный массив. Либо вы просто можете рассматривать массивы в терминах количества необходимых индексов.

Как правило, для обработки трехмерных массивов используется три вложенных цикла, для обработки четырехмерных массивов — циклы с глубиной вложения, равной четырем, и так далее. В наших примерах в основном мы будем пользоваться двумерными массивами.

Указатели и массивы

Указатели, как следует из главы 9, представляют собой символический способ использования адресов. Поскольку аппаратные инструкции вычислительных машин в значительной степени зависят от адресов, указатели позволяют формулировать свои намерения достаточно близко к тому, как машина выражает свои задачи. Это соответствие повышает эффективность программ, использующих указатели. В частности, указатели позволяют эффективно работать с массивами. В самом деле, как вы увидите далее, система обозначения массивов просто представляет собой особый вид использования указателей.

Примером такого особого использования может служить тот факт, что имя массива является также адресом первого элемента массива. Другими словами, если `flizny` есть массив, то приведенное ниже выражение будет истинным:

```
flizny == &flizny[0];  // именем массива является адрес первого элемента
```

Оба имени, `flizny` и `&flizny[0]` представляют адрес в памяти первого элемента массива. (Вспомните, что `&` — это операция адресации.) Оба имени являются константами, поскольку остаются фиксированными в течение всего времени действия программы. В то же время они могут быть присвоены в виде значений *переменной* типа указатель, и вы можете менять значение переменной, как показано в листинге 10.8. Обратите внимание на то, что происходит со значением указателя, когда вы прибавляете к нему число. (Как говорилось ранее, спецификатор `%p`, предназначенный для указателей, обычно отображает шестнадцатеричные значения.)

Листинг 10.8. Программа `pnt_add.c`

```
// pnt_add.c -- сложение указателей
#include <stdio.h>
#define SIZE 4

int main(void)
{
   short dates [SIZE];
   short * pti;
   short index;
   double bills[SIZE];
   double * ptf;

   pti = dates;              // назначение указателю адреса массива
   ptf = bills;
   printf("%23s %10s\n", "short", "double");
   for (index = 0; index < SIZE; index ++)
      printf("указатели + %d: %10p %10p\n",
              index, pti + index, ptf + index);

   return 0;
}
```

Ниже показаны выходные данные этой программы:

```
                    short     double
указатели + 0: 0x0064fd20 0x0064fd28
указатели + 1: 0x0064fd22 0x0064fd30
указатели + 2: 0x0064fd24 0x0064fd38
указатели + 3: 0x0064fd26 0x0064fd40
```

Во второй строке печатаются начальные адреса двух массивов, в следующей строке показан результат от прибавления к адресу 1 и так далее. Не забывайте, что адреса представлены в шестнадцатеричной форме, поэтому 30 на 1 больше, чем 2f, и на 8 больше, чем 28. Почему так?

```
0x0064fd20 + 1 дает в результате 0x0064fd22?
0x0064fd30 + 1 дает в результате 0x0064fd38?
```

Вам все еще это не понятно? Объяснение достаточно простое! В нашей системе реализована побайтная адресация памяти, в то время как тип `short` использует 2 байта, а тип `double` — 8 байтов. Происходит вот что: когда вы говорите "прибавить 1 к указателю", С добавляет одну *единицу хранения*. В случае массивов это означает, что адрес увеличивается до адреса следующего *элемента*, но не до следующего байта (рис. 10.3). Это одна из причин того, почему вы должны объявлять вид объекта, на который указывает указатель. Адреса недостаточно, поскольку компьютер должен знать, сколько нужно байтов для хранения объекта. (Это справедливо и в отношении указателей на скалярные переменные, в противном случае операция `*pt`, выбирающая значение, не будет работать правильно.)

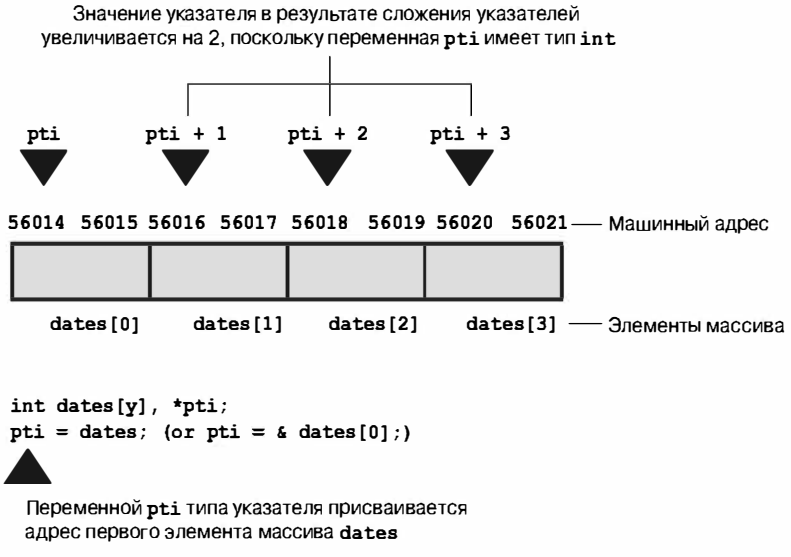


Рис. 10.3. *Массивы и сложение указателей*

Сейчас необходимо дать более четкое определение, что означают выражения "указатель на `int`", "указатель на `float`" или "указатель на любой тип".

- Значение указателя — это адрес объекта, на который он указывает. То, как адрес представлен в памяти машины, зависит от конструктивных особенностей аппаратных средств. Многие компьютеры, в том числе и персональные компьютеры IBM PC и Macintosh, имеют *байтовую адресацию*, это значит, что байты памяти пронумерованы последовательно. В подобных случаях адрес крупного объекта, такого как переменная типа `double`, как правило, представляет собой адрес первого байта объекта.
- Применяя операцию `*` к указателю, получаем значение, которое хранится в объекте, на который нацелен указатель.
- Добавление к указателю 1 увеличивает его значение на величину размера типа в байтах переменной, на которую он указывает.

Благодаря интеллектуальному характеру языка C, имеем следующие равенства:

```
dates +2 == &date[2]        /* тот же адрес   */
*(dates + 2) == dates[2]    /* то же значение */
```

Представленные отношения подводят итог тесной зависимости между массивами и указателями. Это означает, что вы можете использовать указатели для идентификации конкретного элемента массива и получения его значения. По сути, в вашем распоряжении имеются два различных обозначения одного и того же объекта. В самом деле, стандарт языка C описывает массивы через указатели. То есть, он определяет, что `ar[n]` означает `*(ar + n)`. Вы можете интерпретировать второе выражение как "перейти к ячейке памяти `ar`, пройти через `n` единиц и там найти нужное значение".

В то же время, не путайте `*(dates+2)` с `*dates+2`. Операция разыменования (*) имеет более высокий приоритет, чем +, следовательно, последнее выражение означает `(*dates)+2`:

```
*(dates +2)       /* значение третьего элемента массива dates   */
*dates +2         /* 2 прибавляется к значению первого элемента */
```

Такая зависимость между массивами и указателями означает, что вы часто можете пользоваться любым из подходов при написании программы. Программа в листинге 10.9, например, после компиляции генерирует те же выходные данные, что и программа в листинге 10.1.

Листинг 10.9. Программа `day_mon3.c`

```
/* day_mon3.c -- используются обозначения через указатели */
#include <stdio.h>
#define MONTHS 12

int main(void)
{
   int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
   int index;

   for (index = 0; index < MONTHS; index++)
      printf("Месяц %2d имеет %d дней (ltym).\n", index +1,
              *(days + index));       // то же, что и days[index]
   return 0;
}
```

В рассматриваемом случае `days` представляет собой адрес первого элемента массива, индекс `days + index` — адрес элемента `days[index]`, а `*(days + index)` — значение этого элемента, так же как и `days[index]`. Цикл поочередно просматривает элементы массива и выводит на печать значения, которые в них находит. Обладает ли какими-либо преимуществами программа, написанная подобным образом? По сути, никаких преимуществ она не предоставляет. Основное назначение программы в листинге 10.9 состоит в том, чтобы показать, что подходы с использованием записи через массивы и через указатели являются эквивалентными. Этот пример демонстрирует, что вы можете использовать для массивов систему обозначения с помощью указателей. Обратное утверждение также верно — вы можете использовать систему обозначения массивов в отношении указателей. Это имеет значение в случае работы с функцией, аргументом которой является массив.

Функции, массивы и указатели

Предположим, что вы хотите написать функцию, которая выполняет операции над массивами. Например, нужна функция, возвращающая сумму элементов массива. Предположим, что marbles — имя массива значений типа int. Какой вид будет иметь вызов такой функции? Здравый смысл подсказывает, что он должен иметь вид:

```
total = sum(marbles);              // возможный вызов функции
```

Каким должен быть прототип этой функции? Вспомните, что именем массива является адрес его первого элемента, так что фактический аргумент marbles, будучи адресом значения int, должен быть присвоен формальному параметру, каковым является указатель на тип int:

```
int sum(int * ar);                 // соответствующий прототип
```

Какую информацию получает функция sum() благодаря этому аргументу? Она получает адрес первого элемента массива, она также узнает, как найти значение int в этой ячейке. Обратите внимание на то, что эта информация ничего не говорит о количестве элементов в массиве. Мы поставлены перед выбором одного из двух вариантов продолжения определения функции. Первый вариант заключается в том, чтобы каким-то образом закодировать фиксированный размер массива в функцию:

```
int sum(int *ar)                   // соответствующее определение
{
   int i;
   int total = 0;
   for( i = 0; i < 10; i++)        // предполагается наличие 10 элементов
        total += ar[i];            // ar[i] то же, что и *(ar + i)
   return total;
}
```

В данном случае используется тот факт, что аналогично возможности применения системы обозначений через указатели к массивам, можно употреблять систему обозначений массивов к собственно указателям. Кроме того, вспомните, что операция += добавляет значение операнда, стоящего справа от знака операции, к операнду, стоящему слева от знака операции. Следовательно, итоговое значение представляет собой сумму элементов массива.

Определение этой функции ограничено; она может работать только с массивами, содержащими 10 элементов. Более гибкий подход состоит в том, что размер массива передается в качестве второго аргумента:

```
int sum(int * ar, int n)           // более общий подход
{
   int i;
   int total = 0;
   for( i = 0; i < n; i++)         // используются n элементов
      total += ar[i];              // ar[i] - это то же, что и *(ar + i)
   return total;
}
```

В данном случае первый параметр говорит функции о том, где можно найти массив, и какой тип данных элементов массива, а второй параметр уведомляет функцию о том, сколько элементов содержится в массиве.

О параметрах функции необходимо сказать следующее. В контексте прототипа функции или заголовка определения функции, и *только* в этом контексте, можно поставить `int ar[]` вместо `int * ar`:

```
int sum (int ar[], int n);
```

Форма `int * ar` всегда означает, что `ar` является типом указателя на значение `int`. Форма `int ar[]` также означает, то `ar` — это тип указатель на `int`, однако только в тех случаях, когда он используется *только* для объявления формальных параметров. Идея заключается в том, что вторая форма напоминает, что `ar` не просто указывает на значение типа `int`, он указывает на значение `int`, которое является элементом массива.

Объявление параметров массива

Поскольку имя массива — это адрес его первого элемента, фактический аргумент в виде имени массива требует, чтобы соответствующий формальный аргумент был указателем. В этом контексте, и только в нем, С интерпретирует `int ar[]` как `int * ar`, то есть `ar` является указателем на тип `int`. Поскольку прототипы позволяют опускать имя, все четыре показанных ниже прототипа эквивалентны:

```
int sum(int *ar, int n);
int sum(int *, int);
int sum(int ar[], int n);
int sum(int [], int);
```

Вы не можете опускать имена в определениях функций, следовательно, с точки зрения определений, следующие две формы эквивалентны:

```
int sum(int *ar, int n)
{
   // здесь находится программный код
}
int sum(int ar[], int n);
{
   // здесь находится программный код
}
```

Вы должны иметь возможность использовать любой из указанных выше прототипов с любым из двух определений, приведенных выше.

В листинге 10.10 представлена программа, использующая функцию `sum()`. Чтобы продемонстрировать интересные особенности аргументов типа массива, программа распечатывает также размер исходного файла и размер параметра функции, представляющего массив. (Используйте спецификатор `%u` или, возможно, `%lu`, если ваш компилятор не поддерживает спецификатор `%zd` для печати значений функции `sizeof`.)

Листинг 10.10. Программа `sum_arr1.c`

```
// sum_arr1.c -- сумма элементов массива
// используйте спецификаторы %u или %lu, если спецификатор %zd не работает
#include <stdio.h>
#define SIZE 10
int sum(int ar[], int n);
```

```
int main(void)
{
   int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
   long answer;

   answer = sum(marbles, SIZE);
   printf("Общая сумма элементов массива marbles равна %ld.\n", answer);
   printf("Объем памяти, отведенной под массив marbles, составляет %zd байт.\n",
          sizeof marbles);
   return 0;
}

int sum(int ar[], int n)          // каков размер массива?
{
   int i;
   int total = 0;

   for( i = 0; i < n; i++)
        total += ar[i];
   printf("Размер переменной ar составляет %zd байт.\n", sizeof ar);

   return total;
}
```

Выходные данные нашей системы имеют следующий вид:

```
Размер переменной ar составляет 4 байт.
Общая сумма элементов массива marbles равна 190.
Объем памяти, отведенной под массив marbles, составляет 40 байт.
```

Обратите внимание на то, что размер массива `marbles` равен 40 байтов. Это имеет смысл, поскольку массив `marbles` содержит 10 значений типа `int`, каждое из которых занимает 4 байта, что в сумме составляет 40 байт. В то же время размер `ar` равен всего 4 байта. Это объясняется тем, что `ar` не является массивом, это указатель на первый элемент массива `marbles`. Наша система использует четырехбайтовый адрес, таким образом, размер переменной типа указатель составляет 4 байта. (Другие системы могут использовать для этой цели другое число байтов.) Короче говоря, в программе из листинга 10.10 `marbles` — это массив, `ar` — указатель на первый элемент массива `marbles`, а связь в C между массивами и указателями позволяет использовать систему обозначений массива для указателя `ar`.

Использование параметров типа указатель

Функция, работающая с массивом, должна знать, где начинать и где заканчивать свои действия. Функция `sum()` использует параметр типа указатель с тем, чтобы распознать начало массива и целочисленный параметр, задающий количество элементов массива, подлежащих обработке. (Параметр типа указатель также описывает тип данных в массиве.) Но это не единственный путь сообщить функции то, что она должна знать. Другой способ состоит в том, чтобы описать массив, передавая функции два параметра, при этом первый из них указывает, где начинается массив (как и раньше), а второй — где массив заканчивается. Программа в листинге 10.11 служит иллюстрацией этого подхода. Он также использует тот факт, что параметр типа указатель является переменной, а это означает, что вместо использования индекса для указания, к какому

элементу массива осуществлять доступ, эта функция может менять само значение указателя, заставляя его поочередно указывать на каждый элемент массива.

Листинг 10.11. Программа sum_arr2.c

```
/* sum_arr2.c -- суммирует элементы массива */
#include <stdio.h>
#define SIZE 10
int sump(int * start, int * end);
int main(void)
{
   int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
   long answer;

   answer = sump(marbles, marbles + SIZE);
   printf("Общее количество элементов marbles равно %ld.\n", answer);

   return 0;
}

/* использование арифметики указателей */
int sump(int * start, int * end)
{
   int total = 0;

   while (start < end)
   {
      total += *start;     /* добавить значение к total */
      start++;             /* переместить указатель на следующий элемент */
   }

   return total;
}
```

Указатель `start` в начале указывает на первый элемент массива `marbles`, таким образом, выражение `total +=*start` добавляет значение первого элемента массива (`20`) к значению переменной `total`. Затем выражение `start++` увеличивает значение переменной `start`, благодаря чему она указывает на следующий элемент массива. Поскольку `start` указывает на тип `int`, C увеличивает значение `start` на размер типа `int`.

Обратите внимание на то, что функция `sump()` использует различные методы функции `sum()` для окончания цикла суммирования. Функция `sum()` использует количество элементов в качестве второго аргумента, а в цикле это значение применяется как часть проверки конца цикла:

```
for( i = 0; i < n; i++)
```

Однако функция `sump()` для проверки окончания цикла использует второй указатель:

```
while (start < end)
```

Поскольку выполняется проверка на неравенство, последним элементом массива, подвергнутым обработке, будет элемент, непосредственно предшествующий элементу, на который указывает `end`. Это означает, что `end` указывает на ячейку, которая находится сразу же за последним элементом массива. C гарантирует, что когда он распределяет пространство памяти для массива, указатель на первую ячейку после конца

массива будет допустимым. Благодаря этому обстоятельству, конструкция, подобная данной, также допустима, так как последним значением, которое start получает в цикле, будет end. Обратите внимание на то, что использование указателя, нацеленного "за пределы конца массива", позволяет осуществить такой вызов:

```
answer = sump(marbles, marbles + SIZE);
```

Поскольку индексирование начинается с 0, marbles + SIZE указывает на элемент, следующий за концом массива. Если бы end указывал на последний элемент вместо элемента, следующего за концом массива, вы должны бы были воспользоваться следующим кодом:

```
answer = sump(marbles, marbles + SIZE - 1);
```

Этот код не только менее элегантен по внешнему виду, его, к тому же, труднее запомнить, следовательно, из-за чего в программе появляется тенденция к возникновению ошибок. Между прочим, хотя язык C гарантирует допустимость применения указателя marbles + SIZE, тем не менее, нет таких гарантий в отношении marbles[SIZE], значения, хранимого в этой ячейке.

Вы можете также сжать тело цикла в следующую строку:

```
total += *start++;
```

Унарные операции * и ++ имеют один и тот же приоритет, но они выполняются справа налево. Это означает, что операция ++ применяется к start, но не к *start. Иначе говоря, увеличивается указатель, но не значение, на которое он указывает. Использование постфиксной формы (start++ вместо ++start) означает, что значение показателя не увеличивается, пока значение, на которое нацелен указатель, не будет добавлено к total. Если бы программа использовала конструкцию *++start, порядок был бы следующий: увеличение значения указателя с последующим использованием значения, на которое нацелен указатель. Однако если в программе используется конструкция (*start)++, она использует значение start, после чего увеличивается значение, но не указатель. При этом указатель будет нацелен на тот же элемент, но этот элемент содержит новое число. И хотя запись вида *start++ используется достаточно широко, мы рекомендуем более удобочитаемую форму записи: *(start++). Программа, показанная в листинге 10.12, служит иллюстрацией всех этих "прелестей" приоритетов операций.

Листинг 10.12. Программа order.c

```
/* order.c -- приоритеты операций с указателями */
#include <stdio.h>
int data[2] = {100, 200};
int moredata[2] = {300, 400};
int main(void)
{
   int * p1, * p2, * p3;
   p1 = p2 = data;
   p3 = moredata;
   printf("  *p1 = %d,   *p2 = %d, *   p3 = %d\n",
             *p1     ,   *p2     ,    *p3);
   printf("*p1++ = %d, *++p2 = %d, (*p3)++ = %d\n",
           *p1++ ,     *++p2     , (*p3)++);
```

```
    printf("  *p1 = %d,   *p2 = %d,     *p3 = %d\n",
              *p1     ,   *p2     ,     *p3);
    return 0;
}
```

Вот как выглядят выходные данные этой программы:

```
  *p1 = 100,   *p2 = 100,     *p3 = 300
*p1++ = 100, *++p2 = 200, (*p3)++ = 300
  *p1 = 200,   *p2 = 200,     *p3 = 301
```

Единственная операция, которая меняет значение массива — `(*p3)++`. Две другие операции приводят к тому, что указатели `p1` и `p2` перемещаются на следующий элемент массива.

Комментарии: указатели и массивы

Как вы уже убедились, функции, выполняющие обработку массивов, по сути, используют указатели в качестве аргументов, однако при создании функций обработки массивов потребуется выбрать форму записи — с помощью массива или с помощью указателей.

Применение формы записи с использованием массива, как в листинге 10.10, позволяет легче определять, что данная функция работает с массивами. Наряду с этим, запись в форме массива более привычна для программистов, работающих в других языках программирования, таких как, например, FORTRAN, Pascal, Modula-2 или BASIC. Другие программисты, возможно, предпочитают работать с указателями, и для них форма записи с использование указателей, подобная продемонстрированной в листинге 10.11, является более привычной.

Что касается языка C, то два выражения `ar[i]` и `*(ar+i)` эквивалентны по смыслу. Оба работают в тех случаях, когда `ar` есть имя массива, оба они также работают, если `ar` — переменная типа указатель. Тем не менее, использование такого выражения, как `ar++`, работает только в тех случаях, когда `ar` представляет собой переменную типа указатель.

Запись с применением указателей, в частности, когда они сопровождается операцией инкремента, ближе к машинному языку, а при использовании некоторых компиляторов дает более эффективный программный код. В то же время, многие разделяют мнение о том, что основная задача программиста — соблюсти правильность и ясность программного кода, а оптимизация кода вменяется в обязанности компилятора.

Операции с указателями

Так что же можно делать с указателями? Язык C предлагает множество базовых операций, которые можно выполнять над указателями, а приведенная ниже программа демонстрирует восемь из существующих возможностей. Чтобы показать результаты каждой операции, программа выводит на печать значение указателя (таковым является адрес, на который он указывает), значение, хранящееся по адресу, на который нацелен указатель, а также адрес самого указателя. (Если ваш компилятор не поддерживает спецификатор `%p`, попытайтесь воспользоваться для вывода адресов спецификатором `%u` или, возможно, `%lu`.) В листинге 10.13 показаны восемь базовых операций,

которые можно выполнять над переменными типа указатель. В дополнение к этим операциям вы можете использовать операции отношений при сравнении указателей.

Листинг 10.13. Программа `ptr_ops.c`

```
// ptr_ops.c -- операции над указателями
#include <stdio.h>
int main(void)
{
   int urn[5] = {100,200,300,400,500};
   int * ptr1, * ptr2, *ptr3;
   ptr1 = urn;                  // присваивание указателю адреса
   ptr2 = &urn[2];              // второй экземпляр
                                // разыменование указателя и взятие
                                // адреса указателя
   printf("значение указателя, разыменованный указатель, адрес указателя:\n");
   printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
          ptr1, *ptr1, &ptr1);
                                // сложение указателей
   ptr3 = ptr1 + 4;
   printf("\nсложение значения int с указателем:\n");
   printf("ptr1 + 4 = %p, *(ptr4 + 3) = %d\n",
          ptr1 + 4, *(ptr1 + 3));
   ptr1++;                      // увеличение значение указателя на 1
   printf("\nзначения после выполнения операции ptr1++:\n");
   printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
          ptr1, *ptr1, &ptr1);
   ptr2--;                      // уменьшение значение указателя на 1
   printf("\nзначения после выполнения операции --ptr2:\n");
   printf("ptr2 = %p, *ptr2 = %d, &ptr2 = %p\n",
          ptr2, *ptr2, &ptr2);
   --ptr1;                      // восстановление исходного значения
   ++ptr2;                      // восстановление исходного значения
   printf("\nвосстановление исходных значений указателей:\n");
   printf("ptr1 = %p, ptr2 = %p\n", ptr1, ptr2);
                                // вычитание одного указателя из другого
   printf("\nвычитание одного указателя из другого:\n");
   printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 = %d\n",
          ptr2, ptr1, ptr2 - ptr1);
                                // вычитание целого значения из указателя
   printf("\nвычитание из указателя значения типа int:\n");
   printf("ptr3 = %p, ptr3 - 2 = %p\n",
          ptr3, ptr3 - 2);
   return 0;
}
```

Ниже показаны выходные данные этой программы:

```
значение указателя, разыменованный указатель, адрес указателя:
ptr1 = 0x0012ff38, *ptr1 =100, &ptr1 = 0x0012ff34

сложение значения int с указателем:
ptr1 + 4 = 0x0012ff48, *(ptr4 + 3) = 400
```

```
значения после выполнения операции ptr1++:
ptr1 = 0x0012ff3c, *ptr1 =200, &ptr1 = 0x0012ff34

значения после выполнения операции --ptr2:
ptr2 = 0x0012ff3c, *ptr2 = 200, &ptr2 = 0x0012ff30

восстановление исходных значений указателей:
ptr1 = 0x0012ff38, ptr2 = 0x0012ff40

вычитание одного указателя из другого:
ptr2 = 0x0012ff40, ptr1 = 0x0012ff38, ptr2 - ptr1 = 2

вычитание из указателя значения типа int:
ptr3 = 0x0012ff48, ptr3 - 2 = 0x0012ff40
```

В представленном ниже перечне описаны базовые операции, которые могут быть выполнены над указателями.

- **Присваивание значений.** Указателю можно присвоить адрес. Обычно это делается путем использования имени массива или операции адресации (`&`). В рассматриваемом примере переменной `ptr1` присваивается адрес начала массива `urn`. Этим адресом оказался номер ячейки памяти `0x0012ff38`. Переменная `ptr2` получает в качестве своего значения адрес третьего, и последнего, элемента `urn[2]`. Обратите внимание, что этот адрес должен быть совместим с типом указателя. Иначе говоря, вы не можете присваивать адрес типа `double` указателю, ссылающемуся на значение типа `int`, по крайней мере, без приведения типов, которое в такой ситуации чревато пагубными последствиями. Это правило было введено стандартом C99.

- **Определение значения (разыменование).** Операция `*` возвращает значение, хранящееся в ячейке, на которую ссылается указатель. По этой причине первоначальным значением `*ptr1` является `100`, это значение хранится в ячейке `0x0012ff38`.

- **Адрес указателя.** Подобно всем переменным, переменные типа указатель имеют адрес и значение. Операция `&` определяет, где хранится сам указатель. В рассматриваемом примере `ptr1` хранится в ячейке `0x0012ff34`. Содержимое этой ячейки памяти — `0x0012ff38`, что является адресом массива `urn`.

- **Сложение целочисленного значения с указателем.** С помощью операции `+` можно сложить целое число с указателем или указатель с целым числом. В любом из этих случаев целое число умножается на количество байт, представляющее тип данных, на который ссылается указатель, после чего результат добавляется к первоначальному адресу. Результат операции `ptr1 + 4` тот же, что и результат операции `&urn[4]`. Результат сложения не определен, если он выходит за пределы массива, на который ссылается исходный указатель, за исключением ссылки на адрес, следующий за концом массива; в этом случае значение считается допустимым.

- **Инкремент значения указателя.** Увеличение значения указателя на величину элемента массива заставляет указатель ссылаться на следующий элемент массива. По этой причине операция `ptr1++` увеличивает значение указателя `ptr1` на `4` (в нашей системе под каждое значение типа `int` отводится 4 байта), в результа-

те указатель ptr1 ссылается на urn[1] (рис. 10.4). Теперь ptr1 имеет значение 0x0012ff3c (адрес следующего элемента массива), а *ptr1 — значение 200 (значение элемента urn[1]). Обратите внимание, что сам адрес ptr1 не меняется и остается равным 0x0012ff34. В конце концов, переменная не перемещается в памяти по причине того, что она поменяла свое значение!

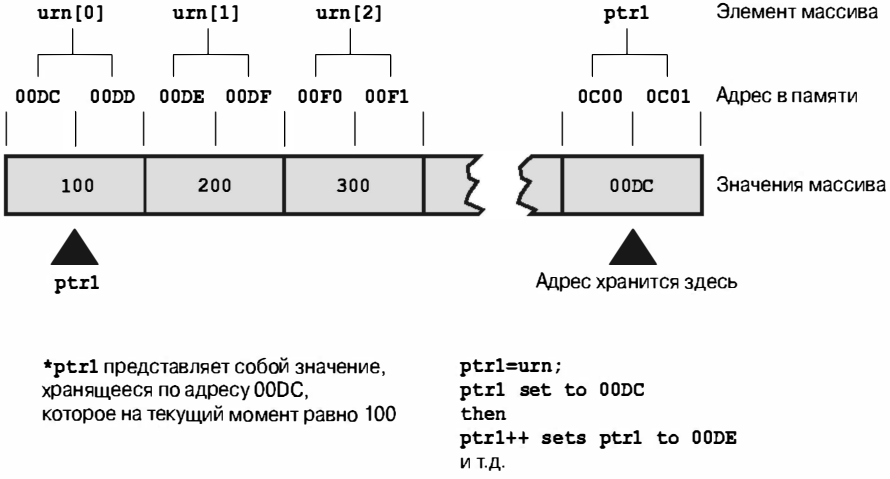


Рис. 10.4. *Увеличение указателя на значение типа* `int`

- **Вычитание целочисленных значений.** С помощью операции – можно вычитать целое число из указателя; указателем должен быть первый операнд или указатель на целое число. Целое число умножается на количество байт в типе, на который ссылается указатель, а результат умножения вычитается из первоначального адреса. Результат операции вычитания ptr3 - 2 аналогичен результату операции &urn[2], поскольку ptr3 указывает на &urn[4]. Результат сложения не определен, если он выходит за пределы массива, на который ссылается исходный указатель, за исключением ссылки на адрес, следующий за концом массива; в этом случае значение считается допустимым.
- **Декремент значения указателя.** Разумеется, значение указателя можно декрементировать. В рассматриваемом примере уменьшение значение ptr2 на единицу приводит к тому, что он ссылается не на третий, а на второй элемент массива. Обратите внимание, что вы можете пользоваться префиксными и постфиксными формами операций инкремента и декремента. Также обратите внимание на то, что оба указателя ptr1 и ptr2 указывали на один и тот же элемент, в данном случае это urn[1] перед тем, как их значение было изменено.
- **Вычисление разности указателей.** Имеется возможность определить разность между двумя указателями. Обычно это делается для двух указателей на элементы, принадлежащие одному и тому же массиву, с тем, чтобы определить, насколько далеко они отстоят друг от друга. Результат представляется в тех же единицах, что и размер типа. Например, в выходных данных программы из листинга 10.13 выражение ptr2 - ptr1 имеет значение 2, что означает, что эти

указатели ссылаются на объекты, разделенные двумя значениями int, но не двумя байтами. Вычитание является гарантировано допустимой операцией, если только оба указателя ссылаются на один и тот же массив (или, возможно, на позицию, следующую непосредственно за концом массива). Применение этой операции к указателям в двух различных массивах может дать какой-то результат, но может привести и к ошибке во время выполнения программы.

- **Сравнения указателей.** Для сравнения значений двух указателей могут использоваться операции отношений в случае, если указатели имеют один и тот же тип.

Обратите внимание на существование двух форм вычитания. Вы можете вычесть один указатель из другого и получить целое число, в то же время вы может вычесть целое число из указателя и получить указатель.

При выполнении операций инкремента и декремента указателя, следует соблюдать определенные меры предосторожности. Компьютер не отслеживает, ссылается ли полученный в результате указатель на какой-то элемент массива. Язык C гарантирует только то, что для заданного массива указатель на любой из элементов этого массива или на позицию, непосредственно следующую за последним элементом массива, является допустимым. Однако результат инкремента или декремента указателя, выходящий за эти пределы, не определен. Наряду с этим, вы можете разыменовать указатель для любого элемента массива. Тем не менее, несмотря на то, что указатель, ссылающийся на элемент, следующий за концом массива, допустим, нет гарантии того, что этот указатель может быть разыменован.

Разыменование неинициализированного указателя

Говоря об осторожности, существует правило, которое должно прочно засесть в вашей памяти: никогда не разыменовывайте неинициализированные указатели. Например, рассмотрим следующий программный код:

```
int * pt;      // неинициализированный указатель
*pt = 5;       // катастрофическая ошибка
```

Почему это настолько плохо? Вторая строка означает необходимость сохранить значение 5 в ячейке, на которую указывает pt. Однако pt, будучи инициализированным, имеет случайное значение, следовательно, неизвестно, куда будет записано значение 5. Оно может не вызвать каких-либо пагубных последствий, но оно может затереть какой-то код или данные, а может вообще привести к аварийному завершению программы. Вспомните, что при создании указателя память выделяется под сам указатель, но для хранения данных память не распределяется. Поэтому, прежде чем вы воспользуетесь указателем, ему должен быть присвоен адрес ячейки памяти, которая уже была распределена. Например, вы можете назначить указателю адрес существующей переменной. (Именно это и происходит, когда вы используете функцию с параметром типа указатель.) Либо вы можете воспользоваться функцией malloc(), которая рассматривается в главе 12, чтобы сначала зарезервировать нужную память.

В любом случае, чтобы не допускать ошибок, не разыменовывайте неинициализированные указатели!

```
double * pd;      // неинициализированный указатель
*pd = 2.4;        // НЕ ДЕЛАЙТЕ этого!
```

Пусть дано

```
int urn[3];
int * ptr1, * ptr2;
```

ниже приведены примеры допустимых и недопустимых операторов:

Допустимый оператор	*Недопустимый оператор*
`ptr1++;`	`urn++;`
`ptr2 = ptr1 + 2;`	`ptr2 = ptr2 + ptr1;`
`ptr2 = urn + 1;`	`ptr2 = urn * ptr1;`

Описанные ранее операции открывают множество возможностей. Программисты, работающие на языке C, создают массивы указателей, указатели на функции, массивы указателей на указатели, массивы указателей на функции и так далее. Однако пусть вас это не волнует, мы ограничимся лишь теми основными видами использования указателей, которые рассматривались выше. Первый основной вид использования указателей состоит в передаче информации из функций и в функции. Вы уже знаете, что вы должны использовать указатели, если хотите, чтобы та или иная функция меняла значения переменных в вызывающей функции. Второй вид использования касается функций, разработанных для манипулирования массивами. Рассмотрим еще один пример программы, использующей функции и массивы.

Защита содержимого массива

Когда вы пишете функцию, которая выполняет обработку данных фундаментального типа, такого как `int`, перед вами встает выбор: передать данные типа `int` по значению или передать указатель на значение типа `int`. Обычное правило предусматривает передачу числовых данных по значению до тех пор, пока в программе не появится потребность изменить это значение, в этом случае вы передаете указатель. Массивы не предоставляют такой возможности, и вы *обязательно* должны передавать указатель. Это делается с целью обеспечить эффективность программы. Если функция передает массив по значению, она должна иметь в своем распоряжении достаточное пространство, чтобы сохранить копию исходного массива, после чего скопировать все данные из исходного массива в новый массив. Гораздо быстрее передать адрес массива, чтобы функция работала с исходным массивом.

При использовании такого метода возникает ряд проблем. Причиной того, что C обычно передает данные по значению, является стремление сохранить целостность данных. Если функция работает с копией исходных данных, она не сможет случайным образом исказить эти данные. В то же время, поскольку функции, выполняющие обработку массива, работают с исходными данными, они *могут* исказить массив. Иногда это желательно. Например, рассмотрим функцию, которая прибавляет одно и то же значение к каждому элементу массива:

```
void add_to(double ar[], int n, double val)
{
   int i;
   for( i = 0; i < n; i++)
      ar[i] += val;
}
```

Следовательно, вызов функции

```
add_to(prices, 100, 2.50);
```

приводит к тому, что каждый элемент массива `prices` получает величину, превосходящую его исходное значение на 2.5; эта функция изменяет содержимое массива. Функция может сделать это, поскольку, работая с указателями, она использует исходные данные.

Другие функции, однако, не должны изменять данные. Следующая функция, например, предназначена для подсчета суммы содержимого массива, и она не меняет массива. Однако, поскольку `ar` есть фактический указатель, программная ошибка может привести к искажению исходных данных. Здесь, например, выражение `ar[i]++` приводит к тому, что значение каждого элемента увеличивается на 1:

```
int sum(int ar[], int n)      // ошибочный программный код
{
   int i;
   int total = 0;
   for( i = 0; i < n; i++)
      total += ar[i]++; //по ошибке увеличивается значение каждого элемента
   return total;
}
```

Использование `const` с формальными параметрами

При использовании компилятора K&R C единственный способ избежать ошибок такого вида — быть постоянно бдительным. В стандарте ANSI C существует альтернатива. Если функция не предназначена для того, чтобы менять содержимое массива, используйте ключевое слово `const` при объявлении формального параметра в прототипе и в определении функции. Например, прототип и определение функций `sum()` должны иметь следующий вид:

```
int sum(const int ar[], int n);       /* прототип */
int sum(const int ar[], int n)        /* определение */
{
   int i;
   int total = 0;
   for( i = 0; i < n; i++)
      total += ar[i];
   return total;
}
```

Это ключевое слово сообщает компилятору о том, что функция должна рассматривать массив, на который ссылается указатель `ar`, как массив, содержащий константные данные. В этом случае, если вы случайно используете такие выражения, как `ar[i]++`, компилятор может отловить этот случай и выдать сообщение об ошибке, уведомляющее о том, что функция предпринимает попытку изменить константные данные.

Важно понимать, что использование ключевого слова `const` в этом плане не требует, чтобы все элементы исходного массива были константами, оно просто говорит о том, что функция должна обращаться с элементами массива *так, как если бы* они были константами.

Использование ключевого слова const подобным образом обеспечивает защиту массивов, какую передача по значению обеспечивает для фундаментальных типов. В общем случае, если вы пишете функцию, предназначенную для модификации массива, не используйте ключевое слово const при объявлении параметров массива. Если вы пишете функцию, не предназначенную для модификации массива, вы можете указать ключевое слово const при объявлении параметров массива.

В программе, показанной в листинге 10.14, одна функция отображает массив, а другая — умножает каждый элемент массива на заданное значение. Поскольку первая функция не должна менять значения элементов массивов, она использует ключевое слово const. Поскольку вторая функция имеет намерение модифицировать массив, в ней словj const не используется.

Листинг 10.14. Программа arf.c

```
/* arf.c -- функции, манипулирующие массивами */
#include <stdio.h>
#define SIZE 5
void show_array(const double ar[], int n);
void mult_array(double ar[], int n, double mult);
int main(void)
{
   double dip[SIZE] = {20.0, 17.66, 8.2, 15.3, 22.22};

   printf("Исходный массив dip:\n");
   show_array(dip, SIZE);
   mult_array(dip, SIZE, 2.5);
   printf("Массив dip после вызова функции mult_array():\n");
   show_array(dip, SIZE);

   return 0;
}

/* выводит содержимое массива */
void show_array(const double ar[], int n)
{
   int i;

   for (i = 0; i < n; i++)
      printf("%8.3f ", ar[i]);
   putchar('\n');
}

/* умножает каждый элемент массива на один и тот же множитель */
void mult_array(double ar[], int n, double mult)
{
   int i;

   for (i = 0; i < n; i++)
      ar[i] *= mult;
}
```

Ниже приводятся выходные данные этой программы:

```
Исходный массив dip:
20.000 17.660 8.200 15.300 22.220
```

```
Массив dip после вызова функции mult_array():
50.000 44.150 20.500 38.250 55.550
```

Обратите внимание, что типом обеих функции является void. Функция mult_array() присваивает новые значения элементам массива dip, однако при этом механизм возврата значений не используется.

Дополнительные сведения о ключевом слове const

Как вы убедились выше, ключевое слово const можно использовать для создания символических констант:

```
const double PI = 3.14159;
```

В этих случаях, наряду с прочим, вы могли кое-что сделать в отношении директивы #define, в то же время ключевое слово const дополнительно позволяет создать массивы констант, константные указатели, а также указатели на константы. В листинге 10.4 показано, как используется ключевое слово const для защиты массива от модификации:

```
#define MONTHS 12
...
const int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Если в дальнейшем программный код попытается изменить массив, вы получите сообщение об ошибке на этапе компиляции:

```
days[9] = 44;     /* ошибка на этапе компиляции */
```

Указатели на константы не могут использоваться для изменения значений. Рассмотрим следующий программный код:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * pd = rates;       // pd указывает на начало массива
```

Вторая строка кода объявляет, что значение типа double, на которое указывает pd, есть const. Это означает, что вы не можете использовать pd для изменения значений, на которые ссылаются указатели:

```
*pd = 29.89;         // не допускается
pd[2] = 222.22;      // не допускается
rates[0] = 99.99;    // допускается, поскольку rates не является константой
```

Независимо от того, употребляете ли вы форму записи с использованием указателей или с использованием массива, вы не можете использовать pd для изменения данных, на которые указывают указатели. Однако, обратите внимание на то, что массив rates не был объявлен как константа, вы можете продолжать использовать rates с тем, чтобы менять его значения. В то же время, вы можете использовать указатель pd для ссылки на какой-то другой объект:

```
pd++;     /* заставляет pd указывать на rates[1] - допустимо */
```

Указатель на константу обычно применяется в качестве параметра функции, чтобы указать, что функция не использует указателей для изменения данных. Например, для функции show_array() из листинга 10.14 можно предусмотреть следующий прототип:

```
void show_array(const double *ar, int n);
```

Существует несколько правил, касающихся назначения указателей и употребления ключевого слова const, которые вы должны твердо знать. Во-первых, допускается присваивание адреса как константных, так и неконстантных данных указателю на константу:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
const double * pc = rates;          // допустимо
pc = locked;                        // допустимо
pc = &rates[3];                     // допустимо
```

В то же время обычным указателям могут быть присвоены только адреса неконстантных данных:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
double * pnc = rates;               // допустимо
pnc = locked;                       // не допустимо
pnc = &rates[3];                    // допустимо
```

Это разумное правило. В противном случае вы можете использовать указатель для изменения данных, которые рассматривались как константные.

Последствия применения этих правил на практике состоят в том, что функция, например, show_array(), может принимать имена обычных массивов и постоянных массивов в качестве фактических аргументов, поскольку каждый из них может быть присвоен указателю на константу:

```
show_array(rates, 5);               // допустимо
show_array(locked, 4);              // допустимо
```

В то же время функция, подобная mult_array(), не может принимать имени константного массива в качестве аргумента:

```
mult_array(rates, 5, 1.2);          // допустимо
mult_array(locked, 4, 1.2);         // не допустимо
```

В силу этого обстоятельства использование ключевого слова в определении параметра функции не только обеспечивает защиту данных, он также позволяет функции работать с массивами, которые были объявлены как const.

Существуют и другие варианты использования ключевого слова const. Например, вы можете объявить и инициализировать указатель, который не может указывать на что угодно. Все зависит от того, где размещается ключевое слово const:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
double * const pc = rates;       // pc указывает на начало массива
pc = &rates[2];                  // не допустимо
*pc = 92.99;                     // правильно - изменяет элемент
rates[0]
```

Такой указатель все еще может использоваться для изменения значений, но он может указывать только на ячейку, первоначально присвоенную ему.

И, наконец, вы можете воспользоваться ключевым словом дважды для создания указателя, который не может изменить ни адрес, на который он указывает, ни значение, на которое он указывает:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * const pc = rates;
pc = &rates[2];                 // не допустимо
*pc = 92.99;                    // не допустимо
```

Указатели и многомерные массивы

Как указатели связаны с многомерными массивами? И почему это необходимо знать? Функции, которые работают с многомерными массивами, используют с этой целью указатели, поэтому вы должны продолжить изучение указателей, прежде чем переходить к работе с такими функциями. А что касается первого вопроса, то рассмотрим несколько примеров и постараемся найти на него ответ. Чтобы упростить этот процесс, будем работать с массивами небольших размеров. Предположим, что имеется следующее объявление:

```
int zippo[4][2];            /* массив массивов значений int */
```

В этом случае `zippo`, будучи именем массива, представляет собой адрес первого элемента массива. Тогда первый элемент массива `zippo` сам является массивом, состоящим их двух значений типа `int`, следовательно, `zippo` — это адрес массива, состоящего из двух значений типа `int`. Проведем дальнейший анализ с учетом свойств указателей:

- Поскольку `zippo` — адрес первого элемента массива, `zippo` имеет то же значение, что и `&zippo[0]`. Далее, `zippo[0]` сам по себе есть массив из двух целых чисел, следовательно, `zippo[0]` имеет то же значение, что и `&zippo[0][0]`, адрес его первого элемента, то есть `int`. Короче говоря, `zippo[0]` есть адрес объекта, размер которого выражается через величину типа `int`, а `zippo` — адрес объекта, имеющего размер, равный размеру двух типов `int`. Поскольку и целое значение, и массив, состоящий из двух целочисленных значений, начинаются в одной и той же ячейке, `zippo` и `zippo[0]` имеют одно и то же числовое значение.
- Добавление 1 к указателю или адресу дает значение, которое больше исходного на размер объекта ссылки. В этом отношении `zippo` и `zippo[0]` отличаются друг от друга, поскольку `zippo` ссылается на объект, имеющий размер двух типов `int`, а `zippo[0]` ссылается на объект размером в один тип `int`. Следовательно, `zippo + 1` имеет значение, отличное от `zippo[0] + 1`.
- Разыменование указателя или адреса (применение операции `*` или операции `[]` с индексом) позволяет получить значение, представляемое объектом ссылки. Поскольку `zippo[0]` является адресом его первого элемента, `(zippo[0][0])`, `*(zippo[0])` представляет значение, хранящееся в `zippo[0][0]`, то есть значение типа `int`. Аналогично, `*zippo` представляет значение его первого элемента, то есть `zippo[0]`, в то же время `zippo[0]` сам по себе есть адрес значения типа `int`. Это адрес `&zippo[0][0]`, следовательно, `*zippo` есть `&zippo[0][0]`. Применение оператора разыменования к обоим выражениям, приводит к тому, что `**zippo` эквивалентно `*&zippo[0][0]`, при этом оба эти выражения приводятся к `zippo[0][0]`, представляющему собой значение типа `int`. Короче говоря, `zippo` — это адрес адреса, к которому операция разыменования должна быть применена дважды, чтобы получить обычное значение. Адрес адреса или указатель указателя представляют собой примеры *двойного разыменования*.

Разумеется, возрастание размерности массива увеличивает сложность представления в виде указателей. В этой точке большинство изучающих язык C начинают понимать, почему указатели считаются одним из наиболее трудных аспектов языка. Возможно, вам потребуется еще раз почитать о свойствах указателей, которые описаны выше, и только после этого вернуться к рассмотрению программы в листинге 10.15, где выводятся значения некоторых адресов и содержимое массивов.

Листинг 10.15. Программа `zippo1.c`

```
/* zippo1.c -- информация о массиве zippo */
#include <stdio.h>
int main(void)
{
   int zippo[4][2] = { {2,4}, {6,8}, {1,3}, {5, 7} };
   printf("   zippo = %p,    zippo + 1 = %p\n",
            zippo,           zippo + 1);
   printf("zippo[0] = %p, zippo[0] + 1 = %p\n",
           zippo[0],      zippo[0] + 1);
   printf("  *zippo = %p,   *zippo + 1 = %p\n",
            *zippo,        *zippo + 1);
   printf("zippo[0][0] = %d\n", zippo[0][0]);
   printf("  *zippo[0] = %d\n", *zippo[0]);
   printf("    **zippo = %d\n", **zippo);
   printf("      zippo[2][1] = %d\n", zippo[2][1]);
   printf("*(*(zippo+2) + 1) = %d\n", *(*(zippo+2) + 1));
   return 0;
}
```

Ниже показаны результаты выполнения этой программы в одной из систем:

```
   zippo = 0x0064fd38,    zippo + 1 = 0x0064fd40
zippo[0] = 0x0064fd38, zippo[0] + 1 = 0x0064fd3c
  *zippo = 0x0064fd38,   *zippo + 1 = 0x0064fd3c
zippo[0][0] = 2
  *zippo[0] = 2
    **zippo = 2
      zippo[1][2] = 3
*(*(zippo+1) + 2) = 3
```

Другие системы могут отображать различные адресные значения, однако отношения будут такими же, что и описанные в этом разделе. Эти выходные данные показывают, что адрес двумерного массива `zippo` и адрес одномерного массива `zippo[0]` — одни и те же адреса. Каждый из них представляет собой адрес первого элемента соответствующего массива, что в числовом виде то же самое, что и `&zippo[0][0]`.

Тем не менее, здесь имеются определенные различия. В нашей системе тип `int` занимает 4 байта. Как уже говорилось выше, адрес `zippo[0]` указывает на 4-байтный объект. Добавляя 1 к `zippo[0]`, мы должны получить адрес, больший исходного на 4. Имя `zippo` представляет собой адрес массива, состоящего из двух значений типа `int`, таким образом, он идентифицирует 8-байтовый объект данных. По этой причине добавление 1 к `zippo` должно иметь своим результатом адрес, превышающий исходный на 8 байтов, что и происходит на самом деле.

Приведенная выше программа показывает, что `zippo[0]` и `*zippo` идентичны, какими они и должны быть. Далее, она показывает, что имя двумерного массива должно быть разыменовано дважды, чтобы можно было получить доступ к значению, хранящемуся в массиве. Это можно сделать, применив дважды операцию разыменования (`*`) или дважды выполнив операцию квадратных скобок (`[ ]`). (Это можно сделать, если выполнить один раз операцию `*` и один раз применив квадратные скобки `[ ]`, однако, рассматривая все эти возможности, давайте не будем отклоняться от главной цели.)

В частности, обратите внимание, что выражение `zippo[2][1]` в системе записи с использованием указателей эквивалентно выражению `*(*(zippo+2) + 1)` в системе записи с использованием указателей. Вы, должно быть, хотя бы раз в своей жизни пытались нарушить это равенство. Будем строить это выражение постепенно, выполняя указанную ниже последовательность шагов:

`zippo`	← адрес первого элемента, состоящего из двух значений типа `int`.
`zippo+2`	← третий элемент массива, состоящий из двух значений типа `int`.
`*(zippo+2)`	← третий элемент, представляющий собой массив из двух значений, следовательно, это адрес его первого элемента, имеющего значение типа `int`.
`*(zippo+2) + 1`	← адрес второго элемента двухэлементного массива, также имеющего значение типа `int`.
`*(*(zippo+2) + 1)`	← значение второго объекта типа `int` в третьей строке (`zippo[2][1]`).

Цель этой причудливой формы записи с использованием указателей состоит не в том, чтобы продемонстрировать только то, что вы овладели ею и можете использовать ее вместо намного более простого выражения `zippo[2][1]`, но и то, что вы вполне можете пользоваться более простой формой записи в виде массива, чтобы извлечь значение, когда встречается указатель на двумерный массив.

На рис 10.5 показана еще одна точка зрения на зависимость между адресами массива, содержимым массива и указателями.

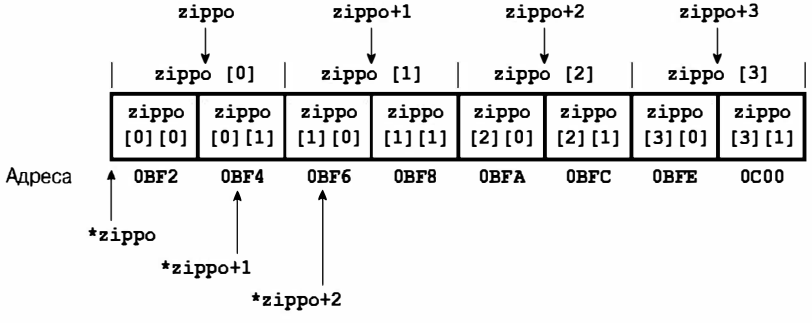


Рис. 10.5. *Массив массивов*

Указатели на многомерные массивы

Как вы объявите переменную pz типа указатель, которая может ссылаться на такой двумерный массив как zippo? Такой указатель может использоваться, например, при написании функции, которая выполняет обработку массивов, подобных zippo. Достаточно ли для этого указателя на значение типа int? Нет, не достаточно. Этот тип совместим с zippo[0], который указывает на какое-то одно значение типа int. Однако zippo есть адрес его первого элемента, который сам представляет собой массив из двух значений типа int. Отсюда следует, что pz должен указывать на массив, содержащий два элемента типа int, а не на какое-то одно значение типа int. Вот как можно поступить в таком случае:

```
int (* pz)[2];     // pz указывает на массив из 2 значений типа int
```

Этот оператор говорит, что pz является указателем на массив из двух значений типа int. Зачем в этом случае нужны круглые скобки? Скобки [ ] имеют более высокий приоритет, чем *. Следовательно, в случае, например, такого объявления

```
int * pax[2];
```

сначала используются квадратные скобки, благодаря которым pax рассматривается как массив двух каких-то объектов данных. Далее, применяется операция *, в результате которой массив pax превращается в массив из двух указателей. В заключение используется значение int, превращающее pax в массив из двух указателей на значения типа int. Это объявление создает *два* указателя на значения типа int, тем не менее, первоначальная версия использует круглые скобки с тем, чтобы сначала применить операцию *, создавая *один* указатель на два значения типа int. Код в листинге 10.16 показывает, как можно использовать такой указатель в качестве исходного массива.

Листинг 10.16. Программа zippo2.c

```
/* zippo2.c -- получение информации о массиве zippo с помощью
               переменной типа указатель */
#include <stdio.h>
int main(void)
{
   int zippo[4][2] = { {2,4}, {6,8}, {1,3}, {5, 7} };
   int (*pz)[2];
   pz = zippo;
   printf("   pz = %p,    pz + 1 = %p\n",
              pz,         pz + 1);
   printf("pz[0] = %p, pz[0] + 1 = %p\n",
           pz[0],      pz[0] + 1);
   printf("  *pz = %p,   *pz + 1 = %p\n",
             *pz,        *pz + 1);
   printf("pz[0][0] = %d\n", pz[0][0]);
   printf("  *pz[0] = %d\n", *pz[0]);
   printf("    **pz = %d\n", **pz);
   printf("      pz[2][1] = %d\n", pz[2][1]);
   printf("*(*(pz+2) + 1) = %d\n", *(*(pz+2) + 1));
   return 0;
}
```

Ниже представлены новые выходные данные:

```
pz = 0x0064fd38,    pz + 1 = 0x0064fd40
pz[0] = 0x0064fd38, pz[0] + 1 = 0x0064fd3c
*pz = 0x0064fd38,     *pz + 1 = 0x0064fd3c
pz[0][0] = 2
  *pz[0] = 2
    **pz = 2
      pz[2][1] = 3
*(*(pz+2) + 1) = 3
```

И снова вы можете получить различные адреса, однако отношения остаются прежними. Как мы обещали, вы можете использовать такую форму записи, как `pz[2][1]`, даже если `pz` является указателем, но не именем массива. В более общих случаях вы можете представлять отдельные элементы с помощью формы записи с использованием массива или с использованием указателей применительно либо к имени массива, либо к указателю:

```
zippo[m][n] == *(*(zippo + m) + n)
pz[m][n] == *(*(pz + m) + n)
```

Совместимость указателей

Правила присвоения одного указателя другому обладают более высоким приоритетом, чем правила для числовых значений. Например, вы можете присвоить значение типа `int` переменной типа `double` без применения преобразования типов, в то же время вы не можете поступить подобным образом в отношении указателей на два эти типа:

```
int n = 5;
double x;
int * p1 = &n;
double * pd = &x;
x = n;              // неявное преобразование типа
pd = p1;            // ошибка этапа компиляции
```

Эти ограничения распространяются на более сложные типы. Предположим, что мы имеем следующие объявления:

```
int * pt;
int (*pa)[3];
int ar1[2][3];
int ar2[3][2];
int **p2;           // указатель на указатель
```

Тогда мы получаем следующие равенства:

```
pt = &ar1[0][0];    // оба - указатели на значение типа int
pt = ar1[0];        // оба - указатели на значение типа int
pt = ar1;           // не допустимо
pa = ar1;           // оба - указатели на int[3]
pa = ar2;           // не допустимо
p2 = &pt;           // оба - указатели на значение типа int *
*p2 = ar2[0];       // оба - указатели на значение типа int
p2 = ar2;           // не допустимо
```

Обратите внимание, что для всех недопустимых случаев присваивания характерно прежде всего то, что используются два указателя, которые указывают на разные типы данных. Например, pt указывает на одно значение int, в то же время ar1 — на массив из трех значений типа int. Аналогично, pa указывает на массив из двух значений типа int, следовательно, он совместим с ar1, но не с ar2, который указывает на массив из двух значений int.

Два последних примера несколько искусственны. Переменная p2 представляет собой указатель на указатель на значение типа int, в то время как ar2 есть указатель на массив, состоящий из двух значений типа int (или, короче, указатель на массив int[2]). Таким образом, p2 и ar2 — разные типы, и вы не можете присвоить ar2 указателю p2. В то же время *p2 имеет тип указателя на тип int, что делает его совместимым с ar2[0]. Напомним, что ar2[0] является указателем на его первый элемент, в данном случае, ar2[0][0], что также делает ar2[0] типом указателя на значение int.

В общем случае многократные операции разыменования также носят искусственный характер. Например, рассмотрим следующий фрагмент кода:

```
int * p1;
const int * p2;
const int ** pp2;
p1 = p2;          // недопустимо – присваивание const значению не const
p2 = p1;          // допустимо - присваивание не const значению const
pp2 = &p1;        // недопустимо - присваивание не const значению const
```

Как вы убедились выше, присваивание указателя типа const указателю типа не const не допускается, поскольку вы можете использовать новый указатель для изменения данных типа const. В то же время указатель типа не const на указатель const допустим при условии, что вы выполняете всего лишь один уровень разыменования:

```
p2 = p1;         // допустимо - присваивание не const значению const
```

Однако такие присваивания теперь чреваты ошибками, если вы перейдете на два уровня операции разыменования. Если бы они были допустимы, вы бы получили возможность выполнить одну из следующих операций:

```
const int **pp2;
int *p1;
const int n = 13;
pp2 = &p1;   // недопустимо, но предполагается допустимость
*pp2 = &n;   // допустимо, оба const, однако устанавливает p1 ссылку на n
*p1 = 10;    // допустимо, но при этом изменяется const
```

Функции и многомерные массивы

Если вы хотите написать функцию, которая выполняет обработку двумерных массивов, то должны иметь достаточно четкое представление об указателях, чтобы делать правильные объявления, касающиеся аргументов функции. В теле самой функции вполне достаточно применять форму записи с помощью массивов.

Напишем функцию, манипулирующую двумерными массивами. Одной из возможностей является использование цикла for для применения функций обработки одномерного массива к каждой строке двумерного массива.

Другими словами, вы можете делать нечто подобное:

```
int junk[3][4] = { {2,4,5,8}, {3,5,6,9}, {12,10,8,6} };
int i, j;
int total = 0;
for (i = 0; i < 3 ; i++)
   total += sum(junk[i], 4);   // junk[i] - одномерный массив
```

Вспомните, что если junk есть двумерный массив, то junk[i] — одномерный массив, который вы можете рассматривать как строку двумерного массива. В этом случае функция sum() вычисляет промежуточную сумму для каждой строки двумерного массива, а в цикле for выполняется суммирование этих промежуточных сумм.

Однако в условиях этого подхода теряется связь с информацией, касающейся строк и столбцов. В этом приложении (суммирование всех значений) такая информация не имеет решающего значения, однако предположим, что каждая строка представляет год, а каждая строка — месяц. В этом случае вам, возможно, понадобится функция, которая, скажем, суммирует значения, содержащиеся в некотором конкретном столбце. В этом случае функция должна иметь в своем распоряжении информацию о столбцах и строках. Этот вопрос можно решить путем объявления правильного вида формальной переменной, благодаря чему функция могла бы правильно передавать массивы. В этом случае массив junk представляет собой массив трех массивов, каждый из которых содержит четыре значения типа int. Как показывают ранее проведенные обсуждения, массив junk — это указатель на массив из четырех значений типа int. Вы можете объявить параметр функции этого типа следующим образом:

```
void somefunction( int (* pt)[4] );
```

С другой стороны, если (и только если) pt является формальным параметром функции, вы можете объявить его следующим образом:

```
void somefunction( int pt[][4] );
```

Обратите внимание на то, что первый набор квадратных скобок пуст. Пустые квадратные скобки показывают, что pt является указателем. Такого рода переменная может быть использована в этом случае так же, как и массив junk. Вот что мы сделали в следующем примере, представленном на листинге 10.17. В этом листинге представлены три эквивалентных формы синтаксиса прототипов.

Листинг 10.17. Программа array2d.c

```
// array2d.c -- функции для двумерных массивов
#include <stdio.h>
#define ROWS 3
#define COLS 4
void sum_rows(int ar[][COLS], int rows);
void sum_cols(int [][COLS], int );      // можно опустить имена
int sum2d(int (*ar)[COLS], int rows);   // другой вид синтаксиса
int main(void)
{
   int junk[ROWS][COLS] = {
           {2,4,6,8},
           {3,5,7,9},
           {12,10,8,6}
   };
```

```
    sum_rows(junk, ROWS);
    sum_cols(junk, ROWS);
    printf("Сумма всех элементов = %d\n", sum2d(junk, ROWS));
    return 0;
}
void sum_rows(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot;
    for (r = 0; r < rows; r++)
    {
        tot = 0;
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
        printf("строка %d: сумма = %d\n", r, tot);
    }
}
void sum_cols(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot;
    for (c = 0; c < COLS; c++)
    {
        tot = 0;
        for (r = 0; r < rows; r++)
            tot += ar[r][c];
        printf("столбец %d: сумма = %d\n", c, tot);
    }
}
int sum2d(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot = 0;
    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
    return tot;
}
```

Ниже показаны выходные данные этой программы:

```
строка 0: сумма = 20
строка 1: сумма = 24
строка 2: сумма = 36
столбец 0: сумма = 17
столбец 1: сумма = 19
столбец 2: сумма = 21
столбец 3: сумма = 23
Сумма всех элементов = 80
```

Программа из листинга 10.17 передает в качестве аргумента имя junk, которое представляет указатель на первый элемент, каковым является подмассив, а также символическую константу ROWS со значением 3, то есть количество строк массива. После этого каждая функция рассматривает ar как массив массивов, содержащих четыре значения типа int. Количество столбцов встраивается в функцию, но количество строк остается неуказанным. Эта же функция будет работать, скажем, с массивом размерности 12×4, если передается число 12 в качестве числа строк. Это объясняется тем, что rows — это количество элементов; в то же время, поскольку каждый элемент представляет собой массив, или строка, rows становится количеством строк.

Обратите также внимание и на то, что ar используется так же, как junk в функции main(). Это становится возможным, поскольку ar и junk имеют один и тот же тип: указатель на массив из четырех значений типа int.

Имейте в виду, что следующее далее объявление не будет работать должным образом:

```
int sum2(int ar[][], int rows);        // ошибочное объявление
```

Вспомните, что компилятор переводит форму записи в стиле массива, в форму записи в стиле указателя. Это значит, например, что ar[1] преобразуется в ar+1. Чтобы компилятор мог вычислить эти выражения, он должен знать размер объекта, на который указывает ar. Объявление

```
int sum2(int ar[][4], int rows);       // правильное объявление
```

говорит о том, что ar указывает на массив, состоящий из четырех значений типа int (в нашей системе, на объект длиной 16 байтов), следовательно, ar+1 означает "прибавить 16 байтов к адресу". В условиях варианта с пустыми квадратными скобками компилятор не будет знать, что делать дальше.

Вы можете также заключить размер в другую пару квадратных скобок, как показано выше, однако компилятор их проигнорирует:

```
int sum2(int ar[3][4], int rows);      // допустимое объявление,
                                       // однако 3 игнорируется
```

Это удобно в тех случаях, когда используются typedef:

```
typedef int arr4[4];        // массив arr4, состоящий из 4 значений int
typedef arr4 arr3x4[3];     // массив arr3x4, состоящий из 3 массивов arr4
int sum2(arr3x4 ar, int rows);     // то же, что и следующее объявление
int sum2(int ar[3][4], int rows); // то же, что и следующее объявление
int sum2(int ar[][4], int rows);  // стандартная форма
```

В общем случае, чтобы объявить указатель, соответствующий *N*-мерному массиву, вы должны снабдить значениями все комплекты квадратных скобок, кроме самой левой пары:

```
int sum4d(int ar[][12][20][30], int rows);
```

Это объясняется тем, что первый комплект квадратных скобок указывает на наличие указателя, в то время как остальные квадратные скобки описывают типы объектов данных, на которые ссылаются указатели, как показано в следующем эквиваленте прототипа:

```
int sum4d(int (*ar)[12][20][30], int rows);   // ar - это указатель
```

В данном случае ar представляет собой указатель на массив 12×20×30 значений типа int.

Массивы переменной длины

Вы, должно быть, уже заметили некоторую странность, характерную для функций, выполняющих обработку двумерных массивов: вы можете описать количество строк посредством параметра функции, однако количество столбцов встроено в функцию. Например, посмотрите на это определение:

```
#define COLS 4
int sum2d(int ar[][COLS], int rows)
{
   int r;
   int c;
   int tot = 0;
   for (r = 0; r < rows; r++)
      for (c = 0; c < COLS; c++)
          tot += ar[r][c];
   return tot;
}
```

Далее, предположим, что были объявлены следующие массивы:

```
int array1[5][4];
int array2[100][4];
int array3[2][4];
```

Вы можете использовать функцию `sum2d()` для обработки следующих массивов:

```
tot = sum2d(array1, 5);      // сумма элементов массива 5 x 4
tot = sum2d(array2, 100);    // сумма элементов массива 100 x 4
tot = sum2d(array3, 2);      // сумма элементов массива 2 x 4
```

Это объясняется тем, что количество строк передается в параметр `rows`, представляющий собой переменную. Однако если вы захотите просуммировать массив размерности 6×5, вам придется воспользоваться другой функцией, той, в которой `COLS` принимает значение 5. Такое поведение есть результатом того факта, что для определения размерности массива вы должны воспользоваться константами; в силу этого обстоятельства, вы должны заменить параметр `COLS` переменной.

Если вы на самом деле хотите создать отдельную функцию, способную работать с любым двумерным массивом, вы сможете сделать это, но при этом получите очень неуклюжий результат. (Вам придется передать массив в виде одномерного массива и иметь под рукой функцию, чтобы вычислять, с какого места начинается каждая строка.) Более того, этот метод недостаточно гладко вписывается в подпрограммы на языке FORTRAN, который позволяет программисту задавать в вызове функции оба размера массива. FORTRAN можно считать древним языком программирования, но в течение многих десятилетий эксперты в области численных методов разработали множество полезных вычислительных библиотек на языке FORTRAN. Ожидалось, что C станет наследником FORTRAN, таким образом, возможность корректного переноса библиотек FORTRAN на C представляется весьма полезной.

Эта потребность для стандарта C99 была главным импульсом, побудившим введение концепции массивов переменной длины, которые позволяют использовать переменные для указания размеров массива.

Например, вы можете сделать следующее:

```
int quarters = 4;
int regions = 5;
double sales[regions][quarters];          // массив переменной длины
```

Как говорилось выше, на использование массивов переменной длины накладываются определенные ограничения. Для них должен быть предусмотрен класс автоматической памяти, это означает, что они объявляются либо в функции, либо как параметры функции. Кроме того, вы не можете их инициализировать в объявлении.

Массивы переменной длины не меняют размера

Термин *переменный* в отношении массивов переменной длины не означает, что можно менять длину массива после того, как вы его создадите. Будучи созданным, массив переменной длины сохраняет свои размеры. Термин *переменный* означает, что вы можете использовать переменные при описании размерности массива.

Поскольку массивы переменной длины представляют собой новое языковое средство, поддержка его в настоящее время пока не отличается полнотой и стабильностью. Рассмотрим простой пример, демонстрирующий, как следует писать функцию, которая выполняет суммирование содержимого любого двумерного массива значений `int`.

Прежде всего, покажем, как объявлять функцию с аргументом в виде двумерного массива переменной длины:

```
int sum2d(int rows, int cols, int ar[rows][cols]);
  // ar - массив переменной длины
```

Обратите внимание на то, что два первых параметра (строки и столбцы) используются как размерность для объявления параметра массива `ar`. Поскольку в объявлении массива `ar` присутствуют строки и столбцы, они должны быть объявлены до того, как `ar` появится в списке параметров. В силу этого обстоятельства следующий прототип является ошибочным:

```
int sum2d(int ar[rows][cols], int rows, int cols); // неправильный порядок
```

Стандарт C99 утверждает, что вы можете в прототипе опускать имена, но в этом случае вы должны заменить опущенные размерности звездочками:

```
int sum2d(int, int, int ar[*][*]);
  // ar – массив переменной длины, имена опущены
```

Во-вторых, посмотрите, как определять функцию:

```
int sum2d(int rows, int cols, int ar[rows][cols])
{
   int r;
   int c;
   int tot = 0;

   for (r = 0; r < rows; r++)
       for (c = 0; c < cols; c++)
           tot += ar[r][c];

   return tot;
}
```

Без учета заголовка новой функции, единственное отличие от классической версии этой функции на языке C (листинг 10.17) состоит в том, что константа COLS была заменена переменной cols. Массив переменной длины в заголовке функции является именно тем средством, которое позволяет проводить такие изменения. Кроме того, наличие переменных, которые представляют как количество строк, так и количество столбцов массива, позволяет использовать новую функцию sum2d() с любыми размерами двумерного массива значений int. Листинг 10.18 служит иллюстрацией этого утверждения. В то же время, для этого требуется компилятор языка C, в котором это новое свойство реализовано. Программа в листинге 10.18 также показывает, что эта функция, построенная на основе свойств массивов переменной длины, может использоваться как с любыми традиционными массивами языка C, так и с массивами переменной длины.

Листинг 10.18. Программа vararr2d.c

```
//vararr2d.c -- функции, использующие массивы переменной длины
#include <stdio.h>
#define ROWS 3
#define COLS 4
int sum2d(int rows, int cols, int ar[rows][cols]); int main(void)
{
   int i, j;
   int rs = 3;
   int cs = 10;
   int junk[ROWS][COLS] = {
         {2,4,6,8},
         {3,5,7,9},
         {12,10,8,6}
   };
   int morejunk[ROWS-1][COLS+2] = {
         {20,30,40,50,60,70},
         {5,6,7,8,9,10}
   };
   int varr[rs][cs];                          // массив переменной длины
   for (i = 0; i < rs; i++)
      for (j = 0; j < cs; j++)
         varr[i][j] = i * j + j;
   printf("Массив размерности 3x5\n");
   printf("Сумма всех элементов = %d\n",
          sum2d(ROWS, COLS, junk));
   printf("Массив размерности 2x6 \n");
   printf("Сумма всех элементов = %d\n",
         sum2d(ROWS-1, COLS+2, morejunk));
   printf("Массив переменной длины размерности 3x10\n");
   printf("Сумма всех элементов = %d\n",
         sum2d(rs, cs, varr)); return 0;
}
```

```
// функция с параметром в виде массива переменной длины
int sum2d(int rows, int cols, int ar[rows][cols])
{
   int r;
   int c;
   int tot = 0;
   for (r = 0; r < rows; r++)
       for (c = 0; c < cols; c++)
           tot += ar[r][c];
   return tot;
}
```

Выходные данные этой программы имеют следующий вид:

```
Массив размерности 3x5
Сумма всех элементов = 80
Массив размерности 2x6
Сумма всех элементов = 315
Массив переменной длины размерности 3x10
Сумма всех элементов = 270
```

Следует отметить тот факт, что объявление массива в списке параметров объявления функции на самом деле не приводит к созданию массива. Как и в случае старого синтаксиса, фактическое имя массива переменной длины является указателем. Это значит, что функция с параметром в виде массива переменной фактически работает с данными исходного массива, и в силу этого обстоятельства способна вносить изменения в массив, переданный ей в качестве аргумента. Следующий фрагмент кода показывает, когда необходимо объявлять указатель, а когда — фактический массив:

```
    int thing[10][6];
    twoset(10,6,thing);
    ...
}
void twoset (int n, int m, int ar[n][m])
  // ar представляет собой указатель на массив из m значений типа int
{
    int temp[n][m];  // temp - массив размерности n x m значений типа int
    temp[0][0] = 2;  // установить значение элемента массива temp равным 2
    ar[0][0] = 2;    // установить значение элемента thing[0][0] равным 2
}
```

Когда вызывается функция `twoset()`, как показано, `ar` становится указателем на элемент `thing[0]`, а `temp` создается как массив размерности 10×6. Поскольку как `ar`, так и `thing` являются указателями на `thing[0]`, то `ar[0][0]` получает доступ к той же ячейке памяти, что и `thing[0][0]`.

Массивы переменной длины позволяют также реализовать динамическое распределение памяти. Это означает, что вы можете задавать размеры массива во время выполнения программы. Для обычных массивов выполняется статическое распределение памяти, означающее, что размер массив известен во время компиляции. Именно по этой причине размеры массива, являющиеся константами, должны быть известны компилятору заранее. Вопросы динамического распределения памяти рассматриваются в главе 12.

Составные литералы

Предположим, что вы хотите передать в функцию некоторое значение с помощью параметра типа int; вы можете передать переменную типа int, но вы также можете передать константу типа int, такую как 5. До появления стандарта C99 возможности реализации функции с аргументом в виде массива были другими; вы могли передать массив, однако тогда не было ничего такого, что могло бы служить эквивалентом константы типа массив. Стандарт C99 изменил эту ситуацию, вводя *составные литералы*. Литералы — это константы, которые не являются символическими константами. Например, 5 есть литеральная константа типа int, 81.3 — литеральная константа типа double, 'Y' — литерал типа char, а "elephant" — строковая литеральная константа. Комитет, который разрабатывал стандарт C99, пришел к заключению, что удобнее будет иметь составные литеральные константы, которые могут представлять содержимое массивов и структур.

С точки зрения массивов, составной литерал выглядит как список инициализации массивов, которому предшествует имя типа, заключенное в круглые скобки. Например, ниже показано обычное объявление массива:

```
int diva[2] = {10, 20};
```

Здесь используется составная литеральная константа, которая создает безымянный массив, содержащий те же значения типа int:

```
(int [2]){10, 20}              // составная литеральная константа
```

Обратите внимание, что именем типа есть то, у вас остается, когда из удалите diva из предыдущего объявления, то есть int [2].

Точно так же, как вы опускаете размерность массива при инициализации именованного массива, вы можете опустить его в составной литеральной константе, а компилятор сам подсчитает, сколько имеется элементов в массиве:

```
(int []){50, 20, 90}           // составной литерал с тремя элементами
```

Поскольку эти составные литеральные константы не имеют имени, вы можете создавать их в каком-то одном операторе и использовать в дальнейшем по мере необходимости. С другой стороны, вы как-то должны их использовать в момент их создания. Один из способов состоит в применении указателя для отслеживания местоположения соответствующей ячейки памяти. То есть, вы можете написать нечто подобное показанному ниже:

```
int * pt1;
pt1 = (int [2]) {10, 20};
```

Следует также отметить, что эта литеральная константа идентифицируется как массив значений тип int. Подобно имени массива, она преобразуется в адрес первого элемента, таким образом, она может быть присвоена указателю на значение типа int. Этот указатель вы можете использовать позже. Например, *pt1 в этом случае будет иметь значение 10, а pt1[1] — значение 20.

Другой прием, который становится возможным благодаря составным литеральным константам, заключается в том, что вы можете передать его в качестве фактического аргумента в функцию с соответствующим формальным параметром:

```
int sum(int ar[], int n);
...
int total3;
total3 = sum((int []){4,4,4,5,5,5}, 6);
```

В данном случае первый аргумент представляет собой массив, состоящий из шести элементов типа int, который действует как адрес первого элемента, то есть так же, как и имя массива. Этот вид использования, в рамках которого вы передаете информацию функции без необходимости заранее создавать массив, типичен для составных литеральных констант.

Вы можете распространить этот метод на двумерные и многомерные массивы. Вот как в этом случае создается двумерный массив значений типа int и сохраняется его адрес:

```
int (*pt2)[4]; //объявление указателя на массив массивов 4 значений типа int
pt2 = (int [2][4]) { {1,2,3,-9}, {4,5,6,-8} };
```

В данном случае типом является int [2][4], то есть массив размерности 2×4 значений типа int.

Листинг 10.19 объединяет все эти примеры в одну полную программу.

Листинг 10.19. Программа `flc.c`

```
// flc.c -- странно выглядящие константы
#include <stdio.h>
#define COLS 4
int sum2d(int ar[][COLS], int rows);
int sum(int ar[], int n);

int main(void)
{
   int total1, total2, total3;
   int * pt1;
   int (*pt2)[COLS];

   pt1 = (int [2]) {10, 20};
   pt2 = (int [2][COLS]) { {1,2,3,-9}, {4,5,6,-8} };

   total1 = sum(pt1, 2);
   total2 = sum2d(pt2, 2);
   total3 = sum((int []){4,4,4,5,5,5}, 6);
   printf("total1 = %d\n", total1);
   printf("total2 = %d\n", total2);
   printf("total3 = %d\n", total3);

   return 0;
}

int sum(int ar[], int n)
{
   int i;
   int total = 0;
   for( i = 0; i < n; i++)
       total += ar[i];
   return total;
}
```

```
int sum2d(int ar[][COLS], int rows)
{
   int r;
   int c;
   int tot = 0;
   for (r = 0; r < rows; r++)
       for (c = 0; c < COLS; c++)
           tot += ar[r][c];
   return tot;
}
```

Для этой программы необходим компилятор, в котором реализовано рассматриваемое дополнение к стандарту C99 (на текущий момент очень немногие компиляторы способны делать это). Ниже показаны выходные данные этой программы:

```
total1 = 30
total2 = 4
total3 = 27
```

Ключевые понятия

Когда возникает необходимость хранить множество значений одного и того же типа, наиболее подходящим средством для решения этой задачи является массив. Язык C рассматривает массивы как *производные типы*, поскольку они построены на основе других типов. Другими словами, вы не просто объявляете массив значений типа `int` или типа `float` или какого-либо другого типа. Этот другой тип сам по себе является типом массива, и в конечном итоге получается массив массивов, или двумерный массив.

Иногда бывает полезно предусмотреть функции для обработки массивов, это позволяет повысить модульность программы за счет решения конкретных задач в рамках специализированных функций. Важно понимать, что когда вы используете имя массива в качестве фактического аргумента, вы не передаете этой функции весь массив, вы просто передаете адрес массива (следовательно, соответствующий формальный параметр функции должен быть указателем). Чтобы обработать этот массив, функция должна знать, где хранится массив и сколько элементов содержит этот массив. Адрес массива указывает "где"; данные о том "сколько" должны либо быть встроены в функцию, либо передаваться ей в качестве отдельного аргумента. Второй подход носит более общий характер, что позволяет одной и той же функции работать с массивами различной размерности.

Связь между массивами и указателями настолько тесная, что вы часто можете представить одну и ту же операцию, употребляя форму записи с использованием массивов и форму записи через указатели. Именно эта связь позволяет применять форму записи массивов в функции, обрабатывающей массивы даже в тех случаях, когда формальный параметр является указателем, а не массивом.

В языке C вы должны задавать размерность обычного массива константным выражением, благодаря чему размерность обычного массива становится известной во время компиляции. Стандарт C99 предлагает альтернативу в виде массива переменной длины, когда спецификатором размерности может быть переменная. Это позволяет задавать размеры массива переменной длины во время выполнения программы.

Резюме

Массив — это набор элементов одного и того же типа. Элементы хранятся в памяти в виде последовательности, а доступ к ним осуществляется с помощью целочисленного индекса, или *смещения*. В языке C первый элемент массива имеет индекс `0`, следовательно, завершающий элемент массива, содержащего `n` элементов, имеет индекс `n - 1`. Контроль за правильным использованием индексов возлагается на программиста, поскольку ни компилятор, ни исполняемая программа не следят за этим процессом.

Для объявления простого *одномерного* массива используется следующая форма:

```
тип имя[размер];
```

Здесь *тип* указывает тип данных каждого элемента массива, *имя* — это имя массива, а размер задает количество элементов. Традиционно язык C требовал, чтобы размер был константным целочисленным выражением. Стандарт C99 позволяет использовать неконстантное целочисленное выражение; в рассматриваемом случае массив трактуется как массив переменной длины.

Язык C интерпретирует имя массива как адрес первого элемента этого массива. В другой терминологии имя массива эквивалентно указателю на первый элемент массива. В общем случае массивы и указатели тесно связаны друг с другом. Если `ar` есть массив, то выражения `ar[i]` и `*(ar + i)` эквивалентны.

Язык C не допускает передачу всего массива в качестве аргумента функции, однако, вы можете передать в качестве аргумента адрес массива. Далее функция может использовать этот адрес для манипулирования исходным массивом. Если функция не предназначается для модификации исходного массива, вы должны воспользоваться ключевым словом `const` при объявлении формального параметра, представляющего массив. Вы можете использовать в вызывающей функции как запись в форме массива, так и запись в форме указателей. В любом случае на практике используется переменная типа указатель.

Добавление к указателю целого значения или инкремент указателя меняет значение указателя на число байтов, которые занимает в памяти объект, на который нацелен указатель. Иначе говоря, если `pd` указывает на 8-байтовое значение типа `double` в массиве, добавление 1 к указателю `pd` увеличивает его значение на 8, следовательно, этот указатель будет ссылаться на следующий элемент массива.

Двумерные массивы представляют собой массивы массивов. Например, объявление

```
double sales[5][12];
```

создает массив с именем `sales`, состоящий из пяти элементов, каждый из которых представляет собой массив из 12 значений типа `double`.

На первый из этих одномерных массивов можно ссылаться как `sales[0]`, на второй — `sales[1]` и так далее, при этом каждый из этих массивов содержит 12 значений типа `double`. Второй индекс служит для доступа к конкретным элементам в этих массивах. Например, `sales[2][5]` — шестой элемент массива `sales[2]`, а `sales[2]` — третий элемент массива `sales`.

Традиционный метод языка C передачи многомерного массива в функцию предусматривает передачу имени массива, которое является адресом этого массива, параметру в форме указателя на значение подходящего типа.

Объявление такого указателя должно описать все размерности массива, кроме первой; размерность первого параметра обычно передается во втором аргументе. Например, чтобы обработать ранее упоминавшийся массив sales, прототип функции и вызов функции должны иметь вид:

```
void display(double ar[][12], int rows);
...
display(sales, 5);
```

Массивы переменной длины позволяют применять второй синтаксис, в рамках которого обе размерности передаются функции в качестве аргументов. В этом случае прототип функции и вызов функции принимают следующий вид:

```
void display(int rows, int cols, double ar[rows][cols]);
...
display(5, 12, sales);
```

В ходе описанных выше рассуждений были использованы массивы значений типа int и массивы значений типа double, однако все, что было сказано выше, применимо и к массивам других типов. В то же время для символьных строк предусматривается целый набор специальных правил. Это объясняется тем фактом, что завершающий символ пробела в строке позволяет функции обнаружить конец строки без необходимости передачи ей размера. Символьные строки подробно рассматриваются в главе 11.

Вопросы для самоконтроля

1. Что выводит на экран следующая программа?

```
#include <stdio.h>
int main(void)
{
   int ref[] = {8, 4, 0, 2};
   int *ptr;
   int index;
   for (index = 0, ptr = ref; index < 4; index++, ptr++)
      printf("%d %d\n", ref[index], *ptr);
   return 0;
}
```

2. Сколько элементов содержит массив ref из вопроса 1?

3. Адресом чего является ref в вопросе 1? Что можно сказать о ref + 1? На что ссылается выражение ++ref?

4. Какими являются значения *ptr и *(ptr + 2) в каждом из следующих случаев?

   а.
```
int *ptr;
int torf[2][2] = {12, 14, 16};
ptr = torf[0];
```

   б.
```
int * ptr;
int fort[2][2] = { {12}, {14,16} };
ptr = fort[0];
```

5. Какие значения принимают выражения `**ptr` и `**(ptr + 1)` в каждом из следующих функций?

   а.
   ```
   int (*ptr)[2];
   int torf[2][2] = {12, 14, 16};
   ptr = torf;
   ```

   б.
   ```
   int (*ptr)[2];
   int fort[2][2] = { {12}, {14,16} };
   ptr = fort;
   ```

6. Предположим, что имеет следующее выражение:

   ```
   int grid[30][100];
   ```

   а. Выразите адрес `grid[22][56]` одним способом.

   б. Выразите адрес `grid[22][0]` двумя способами.

   в. Выразите адрес `grid[0][0]` тремя способами.

7. Напишите соответствующее объявление для каждой из следующих переменных:

   а. `digits` представляет собой массив из 10 значений типа `int`.

   б. `rates` представляет собой массив из шести значений типа `float`.

   в. `mat` представляет собой массив, состоящий из трех массивов, каждый из которых содержит 5 целых значений.

   г. `psa` представляет собой массив, состоящий из 20 указателей, ссылающихся на значение типа `char`.

   д. `pstr` представляет собой указатель на массив, состоящий из 20 значений типа `char`.

8. а. Объявите массив, состоящий из шести значений типа `int`, и инициализируйте его значениями 1, 2, 4, 8, 16 и 32.

   б. Используйте запись в форме массива для представления третьего элемента (имеющего значение 4) массива, заданного в пункте а).

   в. Предполагая, что правила стандарта C99 вступили в силу, объявите массив из 100 значений типа `int` и инициализируйте его таким образом, чтобы последний элемент получил значение `-1`; значения остальных элементов могут быть произвольными.

9. Каким должен быть диапазон значений индексов массива, состоящего из 10 элементов?

10. Предположим, что имеются следующие объявления:

    ```
    float rootbeer[10], things[10][5], *pf, value = 2.2;
    int i = 3;
    ```

    Назовите, какие из приведенных ниже операторов допустимы, а какие — нет:

    а. `rootbeer[2] = value;`

    б. `scanf("%f", &rootbeer );`

    в. `rootbeer = value;`

    г. `printf("%f", rootbeer);`

    д. `things[4][4] = rootbeer[3];`

    е. `things[5] = rootbeer;`

    ж. `pf = value;`

    з. `pf = rootbeer;`

11. Объявите массив значений типа int размерности 800×600.
12. Пусть заданы объявления трех массивов:

```
double trots[20];
short clops[10][30];
long shots[5][10][15];
```

   а. Напишите прототип и оператор вызова для обычной функции типа void, которая обрабатывает элементы массива trots, и для функции, использующей массивы переменной длины.

   б. Напишите прототип и оператор вызова для обычной функции типа void, которая обрабатывает элементы массива clops, и для функции, использующей массивы переменной длины.

   в. Напишите прототип и оператор вызова для обычной функции типа void, которая обрабатывает элементы массива shots, и для функции, использующей массивы переменной длины.

13. Заданы два прототипа функций:

```
void show(double ar[], int n);          // n - количество элементов
void show2(double ar2[][3], int n);     // n - количество строк
```

   а. Напишите оператор вызова функции show(), который передает функции составную литеральную константу, содержащую значения 8, 3, 9 и 2.

   б. Напишите оператор вызова функции show(), который передает функции составную литеральную константу, содержащую 8, 3 и 9 в качестве первой строки и значения 5, 4 и 1 в качестве второй строки.

Упражнения по программированию

1. Внесите изменения в программу rain, представленную в листинге 10.7, с таким расчетом, чтобы она выполняла вычисления, используя указатели вместо индексов массивов. (Вам по-прежнему придется объявлять и инициализировать соответствующий массив.)
2. Напишите программу, которая инициализирует некоторый массив значений типа double, а затем копирует содержимое этого массива в два других массива. (Все три массива должны быть объявлены в основной программе.) Для создания первой копии воспользуйтесь функцией, в которой применяется запись в форме массива. Для создания второй копии воспользуйтесь функцией, в которой применяется запись в форме указателя и инкрементирование указателя. Каждая функция должна принимать в качестве аргументов имя искомого (целевого) массива и количество элементов, подлежащих копированию. Иначе говоря, с учетом соответствующих объявлений, вызовы функций должны иметь следующий вид:

```
double source[5] = {1.1, 2.2, 3.3., 4.4, 5.5};
double target1[5];
double target2[5];
copy_arr(source, target1, 5);
copy_ptr(source, target1, 5);
```

3. Напишите функцию, которая возвращает наибольшее значение из массива значений типа `int`. Протестируйте эту функцию с помощью простой программы.
4. Напишите функцию, которая возвращает индекс наибольшего значения из массива значений типа `double`. Протестируйте эту функцию с помощью простой программы.
5. Напишите функцию, которая возвращает разность между наибольшим и наименьшим значениями из массива типа `double`. Протестируйте эту функцию с помощью простой программы.
6. Напишите программу, которая инициализирует двумерный массив значений типа `double` и использует одну из функций копирования из упражнения 2 для его копирования во второй двумерный массив. (Поскольку двумерный массив представляет собой массив массивов, функция, предназначенная для копирования одномерных массивов, может использоваться для работы с каждым из подмассивов.)
7. Воспользуйтесь одной из функций копирования из упражнения 2 для копирования элементов с третьего по пятый семиэлементного массива в массив, состоящий из трех элементов. Саму функцию менять не надо, достаточно всего лишь правильно выбрать фактические аргументы. (Фактическими аргументами не обязательно должны быть имя массива и размер массива. Достаточно, чтобы такими аргументами были адрес элемента массива и количество обрабатываемых элементов.)
8. Напишите программу, которая инициализирует двумерный массив значений типа `double` размерности 3×5 и использует функцию, ориентированную на работу с массивами переменной длины, для копирования этого массива во второй двумерный массив. Кроме того, напишите функцию, ориентированную на работу с массивами переменной длины, для отображения содержимого этих двух массивов. В общем случае обе эти функции должны быть способны выполнять обработку произвольных массивов размерности $N \times M$. (Если вы не имеете доступа к компилятору, способному работать с массивами переменной длины, воспользуйтесь традиционным подходом языка C, применяемым к функциям, которые могут выполнять обработку массивов $N \times 5$).
9. Напишите функцию, которая устанавливает значение элемента массива равным сумме соответствующих элементов двух других массивов. Иначе говоря, если массив 1 имеет значения `2`, `4`, `5` и `8`, а массив 2 — значения `1`, `0`, `4` и `6`, эта функция присваивает массиву 3 значения `3`, `4`, `9` и `14`. Эта функция должна принимать имена трех массивов и их размерности в качестве аргументов. Протестируйте эту функцию с помощью простой программы.
10. Напишите программу, которая объявляет массив размерностью 3×5 и выполняет его инициализацию значениями по вашему выбору. Программа должна вывести эти значения на экран, удвоить все значения, после чего вывести на экран новые значения. Напишите одну функцию для вывода значений на экран и другую функцию для удваивания значений. В качестве аргументов функции принимают имя массивов и количество строк.

11. Перепишите программу `rain` из листинга 10.7 таким образом, чтобы основные задачи выполнялись соответствующими функциями, но не в теле функции `main()`.

12. Напишите программу, которая предлагает пользователю ввести три набора, каждый из которых содержит по пять чисел типа `double`.
    Программа должна выполнять следующие действия:
    а. Запоминать информацию в массиве размерности 3×5.
    б. Вычислять среднее значение каждого набора из пяти чисел.
    в. Вычислять среднее значение всех чисел.
    г. Определять наибольшее из 15 значений.
    д. Выводить на экран сообщение с результатами вычислений.

    Каждая более-менее крупная задача должна решаться специальной функцией с использованием традиционного для языка C подхода к обработке массивов. Выполните задачу б) с помощью функции, которая вычисляет и возвращает среднее значение одномерного массива; воспользуйтесь циклом для вызова этой функции три раза. Функции, реализующие остальные задачи, должны получать в качестве аргумента весь массив, а функции, выполняющие задачи в) и г) должны возвращать ответ в вызывающую программу.

13. Выполните упражнение 12, но в качестве параметров функции используйте массивы переменной длины.

ГЛАВА 11

Символьные строки и строковые функции

В этой главе:

- Функции: `gets()`, `puts()`, `strcat()`, `strncat()`, `strcmp()`, `strncmp()`, `strcpy()`, `strncpy()`, `sprintf()`, `strchr()`
- Создание и использование строк
- Использование строковых и символьных функций из библиотеки C и создание собственных строковых функций
- Использование аргументов командной строки

Символьная строка — это один из наиболее полезных и важных типов данных в языке C. До сих пор вы постоянно использовали символьные строки, и вам еще многое предстоит узнать о них. Библиотека функций C предлагает широкий спектр функций для чтения и записи, копирования, сравнения, комбинирования, поиска и выполнения других операций со строками. Эта глава поможет вам включить эти функции в арсенал используемых программных средств.

Введение в строки и строковый ввод-вывод

Разумеется, вам уже известен наиболее важный факт: *символьная строка* представляет собой массив значений типа `char`, концевым элементом которого является нулевой символ (`\0`). В силу этого, все сведения о массивах и указателях, которые вы получили в предыдущих главах, могут быть перенесены и на символьные строки. Однако в связи с интенсивным использованием символьных строк C предоставляет пользователю множество функций, предназначенных для работы со строками. В данной главе рассматривается природа строк, способы объявления и инициализации строк, включение и исключение их из программы, а также манипуляции со строками.

В листинге 11.1 показана довольно насыщенная программа, которая служит иллюстрацией нескольких способов формирования, считывания и печати строк. В ней используются две новых функции — функция `gets()`, которая читает строку, и функция `puts()`, которая выводит строку на печать. (Вы, должно быть, обратили внимание на

их семейное сходство с функциями getchar() и putchar().) В остальном программа не содержит ничего необычного и незнакомого.

Листинг 11.1. Программа strings.c

```
// strings.c -- коллекционирование пользователей
#include <stdio.h>
#define MSG "вы должны обладать многими талантами. Назовите некоторые."
                                        // константа символьной строки
#define LIM 5
#define LINELEN 81                      // максимальная длина строки + 1
int main(void)
{
   char name[LINELEN];
   char talents[LINELEN];
   int i;
                                        // инициализация массива значений
                                        // типа char заданной размерности
   const char m1[40] = "Постарайтесь уложиться в одну строку.";
                                        // пусть компилятор сам вычислит
                                        // размеры массива
   const char m2[] = "Если вам ничего не приходит в голову, придумайте что-нибудь.";
                                        // инициализация указателя
   const char *m3 = "\nВсе, о себе достаточно, а вас как зовут?";
                                        // инициализация массива
                                        // указателей на строку
   const char *mytal[LIM] = {           // массив из 5 указателей
         "Мгновенное складывание чисел",
         "Точное умножение", "Накапливание данных",
         "Исполнение инструкций с точностью до последней буквы",
         "Знание языка программирования C"
         };
   printf("Здравствуйте! Я компьютер по имени Клайд."
          " У меня масса талантов.\n");
   printf("Сейчас я расскажу кое-что о них.\n");
   puts("Что у меня за таланты? Вот только частичный их перечень.");
   for (i = 0; i < LIM; i++)
        puts(mytal[i]);                 // печать талантов компьютера
   puts(m3);
   gets(name);
   printf("Хорошо, %s, %s\n", name, MSG);
   printf("%s\n%s\n", m1, m2);
   gets(talents);
   puts("Посмотрим, есть ли у меня этот перечень:");
   puts(talents);
   printf("Благодарю за информацию, %s.\n", name);
   return 0;
}
```

Чтобы продемонстрировать, на что способна эта программа, запустим ее на выполнение:

```
Здравствуйте! Я компьютер по имени Клайд. У меня масса талантов.
Сейчас я расскажу кое-что о них.
Что у меня за таланты? Вот только частичный их перечень.
Мгновенное складывание чисел
Точное умножение
Накапливание данных
Исполнение инструкций с точностью до последней буквы
Знание языка программирования С

Все, о себе достаточно, а вас как зовут?
Смарт Компетентный
Хорошо, Смарт Компетентный, вы должны обладать многими талантами.
Назовите некоторые.
Постарайтесь уложиться в одну строку.
Если вам ничего не приходит в голову, придумайте что-нибудь.
Словесная перепалка, несение чепухи, симуляция, фальшь и вздохи.
Посмотрим, есть ли у меня этот перечень:
Словесная перепалка, несение чепухи, симуляция, фальшь и вздохи.
Благодарю за информацию, Смарт Компетентный.
```

Мы сейчас не будем изучать листинг 11.1 строка за строкой, а применим более общий подход. Прежде всего, рассмотрим способы определения строк в программах. Затем покажем, что необходимо для чтения строки в программу. И, наконец, рассматрим способы вывода строк.

Определение строк в программе

Как вы, возможно, уже заметили, знакомясь с листингом 11.1, существует множество способов определения строк. Основные способы предполагают использование строковых констант, массивов типа `char`, указателей типа `char` и массивов символьных строк. При этом необходимо убедиться в наличии места в памяти, где можно сохранить строку, и позже этот вопрос еще будет рассматриваться.

Константы типа символьной строки (строковые литералы)

Строковая константа, которую также называют *строковым литералом*, представляет собой произвольную последовательность символов, заключенную в кавычки. Заключенные в кавычки символы и завершающий символ `\0`, который автоматически добавляется компилятором, сохраняются в памяти в виде символьной строки. Программа использует несколько таких констант типа символьной строки, которые чаще всего выступают в качестве аргументов функций `printf()` и `puts()`. Обратите также внимание и на возможность использования директивы `#define` для определения констант типа символьной строки.

Вспомните, что ANSI C выполняет конкатенацию строковых литералов, если они не отделены друг от друга или если разделены пробельными символами. Например,

```
char greeting[50] = "Здравствуйте, ну и"" как вы себя"  " чувствуете"
                    " сегодня?";
```

эквивалентно следующей конструкции:

```
char greeting[50] = "Здравствуйте, ну и как вы себя чувствуете сегодня?";
```

Если вы хотите использовать двойные кавычки в строке, поставьте перед этими кавычками обратную косую черту, как показано в следующем примере:

```
printf("\"Беги, Спот, беги!\" - воскликнул Дик.\n");
```

В этом случае получим следующие данные на выходе программы:

```
"Беги, Спот, беги!" - воскликнул Дик.
```

Константы типа символьной строки размещаются в классе *статической памяти*, это значит, что если вы используете в функции строковую константу, эта константа запоминается только один раз и остается неизменной в течение времени выполнения программы, даже если эта функции вызывается многократно. Вся фраза, заключенная в кавычки, выступает в роли указателя на то место в памяти, в котором хранится эта строка. Это действие аналогично имени массива, трактуемого как указатель на ячейку памяти, в которой хранится массив. Если это так, то какие выходные данные должна генерировать программа, приведенная в листинге 11.2?

Листинг 11.2. Программа `quotes.c`

```
/* quotes.c -- строки как указатели */
#include <stdio.h>
int main(void)
{
   printf("%s, %p, %c\n", "We", "are", *"space farers");
   return 0;
}
```

Формат `%s` должен печатать строку `We`. Формат `%p` печатает адрес. Следовательно, если фраза `"are"` представляет собой адрес, то формат `%p` должен печатать адрес первого символа строки. (Все реализации, применяемые до появления стандарта ANSI, могли пользоваться спецификаторами `%u` или `%lu` вместо `%p`.) Наконец, операция разыменования `*"space farers"` должна генерировать значение адреса, на который ссылается указатель и которым должен быть первый символ строки `"space farers"`. Случится ли это на самом деле? Ниже показаны выходные данные:

```
We, 0x0040c010, s
```

Массивы символьных строк и инициализация

Когда вы определяете массив символьных строк, вы должны уведомить компилятор о том, сколько памяти требуется для его хранения. Один из способов состоит в том, чтобы указать размер массива, достаточный для хранения в нем соответствующей строки. Приведенное ниже объявление инициализирует массив `m1` символами заданной строки:

```
const char m1[40] = "Постарайтесь уложиться в одну строку.";
```

Ключевое слово `const` отражает желание сохранять эту строку неизменной.

Эта форма является сокращенной формой инициализации стандартного массива:

```
const char m1[40] = { 'П',
'о', 'с', 'т', 'а', 'р', 'а', 'й', 'т', 'е', 'с', 'ь', ' ',
'у', 'л', 'о', 'ж', 'и', 'т', 'ь', 'с', 'я',
' ', 'в', ' ', 'о', 'д', 'н', 'у', ' ', 'с', 'т',
'р', 'о', 'к', 'у', '.', '\0'
};
```

Обратите внимание на завершающий нулевой символ. Без него вы получили бы символьный массив, а не строку.

Когда вы задаете размер массива, убедитесь с том, что количество элементов, по меньшей мере, на единицу больше (опять-таки для нулевого символа) длины строки. Любые неиспользованные элементы инициализируются значением 0 (который в форме char представляет нулевой символ, но не символ цифры "0"). Соответствующая иллюстрация показана на рис. 11.1.

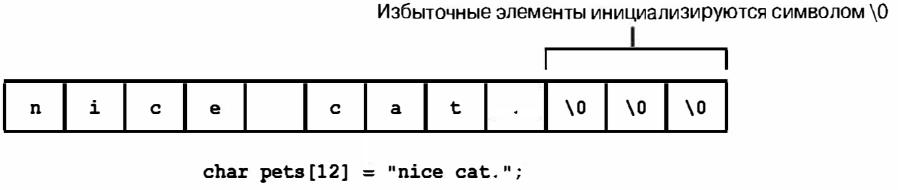


Рис. 11.1. *Инициализация массива*

Часто удобно оставить задачу определения размера массива за компилятором; если вы указываете размер в инициализирующем объявлении, компилятор самостоятельно определит этот размер:

```
const char m2[] = "Если вам ничего не приходит в голову, придумайте что-нибудь.";
```

Инициализация символьных массивов представляет собой один из тех случаев, когда имеет смысл возложить задачу определения размеров массива на компилятор. Вот почему функциям, выполняющим обработку массивов, не обязательно знать размер массива, поскольку им просто достаточно найти нулевой символ, отмечающий конец массива.

Обратите внимание, что в программе должен быть явно назначен размер массива name:

```
#define LINELEN 81      // максимальная длина строки + 1
...
char name[LINELEN];
```

Поскольку содержимое массива name считывается во время выполнения программы, у компилятора нет возможности заранее знать, какой объем пространства памяти необходимо зарезервировать, пока вы не сообщите конкретное значение. В этом случае строковые константы, символы которых компилятор мог бы подсчитать, отсутствуют, поэтому мы произвольно выбрали значение 80 символов для имени пользователя в надежде, что этого хватит в любом случае. При объявлении массива его размер можно задать в виде целочисленной константы. Вы не можете использовать переменную, значение которой устанавливается во время выполнения программы.

Размер массива фиксируется в программе во время ее компиляции. (Вообще говоря, в стандарте C99 вы могли бы воспользоваться массивом переменной длины, но и в этом случае вы знаете заранее, насколько большим он должен быть.)

```
int n = 8;
char cakes[2 + 5];  // допустим, поскольку размер является
                    // константным выражением
char crumbs[n];     // недопустим до появления стандарта C99,
                    // массив переменной длины в стандарте C99
```

Имя символьного массива, как и любое имя массива, получает адрес первого элемента массива. По этой причине для массива `m1` правильны следующие отношения:

```
m1 == &m1[0] , *m1 == 'П' И *(m1+1) == m1[1] == 'о'
```

Действительно, вы можете использовать запись в форме указателя для установки значения строки. Например, в листинге 11.1 имеется следующее определение:

```
const char *m3 = "\nВсе, о себе достаточно, а вас как зовут?";
```

Это объявление очень похоже на объявление, показанное ниже:

```
char m3[] = "\nВсе, о себе достаточно, а вас как зовут?"
```

Оба объявления говорят о том, что `m3` представляет собой указатель на заданную строку. В обоих случаях строка, заключенная в скобки, сама определяет необходимый объем памяти, который будет для нее зарезервирован. Тем не менее, формы записи не идентичны.

Массивы или указатели

В чем, собственно говоря, заключается различие между записью в форме массива и записью в форме указателя? Запись в форме массива (`m3[]`) приводит к тому, что массив, состоящий из 42 элементов (по одному на каждый символ плюс один элемент для завершающего символа `'\0'`), размещается в памяти компьютера. Каждый элемент инициализируется соответствующим символом. Обычно строка, заключенная в кавычки, хранится в сегменте данных, который является частью исполняемого файла; когда программа загружается в память, вместе с ней загружается и эта строка. Говорят, что строка в кавычках находится в *статической памяти*. Однако память под массив будет распределена после того, как программа начнет выполняться. В этот момент строка, заключенная в кавычки, копируется в массив. (В главе 12 вопросы управления памятью рассматриваются более подробно.) Далее, компилятор рассматривает имя массива `m3` как синоним адреса первого элемента массива, в данном случае, `&m3[0]`. Одна важная деталь в данном случае состоит в том, что при использовании записи в форме массива `m3` — это адресная константа. Вы не можете изменить значений `m3`, поскольку это означает изменение места хранения (адреса), в котором хранится массив. Вы можете использовать операции, подобные `m3+1`, с целью идентификации следующего элемента массива, в то же время операция `++m3` является недопустимой. Операция инкремента может применяться только к именам переменных, но не к константам.

Запись в форме указателя (`*m3`) также приводит к тому, что в статической памяти под данную строку резервируется 42 элемента. Наряду с этим, как только начинается выполнение программы, она резервирует еще одну ячейку памяти для *переменной* `m3` типа указатель и запоминает в ней адрес строки. Эта переменная первоначально ука-

зывает на первый символ строки, но это значение может быть изменено. В силу этого вы можете пользоваться операцией инкремента. Например, `++m3` будет указывать на второй символ (`B`).

Короче говоря, инициализация массива приводит к копированию строки из статической памяти, отведенной под этот массив, в то время как инициализация указателя просто копирует адрес строки.

Важны ли эти различия? Часто нет, однако это зависит от того, что именно вы пытаетесь сделать. Соответствующие примеры представлены в последующих обсуждениях.

Различия между массивами и указателями

Рассмотрим различия между инициализацией символьного массива, предназначенного для хранения строки, и инициализацией указателя, который указывает на эту строку. (Под "указыванием на строку" подразумевается указание на первый символ строки.) Например, рассмотрим два следующих объявления:

```
char heart[] = "Я люблю Тилли!";
char *head = "Я люблю Милли!";
```

Основное отличие между ними состоит в том, что имя массива `heart` является константой, в то время как `head` представляет собой переменную. Каково различие между ними с практической точки зрения?

Прежде всего, в обоих случаях может использоваться запись в форме массива:

```
for (i = 0; i < 7; i++)
   putchar(heart[i]);
putchar('\n');
for (i = 0; i < 7; i++)
   putchar(head[i]));
putchar('\n');
```

Получаем следующие выходные данные:

```
Я люблю
Я люблю
```

Во-вторых, в обоих случаях можно использовать операцию сложения с указателем:

```
for (i = 0; i < 7; i++)
   putchar(*(heart + i));
putchar('\n');
for (i = 0; i < 7; i++)
   putchar(*(head + i));
putchar('\n');
```

И снова получаем те же выходные данные:

```
Я люблю
Я люблю
```

Однако только в записи в форме указателя может применяться операция инкремента:

```
while (*(head) != '\0')    /* остановиться в конце строки */
   putchar(*(head++));     /* вывести символ, переместить указатель */
```

Получаем следующие выходные данные:

```
Я люблю Милли!
```

Предположим, что необходимо, чтобы head и heart совпадали. Это можно сделать следующим образом:

```
head = heart;           /* head теперь указывает на массив heart */
```

Представленный оператор заставляет указатель head указывать на первый элемент массива heart.

Тем не менее, вы не можете поступить так:

```
heart = head;           /* недопустимая конструкция */
```

Ситуация аналогична ситуации с операторами x = 3; и 3 = x;. В левой части оператора присваивания должна быть переменная, или в более общем случае, *l-значение*, такое как *p_int. Кстати, оператор head = heart; не приводит к затиранию строки о Милли; она всего лишь меняет адрес, сохраненные в head. Однако если вы где-то не сохраните адрес строки "Я люблю Милли!", то не сможете получить доступ к этой строке после того, как head станет указывать на другую ячейку памяти.

Существует способ, позволяющий изменить сообщение heart, для этого нужно обращаться к отдельным элементам массива:

```
heart[8]= 'M';
```

или

```
*(heart + 8) = ' M ';
```

Элементы массива представляют собой переменные (если только массив не объявлен как const), в то же время *имя* массива переменной не является.

Вернемся к теме инициализации указателя:

```
char * word = "frame";
```

Можно ли воспользоваться указателем для изменения строки?

```
word[1] = 'l';              // допустимо??
```

Ваш компилятор, возможно, позволяет это, однако по условиям текущего стандарта C последствия действий подобного рода не прогнозируются. Такой оператор может, например, стать источником ошибки при доступе к памяти. Причина заключается в том, что компилятор может выбрать вариант представления всех идентичных строковых литералов в виде единственной копии в памяти. Например, все приведенные ниже операторы могут ссылаться на единственную ячейку памяти, в которой хранится строка "Klingon":

```
char * p1 = "Klingon";
p1[0] = 'F';                // все ли правильно?
printf("Klingon");
printf(": Beware the %ss!\n", "Klingon");
```

Другими словами, компилятор может заменить каждый экземпляр строкового литерала "Klingon" ссылкой на один и тот же адрес. Если компилятор использует представление в виде единственной строки и позволит выполнить замену p1[0] на 'F', это сразу же затронет все случаи использования этой строки, следовательно, операторы,

распечатывающие строковый литерал "Klingon", фактически отобразят на экране строку "Flingon":

```
Flingon: Beware the Flingons!
```

Многие компиляторы будут демонстрировать это достаточно противоречивое поведение, в то время как другие генерируют программы, которые в подобных случаях завершаются аварийно. В силу этого на практике рекомендуется при инициализация указателя строковым литералом использовать модификатор const:

```
const char * pl = "Klingon";          // рекомендуемое использование
```

Однако инициализация массива не-const строковым литералом не влечет за собой проблем подобного рода, поскольку такой массив получает копию исходной строки.

Массивы символьных строк

Часто бывает удобным иметь массив символьных строк. В этом случае вы получаете возможность доступа к нескольким различным строкам. В листинге 11.1 используется такой массив:

```
const char *mytal[LIM] = {"Мгновенное складывание чисел",
      "Точное умножение", "Накапливание данных",
      "Исполнение инструкций с точностью до последней буквы",
      "Знание языка программирования C"};
```

Рассмотрим это объявление более внимательно. Поскольку константа LIM равна 5, вы можете утверждать, что mytal — это массив, состоящий из пяти указателей на значения типа char. Иначе говоря, mytal является одномерным массивом, каждый элемент которого содержит адрес значения типа char. Первым указателем является mytal[0], он указывает на первый символ первой строки. Второй указатель — mytal[1], он указывает на начало второй строки. В общем случае каждый указатель ссылается на первый символ соответствующей строки

```
*mytal[0] == 'M', *mytal[1] == 'T', *mytal[2] == 'H'
```

и так далее. Массив mytal на самом деле не содержит строк, он содержит только адреса этих строк. (Строки находятся в разделе памяти, который программа использует для хранения констант.) Вы можете рассматривать mytal[0] как конструкцию, представляющую первую строку, и *mytal[0] — как первый символ первой строки. В силу отношений, существующих между записью в виде массива и указателями, вы также можете воспользоваться конструкцией mytal[0][0] для представления первого символа первой строки, даже если mytal не объявлен как двухмерный массив.

Инициализация выполняется по правилам, предназначенным для массивов. Фрагмент, заключенный в фигурные скобки, эквивалентен следующей конструкции:

```
{{...}, {...}, {...}, {...}};
```

Многоточиями заменены вещи, которые просто лень было набирать. Особо здесь следует отметить тот факт, что первый комплект двойных кавычек соответствует паре фигурных скобок и в этом порядке используется при инициализации первого символа строкового указателя. Следующий комплект фигурных скобок инициализирует второй указатель и так далее. Запятая разделяет соседние строки.

Другой подход предусматривает создание двухмерного массива:

```
char mytal_2[LIM][LINLIM];
```

В данном случае mytal_2 — это массив из пяти элементов, каждый из которых сам является массивом из 81 значений типа char. Одно из различий состоит в том, что второй вариант представляет *прямоугольный* массив, в котором строки имеют одну и ту же длину. То есть, для хранения каждой строки используется 81 элемент. Однако массив указателей образует массив *записей разной длины*, где длина каждой строки определяется строкой, которой она была инициализирована:

```
char *mytal[LIM];
```

Массив переменной длины не требует резервирования лишнего пространства памяти. На рис. 11.2 показаны два вида массивов. (На самом деле строки, на которые указывают элементы массива mytal, не обязательно должны храниться последовательно в памяти, однако этот рисунок может служить иллюстрацией различий в требованиях, предъявляемых к памяти.)

Другое различие заключается в том, что объекты данных mytal и mytal_2 имеют разные типы; mytal — это массив указателей на значения типа char, в то время как mytal_2 представляет собой массив массивов значений типа char. Короче говоря, mytal содержит пять адресов, в то время как mytal_2 хранит пять полных символьных массивов.

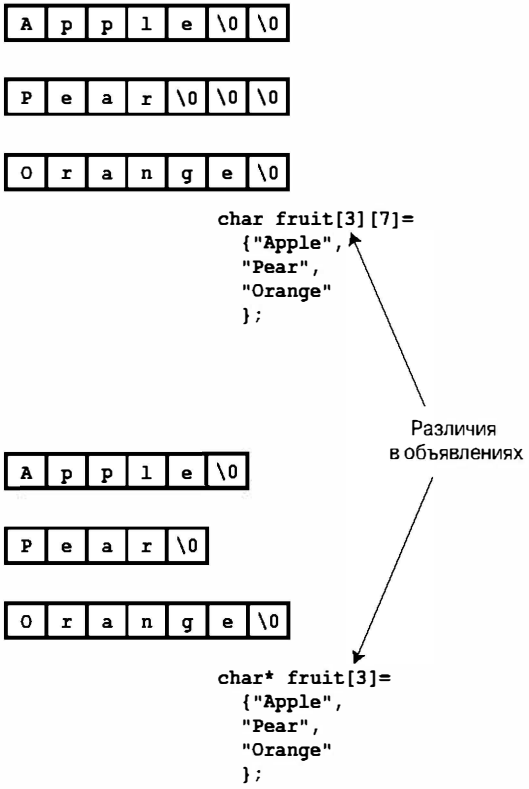


Рис. 11.2. *Прямоугольные массивы и массивы записей разной длины*

Указатели и строки

Вы, возможно, уже заметили, что во время обсуждения строк в предыдущем разделе мы время от времени ссылались на указатели. Большая часть операций над строками языка C фактически использует указатели. В качестве примера рассмотрим учебную программу, представленную в листинге 11.3.

Листинг 11.3. Программа `p_and_s.c`

```
/* p_and_s.c -- указатели и строки */
#include <stdio.h>
int main(void)
{
   const char * mesg = "Не позволяйте себя запутать!";
   const char * copy;
   copy = mesg;
   printf("%s\n", copy);
   printf("mesg = %s; &mesg = %p; value = %p\n",
           mesg, &mesg, mesg);
   printf("copy = %s; &copy = %p; value = %p\n",
           copy, &copy, copy);
   return 0;
}
```

На заметку!

Если ваш компилятор не поддерживает спецификатор `%p`, используйте вместо него спецификаторы `%u` или `%lu`.

Глядя на эту программу, вы можете подумать, что она создает копию строки `"Не позволяйте себя запутать!"`, и первое ознакомление с выходными данными скорее подтверждает это предположение:

```
Не позволяйте себя запутать!!
mesg = Не позволяйте себя запутать!; &mesg = 0x0012ff48; value = 0x0040a000
copy = Не позволяйте себя запутать!; &copy = 0x0012ff44; value = 0x0040a000
```

Теперь изучите выходные данные функции `printf()` более внимательно. Во-первых, значения `mesg` и `copy` выводятся в виде строк (`%s`). Здесь мы не сталкиваемся ни с какими неожиданностями; все строки выглядят как `"Не позволяйте себя запутать!"`.

Следующий элемент каждой строки представляет адрес заданного указателя. Два указателя `mesg` и `copy` хранятся, соответственно, в ячейках `0x0064fd58` и `0x0064fd5c`.

Теперь обратите внимание на завершающий элемент с именем `value`. Это значение конкретного указателя. Значение указателя представляет собой адрес, который он содержит. Легко заметить, что указатель `mesg` нацелен на ячейку `0x0040c000`, то же можно сказать и об указателе `copy`. Следовательно, сама строка не копировалась ни разу. Оператор `copy = mesg;` всего лишь создает второй указатель, ссылающийся на ту же самую строку.

Тогда зачем вся эта суета? Почему бы просто не скопировать всю строку? В таком случае задайте себе следующий вопрос: что эффективнее, копировать один адрес или копировать, скажем, 50 отдельных элементов? Во многих случаях вполне достаточно

указать один адрес, чтобы решить проблему. Если вам и в самом деле нужна копия строки, другими словами, ее дубликат, вы можете воспользоваться функцией `strcpy()` или `strncpy()`, которые рассматриваются далее в этой главе.

Теперь, когда мы посмотрели, как строки определяются внутри программы, рассмотрим особенности ввода строк с клавиатуры.

Ввод строк

Если возникает необходимость ввода строки в программу, вы должны сначала зарезервировать пространство памяти для хранения строки, а затем воспользоваться функцией ввода для считывания строки.

Выделение пространства памяти под строки

Прежде всего потребуется подготовить место в памяти, в котором можно будет запомнить строку после считывания. Как уже говорилось выше, это означает, что нужно зарезервировать достаточное пространство памяти в известном программе месте для последующего считывания. Не рассчитывайте на то, что компьютер подсчитает длину строки в момент ее чтения и выделит необходимое пространство памяти. Компьютер этого не сделает (пока вы не напишите специальную функцию, решающую эту задачу). Например, предположим, что имеются следующие операторы:

```
char *name;
scanf("%s", name);
```

Компилятор, по-видимому, не найдет в них ошибок, тем не менее, когда имя будет прочитано, оно, возможно, будет записано поверх данных или кода вашей программы и может привести к аварийному ее завершению. Это объясняется тем, что функция `scanf()` копирует информацию по адресу, заданному аргументом, в рассматриваемом случае таким аргументом является неинициализированный указатель; `name` может указывать на любое место в памяти. Большинство программистов находят эту ситуацию крайне забавной, правда, только в чужих программах.

Простейшее решение этой проблемы — включение явно заданного размера массива в само определение массива:

```
char name[81];
```

Теперь `name` — это адрес зарезервированного блока памяти размером 81 байта. Другая возможность предполагает использование функций распределения памяти библиотеки C, и этот подход будет рассматриваться в главе 12.

После того, как пространство памяти, необходимое для размещения строки, будет выделено, можно приступать к считыванию строки. Библиотека C предоставляет в ваше распоряжение три функции, способные считывать строки, а именно, `scanf()`, `gets()` и `fgets()`. Чаще других используется функция `gets()`, поэтому ее мы изучим в первую очередь.

Функция `gets()`

Функция `gets()` (“*get* string” — получить строку) очень удобна для применения в интерактивных программах. Она получает строку со стандартного устройства ввода ва-

шей системы, которым в большинстве случаев является клавиатура. Поскольку длина такой строки заранее не устанавливается, функции gets() нужно знать, когда остановиться. Используемый ею метод требует считывания символов до тех пор, пока не встретится символ новой строки (\n), который генерируется при каждом нажатии клавиши <Enter>. Она принимает все символы, предшествующие символу новой строки (не включая его), добавляет в конец последовательности нулевой символ (\0) и передает строку в вызывающую программу. Символ новой строки также считывается, но затем отбрасывается, благодаря чему новая операция считывания начинается в начале следующей строки. В программе, показанной в листинге 11.4, демонстрируются простые способы использования функции gets().

Листинг 11.4. Программа name1.c

```
/* name1.c -- программа считывает имя */
#include <stdio.h>
#define MAX 81
int main(void)
{
   char name[MAX];            /* выделить пространство памяти */
   printf("Как вас зовут?\n");
   gets(name);                /* поместить строку в массив name */
   printf("Прекрасное имя, %s.\n", name);
   return 0;
}
```

Вот результаты выполнения этой учебной программы:

```
Как вас зовут?
Васисуалий Лоханкин
Прекрасное имя, Васисуалий Лоханкин
```

Программа из листинга 11.4 принимает и сохраняет в памяти любое имя (включая пробелы) длиной до 80 символов. (Не забудьте зарезервировать в массиве один элемент для символа \0.) Обратите внимание, что функция gets() должна оказать воздействие на что-нибудь (name) из вызывающей функции. Это значит, что вы должны использовать в качестве аргумента адрес, а имя массива, естественно, является адресом.

Функция gets() сложнее, чем может показаться при изучении предыдущего примера. Взгляните на листинг 11.5.

Листинг 11.5. Программа name2.c

```
/* name2.c -- программа считывает имя */
#include <stdio.h>
#define MAX 81
int main(void)
{
   char name[MAX];
   char * ptr;
   printf("Как вас зовут?\n");
   ptr = gets(name);
   printf("%s? A! %s!\n", name, ptr);
   return 0;
}
```

Вот результаты выполнения этой учебной программы:

```
Как вас зовут?
Алибаба Сорокаразбойников
Алибаба Сорокаразбойников? А! Алибаба Сорокаразбойников!
```

Функция gets() получает ввод двумя способами:

- Она использует адрес для загрузки строки в массив name.
- В коде функции gets() используется ключевое слово return для возврата адреса строки, а программа присваивает этот адрес указателю ptr. Следует отметить, что ptr является указателем на char. Это значит, что функция gets() должна возвратить значение, которое представляет собой указатель на char.

В соответствии с требованиями стандарта ANSI прототип функции gets() должен содержаться в заголовочном файле stdio.h. Вам не надо самому объявлять эту функцию в тех случаях, когда вы не забываете включить в свою программу упомянутый заголовочный файл.

В то же время некоторые очень старые версии языка C требуют, чтобы вы представили собственное объявление функции gets().

Структура функции gets() имеет примерно такой вид:

```
char *gets(char * s)
{
   ...
   return(s);
}
```

Заголовок функции gets() показывает, что она возвращает указатель на значение char. Следует отметить, что функция gets() возвращает тот же указатель, что и ей передан. Существует только одна копия вводимой строки, а именно — строка, размещенная по адресу, переданному функции в качестве аргумента; в силу этого ptr в листинге 11.5 в конечном итоге указывает на массив name. Фактическая структура функции gets() несколько сложнее, поскольку в ней предусмотрены два возвращаемых значения. Если все идет хорошо, она возвращает адрес считываемой строки, как отмечалось выше. Если что-то не в порядке или если функция gets() встречает символ конца файла, она возвращает нулевой адрес. Этот нулевой адрес называется *нулевым указателем* и представлен в stdio.h объявленной константой NULL. Таким образом, функция gets() включает в себя некоторую проверку ошибок, благодаря чему удобно пользоваться конструкциями следующего вида:

```
while (gets(name) != NULL)
```

Такая конструкция дает возможность одновременно отслеживать символ конца файла и считывать значение. Если встречается символ конца файла, в массив name ничего не считывается. Такой двойной подход отличается большей компактностью, чем подход, реализованный в функции getchar(), которая имеет возвращаемое значение, но не имеет аргументов:

```
while ((ch = getchar()) != EOF)
```

Кстати, не следует путать нулевой указатель и нулевой символ. Нулевой указатель — это адрес, а нулевой символ — объект данных типа char со значением, равным нулю.

В цифровой форме оба эти объекта могут быть представлены с помощью 0, однако они принципиально отличаются друг от друга: NULL — это указатель, а \0 — константа типа char.

Функция fgets()

Одно из слабых мест функции gets() связано с тем, что она не выполняет проверку того, что входные данные умещаются в зарезервированную область памяти. В случае переполнения этой области лишние символы попадают в соседние области памяти. Функция fgets() исправляет это довольно-таки безответственное поведение, предоставляя возможность задать верхний предел количества считываемых символов. Поскольку функция fgets() разрабатывалась для файлового ввода-вывода, она менее удобна для чтения с клавиатуры, чем функция gets(). Она отличается от функции gets() в трех аспектах:

- Она принимает второй аргумент, задающий максимальное количество символов для считывания. Если этот аргумент имеет значение n, то функция fgets() выполнит считывание n-1 символов либо считывание до следующего символа новой строки, в зависимости от того, что произойдет первым.
- Если функция fgets() сталкивается с символом новой строки, она сохраняет его в строке, в отличие от функции gets(), которая просто его отбрасывает.
- Она принимает третий аргумент, показывающий, из какого файла должно выполняться считывание. Чтобы прочитать данные с клавиатуры, в качестве аргумента используется идентификатор stdin (от *standard input* — стандартный ввод); определение этого идентификатора находится в заголовочном файле stdio.h.

Листинг 11.6 представляет собой модификацию листинга 11.5, он ориентирован на использование функции fgets() вместо функции gets().

Листинг 11.6. Программа name3.c

```
/* name3.c -- программа считывает имена, пользуясь функцией fgets() */
#include <stdio.h>
#define MAX 81
int main(void)
{
   char name[MAX];
   char * ptr;
   printf("Как вас зовут?\n");
   ptr = fgets(name, MAX, stdin);
   printf("%s? A! %s!\n", name, ptr);
   return 0;
}
```

Ниже показан пример выходных данных этой программы, который демонстрирует некоторые неудобные аспекты функции fgets():

```
Как вас зовут?
Киса Воробьянинов
Киса Воробьянинов
? A! Киса Воробьянинов
!
```

Проблема заключается в том, что функция `fgets()` сохраняет символ новой строки, следовательно, этот символ отображается всякий раз, когда вы выводите строку на экран. Далее в этой главе в конце раздела "Другие строковые функции" вы узнаете, как использовать функцию `strchr()` для поиска и удаления символа новой строки.

Поскольку функция `gets()` не проверяет, поместится ли ввод в массив назначения, она не считается надежной функцией. В самом деле, несколько лет тому назад кто-то заметил, что некоторые программные коды операционной системы Unix применяют функцию `gets()`. Этот "кто-то" воспользовался упомянутым слабым местом при создании "червей", которые распространялись по сетям на базе Unix, задавая длинные последовательностей символов, затирающие код операционной системы. Код системы с тех пор был заменен кодом, в котором не употреблялась функция `gets()`. Следовательно, создавая критичные программы, вы должны применять функцию `fgets()`, но не функцию `gets()`, однако в данной книге используется более либеральный подход.

Функция scanf()

Ранее для чтения строки мы использовали функцию `scanf()` с форматом `%s`. Основное отличие между функциями `scanf()` и `gets()` заключается в том, как они определяют момент достижения конца строки: функция `scanf()` больше ориентирована на "получение слова", а не на "получение строки". Функция `gets()`, как вы могли убедиться, принимает все символы вплоть до первого символа новой строки. Функция `scanf()` может воспользоваться двумя вариантами для прекращения ввода. При любом выборе строка начинается с первого символа, не являющегося пробельным (пробел, символ табуляции или символ новой строки). Если вы указываете формат `%s`, строка продолжается до следующего пробельного символа (не включая его). Если вы зададите ширину поля, как, например, в формате `%10s`, функция `scanf()` считывает до 10 символов или все символы до первого пробельного символа, в зависимости от того, какое условие будет удовлетворено раньше (рис. 11.3).

Вспомните, что функция `scanf()` возвращает целочисленное значение, равное количеству элементов последовательно считанных символов или символ `EOF`, если она сталкивается с концом файла.

Код в листинге 11.7 служит иллюстрацией работы функции `scanf()`, когда задана ширина поля.

Оператор ввода	Исходная очередь ввода *	Содержимое строки имени	Остальная часть очереди
`scanf("%s", name);`	`Fleebert□Hup`	`Fleebert`	`□Hup`
`scanf("%5s", name);`	`Fleebert□Hup`	`Fleeb`	`Ert□Hup`
`scanf("%5s", name);`	`Ann□Ular`	`Ann`	`□Ular`

* □ представляет пробельный символ.

Рис. 11.3. *Ширина поля и функция* `scanf()`

Листинг 11.7. Программа scan_str.c

```
/* scan_str.c -- использование функции scanf() */
#include <stdio.h>
int main(void)
{
   char name1[11], name2[11];
   int count;
   printf("Введите, пожалуйста, два имени.\n");
   count = scanf("%5s %10s",name1, name2);
   printf("Прочитано %d имени: %s и %s.\n",
          count, name1, name2);
   return 0;
}
```

Приводим три варианта выполнения этой программы:

```
Введите, пожалуйста, 2 имени.
Jesse Jukes
Прочитано 2 имени: Jesse и Jukes.

Введите, пожалуйста, 2 имени.
Liza Applebottham
Прочитано 2 имени: Liza и Applebott.

Введите, пожалуйста, 2 имени.
Portensia Callowit
Прочитано 2 имени: Porte и nsia.
```

В первом примере оба имени попадают в допустимые границы размеров. Во втором примере были считаны только первые 10 символов имени `Applebottham` по причине использования формата `%10s`. В третьем примере четыре буквы имени `Portensia` попадают в `name2`, поскольку второй вызов функции `scanf()` возобновляет ввод там, где заканчивается первый ввод, в рассматриваемом случае первый ввод заканчивается внутри слова `Portensia`.

В некоторых случаях для ввода текста с клавиатуры удобнее пользоваться функцией `gets()`. Она проще в использовании, обладает приличным быстродействием, к тому же более компактна. Обычным способом применения функции `scanf()` является считывание и преобразование смеси данных различных видов, и приведение их к некоторой стандартной форме. Например, если входная строка содержит имя инструмента, складской номер и стоимость за единицу, то вы можете воспользоваться функцией `scanf()` либо написать собственную функцию, которая будет выполнять проверку на наличие ошибок. Если вы планируете обрабатывать ввод по одному слову за раз, можете прибегнуть к услугам функции `scanf()`.

Вывод строк

Теперь перейдем от темы ввода строк к теме выбора строк, снова прибегнув к помощи библиотечных функций. В языке C доступны три стандартных библиотечных функции печати строк: `puts()`, `fputs()` и `printf()`.

Функция puts()

Функция puts() очень удобна в использовании. Ей достаточно передать в качестве аргумента адрес строки. Программа в листинге 11.8 может служить иллюстрацией различных способов применения функции puts().

Листинг 11.8. Программа put_out.c

```
/* put_out.c -- использование функции puts() */
#include <stdio.h>
#define DEF "Я – строка, определенная директивой #define."
int main(void)
{
   char str1[80] = "Массив был инициализирован моим значением.";
   const char * str2 = "Указатель был инициализирован моим значением.";

   puts("Я – аргумент функции puts().");
   puts(DEF);
   puts(str1);
   puts(str2);
   puts(&str1[2]);
   puts(str2+4);

   return 0;
}
```

Выходные данные этой программы имеют следующий вид:

```
Я – аргумент функции puts().
Я – строка, определенная директивой #define.
Массив был инициализирован моим значением.
Указатель был инициализирован моим значением.
сив был инициализирован моим значением.
атель был инициализирован моим значением.
```

Обратите внимание, что каждая строка появляется в своей строке вывода. В отличие от printf(), функция puts() автоматически добавляет символ новой строки при отображении строки.

Этот пример напоминает вам, что фразы, заключенные в двойные кавычки, представляют собой строковые константы, и программа манипулирует ими как адресами. Кроме того, имена строковых массивов также трактуются программой как адреса. Выражение &str1[2] – это адрес третьего элемента массива str1. Этот элемент содержит символ 'c', именно этот символ функция puts() использует в качестве отправной точки. Аналогично, str2+4 указывает на ячейку памяти, содержащую символ 'а' строки "Указатель", и вывод начинается с него.

Знает ли функция puts(), когда остановиться? Она прекращает ввод, когда сталкивается с нулевым символом, поэтому желательно, чтобы в строке такой символ присутствовал. Не пытайтесь кодировать так, как показано в программе из листинга 11.9!

Листинг 11.9. Программа nono.c

```
/* Программа nono.c -- не делайте так! */
#include <stdio.h>
```

```
int main(void)
{
   char side_a[] = "Сторона А";
   char dont[] = {'В', 'А', 'У', '!' };
   char side_b[] = "Сторона Б";
   puts(dont);          /* dont не является строкой */
   return 0;
}
```

Поскольку в массиве dont отсутствует завершающий символ, эта последовательность не является строкой, поэтому функция puts() не знает, в каком месте остановиться. Она будет выводить на печать содержимое памяти, примыкающей к массиву dont, пока не найдет нулевой символ где-нибудь в другом месте. Чтобы нулевой символ не оказался слишком далеко, массив dont помещен между двумя настоящими строками. Вот как выглядит вывод этой программы:

```
ВАУ!Сторона Б
```

Конкретный компилятор, который был использован в рассматриваемом случае, размещает в памяти массив side_a сразу после массива dont, следовательно, функция puts() продолжает выполняться до тех пор, пока не столкнется с нулевым символом в массиве side_a. Вы можете получить другие результаты в зависимости от того, как ваш компилятор разместит данные в памяти. Что произойдет, если программа пропустит массивы side_a и side_b? В любой момент в памяти содержится множество нулевых символов, и если вам повезет, функция puts() достаточно быстро найдет один из них, однако не следует на это особо рассчитывать.

Функция fputs()

Функция fputs() представляет собой версию функции gets(), ориентированную на работу с файлами. Между ними существуют следующие основные различия:

- Функция fputs() принимает второй аргумент, указывающий на файл, в который должна выполняться запись. Для вывода на устройство отображения вы можете воспользоваться аргументом stdout (от *standard output* — стандартный вывод), который определяется в заголовочном файле stdio.h.
- В отличие от puts(), функция fputs() не добавляет символ новой строки в выходные данные.

Обратите внимание на то, что функция gets() отбрасывает символ новой строки из входных данных, однако функция puts() добавляет этот символ в выходные данные. С другой стороны, функция fgets() сохраняет символ новой строки во входных данных, а функция fputs() не добавляет упомянутый символ в выходные данные. Предположим, что вы хотите построить цикл, который считывает строку и воспроизводит ее в следующей строке выходных данных. Вы можете поступить следующим образом:

```
char line[81];
while (gets(line))
    puts(line);
```

Вспомните, что функция gets() возвращает нулевой указатель, как только встречает конец файла. Нулевой указатель интерпретируется как ноль, или ложное значение, следовательно, выполнение цикла прекращается. Либо можно выполнить следующие операторы:

```
char line[81];
while (fgets(line, 81, stdin))
    fputs(line, stdout);
```

В первом цикле строка массива line отображается в собственной строке вывода, поскольку функция puts() добавляет символ новой строки. Во втором цикле строка массива line отображается в собственной строке вывода по той причине, что функция fgets() запоминает символ новой строки. Следует отметить, что если вы смешаете ввод функции fgets() с выводом функции puts(), то получите два символа новой строки, отображенных в каждой строке. Главное заключается в том, что функция puts() разрабатывалась для работы в паре с gets(), а функция fputs() — для совместной работы с fgets().

Функция printf()

Мы достаточно подробно рассматривали функцию printf() в главе 4. Подобно функции puts(), она принимает адрес строки в качестве аргумента. Функция printf() менее удобна в употреблении, чем функция puts(), но в то же время она более универсальна, поскольку способна выполнять форматирование различных типов данных.

Одно из различий состоит в том, что функция printf() не печатает автоматически каждую строку в новой строке. Вместо этого необходимо указать, куда должны быть помещены символы новой строки. Поэтому

```
printf("%s\n", string);
```

приводит к тому же результату, что и

```
puts(string);
```

Легко заметить, что первая форма требует вывода на печать большего объема информации. Она также требует больших затрат машинного времени (но не настолько, чтобы вы смогли это заметить). С другой стороны, функция printf() упрощает размещение нескольких строк в одной строке вывода. Например, следующий оператор объединяет в одну строку вывода слово Well, имя пользователя и символьную строку, определенную с помощью #define:

```
printf("Well, %s, %s\n", name, MSG);
```

Возможность создания собственных функций

Ваши возможности в смысле ввода и вывода ограничены стандартными библиотечными функциями. Если у вас нет таких функций, вы можете подготовить собственные варианты на базе функций getchar() и putchar(). Предположим, вам нужна функция, подобная puts(), но которая автоматически не добавляет символ новой строки.

В листинге 11.10 продемонстрирован один из способов создания такой функции.

Листинг 11.10. Программа `put1.c`

```
/* put1.c -- печатает строку без добавления символа \n */
#include <stdio.h>
void put1(const char * string)          /* строка не меняется */
{
   while (*string != '\0')
       putchar(*string++);
}
```

Указатель `string` на `char` первоначально ссылается на первый элемент переданного аргумента. Поскольку эта функция не изменяет строку, воспользуйтесь модификатором `const`. После того, как содержимое этого элемента выведено на печать, значение указателя увеличивается и указывает на следующий элемент. Это продолжается до тех пор, пока указатель не укажет на элемент, содержащий нулевой символ. Вспомните, что операция ++ имеет больший приоритет, чем *, а это означает, что вызов `putchar(*string++)` выводит на печать значение, на которое указывает `string`, и увеличивает значение самого указателя `string`, но не символа, на который он ссылается.

Вы можете рассматривать файл `put1.c` как модель для написания функций, выполняющих обработку строк. Поскольку каждая строка содержит нулевой символ, обозначающий конец строки, вы не должны передавать размер строки в качестве параметра функции. Вместо этого функция последовательно выполняет обработку символов, пока не столкнется с нулевым символом. Несколько более длинный способ написания функции получается при использовании записи в форме массива:

```
int i = 0;
while (string[i]!= '\0')
   putchar(string[i++]);
```

Это требует дополнительной переменной для индекса.

Многие программисты, работающие на C, предпочитают использовать следующее условие для проверки конца цикла:

```
while (*string)
```

Когда указатель `string` ссылается на нулевой символ, выражение `*string` принимает значение `0`, что приводит к завершению цикла. Такой подход, несомненно, требует ввода меньших объемов данных с клавиатуры по сравнению с предыдущей версией. Для тех, кто не обладает достаточным опытом практической работы с языком C, это не столь очевидно. Данный подход получил весьма широкое распространение, и опытные программисты, работающие на C, как мы полагаем, должны быть с ним знакомы.

На заметку!

Почему в программе, показанной в листинге 11.10, в качестве формального аргумента используется конструкция `const char * string`, а не `const char string[]`? В техническом плане они эквивалентны, так что обе формы будут работать. Одна из причин, по которой применяется форма записи с квадратными скобками, состоит в том, чтобы напомнить пользователю, что данная функция выполняет обработку массива. Однако при работе со строками фактическим аргументом может быть имя массива, строка, заключенная в кавычки, либо переменная, которая была объявлена с типом `char *`. Использование конструкции `const char * string` напоминает о том, что фактическим аргументом может быть не только массив.

Предположим, что вам нужна функция puts(), которая показывает также, сколько символов было выведено на печать. Как следует из листинга 11.11, добавление этого свойства не сопряжено с большими трудностями.

Листинг 11.11. Программа put2.c

```
/* put2.c -- печатает строку и подсчитывает выведенные символы */
#include <stdio.h>
int put2(const char * string)
{
   int count = 0;
   while (*string)              /* общая идиома */
   {
      putchar(*string++);
      count++;
   }
   putchar('\n');               /* символ новой строки не подсчитывается */
   return(count);
}
```

Следующий вызов функции печатает строку pizza:

```
put1("pizza");
```

Приведенный ниже вызов возвращает также количество символов, присвоенных переменной num (в данном случае, 5):

```
num = put2("pizza");
```

В листинге 11.12 представлен драйвер, использующий функции put1() и put2(), а также вложенные вызовы функций.

Листинг 11.12. Программа put_put.c

```
//put_put.c -- функции обработки ввода, объявленные пользователем
#include <stdio.h>
void put1(const char *);
int put2(const char *);
int main(void)
{
   put1("Если бы у меня было столько денег,");
   put1(" сколько можно было бы потратить,\n");
   printf("Получилось %d символов.\n",
          put2("я никогда не чинила бы старые вещи."));
   return 0;
}
void put1(const char * string)
{
   while (*string)              /* то же что и *string != '\0' */
       putchar(*string++);
}
```

```
int put2(const char * string)
{
   int count = 0;
   while (*string)
   {
      putchar(*string++);
      count++;
   }
   putchar('\n');
   return(count);
}
```

Таким образом, мы использует функцию `printf()` для вывода на печать значений функции `put2()`, однако в процессе поиска значения `put2()` компьютер сначала должен выполнить эту функцию, в результате чего появляется строка, которая затем распечатывается.

Ниже показаны выходные данные этой программы:

```
Если бы у меня было столько денег, сколько можно было бы потратить
я никогда не чинила бы старые вещи.
Получилось 35 символов.
```

Строковые функции

Библиотека C предлагает программистам несколько функций обработки строк; в ANSI C прототипы этих функций содержатся в заголовочном файле `string.h`. Мы рассмотрим наиболее полезные и распространенные из этих функций, а именно — `strlen()`, `strcat()`, `strncat()`, `strcmp()`, `strncmp()`, `strcpy()` и `strncpy()`. Мы также опробуем функцию `sprintf()`, которая поддерживается заголовочным файлом `stdio.h`. Для ознакомления со всем списком семейства функций `string.h` обратитесь в раздел V справочника (приложение Б).

Функция `strlen()`

Функция `strlen()`, как вам уже известно, вычисляет длину строки. Она используется в следующем примере, в котором рассматривается функция, укорачивающая строку:

```
/* fit.c -- функция "прокрустова ложа" */
void fit(char * string, unsigned int size)
{
   if (strlen(string) > size)
       *(string + size) = '\0';
}
```

Эта функция изменяет строку, следовательно, заголовочная функция не использует модификатор `const` при объявлении формального параметра `string`.

Проверьте, как работает функция `fit()` в учебной программе, показанной в листинге 11.13. Обратите внимание, что этот код использует возможность конкатенации строковых литералов.

Листинг 11.13. Программа `test_fit.c`

```
/* test_fit.c -- использование функции укорачивания строки */
#include <stdio.h>
#include <string.h>          /* содержит прототипы строковых функций */
void fit(char *, unsigned int);
int main(void)
{
   char mesg[] = "Вещи должны быть максимально простыми,"
                 " но никак не проще.";
   puts(mesg);
   fit(mesg,37);
   puts(mesg);
   puts("Рассмотрим еще несколько строк.");
   puts(mesg + 38);
   return 0;
}
void fit(char *string, unsigned int size)
{
   if (strlen(string) > size)
       *(string + size) = '\0';
}
```

Выходные данные этой программы имеют вид:

```
Вещи должны быть максимально простыми, но никак не проще.
Вещи должны быть максимально простыми
Рассмотрим еще несколько строк.
 но никак не проще.
```

Функция `fit()` помещает символ `'\0'` в 38-й элемент массива вместо запятой. Выполнение функции `puts()` прекращается при появлении нулевого символа, при этом остальная часть массива игнорируется. В то же время остальная часть массива все еще находится в пределах досягаемости, как показывает следующий вызов функции:

```
puts(mesg + 38);
```

Выражение `mesg + 38` — это адрес элемента массива `mesq[38]`, таковым является символ пробела. Таким образом, функция `puts()` отображает этот символ и продолжает работу до тех пор, пока не столкнется с исходным нулевым символом. Рисунок 11.4 служит иллюстрацией того, что происходит в этой программе (в условиях укороченной строки). (Варианты этого изречения приписываются Альберту Эйнштейну, но это более чем цитата, это, скорее, формулирование целой философии.)

Заголовочный файл ANSI `string.h` содержит прототипы семейства строковых функций языка C, именно по этой причине этот файл включен в данную учебную программу.

На заметку!

Некоторые системы, созданные до появления стандарта ANSI, используют заголовочный файл `strings.h` вместо указанного выше файла, в других системах строковый заголовочный файл может вообще отсутствовать.

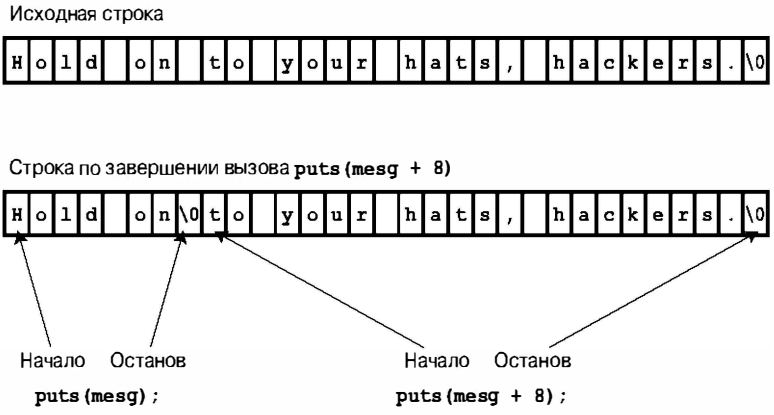


Рис. 11.4. *Функция puts() и нулевой символ*

Функция strcat()

В качестве аргументов функция strcat() (от *string concatenation* — конкатенация строк) принимает две строки. Копия второй строки присоединяется в конец первой строки, полученная в результате комбинация строк становится новой версией первой строки. Вторая строка не меняется. Функция strcat() имеет тип char * (иначе говоря, указатель на значение типа char). Она возвращает значение своего первого аргумента, то есть адрес первого символа строки, в конец которой была добавлена вторая строка. Листинг 11.14 служит иллюстрацией того, что может делать функция strcat().

Листинг 11.14. Программа str_cat.c

```
/* str_cat.c -- объединяет две строки */
#include <stdio.h>
#include <string.h>              /* объявление функции strcat()*/
#define SIZE 80
int main(void)
{
   char flower[SIZE];
   char addon[] = " пахнет как старые валенки.";
   puts("Какой ваш любимый цветок?");
   gets(flower);
   strcat(flower, addon);
   puts(flower);
   puts(addon);
   return 0;
}
```

Ниже показан результат выполнения этой программы:

```
Какой ваш любимый цветок?
Роза
Роза пахнет как старые валенки.
 пахнет как старые валенки.
```

Функция strncat()

Функция strcat() не выполняет проверку для определения, уместится ли вторая строка в первом массиве. Если вам не удастся зарезервировать пространство памяти, достаточное для размещения первого массива, вы столкнетесь с проблемами, такими как переполнение соседних ячеек памяти избыточными символами первой строки. Разумеется, вы предварительно можете воспользоваться функцией strlen(), как показано в листинге 11.15. Следует отметить, что эта функция добавляет 1 к общей длине, чтобы зарезервировать место для нулевого символа. С другой стороны, вы можете воспользоваться функцией strncat(), которая принимает второй аргумент, представляющий собой максимальное количество символов, которое нужно добавить. Например, вызов strncat(bugs, addon, 13) будет добавлять содержимое строки addon к строке bugs, и этот процесс прекратится после того, как будут добавлены 13 дополнительных символов или при добавлении нулевого символа, в зависимости от того, какое из этих событий произойдет раньше. Следовательно, с учетом нулевого символа (который добавляется в любом случае), массив bugs должен иметь размеры, достаточные для того, чтобы сохранить исходную строку (без нулевого символа), максимум 13 дополнительных символов, а также завершающий нулевой символ. В листинге 11.15 эта информация используется для вычисления значения переменной available, которая служит в качестве максимально допустимого числа дополнительных символов.

Листинг 11.15. Программа join_chk.c

```
/* join_chk.c -- объединяет две строки, сначала проверяя размер */
#include <stdio.h>
#include <string.h>
#define SIZE 30
#define BUGSIZE 11
int main(void)
{
   char flower[SIZE];
   char addon[] = " пахнет как старые валенки. ";
   char bug[BUGSIZE];
   int available;
   puts("Какой ваш любимый цветок?");
   gets(flower);
   if ((strlen(addon) + strlen(flower) + 1) <= SIZE)
       strcat(flower, addon);
   puts(flower);
   puts("Какое ваше любимое насекомое?");
   gets(bug);
   available = BUGSIZE - strlen(bug) - 1;
   strncat(bug, addon, available);
   puts(bug);
   return 0;
}
```

Приводим примеры выходных данных этой программы:

```
Какой ваш любимый цветок?
Роза
Роза пахнет как старые валенки?
Тля
Тля пахнет
```

Функция strcmp()

Предположим, что вы хотите сравнить чей-то ответ со строкой, хранящейся в памяти, как показано в листинге 11.16.

Листинг 11.16. Программа nogo.c

```
/* nogo.c -- будет ли это работать? */
#include <stdio.h>
#define ANSWER "Грант"
int main(void)
{
   char try[40];

   puts("Кто похоронен в могиле Гранта?");
   gets(try);
   while (try != ANSWER)
   {
      puts("Нет, неправильно. Попытайтесь еще раз.");
      gets(try);
   }
   puts("Теперь правильно!");

   return 0;
}
```

Как бы ни красиво выглядела эта программа, она не будет работать правильно. ANSWER и try — на самом деле указатели, следовательно, сравнение try != ANSWER не работает в плане проверки тождественности двух сравниваемых строк. Скорее, оно используется для проверки, имеют ли эти две строки один и тот же адрес. Поскольку ANSWER и try хранятся в различных ячейках памяти, эти два адреса никогда не бывают тождественными, а пользователь каждый раз получает сообщение об ошибке. Подобного рода программы часто сбивают пользователей с толку.

По сути дела вам требуется функция, которая сравнивает *содержимое* строк, но не их *адреса*. Конечно, вы можете сами разработать такую функцию, тем не менее, она уже существует и носит имя strcmp() (от *string comparison* — сравнение строк). Эта функция выполняет ту же обработку строк, какую выполняют операции отношения над числами. В частности, она возвращает 0, если оба строковых аргумента идентичны. Программа с соответствующими изменениями показана в листинге 11.17.

Листинг 11.17. Программа compare.c

```
/* compare.c -- эта программа будет работать */
#include <stdio.h>
#include <string.h> /* объявляет функцию strcmp() */
#define ANSWER "Грант"
#define MAX 40
int main(void)
{
   char try[MAX];

   puts("Кто похоронен в могиле Гранта?");
   gets(try);
```

```
   while (strcmp(try,ANSWER) != 0)
   {
      puts("Нет, неправильно. Попытайтесь еще раз.");
      gets(try);
   }
   puts("Теперь правильно!");
   return 0;
}
```

На заметку!

Поскольку любое ненулевое значение трактуется как истинное, большинство квалифицированных программистов, работающих на C, отдадут свое предпочтение сокращенной форме записи оператора цикла `while` в виде:

```
while(strcmp(try,ANSWER))
```

Одно из примечательных свойств функции `strcmp()` заключается в том, что она сравнивает строки, а не массивы. И хотя массив `try` занимает 40 ячеек памяти, а слово "Грант" только шесть (одна ячейка отводится под нулевой символ), при выполнении операции сравнения просматривается только часть массива `try`, расположенная до первого нулевого символа. В результате `strcmp()` может использоваться для сравнения строк, хранящихся в массивах различных размеров.

Что произойдет, если ответом пользователя будет "ГРАНТ" или "грант" или "Улисс С. Грант"? Программа сообщит пользователю о том, что он ошибся. Чтобы сделать программу более дружественной, вы должны быть готовы к приему всех возможных правильных ответов. Существует несколько приемов, которыми вы можете воспользоваться. Например, вы можете воспользоваться директивой `#define` для определения такого ответа, как "ГРАНТ", и написать функцию, которая преобразует все ответы в символы верхнего регистра. Такой подход устраняет проблему употребления заглавных букв, однако следует также подумать и о других формах ответа. Мы оставляем эту задачу в качестве упражнения для самостоятельного выполнения.

Возвращаемое значение функции `strcmp()`

Какое значение возвращает функция `strcmp()` в тех случаях, когда строки не одинаковы? Листинг 11.18 может служить иллюстрацией этого случая.

Листинг 11.18. Программа `compback.c`

```
/* compback.c — значения, возвращаемые функцией strcmp() */
#include <stdio.h>
#include <string.h>
int main(void)
{
   printf("strcmp(\"A\", \"A\") равно ");
   printf("%d\n", strcmp("A", "A"));
   printf("strcmp(\"A\", \"B\") равно ");
   printf("%d\n", strcmp("A", "B"));
   printf("strcmp(\"B\", \"A\") равно ");
   printf("%d\n", strcmp("B", "A"));
```

```
    printf("strcmp(\"C\", \"A\") равно ");
    printf("%d\n", strcmp("C", "A"));
    printf("strcmp(\"Z\", \"a\") равно ");
    printf("%d\n", strcmp("Z", "a"));
    printf("strcmp(\"apples\", \"apple\") равно ");
    printf("%d\n", strcmp("apples", "apple"));
    return 0;
}
```

В одной из систем выходные данные могут иметь следующий вид:

```
strcmp("A", "A") равно 0
strcmp("A", "B") равно -1
strcmp("B", "A") равно 1
strcmp("C", "A") равно 1
strcmp("Z", "a") равно -1
strcmp("apples", "apple") равно 1
```

Сравнение символа "A" с самим собой возвращает 0. Сравнение "A" и "B" возвращает -1, а обращение сравнения возвращает 1. Эти результаты заставляют предположить, что функция strcmp() возвращает отрицательное значение в тех случаях, когда первая строка предшествует второй в алфавитном порядке, и что она возвращает положительное значение, если порядок следования строк противоположный. В силу этого, сравнение "C" с "A" дает в результате 1. Другие системы могут возвратить 2, что объясняется различием значений кодов ASCII. Стандарт ANSI требует, чтобы функция strcmp() возвращала отрицательное число в тех случаях, когда первая строка предшествует второй в алфавитном порядке, 0, если строки совпадают, и положительное значение, если первая строка следует за второй в алфавитном порядке. В то же время, точные числовые значения выбираются на этапе реализации. Например, ниже показан вывод для другой реализации, а именно, для той, которая возвращает разность значений кодов символов:

```
strcmp("A", "A") равно 0
strcmp("A", "B") равно -1
strcmp("B", "A") равно 1
strcmp("C", "A") равно 2
strcmp("Z", "a") равно -7
strcmp("apples", "apple") равно 115
```

А что произойдет, если начальные символы строк одинаковы? В общем случае, функция strcmp() продвигается по строками до тех пор, пока не найдет первую пару несовпадающих символов. После этого она возвращает соответствующий код. Например, в самом последнем примере, строки "apples" и "apple" совпадают вплоть до последнего символа первой строки. Соответствие соблюдается до шестого символа строки "apple", каковым является нулевой символ, то есть 0 в кодировке ASCII. Поскольку нулевой символ идет первым в последовательности ASCII, а символ s — после него, рассматриваемая функция возвращает положительное значение.

Последнее сравнение показывает, что функция strcmp() сравнивает сразу все символы, а не только отдельные буквы, следовательно, вместо того, чтобы утверждать, что сравнение проводится в алфавитном порядке, мы можем сказать, что функция

strcmp() выполняет просмотр в *сортирующей последовательности* (или, другими словами, в соответствие со *схемой упорядочения*). Это означает, что символы сравниваются по их числовым представлениям, которыми обычно являются ASCII-коды. В кодировке ASCII буквы верхнего регистра предшествуют буквам нижнего регистра. По этой причине вызов strcmp("Z", "a") возвращает отрицательное значение.

Довольно часто вас не интересует точное возвращаемое значение. Вам вполне достаточно знать, принимает ли оно нулевое или ненулевое значение, то есть совпадают строки или нет. Когда же вы хотите расположить строки в алфавитном порядке, тогда вам нужно знать, каким является результат сравнения — положительный, отрицательный или нулевой.

На заметку!

Функция strcmp() предназначена для сравнения *строк*, но не *символов*. Следовательно, вы можете использовать такие аргументы как "apples" и "A", но не аргументы наподобие 'A'. Однако напомним, что тип char относится к семейству целочисленных типов, следовательно, для сравнения символов вы можете использовать операции отношения. Предположим, что word — это строка, хранящаяся в массиве значений типа char, и что ch — переменная типа char. В таком случае допустимыми являются следующие операторы:

```
if (strcmp(word, "quit") == 0)     // функция strcmp() используется
                                   // применительно к строкам
   puts("Всего хорошего!");
if (ch == 'q')                     // операция == используется
                                   // применительно к строкам
   puts("Всего хорошего!");
```

В то же время в качестве аргументов функции strcmp() нельзя использовать ch или 'q'.

В программе, представленной в листинге 11.19, функция strcmp() применяется для определения момента, когда она должна прекратить чтение ввода.

Листинг 11.19. Программа quit_chk.c

```
/* quit_chk.c -- начало некоторой программы */
#include <stdio.h>
#include <string.h>
#define SIZE 81
#define LIM 100
#define STOP "quit"
int main(void)
{
   char input[LIM][SIZE];
   int ct = 0;
   printf("Введите не более %d строк (или quit для завершения):\n", LIM);
   while (ct < LIM && gets(input[ct]) != NULL &&
          strcmp(input[ct],STOP) != 0)
   {
      ct++;
   }
   printf("Введено %d строк\n", ct);
   return 0;
}
```

Эта программа прекращает чтение входных данных, как только встречается символ EOF (функция gets() возвращает в этом случае нулевое значение), при вводе слова *quit* (выход из программы) или когда будет достигнуто предельное значение LIM.

Между прочим, иногда удобнее прекратить ввод путем ввода пустой строки, иначе говоря, нажатием клавиши <Enter> или <Return>, не набирая при этом никаких других символов. Чтобы сделать это, вы можете модифицировать оператор цикла while следующим образом:

```
while (ct < LIM && gets(input[ct]) != NULL
                && input[ct][0] != '\0')
```

В данном случае input[ct] — это только что введенная строка, а input[ct][0] — первый символ этой строки. Если пользователь вводит пустую строку, функция gets() помещает нулевой символ в первый элемент, таким образом, выражение

```
input[ct][0] != '\0'
```

выполняет проверку на наличие пустой строки ввода.

Варианты функции strncmp()

Функция strcmp() сравнивает строки до тех пор, пока не найдет пару соответствующих символов, которые отличаются друг от друга, и этот поиск может продолжаться до тех пор, пока не будет достигнут конец одной из строк. Функция strncmp() сравнивает строки до тех пор, пока не обнаружит в них различия, либо пока не сравнит количество символов обеих строк, заданное третьим аргументом. Например, если вы хотите обнаружить строку, начинающуюся фрагментом "astro", вы можете ограничить этот поиск первыми пятью символами. В листинге 11.20 показано, как это делается.

Листинг 11.20. Программа starsrch.c

```
/* starsrch.c -- использование функции strncmp() */
#include <stdio.h>
#include <string.h>
#define LISTSIZE 5
int main()
{
   const char * list[LISTSIZE] =
   {
      "astronomy", "astounding",
      "astrophysics", "ostracize",
      "asterism"
   };
   int count = 0;
   int i;

   for (i = 0; i < LISTSIZE; i++)
       if (strncmp(list[i],"astro", 5) == 0)
       {
           printf("Найдено: %s\n", list[i]);
           count++;
       }
```

```
   printf("Список содержит %d слов(о,а), начинающихся"
          " с astro.\n", count);
   return 0;
}
```

Получаем следующие выходные данные:

```
Найдено: astronomy
Найдено: astrophysics
Список содержит 2 слов(о,а), начинающихся с astro.
```

Функции strcpy() и strncpy()

Выше мы говорили, что если pts1 и pts2 — указатели на строки, то выражение

```
pts2 = pts1;
```

копирует только адрес строки, но не саму строку. Предположим, однако, что нужно скопировать строку. В таком случае вы можете воспользоваться функцией strcpy(). Программа, представленная в листинге 11.21, предлагает пользователю ввести слова, начинающиеся с буквы *q*. Эта программа копирует ввод во временный массив, и если первая буква есть *q*, программа использует функцию strcpy() для копирования этой строки из временного файла в место ее постоянного хранения. Функция strcpy() представляет собой строковый эквивалент оператора присваивания.

Листинг 11.21. Программа copy1.c

```
/* copy1.c -- демонстрационная программа использования функции strcpy() */
#include <stdio.h>
#include <string.h>                    /* объявление функции strcpy() */
#define SIZE 40
#define LIM 5
int main(void)
{
   char qwords[LIM][SIZE];
   char temp[SIZE];
   int i = 0;
   printf("Введите %d слов, начинающихся с буквы q:\n", LIM);
   while (i < LIM && gets(temp))
   {
      if (temp[0] != 'q')
         printf("%s не начинается с буквы q!\n", temp);
      else
      {
         strcpy(qwords[i], temp);
         i++;
      }
   }
   puts("Перечень слов, удовлетворяющих заданному критерию:");
   for (i = 0; i < LIM; i++)
       puts(qwords[i]);
   return 0;
}
```

Ниже показаны результаты выполнения этой учебной программы:

```
Введите 5 слов, начинающиеся с буквы q:
quackery
quasar
quilt
quotient
no more
no more не начинается с буквы q!
quiz
Перечень слов, удовлетворяющих заданному критерию:
quackery
quasar
quilt
quotient
quiz
```

Обратите внимание, что значение счетчика `i` увеличивается только в том случае, когда вводимое слово проходит проверку на наличие буквы *q*. Отметим также, что в рассматриваемой программе выполняется следующая проверка на наличие символа:

```
if (temp[0] != 'q')
```

Иначе говоря, является ли первый символ массива `temp` буквой *q*? Еще одна возможность состоит в использовании построчной проверки:

```
if (strncmp(temp, "q", 1) != 0)
```

Другими словами, различаются ли строки `temp` и `"q"` уже в первом элементе?

Следует отметить, что строка, на которую ссылается второй аргумент (`temp`), копируется в массив, на который указывает первый аргумент (`qword[i]`). Копия называется *целью*, а исходная строка — *источником*. Вы легко можете запомнить порядок аргументов, заметив, что он совпадает с порядком в операторе присваивания (целевая строка находится слева):

```
char target[20];
int x;
x = 50;                          /* присвоение чисел */
strcpy(target, "Hi ho!");        /* присвоение строк */
target = "So long";              /* синтаксическая ошибка */
```

Ответственность за то, чтобы массив назначения содержал достаточно пространства для размещения копии источника, лежит на программисте. Показанный ниже программный код может привести к неприятностям:

```
char * str;
strcpy(str, "The C of Tranquility");              /* проблема */
```

Эта функция копирует строку `"The C of Tranquility"` по адресу, указанному переменной `str`, но переменная `str` не инициализирована, поэтому копия может оказаться в любом месте памяти!

Короче говоря, функция `strcpy()` принимает два строковых указателя в качестве аргументов. Второй указатель, который ссылается на исходную строку, может быть объявленным указателем, именем массива или строковой константой. Первый указатель, который ссылается на копию, должен указывать на объект данных, такой как,

например, массив, располагающий пространством памяти, достаточным для размещения этой строки. Напоминаем, что при объявлении массива выделяется память для хранения данных, при объявлении указателя выделяется пространство памяти, достаточное для размещения одного адреса.

Остальные свойства функции strcpy()

Функция strcpy() обладает еще двумя свойствами, которые могут оказаться полезными. Во-первых, она имеет тип char *. Она возвращает значение своего первого аргумента, а именно, адрес символа. Во-вторых, первый аргумент не обязательно должен указывать на начало массива; это дает возможность копировать только нужную часть массива. Листинг 11.22 служит иллюстрацией обоих свойств.

Листинг 11.22. Программа copy2.c

```
/* copy2.c — демонстрационная программа использования функции strcpy() */
#include <stdio.h>
#include <string.h>                 /* объявление функции strcpy()*/
#define WORDS "beast"
#define SIZE 40

int main(void)
{
   const char * orig = WORDS;
   char copy[SIZE] = "Be the best that you can be.";
   char * ps;

   puts(orig);
   puts(copy);
   ps = strcpy(copy + 7, orig);
   puts(copy);
   puts(ps);

   return 0;
}
```

Приводим результаты выполнения этой учебной программы:

```
beast
Be the best that you can be.
Be the beast
beast
```

Функция strcpy() копирует нулевой символ из исходной строки. В этом примере нулевой символ затирает букву *t*, идущую первой в слове that в массиве copy, так что новая строка заканчивается словом beast (рис. 11.5). Кроме того, ps указывает на восьмой элемент (индекс 7) массива copy, поскольку первым аргументом является copy + 7. В силу этого вызов puts(ps) печатает строку, которая начинается в этой точке.

Тщательный выбор: функция strncpy()

Функцию strcpy() объединяет с функцией gets() общая проблема — ни та, ни другая не проверяют, умещается ли на самом деле исходная строка в целевой строке.

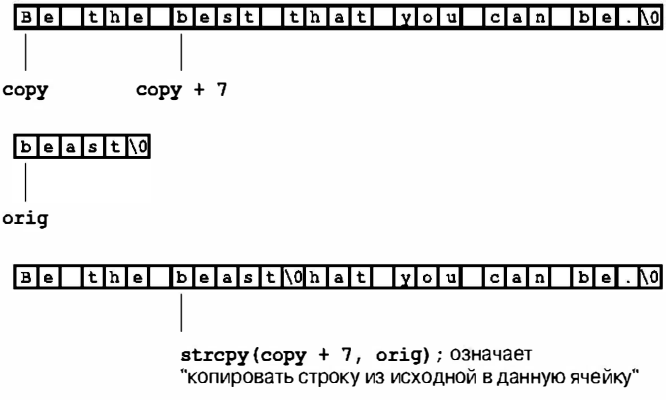


Рис. 11.5. *Функция* `strcpy()` *использует указатели*

Более безопасный способ копирования строк предполагает использование функции `strncpy()`. Эта функция принимает третий аргумент, представляющий собой максимальное количество символов, предназначенных для копирования. Листинг 11.23 представляет собой версию листинга 11.21, в которой вместо функции `strcpy()` используется `strncpy()`. Чтобы показать, что происходит в тех случаях, когда размер исходной строки превосходит размер целевой строки, в нем выбраны небольшие размеры (семь элементов, шесть символов) для целевых строк.

Листинг 11.23. Программа `copy3.c`

```
/* copy3.c -- демонстрационная программа использования функции strncpy() */
#include <stdio.h>
#include <string.h>                    /* объявление функции strncpy() */
#define SIZE 40
#define TARGSIZE 7
#define LIM 5
int main(void)
{
   char qwords[LIM][TARGSIZE];
   char temp[SIZE];
   int i = 0;

   printf("Введите %d слов, начинающихся с буквы q:\n", LIM);
   while (i < LIM && gets(temp))
   {
      if (temp[0] != 'q')
          printf("%s не начинается с буквы q!\n", temp);
      else
      {
          strncpy(qwords[i], temp, TARGSIZE - 1);
          qwords[i][TARGSIZE - 1] = '\0';
          i++;
      }
   }

   puts("Перечень слов, удовлетворяющих заданному критерию:");
```

```
    for (i = 0; i < LIM; i++)
        puts(qwords[i]);
    return 0;
}
```

Ниже показаны результаты выполнения этой учебной программы:

```
Введите 5 слов, начинающихся с буквы q:
quack
quadratic
quisling
quota
quagga
Перечень слов, удовлетворяющих заданному критерию:
quack
quadra
quisli
quota
quagga
```

Вызов функции `strncpy(target, source, n)` копирует максимум `n` символов либо все символы, предшествующие нулевому (в зависимости от того, какое из этих событий произойдет первым), из строки `source` в строку `target`. Следовательно, если количество символов в строке `source` меньше `n`, копируется вся строка, в том числе и нулевой символ. Эта функция никогда не копирует более `n` символов, следовательно, если она достигает предела до того, как достигнет конца исходной строки, нулевой символ не добавляется. В результате в окончательном виде скопированная строка может содержать, но может и не содержать нулевой символ. По этой причине рассматриваемая программа устанавливает значение `n` на единицу меньше, чем размер целевого массива, а затем устанавливает финальный элемент массива равным нулевому символу:

```
strncpy(qwords[i], temp, TARGSIZE - 1);
qwords[i][TARGSIZE - 1] = '\0';
```

Тем самым гарантируется сохранение строки. Если исходная строка на самом деле умещается в целевой строке, нулевой символ, скопированный вместе с нею, отмечает фактический конец строки. Если исходная строка не умещается в целевой, конец строки будет обозначать нулевой символ в финальном элементе массива.

Функция `sprintf()`

Функция `sprintf()` объявлена в заголовочном файле `stdio.h`, а не в `string.h`. Она ведет себя подобно функции `printf()`, но при этом выполняет запись в строку, а не выводит ее на устройство отображения. По этой причине она обеспечивает способ комбинирования нескольких элементов в единую строку. В качестве первого аргумента функция `sprintf()` принимает адрес целевой строки. Остальные аргументы аналогичны аргументам функции `printf()` — строка спецификации преобразования и список элементов, которые эта функция должна записать.

В программе в листинге 11.24 с помощью функции `sprintf()` в одну строку объединяются три элемента (две строки и число). Обратите внимание, что эта программа

использует функцию sprintf() так же, как вы использовали бы функцию printf(), за исключением того обстоятельства, что полученная в результате объединения строка сохраняется в массиве formal вместо отображения ее на экране.

Листинг 11.24. Программа format.c

```
/* format.c -- форматирование строки */
#include <stdio.h>
#define MAX 20
int main(void)
{
   char first[MAX];
   char last[MAX];
   char formal[2 * MAX + 10];
   double prize;

   puts("Введите свое имя:");
   gets(first);

   puts("Введите свою фамилию:");
   gets(last);

   puts("Введите сумму денежного приза:");
   scanf("%lf", &prize);

   sprintf(formal, "%s, %-19s: $%6.2f\n", last, first, prize);
   puts(formal);

   return 0;
}
```

Ниже представлены результаты выполнения этой учебной программы:

```
Введите свое имя:
Остап
Введите свою фамилию:
Бендер
Введите сумму денежного приза:
2000
Бендер, Остап               : $2000.00
```

Команда sprintf() принимает ввод и приводит его к стандартному формату, после чего он сохраняется в строке formal.

Другие строковые функции

Библиотека ANSI C содержит более 20 функций, предназначенных для работы со строками, ниже приводится список наиболее часто используемых функций с краткими описаниями некоторых из них:

- `char *strcpy(char * s1, const char * s2);`

  Эта функция копирует строку (включая нулевой символ), на которую нацелен указатель s2, в ячейку, на которую указывает s1. Возвращаемым значением функции является s1.

- `char *strncpy(char * s1, const char * s2, size_t n);`

  Эта функция копирует в ячейку, на которую указывает `s1`, не более `n` символов из строки, на которую указывает `s2`. Возвращаемым значением является `s1`. Символы, следующие за нулевым символом, не копируются, и если исходная строка короче `n` символов, оставшаяся часть целевой строки заполняется нулевыми символами. Если исходная строка содержит `n` или большее количество символов, нулевые символы не копируются. Возвращаемым значением является `s1`.

- `char *strcat(char * s1, const char * s2);`

  Строка, на которую указывает `s2`, копируется в конец строки, на которую указывает `s1`. Первый символ строки `s2` копируется поверх нулевого символа строки `s1`. Возвращаемым значением является `s1`.

- `char *strncat(char * s1, const char * s2, size_t n);`

  К строке `s1` добавляется не более `n` символов строки `s2`, при этом первый символ строки копируется поверх нулевого символа строки `s1`. Нулевой символ и любые другие символы, которые за ним следуют в строке `s2`, не копируются, нулевой символ добавляется в конец полученной при этом строки. Возвращаемым значением является `s1`.

- `int strcmp(const char * s1, const char * s2);`

  Эта функция возвращает положительное значение, если в машинной схеме упорядочения строка `s1` следует за строкой `s2`, значение `0`, если строки идентичны, и отрицательное значение, если первая строка предшествует второй.

- `int strncmp(const char * s1, const char * s2, size_t n);`

  Эта функция ведет себя так же, как и функция `strcmp()`, за исключением того, что процедура сравнения завершается после просмотра `n` пар символов, либо когда встречается первый нулевой символ, в зависимости от того, какое из этих событий произойдет первым.

- `char *strchr(const char * s, int c);`

  Эта функция возвращает указатель на первую ячейку строки `s`, которая содержит символ `c`. (Завершающий нулевой символ является частью строки, поэтому требуется выполнить его поиск.) Функция возвращает нулевой указатель, если этот символ не найден.

- `char *strpbrk(const char * s1, const char * s2);`

  Эта функция возвращает указатель на первую ячейку строки `s1`, в которой содержится любой символ, найденный в строке `s2`. Эта функция возвращает нулевой указатель, если ни одного символа не найдено.

- `char *strrchr(const char * s, int c);`

  Эта функция возвращает указатель на последнее появление символа `c` в строке `s`. (Завершающий нулевой символ является частью строки, поэтому его поиск также вполне возможен.) Функция возвращает нулевой указатель, если заданный символ не найден.

- `char *strstr(const char * s1, const char * s2);`

  Эта функция возвращает указатель на первое появление строки `s2` в строке `s1`. Функция возвращает нулевой указатель, если строка не найдена.

- `size_t strlen(const char * s);`

  Эта функция возвращает количество символов, найденных в строке s, при этом нулевой символ не учитывается.

Следует отметить, что все эти прототипы используют ключевое слово `const` для того, чтобы показать, какие строки не подвергаются изменению со стороны функции. Например, рассмотрим следующий прототип:

```
char *strcpy(char * s1, const char * s2);
```

Это означает, что `s2` указывает на строку, изменение которой не допускается, по меньшей мере, функцией `strcpy()`, в то же время `s1` указывает на строку, которая может быть изменена. В этом есть смысл, ибо `s1` — целевая строка, которая подвергается изменениям, а `s2` — исходная строка, которая должна оставаться неизменной.

Тип `size_t`, как следует из обсуждения, проведенного в главе 5, может быть любым типом, который возвращает операция `sizeof`. Стандарт C утверждает, что операция `sizeof` возвращает целочисленный тип, но при этом не уточняется, какой это тип, следовательно, тип `size_t` в одной системе может быть `int` без знака, тогда как в другой — `long`. Ваш заголовочный файл `string.h` определяет тип `size_t` для вашей конкретной системы либо ссылается на другой заголовочный файл, в котором имеется необходимое определение.

Как было указано выше, в разделе V справочника (приложение Б) содержится список всех функций семейства `string.h`. Многие реализации добавляют в этот список дополнительные функции, не предусмотренные стандартом ANSI. Вам следует ознакомиться с документацией по конкретной реализации, чтобы знать, какие функции имеются в вашем распоряжении.

Рассмотрим, как одна из этих функций используется в простейшем случае. Ранее было показано, что функция `fgets()` при считывании строки ввода сохраняет символ новой строки в строке назначения. Вы можете воспользоваться функцией `strchr()` для замены символа новой строки нулевым символом. Во-первых, воспользуйтесь функцией `strchr()` для поиска символа новой строки, если, конечно, он есть. Если эта функция обнаруживает символ новой строки, она возвращает его адрес, и вы затем по этому адресу можете поместить нулевой символ:

```
char line[80];
char * find;

fgets(line, 80, stdin);
find = strchr(line, '\n');      // поиск символа новой строки
if (find)                       // если адрес не является нулевым,
   *find = '\0';                // поместить туда нулевой символ
```

Если функция `strchr()` не сможет найти символ новой строки, функция `fgets()` вступит противоречие с предельным значением размера, прежде чем достигнет конца строки. Вы можете добавить конструкцию `else` к оператору `if` и обработать эту ситуацию.

Далее рассмотрим полноценную программу, выполняющую обработку строк.

Пример обработки строк: сортировка строк

Попробуем теперь решить практическую задачу сортировки строк в алфавитном порядке. Эту задачу приходится решать при построении списков имен, при индексации и во многих других ситуациях. Одним из основных инструментальных средств в такой программе является функция `strcmp()`, поскольку она может использоваться для определения порядка следования двух строк. Универсальный подход предусматривает считывание массива строк, их сортировку и вывод на печать. Выше мы предлагали схему считывания строк, с этого мы начнем и данную программу. Печать строк не представляет собой особой проблемы. Мы применим стандартный алгоритм сортировки, который изучим несколько позже. В то же время мы воспользуемся одним необычным приемом, постарайтесь самостоятельно обнаружить его. Программа показана в листинге 11.25.

Листинг 11.25. Программа `sort_str.c`

```
/* sort_str.c -- считывает строки и сортирует их */
#include <stdio.h>
#include <string.h>
#define SIZE 81            /* предельная длина строки, включая \0 */
#define LIM 20             /* максимальное количество считываемых строк */
#define HALT ""            /* нулевая строка для прекращения ввода */
void stsrt(char *strings[], int num);      /* функция сортировки строк */

int main(void)
{
   char input[LIM][SIZE];  /* массив, в котором сохраняются результаты ввода*/
   char *ptstr[LIM];       /* массив переменных типа указатель */
   int ct = 0;             /* счетчик ввода */
   int k;                  /* счетчик вывода */

   printf("Введите не более %d строк, и они будут отсортированы.\n",LIM);
   printf("Чтобы остановить ввод, нажмите клавишу Enter в начале строки.\n");
   while (ct < LIM && gets(input[ct]) != NULL
                   && input[ct][0] != '\0')
   {
      ptstr[ct] = input[ct];         /* установка указателей на строки */
      ct++;
   }
   stsrt(ptstr, ct);                 /* сортировщик строк */
   puts("\nОтсортированный список:\n");
   for (k = 0; k < ct; k++)
       puts(ptstr[k]) ;              /* отсортированные указатели */

   return 0;
}

/* функция сортировки указателей строк */
void stsrt(char *strings[], int num)
{
   char *temp;
   int top, seek;
```

```
    for (top = 0; top < num-1; top++)
        for (seek = top + 1; seek < num; seek++)
            if (strcmp(strings[top],strings[seek]) > 0)
            {
               temp = strings[top];
               strings[top] = strings[seek];
               strings[seek] = temp;
            }
}
```

Для тестирования программы, представленной в листинге 11.22, был выбран отрывок из поэмы А. С. Пушкина "Руслан и Людмила":

```
Введите не более 20 строк, и они будут отсортированы.
Чтобы остановить ввод, нажмите клавишу Enter в начале строки.
У лукоморья дуб зеленый;
Златая цепь на дубе том:
И днем и ночью кот ученый
Все ходит по цепи кругом;

Отсортированный список:

Все ходит по цепи кругом;
Златая цепь на дубе том:
И днем и ночью кот ученый
У лукоморья дуб зеленый;
```

Похоже, что стих не слишком пострадал от упорядочения строк в алфавитном порядке.

Сортировка указателей вместо строк

Необычный прием в рассматриваемой программе состоит в том, что вместо изменения порядка следования мы просто упорядочиваем *указатели* на эти строки. Приведем соответствующие пояснения. Первоначально элементу `ptrst[0]` устанавливается значение `input[0]` и так далее. Это означает, что указатель `ptrst[i]` ссылается на первый символ массива `input[i]`. Каждый элемент `input[i]` представляет собой массив из 81 элемента, а каждый элемент `ptrst[i]` — отдельную переменную. Процедура сортировки упорядочивает массив указателей `ptrst`, оставляя `input` неизменным. Если, например, `input[1]` предшествует `input[0]` в алфавитном порядке, программа переключает указатели `ptrst`, в результате чего `ptrst[0]` указывает на начало строки `input[1]`, а `ptrst[1]` — на начало строки `input[0]`. Это намного проще, чем применение, скажем, функции `strcpy()` для обмена содержимым двух входных строк. На рис. 11.6 представлена еще одна точка зрения на этот процесс. Кроме того, он обладает еще и тем преимуществом, что сохраняет первоначальный порядок массива `input`.

Выбор алгоритма сортировки

Для сортировки указателей мы воспользуемся алгоритмом *сортировки методом выбора*. Идея этого алгоритма заключается в том, что каждый элемент поочередно сравнивается с первым элементом. Если сравниваемый элемент предшествует текущему первому элементу, программа меняет их местами.

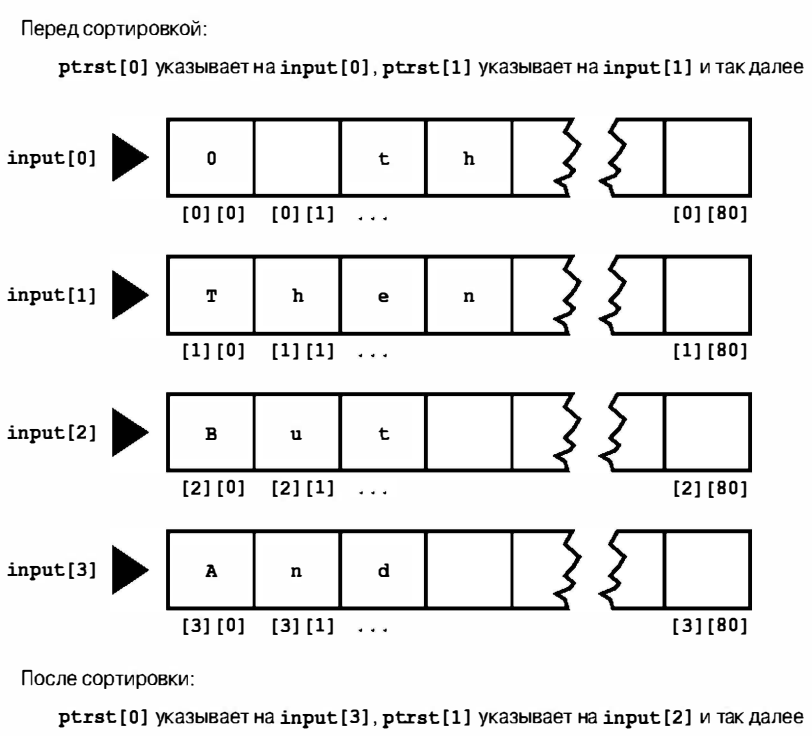


Рис. 11.6. *Сортировка указателей на строки*

К тому моменту, когда программа достигнет конца цикла, первый элемент содержит указатель на элемент, идущий первым в машинной схеме упорядочения. Затем внешний цикл `for` повторяет этот процесс, начиная со второго элемента `input`. Когда внутренний цикл завершает свою работу, указатель на строку, идущую второй, становится вторым элементом массива `ptrst`. Процесс продолжается до тех пор, пока все элементы не будут отсортированы.

Теперь рассмотрим сортировку методом выбора более подробно. Схема, в соответствии с которой работает эта сортировка, выражается в псевдокоде следующим образом:

```
для n = первый до n = предпоследний элемент,
   найти наибольшее из оставшихся чисел и поместить его в n-й элемент
```

Эта схема функционирует следующим образом. Работа начинается с `n = 0`. Выполните просмотр всего массива, найдите наибольшее число и поменяйте его местами с первым элементом. Далее установите `n = 1` и просмотрите все элементы массива, кроме первого. Найдите наибольший из оставшихся элементов и поменяйте его местами со вторым элементом. Продолжайте этот процесс до тех пор, пока не достигнете предпоследнего элемента. Теперь остаются только два элемента. Сравнение их и поместите больший в позицию предпоследнего элемента. Теперь наименьший элемент занял свою окончательную позицию.

Создается впечатление, что эту задачу можно решать с помощью цикла `for`, тем не менее, нам все-таки придется более подробно описать процесс “найти и разместить”.

Один из способов выбора максимального значения из числа оставшихся значений предусматривает операцию сравнения первого и второго элементов оставшегося массива. Если второй элемент больше первого, необходимо выполнить обмен их значений. Далее первый элемент сравнивается с третьим. Если третье значение больше, нужно выполнить обмен значений первого и третьего элементов. Если третий элемент больше, обменяйте значениями эти два элемента. Каждый такой обмен перемещает больший элемент в вершину списка. Продолжайте действовать таким образом до тех пор, пока не произойдет сравнение первого элемента с последним. Когда вы закончите этот процесс, наибольшее значение будет в первом элементе получившегося при этом массива. Вам удалось выявить первый элемент, однако оставшийся массив остается неупорядоченным. Представим дальнейшую процедуру в псевдокоде:

```
для n = второй до последнего элемента,
   сравнить n-й элемент с первым элементом;
   если n-й элемент больше, выполнить обмен их значениями
```

Этот процесс аналогичен еще одному циклу `for`. Он должен быть вложен в первый цикл. Внешний цикл указывает, какой элемент массива должен быть заполнен, а внутренний цикл находит значение, которое в него следует поместить. Объединяя обе части псевдокода в единое целое и транслируя его в язык C, мы получаем функцию, показанную в листинге 11.25. Между прочим, в библиотеке C имеется более совершенная функция сортировки под именем `qsort()`. Среди всего прочего она использует указатель на функцию, выполняющую сортирующее сравнение. В главе 16 приводятся примеры ее использования.

Символьные функции `ctype.h` и строки

В главе 7 было представлено описание семейства `ctype.h` функций обработки символов. Эти функции не могут быть применены к строке как к единому целому, в то же время они могут применяться к отдельным символам строки. В листинге 11.26, например, дано определение функции, которая применяет функцию `toupper()` к каждому символу строки, тем самым, преобразуя всю строку к верхнему регистру. Она также определяет функцию, которая использует функцию `ispunct()` для подсчета знаков препинания в строке.

Листинг 11.26. Программа `mod_str.c`

```
/* mod_str.c -- модифицирует строку */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define LIMIT 80
void ToUpper(char *);
int PunctCount(const char *);

int main(void)
{
   char line[LIMIT];

   puts("Введите строку:");
   gets(line);
   ToUpper(line);
```

```
   puts(line);
   printf("Эта строка содержит %d знаков препинания.\n",
           PunctCount(line));
   return 0;
}
void ToUpper(char * str)
{
   while (*str)
   {
       *str = toupper(*str);
        str++;
   }
}
int PunctCount(const char * str)
{
   int ct = 0;
   while (*str)
   {
       if (ispunct(*str))
          ct++;
       str++;
   }
   return ct;
}
```

Цикл `while (*str)` выполняет обработку каждого символа строки, на которую указывает `str`, пока не встретится нулевой символ. В этой точке `*str` получает значение `0` (код нулевого символа), или ложное значение, и по этой причине цикл прекращается. Ниже показан пример выполнения этой демонстрационной программы:

```
Введите строку:
Спокойно, спокойно. За дело берусь я. Заседание продолжается.
СПОКОЙНО, СПОКОЙНО. ЗА ДЕЛО БЕРУСЬ Я. ЗАСЕДАНИЕ ПРОДОЛЖАЕТСЯ.
Эта строка содержит 4 знаков препинания.
```

Функция `Upper()` применяет функцию `toupper()` к каждому символу строки. (Тот факт, что язык C различает символы верхнего и нижнего регистра, требует, чтобы имена этих функций отличались друг от друга.) В соответствии с требованиями стандарта ANSI C, функция `toupper()` может изменять только символы нижнего регистра. В то же время, в очень старых реализациях языка C такая проверка не выполняется автоматически, поэтому потребуется примерно такой код:

```
if (islower(*str)) /* до появления стандарта ANSI C — проверка
                      перед преобразованием */
   *str = toupper(*str);
```

Кроме того, функции семейства `ctype.h` обычно реализованы как *макросы*. Таковыми являются препроцессорные конструкции языка C, которые в своих действиях напоминают функции, но с некоторыми довольно существенными отличиями. Макросы рассматриваются в главе 16.

Далее восполним некоторый пробел в знаниях, выяснив, все о круглых скобках функции `main()`.

Аргументы командной строки

До появления современного графического интерфейса существовал интерфейс командной строки. Примерами могут служить операционные системы DOS и Unix. *Командная строка* — это последовательность символов, которую вы вводите с клавиатуры в среде командной строки. Предположим, что ваша программа хранится в файле с именем `fuss`. Командная строка для запуска этой программы в среде Unix может иметь следующий вид:

```
$ fuss
```

В операционной системе Windows, функционирующей в режиме командной строки, она может выглядеть так:

```
C> fuss
```

Аргументы командной строки — это дополнительные элементы той же строки. Рассмотрим соответствующие примеры:

```
% fuss -r Ginger
```

Программа на языке C может считывать эти дополнительные элементы для собственных целей (рис. 11.7).

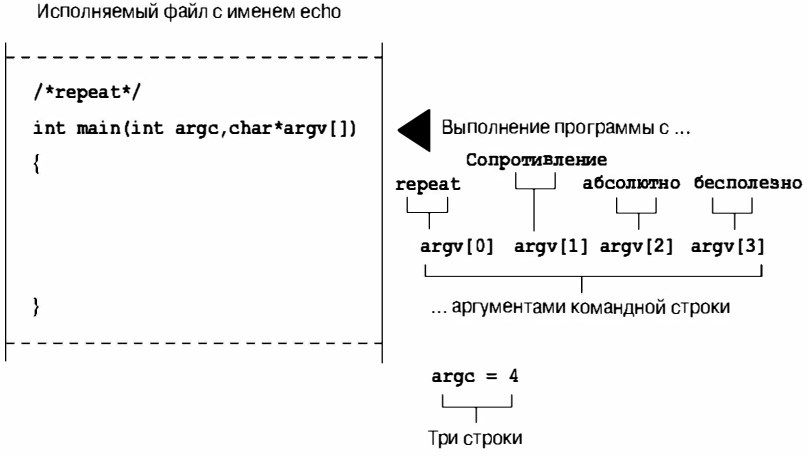


Рис. 11.7. *Аргументы командной строки*

Программа на C читает эти элементы, используя аргументы функции `main()`. В листинге 11.27 показан типичный пример.

Листинг 11.27. Программа `repeat.c`

```
/* repeat.c -- функция main() с аргументами */
#include <stdio.h>
int main(int argc, char *argv[])
{
   int count;
   printf("В командной строке имеются %d аргументов:\n", argc - 1);
```

```
    for (count = 1; count < argc; count++)
        printf("%d: %s\n", count, argv[count]);
    printf("\n");
    return 0;
}
```

Скомпилируйте эту программу в исполняемый файл с именем `repeat`. Вот что происходит, когда вы запускаете ее в командной строке:

```
C>repeat Сопротивление абсолютно бесполезно
В командной строке имеются 3 аргументов:
1: Сопротивление
2: абсолютно
3: бесполезно
```

Наверняка вы поняли, почему программа называется `repeat` ("повторить"), но, возможно, вы езе и хотите знать, как она работает. Ниже даются необходимые пояснения.

Компиляторы языка C позволяют функции `main()` не иметь аргументов либо иметь два аргумента. (В некоторых реализациях допускаются дополнительные аргументы, но это рассматривается как расширение стандарта.) В случае двух аргументов первым является количество строк в командной строке. По традиции (но не обязательно), этот аргумент типа `int` имеет имя `argc` (сокращение от *argument count* — количество аргументов). С помощью пробелов строки отделяются друг от друга. В силу этого, в примере с `repeat` используются четыре строки, включая строку с именем команды, а в примере с `fuss` — три строки. Второй аргумент — это массив указателей на строки. Каждая строка командной строки хранится в памяти, при этом для каждой из них предусмотрен соответствующий указатель. По соглашению этот массив указателей получил имя `argv` (сокращение от *argument values* — значения аргументов). Там, где это возможно (некоторые операционные системы не допускают этого), элементу `argv[0]` присваивается имя самой программы. Затем элементу `argv[1]` присваивается первая из последующих строк и так далее. Что касается нашего примера, то справедливы следующие утверждения:

`argv[0]` указывает на `repeat` (в большинстве систем)

`argv[1]` указывает на `Сопротивление`

`argv[2]` указывает на `абсолютно`

`argv[3]` указывает на `бесполезно`

Программа, показанная в листинге 11.27, использует цикл `for` для вывода строк на печать в порядке очередности. Напомним, что спецификатор `%s` функции `printf()` настроен на то, что в качестве аргумента будет передан адрес строки. Каждый элемент, а именно, `argv[0]`, `argv[1]` и так далее, представляет собой адрес.

Эта форма аналогична форме любой другой функции, принимающей формальные аргументы. Многие программисты используют другое объявление аргумента `argv`:

```
int main(int argc, char **argv)
```

Это альтернативное объявление аргумента `argv` на самом деле эквивалентно `char *argv[]`. Оно говорит о том, что `argv` является указателем на указатель на `char`.

Рассматриваемый пример сводится к тому же. В нем содержится массив из семи элементов. Именем массива является указатель на первый элемент, следовательно, argv указывает на argv[0], а argv[0] — указатель на значение char. Следовательно, даже в случае исходного определения argv представляет собой указатель на указатель на char. Вы можете использовать любую форму, тем не менее, мы полагаем, что первая форма более отчетливо говорит о том, что argv представляет некоторое множество строк.

Кроме того, многие операционные среды, в том числе Unix и DOS, допускают использование кавычек для объединения нескольких слов в единый аргумент. Например, команда

```
repeat "Ух, как я зол" сейчас
```

присваивает строку "Ух, как я зол" элементу argv[1], а строку "сейчас" — элементу argv[2].

Аргументы командной строки в интегрированных средах

В интегрированных средах Windows, таких как Metrowerks CodeWarrior, Microsoft Visual C++ и Borland C/C++, командные строки для запуска программ не используются. В то же время, в некоторых из них имеются меню, с помощью которых можно задать аргументы командной строки. В других случаях вы можете получить возможность скомпилировать программу в IDE (Integrated Development Environment — интегрированная среда разработки), а затем открыть окно MS-DOS для запуска программы на выполнение в режиме командной строки.

Аргументы командной строки в среде Macintosh

Операционная система Macintosh не использует командные строки, однако Metrowerks CodeWarrior позволяет смоделировать среду командной строки с помощью функции ccommand(). Воспользуйтесь заголовочным файлом console.h и запустите программу следующим образом:

```
#include <stdio.h>
#include <console.h>
int main(int argc, char *argv[])
{
   ...                              /* объявления переменных */
   argc = ccommand(&argv);
   ...
}
```

Когда в программе очередь доходит до вызова функции ccommand(), она выводит на экран диалоговое окно с полем, в котором можно ввести командную строку. Затем команда помещает слова командной строки в строки argv и возвращает количество слов. Имя текущего проекта появится в качестве первого слова в прямоугольнике командной строке, следовательно, вы должны ввести аргументы командной строки после этого имени. Функция ccommand() позволяет также эмулировать перенаправление.

Почему такие вопросы возникают в системе Macintosh? По-видимому, единственная причина этого состоит в том, чтобы освоить работу с идиомами командной строки на тот случай, когда в один прекрасный момент вам придется писать программы на базе командной строки. Теперь, когда компания Macintosh перешла на работу под управлением Unix-подобной операционной системы Mac OS X, программисты, работающие в системе Mac, могут ближе ознакомиться с программами, функционирующими в режиме командной строки.

Преобразование строк в числа

Числа могут храниться в форме строк или в числовой форме. Сохранение числа в виде строки означает сохранение цифровых символов. Например, число 213 может быть сохранено в виде строки символов, то есть как последовательность цифр `'2'`, `'1'`, `'3'`, `'\0'`. Сохранение числа 213 в числовой форме означает его хранение как значения, скажем, типа `int`.

Язык C требует числовых форм для операций над числами, таких как сложение или сравнение чисел, однако отображение чисел на экране требует строковой формы, поскольку экран воспроизводит символы. Функции `printf()` и `sprintf()`, независимо от спецификатора `%d` и других спецификаторов, которые они используют, преобразуют числовые формы в строковые и наоборот. В C также имеются функции, единственная цель которых заключается в преобразовании строковых форм в числовые.

Предположим, например, что вам нужна программа, использующая аргумент командной строки. К сожалению, аргументы командной строки считываются как строки. В связи с этим, чтобы использовать числовое значение, вы сначала должны преобразовать строку в число. Если число целое, вы можете воспользоваться функцией `atoi()` (сокращение от *alphanumeric to integer* — преобразование алфавитно-цифрового значения в целочисленное). Эта функция принимает строку в качестве аргумента и возвращает соответствующее целочисленное значение. В листинге 11.28 показан пример ее использования.

Листинг 11.28. Программа `hello.c`

```
/* hello.c -- преобразует аргумент командной строки в число */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   int i, times;

   if (argc < 2 || (times = atoi(argv[1])) < 1)
       printf("Использование: %s положительное-число\n", argv[0]);
   else
       for (i = 0; i < times; i++)
           puts("Всего хорошего!");

   return 0;
}
```

Приводим результаты выполнения этой учебной программы:

```
% hello 3
Всего хорошего!
Всего хорошего!
Всего хорошего!
```

Знак % — это приглашение операционной системы Unix. Аргумент командной строки, представленный числом 3, сохраняется в виде строки 3\0. Функция atoi() преобразует эту строку в целочисленное значение 3, которое присваивается переменной times. Это число определяет количество итераций цикла for.

Если вы запустите эту программу без аргумента командной строки, проверка условия argc < 2 приводит к прекращению выполнения программы и выводу сообщения, в котором описаны правила использования программы. То же происходит, если переменная times получает 0 или отрицательное значение. Порядок вычисления логических операций языка C гарантирует, что при argc < 2 вычисление atoi(argv[1]) не производится.

Функция atoi() работает, если строка начинается с целого числа. В этом случае она выполняет преобразование символов до тех пор, пока не встретит нечто такое, что не является частью целого числа. Например, atoi("42regular") возвращает целое значение 42. А что произойдет, если командная строка примет значение hello what или что-то в этом роде? В реализации языка C, с которой мы имеем дело, функция atoi() возвращает значение 0, если ее аргумент не распознан как числовой. В то же время в соответствии со стандартом ANSI поведение этой функции в данном случае не определено. Функция strtol(), к изучению которой мы вскоре перейдем, обеспечивает контроль ошибок, тем самым повышая надежность.

Мы включили в программу заголовочный файл stdlib.h, который в условиях стандарта ANSI C содержит объявление функции atoi(). Этот заголовочный файл включает объявления функций atof() и atol(). Функция atof() преобразует строку в значение типа double, а функция atol() — в значение типа long. Они работают аналогично функции atoi(), и поэтому имеют, соответственно, тип double и тип long.

Стандарт ANSI C предусматривает более сложные версии этих функций: функция strtol() преобразует строку в тип long, функция strtoul() — в тип long без знака, а функция strtod() — в значение типа double. Более сложный аспект этих функций заключается в том, что они обнаруживают и извещают о первом символе строки, который не является частью числа. Кроме того, функции strtol() и strtoul() позволяют задавать основание системы счисления.

Рассмотрим пример программы, в которой используется функция strtol(). Ее прототип имеет вид:

```
long strtol(const char *nptr, char **endptr, int base);
```

В этом случае nptr — это указатель на строку, которую необходимо преобразовать, endptr — адрес указателя, который ссылается на адрес символа, прекращающего ввод числа, а base — основание системы счисления, в которой представлено вводимое число. Пример, приведенный в листинге 11.29, поясняет сказанное.

Листинг 11.29. Программа strcnvt.c

```
/* strcnvt.c -- использование функции strtol() */
#include <stdio.h>
```

```
#include <stdlib.h>
int main()
{
   char number[30];
   char * end;
   long value;
   puts("Введите число (или пустую строку для выхода из программы):");
   while(gets(number) && number[0] != '\0')
   {
      value = strtol(number, &end, 10);  /* десятичная система счисления */
      printf("значение: %ld, останов на %s (%d)\n",
              value, end, *end);
      value = strtol(number, &end, 16);  /* шестнадцатеричная система счисления */
      printf("значение: %ld, останов на %s (%d)\n",
              value, end, *end);
      puts("Следующее число:");
   }
   puts("Всего хорошего!\n");
   return 0;
}
```

Приводим некоторые результаты выполнения этой учебной программы:

```
Введите число (или пустую строку для выхода из программы):
10
значение: 10, останов на  (0)
значение: 16, останов на  (0)
Следующее число:
10atom
значение: 10, останов на atom (97)
значение: 266, останов на tom (116)
Следующее число:

Всего хорошего!
```

Во-первых, обратите внимание, что "10" преобразуется в число 10, когда система счисления десятичная, и в 16, когда шестнадцатеричная. Кроме того, обратите внимание на то, что если end указывает на символ, то *end является символом. Поэтому первое преобразование завершается, когда достигается нулевой символ, следовательно, end указывает на нулевой символ. Вывод указателя end на печать приводит к тому, что на экран выводится пустая строка, а вывод *end в формате %d отображает ASCII-код нулевого символа.

Для второй строки ввода (в интерпретации с десятичным представлением), указатель end получает адрес символа 'a'. Следовательно, вывод на печать указателя end отображает строку "atom", а вывод на печать *end отображает ASCII-код символа 'a'. Однако если в качестве системы счисления будет принята шестнадцатеричная, символ 'a' будет рассматриваться как полноправная шестнадцатеричная цифра, и функция преобразует шестнадцатеричное число 10a в десятичное 266.

Функция strtol() работает вплоть до 36-ричной системы счисления, используя в качестве цифр все буквы алфавита до 'z'. Функция strtoul() делает то же самое, но

выполняет преобразование значений без знака. Функция `strtod()` работает только в десятичной системе счисления, поэтому она принимает только два аргумента.

Во многих реализациях имеются функции `itoa()` и `ftoa()`, обеспечивающие преобразование целочисленных значений и значений с плавающей запятой в строки. Тем не менее, они не являются частью библиотеки ANSI C; дабы не нарушать совместимость, используйте функцию `sprintf()`.

Ключевые понятия

Многие программы выполняют обработку текстовых данных. Программа может предложить ввести ваше имя, список корпораций, адрес, ботаническое название типа папоротника, музыкальное сопровождение и так далее; поскольку мы взаимодействуем с окружающим миром посредством слов, примерам использования текстов буквально нет конца. Строки являются средством, используемым программами на языке C.

Строка в C, независимо от того, представлена ли она в виде символьного массива, указателя или строкового литерала, представляет собой последовательность байтов в памяти, содержащих коды символов, причем эта последовательность завершается нулевым символом. Язык C признает пользу от применения строк, о чем свидетельствует наличие библиотеки функций, манипулирующих строками, осуществляющих поиск и анализ строк. В частности, не упускайте из виду тот факт, что вы должны применять функцию `strcmp()` вместо операций отношений при сравнении строк, вы также должны использовать функцию `strcpy()` или `strncpy()` вместо операций присваивания строки символьному массиву.

Резюме

Строка в языке C представляет собой последовательность значений типа `char`, которая оканчивается нулевым символом `'\0'`. Строка может храниться в символьном массиве. Строка может быть представлена *строковой константой*, в которой символы, за исключением нулевого, заключены в двойные кавычки. Нулевой символ расставляет компилятор. В силу этого строка `"joy"` сохраняется в памяти как последовательность из четырех символов `j`, `o`, `y` и `\0`. Длина строки, измеренная с помощью функции `strlen()`, не учитывает нулевого символа.

Строковые константы, известные также как *строковые литералы*, могут использоваться для инициализации символьных массивов. Размер массива должен быть, по меньшей мере, на единицу больше, чем длина строки, чтобы можно было включить нулевой символ. Строковые константы также могут применяться для инициализации указателей на значение типа `char`. Для идентификации обрабатываемой строки функции используют указатель на первый символ этой строки. В общем случае соответствующим фактическим аргументом является имя массива, переменная типа указатель или строка, заключенная в кавычки. В любом случае передается адрес первого символа. В общем случае нет необходимости передавать длину строки, поскольку функция может использовать завершающий нулевой символ для идентификации конца строки.

Функции `gets()` и `puts()`, соответственно, считывают строку ввода и отображают строку вывода. Они являются частью семейства функций `stdio.h`.

Библиотека C включает несколько функций *обработки строк*. В условиях действия стандарта ANSI C эти функции объявлены в файле `string.h`. Данная библиотека также содержит несколько функций *обработки символов*; они объявлены в заголовочном файле `ctype.h`.

Вы можете предоставить доступ к *аргументам командной строки* путем передачи двух соответствующих формальных переменных функции `main()`. Первый аргумент, который по традиции называется `argc`, имеет тип `int`, ему присваивается количество слов в командной строке. Второй аргумент, которому традиционно присваивается имя `argv`, представляет собой указатель на массив указателей значений типа `char`. Каждый указатель на значение типа `char` ссылается на одну из строк командной строки, при этом `argv[0]` указывает на имя команды, `argv[1]` — на первый аргумент командной строки, `argv[2]` — на второй аргумент и так далее.

Функции `atoi()`, `atol()` и `atof()` преобразуют строковые представления чисел, соответственно, в формы типы `int`, `long` и `double`. Функции `strtol()`, `strtoul()` и `strtod()` преобразуют строковые представления чисел, соответственно, в формы типа `long`, `unsigned long` и `double`.

Вопросы для самоконтроля

1. Какая ошибка допущена в представленном объявлении символьной строки?

```
int main(void)
{
   char name[] = {'F', 'e', 's', 's' };
   ...
}
```

2. Что напечатает следующая программа?

```
#include <stdio.h>
int main(void)
{
   char note[] = "Увидимся в кафе.";
   char *ptr;
   ptr = note;
   puts(ptr);
   puts(++ptr);
   note[7] = '\0';
   puts(note);
   puts(++ptr);
   return 0;
}
```

3. Что напечатает следующая программа?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char food[] = "Yummy";
   char *ptr;
```

```
    ptr = food + strlen(food);
    while (--ptr >= food)
        puts(ptr);
    return 0;
}
```

4. Что напечатает следующая программа?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char goldwyn[40] = "асть этого ";
    char samuel[40] = "Я читал ч";
    const char * quote = "всю дорогу.";
    strcat(goldwyn, quote);
    strcat(samuel, goldwyn);
    puts(samuel);
    return 0;
}
```

5. Приведенное ниже упражнение позволяет приобрести опыт работы со строками, циклами, указателями и инкрементированием указателей. Во-первых, предположим, что имеется следующее определение функции:

```
#include <stdio.h>
char *pr (char *str)
{
    char *pc;
    pc = str;
    while (*pc)
        putchar(*pc++);
    do {
        putchar(*--pc);
        } while (pc - str);
    return (pc);
}
```

Рассмотрим вызов этой функции:

```
x = pr("Хо Хо Хо!");
```

а. Что при этом выводится на печать?

б. Какой тип должна иметь переменная `x`?

в. Какое значение получает переменная `x`?

г. Что означает выражение `*--pc`, и чем оно отличается от выражения `--*pc`?

д. Что будет выведено на печать, если `*--pc` заменить на `*pc--`?

е. Для чего выполняется проверка двух выражений `while`?

ж. Что произойдет, если функции `pr()` будет передана пустая строка в качестве аргумента?

з. Что нужно сделать в вызывающей функции, чтобы функцию `pr()` можно было использовать так, как показано выше?

6. Предположим, что имеется следующее объявление:

```
char sign = '$';
```

Сколько байтов памяти занимает этот знак? А знак '$'? Знак "$"?

7. Что распечатывает следующая программа?

```
#include <stdio.h>
#include <string.h>
#define M1 "How are ya, sweetie? "
char M2[40] = "Beat the clock.";
char * M3 = "chat";
int main(void)
{
   char words[80];
   printf(M1);
   puts(M1);
   puts(M2);
   puts(M2 + 1);
   strcpy(words,M2);
   strcat(words, " Win a toy.");
   puts(words);
   words[4] = '\0';
   puts(words);
   while (*M3)
      puts(M3++);
   puts(--M3);
   puts(--M3);
   M3 = M1;
   puts(M3);
   return 0;
}
```

8. Что распечатывает следующая программа?

```
#include <stdio.h>
int main(void)
{
   char str1[] = "gawsie";
   char str2[] = "bletonism";
   char *ps;
   int i = 0;
   for (ps = str1; *ps != '\0'; ps++) {
        if ( *ps == 'a' || *ps == 'e')
               putchar(*ps);
        else
               (*ps)--;
        putchar(*ps);
      }
   putchar('\n');
   while (str2[i] != '\0' ) {
      printf("%c", i % 3 ? str2[i] : '*');
      ++i;
      }
   return 0;
}
```

9. Функция `strlen()` принимает указатель на строку в качестве аргумента и возвращает длину строки. Напишите свою версию этой функции.
10. Разработайте функцию, которая принимает указатель на строку в качестве аргумента и возвращает указатель на первый символ пробела в строке в позиции, на которую ссылается указатель или после нее. Функция должна возвратить нулевой указатель, если пробелы не найдены.
11. Перепишите программу из листинга 11.17, используя функции из заголовочного файла `ctype.h` для того, чтобы программа распознавала правильный ответ независимо от применения пользователем букв верхнего или нижнего регистров.

Упражнения по программированию

1. Разработайте и протестируйте функцию, которая загружает с устройства ввода очередные `n` символов (включая пробелы, символы табуляции и символы новой строки) и запоминает результаты в массиве, адрес которого передается ей в качестве аргумента.
2. Внесите соответствующие изменения и затем протестируйте функцию из упражнения 1 с тем, чтобы она прекращала ввод после загрузки `n` символов или по достижении первого символа пробела, табуляции или новой строки, в зависимости от того, какой из них поступит первым. (Не ограничивайтесь только использованием функции `scanf()`.)
3. Разработайте и протестируйте функцию, которая читает первое слово строки ввода в массив и игнорирует остальную часть строки. Определите слово как последовательность символов без пробелов, символов табуляции или символов новой строки.
4. Разработайте и протестируйте функцию, которая осуществляет поиск в строке, переданной в первом параметре, первого появления символа, заданного во втором параметре функции. Функция должна возвратить указатель на этот символ, и ноль, если указанный символ в строке не найден. (Поведение этой функции дублирует действия библиотечной функции `strchr()`.) Выполните тестирование функции в рамках завершенной программы, которая использует цикл для передачи входных значений рассматриваемой функции.
5. Напишите функцию под именем `is_within()`, которая принимает символ и указатель на строку как два параметра функции. Функция должна возвращать ненулевое значение (`true`), если заданный символ содержится в строке и ноль (`false`) в противном случае. Протестируйте эту функцию в рамках завершенной программы, которая использует цикл для передачи входных значений рассматриваемой функции.
6. Функция `strncpy(s1,s2,n)` копирует ровно `n` символов из строки `s2` в строку `s1`, усекая при необходимости строку `s2` или заполняя избыточные символы дополнительными нулевыми символами. Целевая строка может не содержать завершающего нулевого символа, если длина строки `s2` равна или больше `n`. Функция возвращает строку `s1`. Напишите собственную версию этой функции. Протестируйте эту функцию в рамках завершенной программы, которая использует цикл для передачи входных значений рассматриваемой функции.

7. Напишите функцию под именем `string_in()`, которая принимает два указателя на строки в качестве аргументов. Если вторая строка содержится в первой, функция должна возвратить адрес, с которого начинается вторая строка в первой строке. Например, вызов `string_in("hats", "at")` возвращает адрес символа `a` в строке `hats`. В противном случае функция должна возвратить нулевой указатель. Протестируйте эту функцию в рамках завершенной программы, которая использует цикл для передачи входных значений рассматриваемой функции.
8. Напишите функцию, которая замещает содержимое заданной строки этой же строкой, но в обратном порядке. Протестируйте эту функцию в рамках завершенной программы, которая использует цикл для передачи входных значений рассматриваемой функции.
9. Напишите функцию, которая принимает строку в качестве аргумента и удаляет из этой строки все пробелы. Протестируйте эту функцию в рамках программы, которая использует цикл для чтения строк до тех пор, пока не будет введена пустая строка. Программа должна применять эту функцию к каждой входной строке и отображать результат на экране.
10. Напишите программу, которая читает строки и прекращает чтение на десятой строке включительно или при появлении символа `EOF`, в зависимости от того, какое из этих событий произойдет первым. Функция должна предложить пользователю меню с пятью вариантами: печать исходного списка строк, печать строк в виде упорядоченной последовательности в кодировке ASCII, печать строк в порядке возрастания их длины, печать строк в порядке возрастания длины первого слова строки и выход из программы. Меню должно выводиться на экран после выполнения каждой операции, указанной меню, пока пользователь не выберет вариант выхода из программы. Разумеется, программа должна выполнять все указанные в меню задачи.
11. Напишите программу, которая читает входные данные до тех пор, пока не встретится символ `EOF`, и выводит количество слов, прописных букв, строчных букв, знаков препинания и цифр. Используйте семейство функций `ctype.h`.
12. Напишите программу, которая воспроизводит аргументы командной строки в обратном порядке. Другими словами, если аргументами командной строки являются `до скорого свидания`, данная программа должна вывести на экран `свидания скорого до`.
13. Напишите программу возведения в степень, которая работает в режиме командной строки. Первым аргументом командной строки должно быть число типа `double`, возводимое в соответствующую степень, а вторым аргументом — целочисленный показатель степени.
14. С помощью функций классификации символов создайте свою реализацию функции `atoi()`.
15. Напишите программу, которая читает входные данные до тех пор, пока не встретится символ `EOF`, и выводит эти данные на экран. Эта функция должна распознавать и реализовывать следующие аргументы командной строки:
    - `-p` Печатать входные данные такими, какими они есть.
    - `-u` Преобразовать входные данные к верхнему регистру.
    - `-l` Преобразовать входные данные к нижнему регистру.

ГЛАВА 12

Классы памяти, компоновка и управление памятью

В этой главе:

- Ключевые слова: `auto`, `extern`, `static`, `register`, `const`, `volatile`, `restricted`
- Функции: `rand()`, `srand()`, `time()`, `malloc()`, `calloc()`, `free()`
- Определение области видимости переменной (насколько широко она известна) и времени жизни переменной (насколько долго она существует) в C
- Разработка более сложных программ

Одно из достоинств языка C заключается в том, что он позволяет управлять тонкими моментами в программах. Система управления памятью языка C служит примером такого управления благодаря тому, что позволяет определить, какая функция использует те или иные переменные и как долго та или иная переменная действует в программе. Использование памяти является еще одним аспектом процесса проектирования программы.

Классы памяти

Язык C предлагает пять различных моделей, или *классов памяти*, для хранения переменных. Существует также и шестая модель, в основу которой положены указатели; к ее изучению мы перейдем далее в этой главе (в разделе "Распределенная память: функции `malloc()` и `free()`"). Вы можете описать переменную (или, если использовать более общий термин, объект данных) через *продолжительность хранения*, указывающую, насколько долго она будет храниться в памяти, и через ее *область видимости* и *связывание*, которые вместе указывают на то, какая часть программы может использовать эту переменную, ссылаясь на ее имя. Различные классы памяти предлагают различные сочетания области видимости, связывания и продолжительности хранения. Вы можете использовать переменные, область видимости которых распространяется на несколько файлов исходного кода, переменные, которые могут использоваться любыми функциями из одного конкретного файла, переменные, которые могут применяться только в одной конкретной функции, и даже переменные, которые могут ис-

пользоваться только в некотором подразделе конкретной функции. Одни переменные могут существовать на протяжении всего времени жизни программы, в то же время другие переменные могут существовать только до тех пор, пока выполняется функция, которая их содержит. Вы также можете хранить данные в памяти, которая выделяется и освобождается явно посредством вызовов соответствующих функций.

Прежде чем приступать к изучению пяти указанных выше классов памяти, следует ознакомиться с такими понятиями, как *область видимости*, *связывание* и *продолжительность хранения*. После этого мы вернемся к изучению конкретных классов памяти.

Область видимости

Область видимости описывает участок или участки программы, из которых можно получить доступ к конкретному идентификатору. Для переменной в языке C характерна одна из следующих областей видимости: область видимости в пределах блока, область видимости в пределах прототипа функции и область видимости в пределах файла. Во всех рассмотренных до сих пор примерах программ использовался область видимости на уровне блока. Блок — это фрагмент программного кода, содержащийся в фигурных скобках, при этом каждой открывающей скобке соответствует своя закрывающая скобка. Например, все тело функции представляет собой блок. Любой составной оператор в рамках конкретной функции также является блоком. Переменная, определенная внутри блока, имеет область видимости в пределах блока, она видна от точки, в которой она определена, до конца блока, который содержит ее определение. Наряду с этим, формальные параметры функции, даже если они появляются раньше, чем открывающая фигурная скобка функции, имеют в качестве области видимости весь блок и принадлежат блоку, содержащему тело функции. Следовательно, все переменные, которые использовались до сих пор, включая формальные параметры функции, имеют блок в качестве области видимости. Таким образом, переменные `cleo` и `patrick`, используемые в приведенном ниже коде, имеют областью видимости блок, простирающийся вплоть до закрывающей фигурной скобки:

```
double blocky(double cleo)
{
   double patrick = 0.0;
   ...
   return patrick;
}
```

Переменные, объявленные во внутреннем блоке, имеют область видимости, ограниченную только этим блоком:

```
double blocky(double cleo)
{
   double patrick = 0.0;
   int i;
   for (i = 0; i < 10; i++)
   {
       double q = cleo * i;     // начало области видимости переменной q
       ...
       patrick *= q;
   }                            // конец области видимости переменной q
   ...
   return patrick;
}
```

В этом примере область видимости переменной q ограничена внутренним блоком, и только программный блок, расположенный внутри этого блока, имеет возможность доступа к переменной q.

По традиции переменные с областью видимости в пределах блока должны быть объявлены в начале блока. Стандарт C99 ослабил это требование, позволяя объявлять переменные в любом месте блока. Одна из новых возможностей касается раздела управления выполнением цикла for. Другими словами, теперь вы можете выполнять следующие операции:

```
for (int i = 0; i < 10; i++)
   printf("Возможность C99: i = %d", i);
```

С учетом этого нового свойства стандарт C99 расширяет понятие блока, обеспечивая возможность включения в него программных кодов, управляемых циклами for, while, do while или оператором if, даже если скобки при этом не используются. Таким образом, в предыдущем примере цикла for переменная i рассматривается как часть блока for. Следовательно, область видимости этой переменной ограничена циклом for. После того, как поток управления выйдет из цикла for, программа больше не увидит этой переменной i.

Область видимости в пределах прототипа функции применяется к именам переменных, которые используются в прототипах функций, как, например, в следующем случае:

```
int mighty(int mouse, double large);
```

Область видимости в пределах прототипа функции простирается от точки, в которой объявлена переменная, до конца объявления прототипа. Это значит, что все, что интересует компилятор, когда он выполняет обработку аргумента прототипа функции — это его тип. Имена, если вы их указываете, обычно его не интересуют, они не обязательно должны совпадать с именами, которые используются при определении функции. Один из тех случаев, когда имена не имеют существенного значения — это параметры типа массивов переменной длины:

```
void use_a_VLA(int n, int m, ar[n][m]);
```

Если вы используете имена в круглых скобках, это должны быть имена, объявленные ранее в прототипе.

Переменная с определением, помещенным за пределами какой бы то ни было функции, имеет *область видимости в пределах файла*. Переменная с областью видимости в пределах файла видна на протяжении от точки ее определения до конца файла, содержащего это определение. Рассмотрим следующий пример:

```
#include <stdio.h>
int units = 0;   /* переменная с областью видимости в пределах файла */
void critic(void);
int main(void)
{

}
void critic(void)
{

}
```

В данном случае переменная units имеет область видимости в пределах файла, поэтому она может использоваться как в функции main(), так и в функции critic().

Поскольку переменные с областью видимости в пределах файла могут использоваться в более чем одной функции, они еще называются *глобальными переменными*.

Существует еще один тип области видимости, получивший название *области видимости в пределах функции*, но такие области видимости относятся только к меткам, используемым в операторах goto. Область видимости функции означает, что метка оператора в некоторой конкретной функции видима коду программы в любом месте функции, независимо от блока, в котором она появляется.

Связывание

Далее рассмотрим вопросы связывания. Переменная в языке C имеет одно из следующих связываний: внешнее связывание, внутреннее связывание или отсутствие связывания. Переменные с областью видимости в пределах блока или с областью видимостью в пределах прототипа связывания не имеют.

Это означает, что они замкнуты в тех блоках или прототипах, которых в которых они определены. Переменная с областью видимости в пределах файла могут иметь внутреннее или внешнее связывание. Переменная с внешним связыванием может использоваться в любом месте многофайловой программы, тогда как переменная с внутренним связыванием — в одиночном файле.

В таком случае, как определить, какое связывание имеет переменная с областью видимости в пределах файла — внутреннее или внешнее? Вы должны посмотреть, используется ли спецификатор класса памяти static во внешнем определении:

```
int giants = 5;         // Область видимости в пределах файла,
                        // внешнее связывание
static int dodgers = 3; // Область видимости в пределах файла,
                        // внутреннее связывание
int main()
{

}
...
```

Переменная giants может использоваться в других файлах, которые являются частями одной и той же программы. Переменная dodgers является приватной для этого конкретного файла, в то же время она может использоваться любой функцией в этом файле.

Продолжительность хранения

Переменная в языке C имеет одну из двух продолжительностей хранения: статическая продолжительность хранения и автоматическая продолжительность хранения. Если переменная имеет статическую продолжительность хранения, она существует на протяжении всего времени выполнения программы. Для переменных с областью видимости в пределах файла характерна статическая продолжительность хранения. Следует отметить, что для переменных с областью видимости в пределах файла ключевое слово static определяет тип связывания, но не продолжительность хранения.

Переменная с областью видимостью в пределах файла, объявленная с ключевым словом static, имеет внутреннее связывание, в то же время все переменные с областью видимости в пределах файла, имеющие внутреннее или внешнее связывание, имеют статическую продолжительность хранения.

Для переменных с областью видимости в пределах блока обычно характерна автоматическая продолжительность хранения. Для этих переменных память уже зарезервирована в тот момент, когда программа передает управление тому блоку, в котором они определены, эта память освобождается, когда управление покидает этот блок. Идея заключается в том, что память, выделяемая для автоматических переменных, представляет собой рабочее пространство или сверхоперативную память, которая используется многократно. Например, когда завершается вызов функции, память, которую она использовала для своих переменных, может быть использована для вызова следующей функции.

Локальные переменные, которые применялись до сих пор, попадают в категорию автоматических функций. Например, в приведенном ниже программном коде переменные number и index возникают всякий раз, когда выполняется вызов функции bore(), и исчезают всякий раз по завершении функции:

```
void bore(int number)
{
   int index;
   for (index = 0; index < number; index++)
      puts("Человеку свойственно ошибаться.\n");
   return 0;
}
```

В языке C такие характеристики, как область видимости, связывание и продолжительность хранения, используются для определения пяти классов памяти: автоматическая, регистровая, статическая с областью видимости в пределах блока, статическая с внешним связыванием и статическая с внутренним связыванием. В табл. 12.1 представлены все комбинации. Теперь, когда мы ознакомились такими понятиями, как область хранения, связывание и продолжительность хранения, можно перейти к более подробному обсуждению всех этих классов памяти.

Таблица 12.1. Пять классов памяти

Класс памяти	*Продолжительность*	*Область видимости*	*Связывание*	*Как определяется*
automatic	автоматическая	блок	нет	В блоке.
register	автоматическая	блок	нет	В блоке с ключевым словом register.
static с внешним связыванием	статическая	файл	внешнее	За пределами всех функций.
static с внутренним связыванием	статическая	файл	внутреннее	За пределами всех функций с ключевым словом static.
static без связывания	статическая	файл	нет	В блоке с ключевым словом static.

Автоматические переменные

Переменная, принадлежащая к классу автоматической памяти, имеет автоматическую продолжительность хранения, видимость в пределах блока и не имеет связывания. По умолчанию любая переменная, объявленная в блоке или в заголовке функции, относится к классу автоматической памяти. Однако вы можете конкретно сформулировать свои намерения, открыто воспользовавшись ключевым словом auto, как показано ниже:

```
int main(void)
{
   auto int plox;
```

Вы можете поступить так, например, чтобы документировать тот факт, что вы намеренно перекрываете определение внешней функции, или что важно не переводить переменную в другой класс памяти. Ключевое слово auto является *спецификатором класса памяти.*

Область видимости в пределах блока и отсутствие связывания говорит о том, что только в блоке, где определена переменная, может осуществляться доступ к этой переменной по имени. (Разумеется, аргументы могут использоваться для передачи значения этой переменной и адреса в другую функцию, но это косвенные сведения.) Другая функция может использовать переменную с тем же именем, но это будет независимая переменная, хранящаяся в другом месте памяти.

Вспомните, что продолжительность автоматического хранения означает, что переменная появляется в тот момент, когда программа входит в блок, который содержит объявление этой переменной. Как только программа выйдет из блока, автоматическая переменная перестает существовать. Ячейки памяти, которые она занимала, могут быть использованы для других целей.

Теперь подробно рассмотрим вложенные блоки. Переменная известна только в блоке, в котором она объявлена? и в любом другом блоке внутри данного блока:

```
int loop(int n)
{
   int m;              // m находится в области видимости
   scanf("%d", &m);
   {
      int i;           // переменные m и i находятся в области видимости
      for (i = m; i < n; i++)
         puts("i локальна в подблоке\n");
   }
   return m;           // m в области видимости, i больше не существует
}
```

В этом коде i видима только во внутренних скобках. Вы получите от компилятора сообщение об ошибке, если попытаетесь использовать эту переменную за пределами этого внутреннего блока. Как правило, это свойство не используется при разработке конкретной программы. Однако время от времени бывает полезно определять переменную в подблоке, если она нигде более не используется. Таким образом, вы должны документировать смысл переменной рядом с тем местом, в котором она применяется. Наряду с этим, переменная не остается неиспользованной и не занимает места, когда она больше не нужна.

Переменные n и m, будучи объявленными в заголовке функции или во внешнем блоке, попадают в область видимости всей функции и существуют до тех пор, пока не завершится выполнение этой функции.

Что произойдет, если вы объявите переменную во внутреннем блоке, который имеет то же имя, что и один из внешних блоков? Тогда имя, определенное внутри блока, является переменной, используемой внутри этого блока. Мы говорит, что она *скрывает* внешнее определение. В то же время, когда управление выходит за пределы внутреннего блока, внешняя переменная снова попадает в область видимости. Листинг 12.1 служит иллюстрацией этих положений, впрочем, и не только их.

Листинг 12.1. Программа `hiding.c`

```
/* hiding.c -- переменные в блоках */
#include <stdio.h>
int main()
{
   int x = 30;           /* исходное значение x */
   printf("x во внешнем блоке: %d\n", x);
   {
      int x = 77;        /* новая переменная x, скрывает первую x */
      printf("x во внутреннем блоке: %d\n", x);
   }
   printf("x во внешнем блоке: %d\n", x);
   while (x++ < 33)      /* исходное значение x */
   {
       int x = 100;      /* новая переменная x, скрывает первую x */
       x++;
       printf("x в цикле while: %d\n", x);
   }
   printf("x во внешнем блоке: %d\n", x);
   return 0;
}
```

Выходные данные этой программы имеют следующий вид:

```
x во внешнем блоке: 30
x во внутреннем блоке: 77
x во внешнем блоке: 30
x в цикле while: 101
x в цикле while: 101
x в цикле while: 101
x во внешнем блоке: 34
```

Во-первых, данная программа создает переменную x, имеющую значение 30, как показывает первый оператор `printf()`. Затем она определяет новую переменную x со значением 77, как показывает второй оператор `printf()`. Это новая переменная, скрывающая первую переменную x, что и демонстрирует третий оператор `printf()`. Он размещен непосредственно под первым внутренним блоком и отображает исходное значение x, показывая, что переменная x никуда не исчезала и не подвергалась изменениям.

Возможно, что частью программы, вызывающей наибольший интерес, является цикл while. В проверке конца цикла while используется исходное значение x:

```
while(x++ < 33)
```

В то же время, внутри цикла программа видит третью переменную x, ту, которая определена внутри блока цикла while. Таким образом, когда программный код использует операцию x++ в теле цикла, это новая переменная x, значение которой увеличивается до 101, после чего отображается на экране. По завершении каждой итерации цикла эта новая переменная x перестает существовать. Затем проверка условия завершения цикла использует и инкрементирует исходное значение x, управление снова передается в блок цикла, и снова создается новая переменная x. В рассматриваемом примере переменная x создается и уничтожается три раза. Обратите внимание, что для прекращения выполнения рассматриваемый цикл должен наращивать значение переменной x в проверяемом условии, так как увеличение значения x в теле цикла приводит к увеличению значения другой переменной x, а не той, которая используется при проверке конца цикла.

Назначение этого примера состоит не в том, чтобы побудить вас писать программный код, подобный рассматриваемой программе. Этот пример служит иллюстрацией того, что происходит, когда переменная определяется внутри блока.

Блоки без фигурных скобок

Согласно упомянутому выше положению стандарта C99, операторы, являющиеся частью цикла или оператора if, рассматриваются как блок, даже если фигурные скобки ({}) не используются. Если более подробно, то полный цикл есть подблок содержащего его блока, а тело цикла — подблок полного блока цикла. Аналогично, оператор if является блоком, а связанные с ним подоператоры — подблоками оператора if. Эти правила определяют, где вы можете объявить переменную, а также область видимости этой переменной. Листинг 12.2 демонстрирует, как все это работает применительно к циклу for.

Листинг 12.2. Программа forc99.c

```
//forc99.c -- правила нового стандарта C99, касающиеся блоков
#include <stdio.h>
int main()
{
   int n = 10;
   printf("Первоначально n = %d\n", n);
   for (int n = 1; n < 3; n++)
       printf("цикл 1: n = %d\n", n);
   printf("По завершении цикла 1 n = %d\n", n);
   for (int n = 1; n < 3; n++)
   {
      printf("индекс цикла 2 n = %d\n", n);
      int n = 30;
      printf("цикл 2: n = %d\n", n);
      n++;
   }
   printf("По завершении цикла 2 n = %d\n", n);
   return 0;
}
```

Ниже показаны выходные данные этой программы, при этом предполагается, что компилятор поддерживает стандарт C99:

```
Первоначально n = 10
цикл 1: n = 1
цикл 1: n = 2
По завершении цикла 1 n = 10
цикл 2 индекс n = 1
цикл 2: n = 30
цикл 2 индекс n = 2
цикл 2: n = 30
По завершении цикла 2 n = 10
```

Поддержка стандарта C99

Некоторые компиляторы могут не поддерживать новые правила стандарта C99, касающиеся правил области видимости. Другие компиляторы могут предоставлять возможность активизации таких правил. Например, на время написания этой книги компилятор `gcc` поддерживал по умолчанию многие средства, предлагаемые стандартом C99, но при этом требовал использования варианта `-std=c99` для их активизации:

```
gcc -std=c99 forc99.c
```

Переменная `n`, объявленная в управляющем разделе первого цикла `for`, находится в области видимости до конца цикла и при этом скрывает исходную переменную `n`. Но после того как управление покидает цикл, исходная переменная `n` возвращается в область видимости.

Во втором цикле `for` переменная `n`, объявленная как индекс цикла, скрывает исходную переменную `n`. Затем `n`, объявленная внутри тела цикла, скрывает индекс цикла `n`. Как только программа прекратит выполнение тела цикла, переменная `n`, объявленная в теле цикла, исчезает, и проверка на окончание цикла использует индекс `n`. Когда завершится выполнения всего цикла, в области видимости появляется исходная переменная `n`.

Инициализация автоматических переменных

Автоматические переменные не инициализируются до тех пор, пока вы не сделаете это явно. Рассмотрим следующие объявления:

```
int main(void)
{
   int repid;
   int tents = 5;
```

Переменная `tents` инициализируется значением `5`, но переменная `repid` получает значение, которое раньше находилось в области памяти, выделенной для этой переменной. Никак нельзя рассчитывать на то, что этим значением будет `0`. Вы можете инициализировать автоматическую переменную неконстантным выражением при условии, что все используемые при этом переменные были объявлены раньше:

```
int main(void)
{
   int ruth = 1;
   int rance = 5 * ruth;    // используется ранее объявленная переменная
```

Регистровые переменные

Обычно переменные хранятся в памяти компьютера. При благоприятном стечении обстоятельств регистровые переменные хранятся в регистрах центрального процессора, или, в общем случае, в наиболее быстродействующей памяти, благодаря чему доступ к ним и операции над ними выполняются намного быстрее, чем в случае обычных переменных. Поскольку регистровые переменные могут располагаться в регистрах, а не в памяти, вы не можете получить адрес регистровой переменной. В большинстве других аспектов регистровые переменные ничем не отличаются от автоматических переменных. Иначе говоря, они имеют область видимости в пределах блока, не имеет связывания, и имеют автоматическую продолжительность хранения. Переменная объявляется путем использования спецификатора класса памяти `register`:

```
int main(void)
{
   register int quick;
```

Мы говорим "при благоприятном стечении обстоятельств", поскольку объявление переменной как регистрового класса скорее просьба, чем прямое указание. Компилятор должен удовлетворить ваши требования при наличии необходимого количества регистров или объема доступной быстродействующей памяти, иначе ваше пожелание не будет выполнено. В этом случае переменная становится обыкновенной автоматической переменной; тем не менее, вы все еще не можете использовать вместе с ней адресную операцию. Вы можете потребовать, чтобы формальные параметры были регистровыми переменными. Для этого достаточно воспользоваться ключевым словом `register` в заголовке функции:

```
void macho(register int n)
```

Типы, которые могут быть объявлены как `register`, могут оказаться ограниченными. Например, регистры в процессоре могут быть недостаточно большими, чтобы уместить тип `double`.

Статические переменные с областью видимости в пределах блока

Термин *статическая переменная* звучит как противоречие, как переменная, которая не подлежит изменениям. На самом деле характеристика *статическая* означает, что переменная остается неизменной. Переменные с областью видимостью в пределах файла автоматически (и в обязательном порядке) получают статическую продолжительность хранения. Существует также возможность создавать локальные переменные, то есть переменные, имеющие область видимости в пределах блока, которая характерна для статистической памяти. Такие переменные имеют такую же область видимости, что и автоматические переменные, однако они не исчезают после того, как содержащая их функция завершает свою работу. Иначе говоря, такие переменные имеют область видимости в пределах блока, не имеют связывания, зато имеют статическую продолжительность хранения. Компьютер помнит их значения от одного вызова функции до следующего. Такие переменные создаются через объявление их в блоке (при этом они получают область видимости в пределах блока без связывания) со спецификатором класса памяти `static` (который обеспечивает статическую продол-

жительность хранения). Пример, представленный в листинге 12.3, служит иллюстрацией этой методологии.

Листинг 12.3. Программа `loc_stat.c`

```
/* loc_stat.c -- использование локальных статических переменных */
#include <stdio.h>
void trystat(void);
int main(void)
{
   int count;
   for (count = 1; count <= 3; count++)
   {
       printf("Начинается итерация %d:\n", count);
       trystat();
   }
   return 0;
}
void trystat(void)
{
   int fade = 1;
   static int stay = 1;
   printf("fade = %d и stay = %d\n", fade++, stay++);
}
```

Обратите внимание, что `trystat()` инкрементирует каждую переменную после вывода ее значения. В результате выполнения этой программы получены следующие результаты:

```
Начинается итерация 1:
fade = 1 и stay = 1
Начинается итерация 2:
fade = 1 и stay = 2
Начинается итерация 3:
fade = 1 и stay = 3
```

Статическая переменная `stay` напоминает, что ее значение было увеличено на 1, в то время как переменная `fade` создается заново каждый раз. Отсюда следует, что инициализация этих значений производится по-разному: переменная `fade` инициализируется каждый раз, когда вызывается функция `trystat()`, а переменная `stay` — только один раз, во время компиляции функции `trystat()`. Статические переменные инициализируются нулем, если вы явно не инициализируете ее каким-то другим значением.

Два следующих объявления выглядят похожими:

```
int fade = 1;
static int stay = 1;
```

В то же время первый оператор фактически является частью функции `trystat()` и выполняется всякий раз, когда вызывается эта функция. Это действие выполняется во время выполнения программы. Второй оператор не является частью функции `trystat()`. Если вы используете отладчик для пошагового выполнения программы, то увидите, что программа как бы пропускает этот шаг. Это объясняется тем, что статические переменные и внешние переменные уже находятся в нужном месте сразу же после того, как программа будет загружена в память. Тот факт, что оператор помещен

в функцию trystat(), говорит о том, что только функции trystat() предоставлена возможность видеть эту переменную; это не тот оператор, что выполняется во время работы программы.

Вы не можете использовать модификатор static применительно к параметрам функции:

```
int wontwork(static int flu);     // не разрешено
```

Если вы читали раннюю литературу, посвященную языку программирования C, вы, скорее всего, обратили внимание, что этот класс памяти тогда назывался *внутренним статическим классом памяти*. Однако слово "внутренний" использовалось по отношению к функции, но не для обозначения внутреннего связывания.

Статические переменные с внешним связыванием

Статическая переменная с внешним связыванием имеет область видимости в пределах файла, внешнее связывание и статическую продолжительность хранения. Этот класс иногда называют *классом внешней памяти*, а переменные этого типа — *внешними переменными*. Вы создаете внешнюю переменную, помещая определяющее объявление вне пределов какой-либо функции. В соответствие с документацией, внешняя переменная может быть дополнительно объявлена внутри функции, которая использует ее, за счет указания ключевого слова extern. Если переменная определена в *другом* файле, объявление такой переменной с ключевым словом extern является обязательным. Объявления имеют следующий вид:

```
int Errupt;                /* внешне объявленная переменная */
double Up[100];            /* внешне объявленный массив */
extern char Coal;          /* обязательное объявление */
                           /* Coal объявлена в другом файле */
void next(void);
int main(void)
{
   extern int Errupt;      /* необязательное объявление */
   extern double Up[];     /* необязательное объявление */
   ...
}
void next(void)
{

}
```

Два указанных выше объявления переменной Errupt представляют собой примеры связывания, поскольку оба они ссылаются на одну и ту же переменную. Внешние переменные имеют внешнее связывание, и к этой детали мы снова обратимся несколько позже.

Обращаем ваше внимание на то, что вы не обязаны задавать размерность массива в необязательном объявлении double Up. Это объясняется тем, что исходное объявление уже сообщило эту информацию. Группу внешних объявлений в функции main() можно опустить полностью, поскольку внешние объявления имеют область видимости в пределах файла, следовательно, они известны от точки объявления до конца файла.

Они служат для подтверждения вашего намерения использовать эти переменные в функции main().

Если ключевое слово extern опущено в объявлении переменной внутри той или иной функции, создается отдельная автоматическая переменная. То есть, замена

```
extern int Errupt;
```

на

```
int Errupt;
```

в main() приводит к тому, что компилятор создает автоматическую переменную с именем Errupt. Это будет отдельная локальная переменная, отличная от исходной переменной Errupt. Эта локальная переменная будет находиться в области видимости в то время, когда программа выполняет функцию main(), однако внешняя переменная Errupt будет находиться в области видимости других функций, таких как next(), расположенных в одном с ней файле. Короче говоря, переменная в области видимости в пределах блока "скрывает" переменную с тем же именем, но с областью видимости в пределах файла, когда программа выполняет операторы в соответствующем блоке.

Внешняя переменная имеет статическую продолжительность хранения. В силу этого массив Up существует и сохраняет свои значения, независимо от того, что выполняет программа — функцию main(), next() или какую-то другую функцию.

В следующих трех примерах показаны четыре возможных комбинации внешних и автоматических переменных. Пример 1 содержит одну внешнюю переменную: Hocus. Она известна как функции main(), так и функции magic().

```
/* Пример 1 */
int Hocus;
int magic();
int main(void)
{
   extern int Hocus;      // Hocus объявлена как внешняя переменная
   ...
}
int magic()
{
   extern int Hocus;      // та же переменная Hocus, что и выше
   ...
}
```

В примере 2 используется одна внешняя переменная Hocus, известная обеим функциям. На этот раз функция magic() знает ее по умолчанию.

```
/* Пример 2 */
int Hocus;
int magic();
int main(void)
{
   extern int Hocus;       // Переменная Hocus объявлена как внешняя
   ...
}
int magic()
{
                           // Переменная Hocus не объявлена, но известна

}
```

В примере 3 создаются четыре переменных. Переменная Hocus в main() — это автоматическая переменная по умолчанию и локальная для main. Переменная Hocus в функции magic() объявлена явно как автоматическая и известна только функции magic(). Внешняя переменная Hocus не известна функции main() или magic(), но будет известна любой другой функции в этом файле, в которой нет свой собственной локальной переменной Hocus. И, наконец, Pocus представляет собой внешнюю переменную, известную функции magic(), но не main(), поскольку Pocus идет вслед за main().

```
/* Пример 3 */
int Hocus;
int magic();
int main(void)
{
   int Hocus;      // переменная Hocus объявлена как auto по молчанию
   ...
}
int Pocus;
int magic()
{
   auto int Hocus; // локальная переменная Hocus объявлена как автоматическая
   ...
}
```

Приведенные примеры иллюстрируют то, что область видимости внешних переменных простирается от точки их объявления до конца файла. Кроме того, примеры иллюстрируют время жизни переменных. Внешние переменные Hocus и Pocus сохраняются в течение всего времени выполнения программы, и, поскольку они не заключены ни в какую другую функцию, они не исчезают после завершения функции.

Инициализация внешних переменных

Подобно автоматическим, внешние переменные могут инициализироваться явно. В отличие от автоматических переменных, внешние переменные автоматически инициализируются нулем, не будучи инициализированными явно. Это правило применяется также к элементам массива, определенного как внешний массив. В отличие от случая автоматических переменных, вы можете использовать только константные выражения для инициализации переменных, имеющих область видимости в пределах файла.

```
int x = 10;                // правильно, 10 есть константа
int y = 3 + 20;            // правильно, константное выражение
size_t z = sizeof(int);    // правильно, константное выражение
int x2 = 2 * x;            // неправильно, x является переменной
```

(До тех пор, пока тип не является переменным массивом, выражение sizeof рассматривается как константное.)

Использование внешних переменных

Рассмотрим простой пример, в котором используется внешняя переменная. В частности, предположим, что вы хотите иметь две функции с именами main() и critic() для доступа к переменным элементам. Вы можете сделать это, объявив элементы вне упомянутых выше двух функций, как показано в листинге 12.4.

Листинг 12.4. Программа `global.c`

```
/* global.c -- использование внешней переменной */
#include <stdio.h>
int units = 0;                  /* внешняя переменная */
void critic(void);
int main(void)
{
   extern int units;            /* необязательное повторное объявление */
   printf("Сколько фунтов весит маленький бочонок меда?\n");
   scanf("%d", &units);
   while ( units != 56)
       critic();
   printf("Вы должны это проверить!\n");
   return 0;
}
void critic(void)
{
   /* необязательное повторное объявление опускается */
   printf("Вам не повезло. Попытайтесь еще раз.\n");
   scanf("%d", &units);
}
```

Ниже показаны выходные данные этой учебной программы:

```
Сколько фунтов весит маленький бочонок меда?
14
Вам не повезло. Попытайтесь еще раз.
56
Вы должны это проверить!
```

(Мы это сделали.)

Обратите внимание на то, как второе значение переменной `units` считывается функцией `critic()`, однако функция `main()` знала новое значение, когда она завершала цикл `while`. Таким образом, как функция `main()`, так и функция `critic()` используют идентификатор `units` для доступа к одной и той же переменной. Пользуясь терминологией языка C, мы говорим, что переменная `units` имеет область видимости в пределах файла, внешнее связывание и статическую продолжительность хранения.

Мы сделали `units` внешней переменной, определив ее вне пределов (то есть, сделав ее внешней по отношению) определения любой функции. То есть, все, что вам остается сделать — это обеспечить, чтобы переменная `units` стала доступной для всех последующих функций данного файла.

Рассмотрим некоторые подробности. Прежде всего, объявление переменной `units` там, где она была объявлена, делает ее доступной для функций, объявленных ниже ее без каких-либо дополнительных усилий. Благодаря этому обстоятельству, функция `critics()` использует переменную `units`.

Аналогично, ничего не надо делать, чтобы обеспечить доступ функции `main()` к переменной `units`. Тем не менее, функция `main()` содержит в себе следующее объявление:

```
extern int units;
```

В рассматриваемом примере это объявление есть не что иное, как еще одна форма документирования. Спецификатор класса памяти `extern` сообщает компилятору, что любое упоминание о переменной `units` в конкретной функции относится к переменной, объявленной вне пределов этой функции, возможно, даже за пределами файла. И снова обе функции `main()` и `critic()` используют переменную `units`, определенную вне их пределов.

Внешние имена

В соответствие с требованиями стандарта C99, компиляторы должны отводить первые 63 символа под локальные идентификаторы и первые 31 символ под внешние идентификаторы. Это вступает в противоречие с требованиями предыдущих стандартов, которые отводили первые 31 символ под локальные идентификаторы и первые шесть символов под внешние идентификаторы. Поскольку стандарт C99 относительно новый, вполне возможно, что вы работаете по старым правилам. Причина того, что правила в отношении имен внешних переменных являются более жесткими, чем правила, предъявляемые к локальным переменным, связана с тем, что внешние имена должны быть совместимыми с правилами, действующими в локальной среде, которые вполне могут оказаться более требовательными.

Определения и объявления

Теперь обратим более пристальное внимание на различие между определением переменной и ее объявлением. Рассмотрим следующий пример:

```
int tern = 1;         /* определение tern */
main()
{
   external int tern; /* используется переменная tern,
                         определенная в другом месте */
```

В рассматриваемом случае переменная `tern` объявлена дважды. Первое объявление приводит к тому, что для этой переменной в памяти отводится место. Это объявление представляет собой определение данной переменной. Второе объявление просто сообщает компилятору, что нужно использовать переменную `tern`, которая была создана раньше, поэтому оно не является определением этой переменной. Первое определение называется *определяющим объявлением*, а второе — *ссылочным объявлением*. Ключевое слово `extern` показывает, что это объявление не есть определение, поскольку оно отсылает компилятор за нужной информацией в другое место.

Предположим, что имеются следующие операторы:

```
extern int tern;
int main(void)
{
```

Компилятор полагает, что фактическое определение переменной `tern` дано где-то в другом месте программы, возможно, даже в другом файле. Это объявление не приводит к выделению пространства памяти. Следовательно, не нужно использовать ключевое слово `extern` для создания внешнего определения, достаточно всего лишь *сослаться* на существующее внешнее определение.

Внешняя переменная может быть инициализирована только один раз, и это должно осуществляться во время определения переменной. Оператор наподобие

```
extern char permis = 'Y';        /* ошибка */
```

является ошибочным, поскольку ключевое слово `extern` указывает ссылочное, а не определяющее объявление.

Статическая переменная с внешним связыванием

Переменные этого класса памяти имеют статическую продолжительность хранения, область видимости в пределах файла и внутреннее связывание. Вы создаете одну из таких переменных, определяя ее вне любой из функций (так же, как и в случае внешней переменной) со спецификатором `static` класса памяти:

```
static int svil = 1;     // статическая переменная, внутреннее связывание
int main(void)
{
```

Такие переменные когда-то получили название *внешних статистических* переменных, но это название несколько сбивает с толку, поскольку для этих переменных характерно внутреннее связывание. К сожалению, новый термин найти не удалось, следовательно, нам остается применять термин *статическая переменная с внутренним связыванием*. Обычная внешняя переменная может использоваться в функции любого файла, который является частью рассматриваемой программы, в то время как статическая переменная с внутренним связыванием может использоваться только в функциях одного и того же файла. Вы можете переопределить область видимости файла внутри функции, используя для этого спецификатор `extern` класса памяти. Такое объявление не изменяет связывание. Рассмотрим следующий код:

```
int traveler = 1;          // внешнее связывание
static int stayhome = 1;   // внутреннее связывание
int main()
{
   extern int traveler;    // использовать глобальную переменную traveler
   extern int stayhome;    // использовать глобальную переменную stayhome
   ...
```

Обе переменных `traveler` и `stayhome` являются глобальными в этой части файла, но только `traveler` может использоваться кодом в других файлах. Два приведенных выше объявления с ключевым словом `extern` документируют тот факт, что `main()` использует две глобальные переменные, в то время как переменная `stayhome` продолжает иметь внутреннее связывание.

Множественные файлы

Различие между внутренним и внешним связыванием проявляется только в тех случаях, когда ваша программа собрана из нескольких файлов. В сложных программах на языке C часто используются несколько отдельных файлов кода. Иногда возникает потребность совместного использования внешней переменной. Способ, который предлагает стандарт ANSI C для достижения этой цели, предусматривает определительное объявление в одном файле и ссылочные объявления в остальных. Иначе го-

воря, все объявления за исключением одного (определительное объявление) должны использовать ключевое слово extern, и только определительное объявление может использоваться для инициализации этой переменной.

Следует обратить внимание на то, что внешняя переменная, определенная в одном файле, недоступна во втором до тех пор, пока она не будет объявлена во втором файле (с использованием ключевого слова extern). Внешнее определение само по себе лишь обеспечивает возможность доступа к конкретной переменной из других файлов.

В то же время исторически сложилось так, что многие компиляторы следуют в этом отношении другим правилам. Многие системы Unix, например, предоставляют возможность объявлять переменные в нескольких файлах без использования ключевого слова extern при условии, что только одно объявление предусматривает инициализацию. Если встречается объявление без инициализации, оно рассматривается компилятором как определение.

Спецификаторы классов памяти

Вы, должно быть, уже заметили, что значение ключевых слов static и extern зависит от контекста. Язык C имеет пять ключевых слов, которые выступают в различных сочетаниях как спецификаторы классов памяти. Таковыми являются auto, register, static, extern и typedef. Ключевое слово typedef ничего не говорит о хранении в памяти, тем не менее, оно включено в список для синтаксических целей. В частности, вы можете использовать в объявлении всего лишь один спецификатор класса памяти, следовательно, вы не можете задавать еще один спецификатор класса памяти как часть typedef.

Спецификатор auto указывает на то, что используется переменная с автоматической продолжительностью хранения. Он может применяться только в объявлениях переменных с областью видимости в пределах блока, которая также имеет автоматическую продолжительность хранения, следовательно, в основном auto используется в целях документирования.

Спецификатор register также может быть использован только с переменными с областью видимостью в пределах блока. Он помещает переменную в регистровый класс памяти, что равносильно требованию размещения этой переменной в регистре для более быстрого доступа к ней. Он также лишает вас возможности работы с адресом этой переменной.

Спецификатор static, когда присутствует в объявлении переменной с областью видимости в пределах блока, наделяет эту переменную статической продолжительностью хранения, благодаря чему она существует и сохраняет свои значения на всем периоде выполнения программы, даже в тех случаях, когда содержащий ее блок не используется. Эта переменная сохраняет область видимости в пределах блока и не имеет связывания. Когда static используется в объявлениях с областью видимости в пределах файла, он указывает на то, что переменная имеет внутреннее связывание.

Спецификатор extern показывает, что вы объявляете переменную, которая была определена в каком-то другом месте. Если объявление, содержащее ключевое слово extern, имеет область видимости в пределах файла, переменная, на которую производится ссылка, должна иметь внешнее связывание. Если объявление, содержащее ключевое слово extern, имеет область видимости в пределах блока, переменная, на кото-

рую производится ссылка, может иметь либо внешнее связывание, либо внутреннее связывание, в зависимости от определительного объявления этой переменной.

Сводка: классы памяти

Автоматические переменные имеют область видимости в пределах блока, не имеют связывания и характеризуются автоматической продолжительностью хранения. Эти переменные локальны и приватны для блока (обычно таким блоком является функция), в котором они определены. Регистровые переменные обладают теми же свойствами автоматических переменных, однако компилятор может использовать более быструю память или регистр для их хранения. Вы не можете получить адрес регистровой переменной.

Переменные со статической продолжительностью хранения могут иметь внешнее связывание, внутреннее связывание или вообще не иметь связывания. Когда переменная объявляется внешней по отношению к любой функции файла, то она представляет собой внешнюю переменную и имеет область видимости в пределах файла, внешнее связывание и статическую продолжительность хранения. Если вы добавите к этому объявлению ключевое слово `static`, то вы получите переменную со статической продолжительностью хранения, с областью видимости в пределах файла и с внутренним связыванием. Если вы объявляете переменную внутри функции и используете ключевое слово `static`, то данная переменная имеет статическую продолжительность хранения, область видимости в пределах блока и не имеет связывания.

Память для переменной с автоматической продолжительностью хранения выделяется, когда поток управления войдет в блок, содержащий объявление переменной, и освобождается сразу после того, как поток управления выйдет из этого блока. Будучи неинициализированной, такая переменная имеет случайное значение. Память под переменную со статической продолжительностью хранения выделяется во время компиляции и сохраняется за ней до тех пор, пока выполняется программа. В процессе инициализации такая переменная получает значение `0`.

Переменная с областью видимости в пределах блока представляет собой локальную переменную по отношению к блоку, в котором содержится ее объявление. Переменная, имеющая область видимости в пределах файла, известна всем функциям файла, которые следуют за ее объявлением. Если переменная с областью видимости в пределах файла имеет внешнее связывание, она может использоваться в других файлах программы. Если переменная с областью видимости в пределах файла имеет внутреннее связывание, она может использоваться только в том файле, в котором она объявлена.

Рассмотрим короткую программу, которая использует все пять классов памяти. Программа состоит из двух файлов (листинг 12.5 и листинг 12.6), таким образом, вы должны выполнить многофайловую компиляцию. (См. главу 9.) Ее главная цель заключается в использовании всех пяти типов памяти, а не в том, чтобы предложить проектную модель; в хорошем проекте переменные с областью видимости в пределах файла не употребляются.

Листинг 12.5. Программа `parta.c`

```
// parta.c -- различные классы памяти
#include <stdio.h>
void report_count();
void accumulate(int k);
int count = 0;    // область видимости в пределах файла, внешнее связывание

int main(void)
{
```

```
   int value;          // автоматическая переменная
   register int i;     // регистровая переменная
   printf("Введите положительное целое число (0 для выхода из программы): ");
   while (scanf("%d", &value) == 1 && value > 0)
   {
      ++count; //использование переменной с областью видимости в пределах файла
      for (i = value; i >= 0; i--)
         accumulate(i);
      printf("Введите положительное целое число (0 для выхода из программы): ");
   }
   report_count();
   return 0;
}
void report_count()
{
   printf("Цикл выполнен %d раз\n", count);
}
```

Листинг 12.6. Программа `partb.c`

```
// partb.c -- вторая часть программы
#include <stdio.h>
extern int count;          // ссылочное определение, внешнее связывание
static int total = 0;      // статическое определение, внутреннее связывание
void accumulate(int k);    // прототип
void accumulate(int k)     // переменная k имеет область видимости в
                           // пределах блока, связывание отсутствует
{
   static int subtotal = 0; // статическая переменная, связывание отсутствует
   if (k <= 0)
   {
      printf("итерация цикла: %d\n", count);
      printf("промежуточная сумма: %d; итого: %d\n", subtotal, total);
      subtotal = 0;
   }
   else
   {
      subtotal += k;
      total += k;
   }
}
```

В этой программе используются статическая переменная `subtotal` с областью видимости в пределах блока, которая подсчитывает текущую сумму значений, передаваемых в функцию `accumulate()`, и переменная `total`, в которой фиксируется промежуточная сумма. Функция `accumulate()` сообщает значения `total` и `subtotal` всякий раз, когда ей передается неположительное значение; в тех случаях, когда функция выдает сообщение, она сбрасывает значение `subtotal` в 0.

Прототип функции accumulate() в программе parta.c обязателен, поскольку этот файл содержит вызов функции accumulate(). Что касается программы partb.c, то прототип не обязателен, поскольку эта функция определена, но не вызывается в этом файле. Эта функция использует внешнюю переменную count, чтобы отслеживать, сколько раз был выполнен цикл while в функции main(). (Между прочим, это хороший пример того, как не надо использовать внешнюю переменную, поскольку нет необходимости перемешивать код файла parta.c с кодом файла partb.c.) В parta.c функции main() и report_count() совместно осуществляют доступ к переменной count.

Ниже показаны выходные данные этой учебной программы:

```
Введите положительное целое число (0 для выхода из программы): 5
итерация цикла: 1
промежуточная сумма: 15; итого: 15
Введите положительное целое число (0 для выхода из программы): 10
итерация цикла: 2
промежуточная сумма: 55; итого: 70
Введите положительное целое число (0 для выхода из программы): 2
итерация цикла: 3
промежуточная сумма: 3; итого: 73
Введите положительное целое число (0 для выхода из программы): 0
Цикл выполнен 3 раза
```

Классы памяти и функции

Функции также имеют свои классы памяти. Функция может быть либо внешней (по умолчанию), либо статической. (Стандарт C99 добавляет третью возможность, а именно — встраиваемую, или подставляемую, функцию, которая подробно рассматривается в главе 16.) К внешней функции доступ могут осуществлять функции из других файлов, в то же время статическая функция может использоваться только в файле, в котором она определена. Рассмотрим, например, файл со следующими объявлениями функций:

```
double gamma();             /* внешняя функция по умолчанию */
static double beta();
extern double delta();
```

Функции gamma() и delta() могут использоваться функциями из других файлов, которые являются частью программы, тогда как beta() — нет. В силу этого применение функции beta() ограничено одним файлом, и в других файлах можно использовать функции с тем же именем. Одна из причин использования класса статической памяти заключается в необходимости создания функций, приватных для конкретных модулей, благодаря чему во многих случаях удается избежать конфликтов имен.

Обычная практика состоит в том, что при объявлении функции, определенной в другом файле, указывается ключевое слово extern. При этом просто достигается большая ясность, поскольку при объявлении функция и так предполагается как extern, если только не задано ключевое слово static.

Какой класс памяти следует выбрать?

На вопрос о том, какой выбран класс памяти, чаще всего следует ответ — автоматический. В свете сказанного выше, почему в качестве значения по умолчанию выбирается автоматический класс памяти? Да, известно, что на первый взгляд внешняя память довольно привлекательна. Достаточно сделать все свои переменные внешними, и у вас не будет никаких проблем с использованием аргументов и указателей при обмене данными между функциями. Однако в этом случае существует всерьез замаскированная ловушка. Вы можете незаметно изменить поведение функции `A()`, меняя значения переменных, используемых в функции `B()`, хотя это совсем не входит в ваши планы. Как показывает опыт работы множества программистов на протяжении многих лет, возникающие при этом трудно обнаруживаемые ошибки намного опаснее, чем те призрачные преимущества, которые несет с собой беспорядочное использование внешней памяти.

Одним из золотых правил надежного программирования есть принцип "необходимости знать", или принцип минимально необходимой области видимости. Держите всю внутреннюю работу каждой функции максимально закрытой по отношению к другим функциям, используя совместно только те переменные, без которых нельзя обойтись по логике программы. Другие классы памяти полезны, и вы можете ими пользоваться. Однако всякий раз задавайте себе вопрос: а есть ли в этом необходимость?

Функция генерации случайных чисел и статическая переменная

Теперь, когда вы получили необходимый минимум знаний о различных классах памяти, рассмотрим несколько завершенных программ, в которых используются некоторые из рассмотренных классов. Во-первых, рассмотрим функцию, которая использует статическую переменную с внутренним связыванием: функцию, генерирующую случайные числа.

В библиотеке ANSI C имеется функция `rand()`, которая генерирует случайные числа. Существует множество алгоритмов генерации последовательности случайных чисел, при этом стандарт ANSI C позволяет любой реализации выбрать алгоритм, максимально учитывающий особенности конкретной машины. В то же время, стандарт ANSI C предлагает переносимый алгоритм, который генерирует такую же последовательность случайных чисел в разных системах. Фактически, функция `rand()` представляет собой "генератор псевдослучайных чисел"; из названия следует, что фактическая последовательность таких чисел предсказуема (для компьютеров не характерны самопроизвольные действия), однако числа достаточно равномерно распределены в диапазоне возможных значений.

Вместо того чтобы использовать встроенную функцию `rand()` компилятора, воспользуемся переносимой версией ANSI с тем, чтобы вы могли видеть, какие процессы происходят внутри. Такая схема начинает работу с так называемого "начального числа". Генератор случайных чисел использует начальное число для получения нового числа, которое, в свою очередь, становится новым начальным числом. Новое начальное число применяется для получения следующего нового начального числа и так далее. Чтобы эта схема работала, функция генерации случайных чисел должна помнить

начальное число, которое она использовала, когда была вызвана последний раз. Вот тут и возникает потребность в статической переменной. В листинге 12.7 представлена версия 0 (версия 1 ожидается очень скоро.)

Листинг 12.7. Функция `rand0.c`

```
/* rand0.c -- генерация случайных чисел */
/* используется переносимый алгоритм ANSI C */
static unsigned long int next = 1;         /* начальное число */
int rand0(void)
{
   /* магическая формула генерации псевдослучайных чисел */
   next = next * 1103515245 + 12345;
   return (unsigned int) (next/65536) % 32768;
}
```

Согласно листингу 12.7, статическая переменная `next` сначала принимает значение 1, которое затем изменяется с помощью магической формулы всякий раз, когда эта функция вызывается. Результатом будет некоторое возвращаемое значение в пределах от 0 до 32767. Следует отметить, что `next` является статической переменной с внутренним связыванием, а не просто статической переменной без связывания. Это объясняется тем, что рассматриваемый пример позже будет расширен до версии, в которой переменная `next` совместно используется двумя функциями, расположенными в одном и том же файле.

Протестируем функцию `rand0()` с помощью простого драйвера, показанного в листинге 12.8.

Листинг 12.8. Драйвер `r_drive0.c`

```
/* r_drive0.c -- тестирование функции rand0() */
/* компилируется с файлом rand0.c */
#include <stdio.h>
extern int rand0(void);
int main(void)
{
   int count;
   for (count = 0; count < 5; count++)
      printf("%hd\n", rand0());
   return 0;
}
```

Это еще одна возможность попрактиковаться в использовании множества файлов. Сохраните код из листинга 12.7 в одном файле, а код из листинга 12.8 — в другом. Ключевое слово `extern` напоминает, что функция `rand0()` определена в отдельном файле.

Выходные данные программы имеют вид:

```
16838
5758
10113
17515
31051
```

Возникает впечатление, что выходные данные носят случайный характер, но давайте выполним программу еще раз. На этот раз получаем следующие результаты:

```
16838
5758
10113
17515
31051
```

Знакомая картина? В этом-то и состоит аспект "псевдо". Каждый раз, когда выполняется главная программа, вы начинаете с одного и того же начального числа, равного 1. Вы можете избежать этой проблемы, если введете следующую функцию с именем `srand1()`, которая позволяет поменять значение начального числа. Проблема заключается в том, чтобы сделать `next` статической переменной с внутренним связыванием, известной только функциям `rand1()` и `srand1()`. (Функция, эквивалентная `srand1()` в библиотеке C, называется `srand()`.) Включите функцию `srand1()` в файл, который содержит функцию `rand1()`. В листинге 12.9 представлена модифицированная версия программы.

Листинг 12.9. Программа `s_and_r.c`

```
/* s_and_r.c -- файл функций rand1() и srand1() */
/* использует переносимый алгоритм ANSI C */
static unsigned long int next = 1;         /* начальное число */
int rand1(void)
{
   /* магическая формула для генерации псевдослучайных чисел */
   next = next * 1103515245 + 12345;
   return (unsigned int) (next/65536) % 32768;
}
void srand1(unsigned int seed)
{
   next = seed;
}
```

Обратите внимание, что `next` является статической переменной с областью видимости в пределах файла и внутренним связыванием. Это означает, что она может использоваться как функцией `rand1()`, так и функцией `srand1()`, но не функциями из других файлов. Чтобы протестировать эти функции, воспользуйтесь драйвером, показанным в листинге 12.10.

Листинг 12.10. Драйвер `r_drive1.c`

```
/* r_drive1.c -- тестирование функций rand1() и srand1() */
/* компилируется с файлом s_and_r.c */
#include <stdio.h>
extern void srand1(unsigned int x);
extern int rand1(void);
int main(void)
{
   int count;
   unsigned seed;
```

```
    printf("Введите начальное число.\n");
    while (scanf("%u", &seed) == 1)
    {
        srand1(seed);                  /* переустановить начальное число */
        for (count = 0; count < 5; count++)
           printf("%hd\n", rand1());
        printf("Введите следующее начальное число (или q для завершения):\n");
    }
    printf("Программа завершена.\n");
    return 0;
}
```

Ниже показан пример выполнения программы.

```
Введите начальное число.
1
16838
5758
10113
17515
31051
Введите следующее начальное число (или q для завершения)
513
20067
23475
8955
20841
15324
Введите следующее начальное число (или q для завершения)
q
Программа завершена.
```

Использование значения 1 в качестве начального числа дает тот же результат, что и раньше, в то время как значение 3 начального числа обеспечивает новый результат.

Автоматическая переустановка начального значения

Если ваша реализация языка C предоставляет доступ к некоторому изменяющемуся числовому значению, такому как, например, показания системных часов, вы можете использовать его (возможно, с усечением) для инициализации начального числа. Например, в версии ANSI C имеется функция `time()`, которая возвращает значение системного времени. Единицы изменения времени зависят от системы, однако в этом случае важно то, что возвращаемое значение имеет арифметический тип, и то, что оно меняется во времени. Точный тип зависит от конструкции системы, этот тип получил метку `time_t`, но вы можете воспользоваться приведением типа. Вот как выглядит базовая установка:

```
#include <time.h>                /* прототип ANSI для функции time()*/
srand1((unsigned int) time(0)); /* инициализация начального числа */
```

В общем случае функция `time()` принимает аргумент, в роли которого выступает адрес объекта типа `time_t`. В этом случае значение времени также сохраняется по этому адресу. Можно передать нулевой указатель (`0`) как аргумент, и в этом случае значение будет передаваться только с помощью механизма возврата из функции.

Вы можете использовать тот же метод применительно к функциям `srand()` и `rand()` стандарта ANSI C. Если вы намерены обращаться к этим функциям, включите в программу заголовочный файл `stdlib.h`. Теперь, когда было показано, как функции `srand1()` и `rand1()` используют статическую переменную с внутренним связыванием, можно воспользоваться их версиями, предлагаемыми компилятором. Это будет сделано в следующем примере.

Игра в кости

Давайте смоделируем широко известную игру со случайным характером — игру в кости. В ее наиболее распространенной форме используются две шестигранных кости, тем не менее, существуют и другие разновидности игры. Во многих азартных играх используются все пять геометрически возможных костей — с 4, 6, 8, 12 и 20 гранями. Умные древние греки доказали, что только у пяти правильных геометрических фигур все грани имеют одинаковую форму и размеры, и эти фигуры послужили базой для всего разнообразия костей. Можно выбрать кости с другим количеством граней, однако грани будут отличаться друг от друга, поэтому вероятность их выпадения будет разной.

В то же время эти геометрические выкладки никак не ограничивают вычисления, выполняемые на компьютере, поэтому мы можем создать электронную кость, имеющую любое необходимое количество граней. Начнем с шести граней, а затем обобщим алгоритм.

Нам необходимо случайное значение в пределах от 1 до 6. В то же время функция `rand()` генерирует целые числа в диапазоне от `0` до `RAND_MAX`; Значение `RAND_MAX` определена в заголовочном файле `stdlib.h`. Обычно это `INT_MAX`. В силу этого мы должны выполнить некоторые настройки. Ниже представлен один из возможных подходов.

1. Получить случайное число по модулю 6. При этом получается целое число в пределах от 0 до 5.
2. Прибавить 1. Новое число попадает в диапазон от 1 до 6.
3. Для обобщения этого алгоритма достаточно заменить 6 в первом шаге любым другим количеством граней.

Приведенный далее программный код реализует описанные идеи:

```
#include <stdlib.h>              /* для доступа к функции rand() */
int rollem(int sides)
{
   int roll;
   roll = rand() % sides + 1;
   return roll;
}
```

Немного ужесточим требования и сделаем так, чтобы функция позволяла бросать произвольное количество костей и подсчитывала общую сумму. Программа в листинге 12.11 решает эту задачу.

Листинг 12.11. Файл `diceroll.c`

```
/* diceroll.c -- эмуляция игры в кости */
#include "diceroll.h"
#include <stdio.h>
#include <stdlib.h>           /* для библиотечной функции rand() */
int roll_count = 0;           /* внешнее связывание */
static int rollem(int sides)  /* переменная, приватная для данного файла */
{
   int roll;

   roll = rand() % sides + 1;
   ++roll_count;               /* счетчик вызовов функции */

   return roll;
}

int roll_n_dice(int dice, int sides)
{
   int d;
   int total = 0;

   if (sides < 2)
   {
       printf("Нужны, по меньшей мере, 2 грани.\n");
       return -2;
   }
   if (dice < 1)
   {
       printf("Нужна, по меньшей мере, 1 кость.\n");
       return -1;
   }
   for (d = 0; d < dice; d++)
       total += rollem(sides);

   return total;
}
```

Для этого кода характерны некоторые особенности. Во-первых, он делает `rollem()` функцией, приватной для этого файла. В рассматриваемом случае это функция `roll_n_dice()`. Во-вторых, чтобы продемонстрировать, как работает внешнее связывание, данный код объявляет внешнюю переменную с именем `roll_count`. Эта переменная подсчитывает, сколько раз была вызвана функция `rollem()`. Рассматриваемый пример довольно сложен, но в то же время он показывает, как работают свойства внешней переменной.

В-третьих, этот файл содержит следующий оператор:

```
#include "diceroll.h"
```

Когда вы используете стандартные библиотечные функции, такие как `rand()`, вы включаете стандартный заголовочный файл (файл `stdlib.h` для функции `rand()`) вместо того, чтобы объявлять функцию. Это объясняется тем, что данный заголовочный файл уже содержит правильное объявление этой функции. Мы эмулируем этот подход, используя заголовочный файл `diceroll.h` для доступа к функции `roll_n_dice()`. Заключение имени файла в двойные кавычки, а не в угловые скобки, говорит компилятору

о том, что этот файл следует искать локально, а не в стандартных расположениях, которые используются для хранения стандартных заголовочных файлов. Смысл выражения "искать локально" зависит от реализации. Некоторые обычные интерпретации предполагают размещение заголовочного файла в том же каталоге или папке, где хранится исходный код, или в том же каталоге или папке, в которой хранится файл проект (если ваш компилятор их использует). В листинге 12.12 показано содержимое заголовочного файла.

Листинг 12.12. Файл diceroll.h

```
//diceroll.h
extern int roll_count;
int roll_n_dice(int dice, int sides);
```

Этот заголовочный файл содержит прототипы функций и объявление extern. Поскольку файл diceroll.c включает этот заголовок, файл diceroll.c фактически содержит два объявления переменной roll_count:

```
extern int roll_count;          // из заголовочного файла
int roll_count = 0;             // из файла исходных кодов
```

Это хорошо. Можно иметь только одно определяющее объявление переменной. Однако объявление с ключевым словом extern — это ссылочное объявление, а таких объявлений может быть сколько угодно. Программа, использующая функцию roll_n_dice(), должна включать этот заголовочный файл. Такой подход не только предоставляет прототип функции roll_n_dice(), он обеспечивает доступ данной программы к переменной roll_count. Листинг 12.13 иллюстрирует все эти моменты.

Листинг 12.13. Файл manydice.c

```
/* manydice.c -- многократное бросание костей */
/* компилируется с файлом diceroll.c */
#include <stdio.h>
#include <stdlib.h>              /* для библиотечной функции srand() */
#include <time.h>                /* для функции time()               */
#include "diceroll.h"            /* для функции roll_n_dice()        */
                                 /* и для переменной roll_count      */
int main(void)
{
   int dice,roll;
   int sides;
   srand((unsigned int) time(0));   /* рандомизация начального числа */
   printf("Введите количество граней кости или 0 для завершения программы.\n");
   while (scanf("%d", &sides) == 1 && sides > 0)
   {
      printf("Сколько нужно костей?\n");
      scanf("%d", &dice);
      roll = roll_n_dice(dice, sides);
      printf("Вы бросали %d раз, используя %d %d-гранные кости.\n",
              roll, dice, sides);
      printf("Сколько должно быть граней? Введите 0 для завершения программы.\n");
   }
```

```
    printf("Функция rollem() была вызвана %d раз.\n",
           roll_count);              /* используется внешняя переменная */
    printf("Удачи!\n");
    return 0;
}
```

Выполните компиляцию программу, представленную на листинге 12.13, с файлом, содержащим листинг 12.11. Чтобы упростить процедуру, поместите листинги 12.11, 12.12 и 12.13 в один и тот же файл или каталог. Выполняет полученную программу. Выходные данные должны иметь примерно такой вид:

```
Введите количество граней кости или 0 для завершения программы.
6
Сколько нужно костей?
2
Вы бросали 12 раз, используя 2 6-гранные кости.
Сколько должно быть граней? Введите 0 для завершения программы.
6
Сколько нужно костей?
2
Вы бросали 4 раз, используя 2 6-гранные кости.
Сколько должно быть граней? Введите 0 для завершения программы.
6
Сколько нужно костей?
2
Вы бросали 5 раз, используя 2 6-гранные кости.
Сколько должно быть граней? Введите 0 для завершения программы.
0
Функция rollem() была вызвана 6 раз.
Удачи!
```

Поскольку программа использует функцию `srand()` для рандомизации начального числа, вы, скорее всего, не получите один и тот же вывод на один и тот же ввод. Обратите внимание, что функция `main()` в программе `manydice.c` имеет доступ к переменной `roll_count`, определенной в `diceroll.c`.

Вы можете использовать функцию `roll_n_dice()` многими способами. Если задается количество граней, равное 2, рассматриваемая программа моделирует процесс бросания монеты, при этом "орел" — это 2, а "решка" — 1 (или наоборот, как вам больше нравится). Вы легко можете изменить программу таким образом, чтобы она показывала как отдельные результаты, так и сумму. Вы также можете написать программу, моделирующую игру в кости. Если вам нужно большое число бросаний, как, например, в ролевых играх, вы легко можете внести изменения в программу, чтобы она выдавала выходные данные следующего вида:

```
Введите количество бросаний или q для завершения программы.
18
Сколько граней и сколько костей?
6 3
Имеем 18 бросаний 3 6-гранных костей.
  12  10   6   9   8  14   8  15   9  14  12  17  11   7  10
  13   8  14
Введите количество бросаний или q для завершения программы.
q
```

Другим применением функций `rand1()` и `rand()` (но не функции `rollem()`) является создание программы по отгадыванию чисел, в которой компьютер выбирает число, а вы его отгадываете. Можете попытаться написать такую программу самостоятельно.

Распределение памяти: функции `malloc()` и `free()`

Все пять классов памяти имеют одну общую особенность. После того, как вы примете решение, какой класс памяти использовать, решения, касающиеся области видимости и продолжительности хранения следуют автоматически. Ваш выбор подчиняется правилам управления предварительно скомпонованной памятью. Однако при этом существует еще один выбор, который обеспечивает большую гибкость. Этот выбор предполагает использование библиотечных функций распределения и управления памятью.

Во-первых, напомним о некоторых особенностях распределения памяти. Все программы должны резервировать пространство памяти, достаточное для хранения данных, с которыми они работают. Некоторые из операций по распределению памяти выполняются автоматически. Например, вы можете объявить

```
float x;
char place[] = "Dancing Oxen Creek";
```

при этом резервируется пространство памяти, достаточное для хранения объявленной переменной типа `float` и объявленной строки, либо вы можете передать системе более конкретное требование и запросить определенное количество памяти:

```
int plates[100];
```

Такое объявление резервирует 100 ячеек памяти, в каждой из них можно хранить значение типа `int`. Во всех этих случаях объявление содержит идентификатор для этой памяти, благодаря чему для идентификации данных можно пользоваться `x` или `place`.

Язык C идет дальше. Во время выполнения программы вы можете выделять дополнительную память по мере надобности. Основным средством распределения памяти является функция `malloc()`, которая принимает один аргумент: количество необходимых байтов памяти. Затем функция `malloc()` отыскивает подходящий блок памяти. Память “анонимна”; другими словами, функция `malloc()` распределяет блок памяти, но не присваивает ему имени. В то же время она возвращает адрес первого байта этого блока. В результате вы можете присваивать этот адрес переменной типа указатель и использовать этот указатель для доступа к памяти. Поскольку `char` представляет байт, функция `malloc()` традиционно объявляется как указатель на тип `char`. Стандарт ANSI C, однако, использует новый тип: указатель на `void`. Этот тип задумывался как “общий указатель”. Функция `malloc()` может применяться для возвращения указателя на массив, структуру или другой объект. В условиях стандарта ANSI C вы все еще должны выполнять приведение типов для ясности, однако присваивание значения указателя на `void` указателю другого типа не рассматривается как конфликт типов. Если функции `malloc()` не удастся найти затребованное пространство памяти, она возвращает нулевой указатель.

Воспользуемся функцией malloc() для решения задачи создания массива. С помощью malloc() можно запросить блок памяти во время выполнения программы. Вам также понадобится указатель, чтобы знать, где в памяти находится выделенный блок. Например, рассмотрим следующий программный код:

```
double * ptd;
ptd = (double *) malloc(30 * sizeof(double));
```

Этот код запрашивает дополнительное распределение памяти для размещения 30 значений типа double и устанавливает значение указателя ptd, нацеленного на соответствующую ячейку памяти. Обратите внимание, что ptd объявлен как указатель, который ссылается на одиночное значение типа double, а не на блок из 30 значений типа double. Помните, что имя массива — это адрес его первого элемента. Поэтому, если вы сделали так, что ptd указывает на первый элемент блока, вы можете использовать его как массив. То есть, вы можете применять выражение ptd[0] для доступа к первому элементу блока, ptd[1] — для доступа ко второму элементу и так далее. Как вам уже известно, вы можете использовать запись в форме указателя для массивов и запись в форме массива для указателей.

Сейчас доступны три способа для создания массива:

- Вы можете объявить массив, воспользовавшись константными выражениями для обозначения размерности, и затем использовать имя массива для доступа к его элементам.
- Вы можете объявить массив переменной длины, используя для этой цели переменные выражения для обозначения размерности массива и использовать имя массива для доступа к его элементам. (Эта возможность предусмотрена стандартом C99.)
- Вы можете объявить указатель, вызвать функцию malloc() и воспользоваться указателем для доступа к элементам.

С помощью второго или третьего метода можно сделать то, что не удастся сделать в условиях обычно объявленного массива — создать *динамический массив*, то есть массив, под который память выделяется во время выполнения программы и размер которого определяется по результатам выполнения программы. Предположим, например, что n — целочисленная переменная. До момента появления стандарта C99 вы не могли выполнить следующее объявление:

```
double item[n]; /*до появления C99 не допускалось, если n — переменная*/
```

Однако следующее допустимо даже при наличии компилятора, не соответствующего требованиям стандарта C99:

```
ptd = (double *) malloc(n * sizeof(double)); /* разрешается */
```

Эта конструкция работает и, как вы убедитесь далее, она обладает несколько большей гибкостью, чем массив переменной длины.

Обычно вы должны следить за сбалансированным использованием функций malloc() и free(). Функция free() принимает в качестве аргумента адрес, возвращенный до этого функцией malloc(), и освобождает ранее зарезервированную память. Следовательно, продолжительность использования выделенной памяти рассчитывается с момента, когда была вызвана функция malloc() для выделения памяти, до

момента, когда вызывается функция free() с целью освобождения памяти, дабы ее можно было использовать повторно. Функции malloc() и free() можно рассматривать как средство управления пулом памяти. Каждое обращение к функции malloc() влечет за собой выделение некоторого пространства памяти для последующего его использования программой, каждое обращение к функции free() означает возвращение этой памяти в пул, из которого она затем будет выделена для повторного использования в других целях. Аргументом функции free() должен быть указатель на блок памяти, выделенный функцией malloc(); вы не можете использовать free() для освобождения памяти, выделенной другими средствами, такими как объявление массива. Обе функции malloc() и free() являются прототипами в заголовочном файле stdlib.h.

Далее с помощью функции malloc() программа получает возможность принять решение, файл какой размерности ей нужен, и создает его во время выполнения программы. Листинг 12.14 служит иллюстрацией этой возможности. В коде из этого листинга осуществляется присваивание адреса блока памяти указателю ptd, а затем ptd используется подобно массиву. Кроме того, с помощью функции exit(), прототип которой содержится в заголовочном файле stdlib.h, выполнение программы прекращается в случае, если не удастся выделить затребованную память. Значение EXIT_FAILURE также определено в этом заголовочном файле. В соответствии со стандартом возможен возврат двух значений, которые гарантированно распознают все операционные системы: EXIT_SUCCESS (или, что одно и то же, значение 0), указывающее на нормальное завершение программы, и EXIT_FAILURE, указывающее на аварийное завершение программы. Некоторые операционные системы, включая Unix, Linux и Windows, распознают и другие целочисленные значения.

Листинг 12.14. Программа dyn_arr.c

```
/* dyn_arr.c -- динамически распределенный массив */
#include <stdio.h>
#include <stdlib.h>   /* для функций malloc(), free() */
int main(void)
{
   double * ptd;
   int max;
   int number;
   int i = 0;

   puts("Введите максимальное количество элементов типа double.");
   scanf("%d", &max);
   ptd = (double *) malloc(max * sizeof (double));
   if (ptd == NULL)
   {
      puts("Не удалось распределить память. Аварийное завершение.");
      exit(EXIT_FAILURE);
   }
   /* ptd теперь указывает на массив элементов max */
   puts("Введите значения (q для выхода из программы):");
   while (i < max && scanf("%lf", &ptd[i]) == 1)
       ++i;
   printf("Введено %d элементов:\n", number = i);
```

```
    for (i = 0; i < number; i++)
    {
       printf("%7.2f ", ptd[i]);
       if (i % 7 == 6)
          putchar('\n');
    }
    if (i % 7 != 0)
       putchar('\n');
    puts("Программа завершена.");
    free(ptd);
    return 0;
}
```

Ниже показаны результаты выполнения этой учебной программы. Во время ее выполнения были введены шесть чисел, но программа обработала только пять из них, поскольку размерность массива была ограничена значением 5.

```
Введите максимальное количество элементов типа double.
5
Введите значения (q для выхода из программы):
20 30 35 25 40 80
Введено 5 элементов:
  20.00 30.00 35.00 25.00 40.00
Программа завершена.
```

Теперь рассмотрим приведенный выше код. Программа получает нужную размерность массива с помощью следующих строк:

```
puts("Введите максимальное количество элементов типа double.");
scanf("%d", &max);
```

Следующая строка кода выделяет в памяти пространство, достаточное для хранения затребованного количества элементов, а затем присваивает адрес выделенного блока памяти указателю `ptd`:

```
ptd = (double *) malloc(max * sizeof (double));
```

Приведение типов (`double *`) не обязательно в C, но необходимо в C++, таким образом, использование операции приведения типов упрощает перевод рассматриваемой программы из C в C++.

Возможно, что функция `malloc()` не сможет предоставить нужное количество памяти. В этом случае эта функция возвращает нулевой указатель, и выполнение программы прекращается:

```
if (ptd == NULL)
{
   puts("Не удалось распределить память. Аварийное завершение.");
   exit(EXIT_FAILURE);
}
```

Если программа преодолеет это препятствие, она может рассматривать `ptd`, как если бы это было имя массива из `max` элементов, что и получается.

Обратите внимание на вызов функции `free()` ближе к концу программы. Она освобождает память, выделенную функцией `malloc()`. Функция `free()` освобождает

только те блоки памяти, на которые указывает ее аргумент. В данном конкретном примере в использовании функции `free()` нет необходимости, поскольку любая выделенная память автоматически освобождается по завершении выполнения программы. Однако в более сложной программе возможность освобождать и повторно использовать одно и то же пространство памяти может оказаться очень важной.

Что мы выиграли из того, что воспользовались динамическим массивом? Прежде всего, увеличилась гибкость программы. Предположим, что вы заранее знаете, что в течение большей части времени выполнения она потребует не менее 100 элементов, но в некоторые моменты ей потребуются 10 000 элементов. Если вы объявите массив, то должны рассчитывать на худший случай и объявить его с 10 000 элементов. Большую часть времени эту память растрачивается впустую. Когда наступает момент, когда необходимо, скажем, 10 001 элемент, программа аварийно завершится. С помощью динамического массива программа сможет приспосабливаться к обстоятельствам.

Важность функции `free()`

Объем статической памяти фиксируется во время компиляции; оно не изменяется, пока программа выполняется. Объем памяти, используемой автоматическими переменными, возрастает и убывает автоматически по мере выполнения программы. Однако объем распределенной памяти будет расти до тех пор, пока вы не вспомните о функции `free()`. Например, предположим, что в вашем распоряжении имеется функция, которая создает временную копию массива, как схематично показано в следующем коде:

```
...
int main()
{
double glad[2000];
int i;
...for (i = 0; i < 1000; i++)
   gobble(glad, 2000);
...}
void gobble(double ar[], int n)
{
   double * temp = (double *) malloc( n * sizeof(double));
...        /* free(temp); // забыли воспользоваться функцией free() */
}
```

Когда функция `gobble()` вызывается в первый раз, она создает указатель `temp` и использует функцию `malloc()` для выделения 16 000 байтов памяти (предполагается, что тип `double` занимает 8 байтов). Предположим, как показано в коде, мы не вызываем `free()`. Как только функция `gobble()` завершит работу, указатель `temp`, будучи автоматической переменной, исчезает. Однако 16 000 байтов памяти, на которые он указывал, все еще зарезервированы. Доступ к этой памяти невозможен, так как у ее адреса нет. Она не может быть использована повторно, поскольку не была вызвана функция `free()`.

Когда функция `gobble()` вызывается во второй раз, она снова создает переменную `temp` и снова вызывает функцию `malloc()` для получения 16 000 байтов памяти. К первому блоку, состоящему из 16 000 байтов памяти, теперь нет доступа, по этой причине

функция `malloc()` должна найти второй блок памяти размером 16 000 байтов. Когда функция завершит работу, этот блок также станет недоступным и не сможет использоваться повторно.

Поскольку описанный цикл повторяется 1000 раз, к моменту окончания цикла из пула свободной памяти будет изъято 16 000 000 байтов памяти. Фактически может получиться так, что если дело зайдет действительно далеко, программе просто не хватит памяти. Подобное явление называется *утечкой памяти*, и такого рода проблем можно избежать, если поместить в конец функции `gobble()` вызов `free()`.

Функция `calloc()`

Следующий вариант распределения памяти предусматривает использование функции `calloc()`. Типичный способ применения этой функции выглядит следующим образом:

```
long * newmem;
newmem = (long *)calloc(100, sizeof (long));
```

Подобно `malloc()`, функция `calloc()` возвращает указатель на тип `char` в варианте, который применялся до вступления в силу стандарта ANSI, и указатель на `void` в условиях действия стандарта ANSI. Эта новая функция принимает два аргумента, оба они должны быть целыми значениями без знака (тип `size_t` в стандарте ANSI). Первый аргумент — необходимое количество ячеек. Второй аргумент — это размер каждой ячейки в байтах. В нашем случае тип `long` использует 4 байта, следовательно, эта команда устанавливает 100 4-байтовых единиц, используя в совокупности 400 байтов для хранения данных.

Используя значение `sizeof (long)` вместо 4 увеличивает переносимость этого кода. Он будет работать в системах, где тип `long` имеет размер, отличный от 4.

Функция `calloc()` привносит еще одно свойство: она устанавливает всем битам блока нулевое значение. (Следует, однако, отметить, что в некоторых системах значение с плавающей точкой, равное нулю, представляется значением, отличным от того, когда все разряды числа установлены в ноль.)

Функция `free()` может также применяться для освобождения памяти, выделенной с помощью функции `calloc()`.

Динамическое распределение памяти является ключевой особенностью многих современных технологий программирования. Мы рассмотрим некоторые из них в главе 17. Ваша библиотека C, по-видимому, может предложить некоторые другие функции управления памятью — одни из них переносимые, другие — нет. Уделите некоторое время на краткое знакомство с ними.

Распределение динамической памяти и массивы переменной длины

У массивов переменной длины и `malloc()` имеются общие характеристики функциональности. Оба средства, например, могут использоваться для создания массива, размер которого определяется во время выполнения программы:

```
int vlamal()
{
```

```
    int n;
    int * pi;
    scanf("%d", &n);
    pi = (int *) malloc (n * sizeof(int));
    int ar[n];              // массив переменной длины
    pi[2] = ar[2] = -5;
...
}
```

Одно из отличий между ними заключается в том, что массив переменной длины представлен автоматической памятью. Одно из последствий употребления динамической памяти состоит в том, что пространство памяти, используемое массивами переменной длины, освобождается автоматически, как только поток управления покинет блок, в котором этот массив определен; в рассматриваемом случае — когда функция `vlamal()` завершит работу. По этой причине вам не надо заботиться о том, чтобы вовремя вызвать функцию `free()`. С другой стороны, доступ к массиву, созданному с помощью функции `malloc()`, не обязательно должен быть ограничен какой-то одной функцией. Например, одна функция может построить массив и вернуть указатель для доступа к нему со стороны вызывающей функции. Затем вызывающая функция по завершении может обратиться к `free()`. Не будет ошибки, если вы употребите переменную типа указатель с функцией `free()`, отличную от аналогичной переменной для функции `malloc()`; тем не менее, адреса, сохраняемые в этих указателях, обязательно должны совпадать.

Массивы переменной длины более удобны для создания многомерных массивов. Вы можете построить двухмерный массив, воспользовавшись для этого функцией `malloc()`, но синтаксис окажется весьма неудобным. Если компилятор не поддерживает возможности создания массивов переменной длины, одна из размерностей должна быть фиксированной как, например, показано ниже:

```
int n = 5;
int m = 6;
int ar2[n][m];          // массив переменной длины размерности n на m
int (* p2)[6];          // действуют стандарты, предшествующие C99
int (* p3)[m];          // требуется поддержка массивов переменной длины
p2 = (int (*)[6]) malloc(n * 6 * sizeof(int));   // массив n * 6
p3 = (int (*)[m]) malloc(n * m * sizeof(int));   // массив n * m
// приведенные выше выражения также требуют поддержки
// массивов переменной длины
ar2[1][2] = p2[1][2] = 12;
```

Стоит посмотреть на объявления указателей. Функция `malloc()` возвращает указатель, поэтому `p2` должен быть указатель подходящего типа. Объявление

```
int (* p2)[6];          // действуют стандарты, предшествующие C99
```

говорит, что `p2` указывает на массив из шести значений типа `int`. Это значит, что `p2[i]` трактуется как элемент данных, состоящий из шести значений типа `int`, и что `p2[i][j]` является одиночным значением типа `int`.

Второе объявление указателя использует переменную для определения размерности массива, на который ссылается указатель `p3`. Это означает, что `p3` рассматривается как указатель на некоторый массив переменной длины, и именно по этой причине данный код не будет работать в условиях действия стандарта C90.

Классы памяти и динамическое распределение памяти

Вы, по-видимому, хотели бы знать, какая связь существует между классами памяти и распределением динамической памятью. Рассмотрим идеализированную модель. Доступную программе память можно условно разделить на три раздела: раздел статических переменных с внешним связыванием, внутренним связыванием и без связывания; раздел автоматических переменных; раздел динамически распределяемой памяти.

Объем памяти, необходимый для классов статической продолжительности хранения становится известным на стадии компиляции, и данные, которые хранятся в этом разделе, доступны на всем протяжении выполнения программы. Каждая переменная этих классов возникает в момент начала выполнения программы и исчезает в момент завершения программы. Однако автоматическая переменная возникает в тот момент, когда осуществляется вход в блок кода, который содержит определение этой переменной, и исчезает в момент выхода потока управления из этого блока. По этой причине, после того, как программа вызовет функции, и эти функции завершат свою работу, объем памяти, использованный автоматическими переменными, сначала возрастает, а затем уменьшается. Этот раздел памяти обычно реализован в виде стека. Это значит, что новые переменные добавляются в память постепенно, по мере их возникновения, а удаляются из памяти в обратном порядке, по мере их исчезновения.

Динамически распределяемая память возникает в тот момент, когда вызывается функция `malloc()` или родственная ей функция, и освобождается после того, как будет вызвана функция `free()`. Неизменность памяти контролируется самим программистом, а не каким-то набором жестких правил, следовательно, блок памяти может создаваться одной функцией и освобождаться другой. В силу этого обстоятельства раздел памяти, который используется для распределения динамической памяти, может оказаться разбитым на множество фрагментов, то есть неиспользованные участки могут быть перемешаны с активными блоками памяти. Однако эксплуатация динамической памяти может оказаться более медленным процессом, чем стековой памяти.

Квалификаторы типов в стандарте ANSI C

Как вам уже известно, переменная характеризуется типом и классом памяти. Стандарт C90 добавляет к этому еще два свойства: постоянство и изменяемость. Эти свойства объявляются с помощью ключевых слов `const` и `volatile`, которые создают *квалифицированные типы*. Стандарт C99 добавляет третий квалификатор — `restrict`, предназначенный для упрощения оптимизации компилятора.

Стандарт C99 наделяет квалификаторы типов новым свойством — теперь они идемпотентны. И хотя этот термин звучит весьма многозначительно, смысл его заключается в том, что вы можете использовать один и тот же квалификатор в объявлении более одного раза, при этом избыточные квалификаторы игнорируются:

```
const const const int n = 6; // то же, что const int n = 6;
```

Например, благодаря этому свойству допустимой становится следующая последовательность:

```
typedef const int zip;
const zip q = 8;
```

Квалификатор типа const

В главах 4 и 10 вы впервые познакомились с const. Напоминаем, что ключевое слово const в объявлении создает переменную, значение которой не может быть изменено с помощью операций присваивания или инкремента/декремента. В компиляторе, совместимом со стандартом ANSI, программный код

```
const int nochange; /* указывает, что m является константой */
nochange = 12; /* не разрешено */
```

должен выдать сообщение об ошибке. Однако вы можете инициализировать переменную const. Поэтому в приведенной ниже строке ошибок нет:

```
const int nochange = 12;              /* правильно */
```

Это объявление делает nochange переменной только для чтения. После того, как она будет инициализирована, ее значение менять нельзя.

Например, вы можете использовать ключевое слово const для создания массива данных, которые система не может менять:

```
const int days1[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Использование const в объявлениях указателей и параметров

Использовать ключевое слово const в объявлении простой переменной и массива достаточно просто. Указатели — более сложные объекты, поскольку следует различать создание указателя const и создание значения const, на которое ссылается этот указатель. Объявление

```
const float * pf; /* pf указывает на константное значение типа float */
```

создает указатель pf, указывающий на значение, которое должно оставаться постоянным. Само значение pf можно изменять. Например, он может быть переустановлен и указывать на другое значение const. В противоположность этому, объявление

```
float * const pt; /* pt - указатель const */
```

говорит о том, что сам указатель pt не может менять свое значение. Он всегда должен ссылаться на один и то же адрес, но значение, на которое он указывает, может изменяться. И, наконец, объявление

```
const float * const ptr;
```

означает, что ptr должен всегда указывать на одну и ту же ячейку памяти и что значение, на которое хранится в этой ячейке, не может изменяться.

Существует и третье положение, куда можно поместить ключевое слово const:

```
float const * pfc;     // то же, что и const float * pfc;
```

Как показывает комментарий, размещение ключевого слова const после имени типа и перед знаком * означает, что указатель не может использоваться для изменения значения, на которое он указывает. Короче говоря, ключевое слово const, поставленное в любом месте слева от знака *, делает данные постоянными, а const справа от знака * делает постоянным сам указатель.

Один из обычных видов использования этого нового ключевого слова предполагает объявление указателей, которые служат формальными параметрами функций.

Например, предположим, что имеется функция с именем display(), которая отображает содержимое массива. Чтобы воспользоваться ею, вы должны передать ей имя массива в качестве фактического аргумента, в то же время имя массива — это адрес. Это позволит данной функции изменять данные в вызывающей функции. Однако приведенный ниже прототип не позволит этому случиться:

```
void display(const int array[], int limit);
```

В прототипе и заголовке функции объявление параметра const int array[] аналогично объявлению const int * array, тем самым первое объявление утверждает, что данные, на которые указывает массив, не подлежат изменению.

Библиотека ANSI C придерживается этой практики. Если указатель используется только для того, чтобы предоставить функции доступ к значениям, этот указатель объявляется с квалификатором const. Если указатель используется для изменения данных в вызывающей функции, ключевое слово const не задается. Например, объявление функции strcat(), соблюдающее требования стандарта ANSI C, имеет следующий вид:

```
char *strcat(char *, const char *);
```

Напоминаем, что функция strcat() добавляет копию второй строки в конец первой строки. Эта операция приводит к изменению первой строки, тогда как вторая строка остается неизменной. Данное объявление отображает это обстоятельство.

Использование спецификатора const с глобальными данными

Вспомните, что использование глобальных переменных рассматривается как рискованный подход, поскольку он в этом случае существует опасность ошибочного изменения данных любыми другими частями программы. Этот риск исчезает, если данные постоянные, поэтому использование глобальных переменных со спецификатором const вполне оправдано. Вы можете иметь переменные const, массивы const и структуры const. (Структуры — это составной тип данных, который рассматривается в следующей главе.)

Однако областью, требующей к себе постоянного внимания, является совместное использование константных данных различными файлами. Существуют две стратегии, которые можно взять на вооружение. Первая заключается в том, чтобы следовать обычным правилам, применяемым к внешним переменным, а именно, использование определительных объявлений в одном файле и ссылочные объявления (при этом используется ключевое слово extern) в других файлах:

```
/* file1.c -- определение некоторых глобальных констант */
const double PI = 3.14159;
const char * MONTHS[12] =
   {"Январь", "Февраль", "Март", "Апрель", "Май", "Июнь", "Июль",
    "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"};
/* file2.c -- использование констант, определенных в другом месте */
extern const double PI;
extern const * MONTHS[];
```

Второй подход предусматривает размещение констант во включаемом файле. В данном случае вы должны выполнить дополнительное действие, предусматривающее использование класс статической внешней памяти:

```
/* constant.h -- определение некоторых глобальных констант */
static const double PI = 3.14159;
static const char * MONTHS[12] =
   {"Январь", "Февраль", "Март", "Апрель", "Май", "Июнь", "Июль",
    "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"};

/*file1.c -- использование глобальных констант, определенных в другом месте*/
#include "constant.h"

/*file2.c -- использование глобальных констант, определенных в другом месте*/
#include "constant.h"
```

Если вы не используете ключевое слово `static`, включение заголовочного файла `constant.h` в `file1.c` и `file2.c` может привести к тому, что в каждом файле появятся определительные объявления одного и того же идентификатора, что не поддерживается стандартом ANSI. (Некоторые компиляторы, однако, допускают эту возможность.) Если сделать идентификатор внешним и статическим, вы фактически предоставляете каждому файлу отдельную копию данных. Это не будет работать, если предполагается, что файлы используют данные для связи между собой, поскольку каждый файл работает со своей собственной копией. Однако поскольку данные являются константными (так как используется ключевое слово `const`) и идентичными (поскольку оба файла включают один и тот же заголовочный файл), это не проблема.

Достоинство подхода с использованием заголовочного файла состоит в том, что вам не обязательно помнить о необходимости использования определительных объявлений в одном файле и ссылочных объявлений в следующем файле, оба эти файла просто включает один и тот же заголовочный файл. Недостаток такого подхода связан с тем, что эти данные дублируются. Что касается предыдущих примеров, то для них это не является проблемой в полном смысле этого слова, тем не менее, оно может стать таковой, если в число константных данных входят крупные массивы.

Квалификатор типа `volatile`

Квалификатор `volatile` (*volatile* — изменчивый) сообщает компилятору, что значение переменной может быть изменено агентами, отличными от данной программы. Он обычно используется для представления аппаратных адресов и данных, совместно используемых с другими программами, которые выполняются одновременно. Например, адрес может содержать текущее время блока. Значение, хранящееся по этому адресу, изменяется с течением времени, независимо от того, что делает ваша программа. Либо адрес может служить для получения информации, скажем, с другого компьютера.

Синтаксис этого квалификатора не отличается от синтаксиса `const`:

```
volatile int loc1;      /* loc1 - это изменчивая ячейка */
volatile int * ploc;    /* ploc указывает на изменчивую ячейку */
```

Эти операторы объявляют `loc1` как значение типа `volatile`, а `ploc` представляет собой указатель на значение типа `volatile`.

Концепция `volatile` достаточно интересна, и вы, скорее всего, хотели бы узнать, почему комитет ANSI посчитал необходимым использовать `volatile` в качестве ключевого слова. Причина состоит в том, что оно облегчает решение компилятором задачи оптимизации.

Предположим, например, имеется следующий программный код:

```
val1 = x;
/* некоторый программный код, не использующий x */
val2 = x;
```

Интеллектуальный (оптимизирующий) компилятор может отметить, что вы используете объект x дважды, не меняя его значения. Он временно может сохранить значение x в регистре. Затем, когда значение x требуется для val2, он может сэкономить время, считывая это значение из регистра, а не из исходной ячейки памяти. Эта процедура называется *кэшированием*. В первоначальном варианте кэширование представляет хорошую форму оптимизации, но не в том случае, когда x изменяется в промежутке между двумя операторами некоторым другим агентом. Если бы не было ключевого слова volatile, у компилятора не было бы возможности знать, может ли это случиться. В силу этого обстоятельства, чтобы избежать ошибки, компилятор не может реализовать кэширование. Такой была ситуация до появления стандарта ANSI. Теперь, однако, если ключевое слово volatile не используется в объявлении, компилятор имеет основания предположить, что значение не будет изменено в промежутке между двумя случаями его использования, и он может предпринять попытку оптимизации данного программного кода.

Значение может быть одновременно и const, и volatile. Например, установки аппаратных часов в обычном режиме не могут быть изменено программно, благодаря чему это значение получает тип const, но оно может быть изменено агентом, отличным от данной программы, и по этой причине значение получает тип volatile. Просто используйте оба эти квалификатора в объявлении, как показано ниже, при этом порядок их следования значения не имеет:

```
volatile const int loc;
const volatile int * ploc;
```

Квалификатор типа restrict

Ключевое слово restrict (*restrict* — ограничить) расширяет вычислительную поддержку, предоставляя компилятору возможность оптимизировать некоторые виды программного кода. Оно может быть применено только к указателям, при этом оно определяет, что указатель является единственным исходным средством доступа к конкретному объекту данных. Чтобы увидеть, в чем польза от этого ключевого слова, необходимо рассмотреть несколько примеров. Взгляните на следующие объявления:

```
int ar[10];
int * restrict restar = (int *) malloc(10 * sizeof(int));
int * par = ar;
```

В рассматриваемом случае указатель restar является единственным исходным средство доступа к памяти, выделенной функцией malloc(). Поэтому он может быть квалифицирован с помощью ключевого слова restrict. Указатель par, однако, не является ни исходным, ни единственным средством доступа к данным массива, поэтому он не может быть квалифицирован как restrict.

Теперь рассмотрим следующий, несколько искусственный пример, по условиям которого n — это переменная типа int:

```
for (n = 0; n < 10; n++)
{
   par[n] += 5;
   restar[n] += 5;
   ar[n] *= 2;
   par[n] += 3;
   restar[n] += 3;
}
```

Зная, что restar — единственное средства доступа к блоку данных, на который он указывает, компилятор может заменить два оператора, в которых используется restar, одним оператором, имеющим тот же эффект:

```
restar[n] += 8;                /* правильная замена */
```

Однако, приведение двух операторов, содержащих объект данных par, к одному, приводит к возникновению вычислительной ошибки:

```
par[n] += 8;                   /* дает неправильный ответ */
```

Причина, почему в этом случае возникает неправильный ответ, заключается в том, что цикл использует переменную ar для изменения значений в промежутке между двумя доступами к тем же данным со стороны par.

Без ключевого слова restrict компилятор должен рассчитывать на худший случай, а именно, на то, что какой-то другой идентификатор может изменить эти данные в промежутке между двумя использованиями соответствующего указателя. Если будет использовано ключевое слово restrict, компилятор получает свободу в поиске сокращенных методов вычислений.

Вы можете использовать ключевое слово restrict в качестве квалификатора параметров функции, являющихся указателями. Это значит, что компиляторы могут действовать в соответствии с предположением, согласно которому никакие другие идентификаторы не вносят изменений в данные, на которые ссылается конкретный указатель, в теле этой функции, и что компилятор может приступать к оптимизации, которая была бы невозможной в других случаях. Например, в библиотеке C определены две функции для копирования данных из одной ячейки в другую. Согласно стандарту C99, они имеют следующие прототипы:

```
void * memcpy(void * restrict s1, const void * restrict s2, size_t n);
void * memmove(void * s1, const void * s2, size_t n);
```

Каждый из них копирует n байтов из ячейки s2 в ячейку s1. Функция memcpy() требует, чтобы эти ячейки не накладывались друг на друга, в то же время функция memmove() такого требования не выдвигает. Применение к указателям s1 и s2 ключевого слова restrict означает, что каждый указатель является единственным средством доступа, следовательно, они не могут осуществлять доступ к одному и тому же блоку данных. Это вполне соответствует требованию отсутствия взаимного перекрытия. Используя функции memmove(), допускающей такие перекрытия, следует соблюдать осторожность при копировании данных с тем, чтобы эти данные не были затерты до того, как они будут использованы.

С ключевым словом restrict связаны две области применения. Одной из них является компилятор, и restrict говорит компилятору, что тот свободен выдвигать конкретные предположения, касающиеся оптимизации программы.

Другой такой областью применения является пользователь, и ключевое слово restrict говорит пользователю, что следует применять только те аргументы, которые удовлетворяют требованиям квалификатора restrict. В общем случае компилятор не может проверить, соблюдаете ли вы эти ограничения, при этом вы можете игнорировать их на свой страх и риск.

Новые места для старых ключевых слов

Стандарт C99 позволяет помещать квалификаторы типа и спецификатор класса памяти static внутрь круглых скобок формального параметра в прототипе и заголовке функции. В случае квалификаторов типа это позволяет использовать дополнительный синтаксис для существующих возможностей. Например, ниже представлено объявление, соблюдающее требования старого синтаксиса:

```
void ofmouth(int * const a1, int * restrict a2, int n);  // старый стиль
```

Из него следует, что a1 — указатель const на значение типа int, который, как известно, означает, что указатель является величиной постоянной, но отнюдь не означает, что постоянными являются данные, на которые он указывает. Это объявление говорит о том, что a2 — ограниченный указатель, о котором шла речь в предыдущем разделе. Новый и эквивалентный синтаксис этого объявления имеет вид:

```
void ofmouth(int a1[const], int a2[restrict], int n);    // стандарт C99
```

В случае класса памяти static возникают отличия, поскольку ситуация существенно меняется. Например, рассмотрим следующий прототип:

```
double stick(double ar[static 20]);
```

Такое применение спецификатора static показывает, что фактический аргумент в вызове функции является указателем на первый элемент массива, содержащего, по меньшей мере, 20 элементов. Цель этого состоит в том, чтобы предоставить компилятору возможность использовать эту информацию для оптимизации программного кода функции.

Подобно квалификатору restrict, ключевое слово static имеет две области применения. Одной из них является компилятор, и он сообщает компилятору, что тот свободен делать определенные предположения относительно оптимизации. Другой областью применения является пользователь, и static рекомендует ему использовать только те аргументы, которые удовлетворяют требованиям static.

Ключевые понятия

Язык C предлагает несколько моделей управления памятью. Вы должны ознакомиться с различными вариантами. Вы также должны выработать критерии, когда исследовать тот или иной вариант. В большинстве случаев автоматическая переменная будет наилучшим выбором. Если вы решите использовать другой тип переменной, то должны иметь на это основания. Для обмена данными между функциями лучше всего использовать автоматические переменные, параметры функции и возвращаемые значение, чем глобальные переменные. С другой стороны, глобальные переменные идеально подходят для константных данных.

Вы должны научиться понимать свойства статической памяти, автоматической памяти и распределенной памяти. В частности, имейте в виду, что объем используемой статической памяти определяется во время компиляции, а статические данные загружаются в память, когда программа загружается в память. Автоматические переменные размещаются в памяти, а память, отведенная под них, освобождается в процессе выполнения программы. Вы можете представлять себе автоматическую память как рабочее пространство, в которое многократно записываются данные. Распределяемая память увеличивается и уменьшается, однако в подобных случаях этот процесс управляется вызовами функций, а не происходит автоматически.

Резюме

Память, использованная для хранения данных, которыми манипулирует программа, может быть охарактеризована продолжительностью хранения, областью видимости и связыванием. Продолжительность хранения может быть статической, автоматической или распределенной. Если продолжительность хранения статическая, память распределяется в начале выполнения программы и остается занятой на протяжении всего выполнения. Если продолжительность хранения автоматическая, то память под переменную выделяется в момент, когда выполнение программы входит в блок, в котором эта переменная определена, и освобождается, когда выполнение программы покидает этот блок. Если память выделяется, то она выделяется с помощью функции `malloc()` (или родственной функции) и освобождается посредством функции `free()`. Область видимости определяет, какая часть программы может получить доступ к данным. Переменные, определенные вне пределов функции, имеют область видимости в пределах файла и видимы в любой функции, определенной после объявления этой переменной. Переменная, определенная в блоке или как параметр функции, видима только в этом блоке и в любом из блоков, вложенных в этот блок.

Связывание описывает экстент, в пределах которого переменная, определенная в одной части программы, может быть привязана к любой другой части программы. Переменная с областью видимости в пределах блока, будучи локальной, не имеет связывания. Переменная с областью видимости в пределах файла, имеет внутреннее или внешнее связывание. Внутреннее связывание означает, что переменная может быть использована в файле, содержащем ее определение. Внешнее связывание означает, что переменная может быть использована в других файлах.

Ниже приведен список пяти классов памяти языка C.

- **Автоматический**. Переменная, объявленная в блоке (или как параметр в заголовке функции) без спецификатора класса памяти или со спецификатором автоматического класса памяти, принадлежит автоматическому классу памяти. Для нее характерны автоматическая продолжительность хранения, область видимости в пределах блока и отсутствие связывания. Ее значение, если она не инициализирована, не определено.
- **Регистровый**. Переменная, объявленная в блоке (или как параметр в заголовке функции) со спецификатором класса памяти `register`, принадлежит регистровому классу памяти. Она имеет автоматическую продолжительность хранения, область видимости в пределах блока, а связывание отсутствует. Адрес такой переменной получить невозможно. Объявление переменной как регистровой под-

сказывает компилятору, что нужно обеспечить максимально быстрый доступ. Ее значение, если она не инициализирована, не определено.

- **Статический, отсутствие связывания**. Переменная, объявленная в блоке со спецификатором класса статической памяти, принадлежат к классу памяти "статический, отсутствие связывания". Она имеет статическую продолжительность хранения, связывание отсутствует. Она инициализируется только раз, во время компиляции. Если переменная не инициализирована явно, ее байты устанавливаются в 0.
- **Статический, внешнее связывание**. Переменная, объявленная как внешняя по отношению к любой функции и без использования класса статической памяти, принадлежит к классу памяти "статический, внешнее связывание". Она имеет статическую продолжительность хранения, область видимости в пределах файла и внешнее связывание. Она инициируется только раз, во время компиляции. Если не инициализирована явно, ее байты устанавливаются в 0.
- **Статический, внутреннее связывание**. Переменная, объявленная как внешняя по отношению к любой функции и с использованием класса статической памяти, принадлежит к классу памяти "статический, внутреннее связывание". Она имеет статическую продолжительность хранения, область видимости в пределах файла и внутреннее связывание. Она инициируется только раз, во время компиляции. Если не инициализирована явно, ее байты устанавливаются в 0.

Распределение памяти обеспечивается функцией `malloc()` (или ей подобной), которая возвращает указатель на блок памяти, содержащий затребованное количество байтов. Эта память может стать доступной для повторного использования после вызова функции `free()`, в этом вызове в качестве аргумента используется адрес.

Квалификаторами типа являются `const`, `volatile` и `restrict`. Квалификатор `const` определяет данные как константные. Будучи использованным с указателями, квалификатор `const` может указывать на то, что указатель сам является константой, в зависимости от того, какое место `const` занимает в объявлении. Квалификатор `volatile` говорит о том, что данные могут быть изменены процессами, отличными от программы. Его назначение состоит в предупреждении компилятора о том, что он не может выполнять оптимизацию компилируемой программы, которую он предпринимает в отсутствие этого квалификатора. Квалификатор `restrict` также предназначен для целей оптимизации. Указатель, отмеченный квалификатором `restrict`, идентифицируется как единственное средство доступа к конкретному блоку данных.

Вопросы для самоконтроля

1. Какие классы памяти создают переменные, локальные по отношению к функции, которая их содержит?
2. Какие классы памяти создают переменные, которые сохраняются на протяжении продолжительности хранения содержащей их программы?
3. Какой класс памяти создает переменные, которые могут использоваться в нескольких файлах? В одном файле?

4. Какой вид связывания имеют переменные с областью видимости в пределах блока?
5. Для чего служит ключевое слово extern?
6. Рассмотрим следующий фрагмент программного кода:

```
int * p1 = (int *) malloc(100 * sizeof(int));
```

Насколько от него отличается приведенный ниже оператор, в смысле конечного результата?

```
int * p1 = (int *) calloc(100, sizeof(int));
```

7. Какие функции знают каждую переменную в следующем программном коде? Содержит ли код какие-то ошибки?

```
/* файл 1 */
int daisy;
int main(void)
{
   int lily;
   ...;
}
int petal()
{
   extern int daisy, lily;
   ...;
}
/* файл 2 */
extern int daisy;
static int lily ;
int rose;
int stem()
{
   int rose;
   ...;
}
void root()
{
   ...;
}
```

8. Что будет печатать следующая программа?

```
#include <stdio.h>
char color= 'B';
void first(void);
void second(void);
int main(void)
{
   extern char color;
   printf("color в main() равно %c\n", color);
   first();
   printf("color в main() равно %c\n", color);
   second();
   printf("color в main() равно %c\n", color);
   return 0;
}
```

```
void first(void)
{
   char color;
   color = 'R';
   printf("color в first() равно %c\n", color);
}
void second(void)
{
   color = 'G';
   printf("color в second() равно %c\n", color);
}
```

9. Файл начинается со следующих объявлений:

```
static int plink;
int value_ct(const int arr[], int value, int n);
```

   а. Что говорят эти объявления о намерениях программиста?

   б. Повысится ли защищенность значений в вызывающей программе, если заменить `int` и `int n` на `const int` и `const int n`?

Упражнения по программированию

1. Внесите в программу, представленную в листинге 12.4, такие изменения, чтобы в ней не использовались глобальные переменные.

2. Расход горючего обычно вычисляется в милях на один галлон в США и в литрах на 100 километров в Европе. Ниже приводится часть программы, которая предлагает пользователю выбрать режим (метрический или принятый в США), а затем выполняет сбор данных и вычисляет расход горючего:

```
// Файл pe12-2b.c
#include <stdio.h>
#include "pe12-2a.h"
int main(void)
{
   int mode;
   printf("Введите 0 для выбора метрического режима или 1 для выбора
режима, принятого в США: ");
   scanf("%d", &mode);
   while (mode >= 0)
   {
      set_mode(mode);
      get_info();
      show_info();
      printf("Введите 0 для выбора метрического режима или 1 для
выбора режима, принятого в США");
      printf(" (или -1 для выхода из программы): ");
      scanf("%d", &mode);
   }
   printf("Программа завершена.\n");
   return 0;
}
```

Ниже показан пример выходных данных этой программы:

```
Введите 0 для выбора метрического режима или 1 для выбора режима,
принятого в США: 0
Введите пройденное расстояние в километрах: 600
Введите количество израсходованного горючего в литрах: 78.8
Расход горючего составляет 13.13 литров на 100 км.
Введите 0 для выбора метрического режима или 1 для выбора режима,
принятого в США (или -1 для выхода из программы): 1
Введите пройденное расстояние в милях: 434
Введите количество израсходованного горючего в галлонах: 12.7
Расход горючего составляет 34.2 миль на один галлон.
Введите 0 для выбора метрического режима или 1 для выбора режима,
принятого в США (или -1 для выхода из программы): 3
Неправильно выбран режим. Используется режим 1 (принятый в США).
Введите пройденное расстояние в милях: 388
Введите количество израсходованного горючего в галлонах: 15.3
Расход горючего составляет 25.4 миль на галлон.
Введите 0 для выбора метрического режима или 1 для выбора режима,
принятого в США (или -1 для выхода из программы): -1
Программа завершена.
```

Если пользователь неправильно выберет режим, программа должным образом прокомментирует его выбор и воспользуется режимом, выбранным в последний раз.

Напишите заголовочный файл `pe12-2a.h` и файл исходного кода `pe12-2a.c`, чтобы решить поставленную задачу. Файл исходного кода должен определить три переменных с областью видимости в пределах файла и внутренним связыванием. Одна переменная представляет режим, вторая переменная — расстояние и третья — расход топлива. Функция `get_info()` запрашивает ввод данных в соответствии с выбранным режимом и сохраняет ответы в переменных с областью видимости в пределах файла. Функция `show_info()` вычисляет и отображает расход топлива в соответствии с форматом выбранного режима.

3. Внесите изменения в программу, описанную в упражнении 2, с таким расчетом, чтобы в ней использовались только автоматические переменные. Программа должна обеспечить тот же пользовательский интерфейс, в частности, она должна предложить пользователю выбрать соответствующий режим и далее придерживаться знакомого сценария. В то же время вам придется прибегнуть к помощи другой последовательности вызовов функций.
4. Напишите и протестируйте в цикле функции, которая возвращает количество ее вызовов.
5. Напишите программу, которая генерирует список 100 случайных чисел в диапазоне от 1 до 10 в порядке убывания. (Вы можете приспособить для этой цели алгоритм, представленный в главе 11, для печати значений типа `int`. В данном случае выполните сортировку этих чисел самостоятельно.)
6. Напишите программу, которая генерирует 1000 случайных чисел в диапазоне от 1 до 10. Не надо сохранять или выводить на печать эти числа, вместо этого программа должна выводить, сколько раз появлялось каждое число. Программа должна выполнить эту процедуру для 10 различных начальных чисел. Одинаково ли количество появлений этих чисел? Можете использовать функции, рассмотренные в этой главе или функции `rand()` и `srand()` стандарта ANSI C того

же формата. Это один из способов проверки, насколько случайны числа, выдаваемые генератором случайных чисел.

7. Напишите программу, которая ведет себя так же, как и вариант, представленный в листинге 12.13, исследование которого мы проводили после того, как показали выходные данные программы из листинга 12.13. То есть программа должна генерировать выходные данные, аналогичные показанным ниже:

```
Введите количество бросаний или q для завершения программы.
18
Сколько граней и сколько костей?
6 3
Имеем 18 бросаний 3 6-гранных костей.
  12  10   6   9   8  14   8  15   9  14  12  17  11   7  10
  13   8  14
Введите количество бросаний или q для завершения программы.
q
```

8. Пусть имеется следующий фрагмент программы:

```
// Файл pe12-8.c
#include <stdio.h>
int * make_array(int elem, int val);
void show_array(const int ar[], int n);
int main(void)
{
   int * pa;
   int size;
   int value;
   printf("Введите количество элементов: ");
   scanf("%d", &size);
   while (size > 0)
   {
       printf("Введите значение для инициализации: ");
       scanf("%d", &value);
       pa = make_array(size, value);
       if (pa)
       {
           show_array(pa, size);
              free(pa);
       }
       printf("Введите количество элементов (или значение < 1 для
выхода из программы): ");
       scanf("%d", &size);
   }
   printf("Программа завершена.\n");
   return 0;
}
```

Завершите показанный код, добавив определения функций `make_array()` и `show_array()`. Функция `make_array()` принимает два аргумента. Первый аргумент — количество элементов массива значений типа `int`, второй аргумент — значение, которое должно быть присвоено каждому элементу. Указанная функция использует функцию `malloc()` для создания массива требуемого размера, присваивает каждому элементу заданное значение и возвращает указатель на массив. Функция `show_array()` отображает содержимое массива по восемь элементов в строке.

ГЛАВА 13

Файловый ввод-вывод

В этой главе:

- Функции: `fopen()`, `getc()`, `putc()`, `exit()`, `fclose()`, `fprintf()`, `fscanf()`, `fgets()`, `fputs()`, `rewind()`, `fseek()`, `ftell()`, `fflush()`, `fgetpos()`, `fsetpos()`, `feof()`, `ferror()`, `ungetc()`, `setvbuf()`, `fread()`, `fwrite()`
- Обработка файлов с помощью семейства стандартных функций ввода-вывода
- Текстовое и двоичное представления, текстовые и двоичные форматы, буферизованный и небуферизованный ввод-вывод
- Использование функций, которые могут осуществлять последовательный и произвольный доступ к файлам

Файлы являются важной частью современных компьютерных систем. Они используются для хранения программ, данных, корреспонденции, форм, графических данных и множества другой информации. Будучи программистом, вы должны знать, как писать программы, которые создают, записывают и читают файлы. В этой главе мы покажем, как это делается.

Обмен данными с файлами

Часто вы будете испытывать потребность в программах, которые могут считывать информацию из файла либо записывать результаты выполнения в файл. Одной из таких форм обмена данными между программой и файлом является переадресация файла, в чем вы могли убедиться, изучив материал главы 8. Этот метод отличается простотой, однако, его применение ограничено. Например, предположим, что вы хотите написать интерактивную программу, которая запрашивает названия книг, а затем сохраняет полный список книг в соответствующем файле. Если вы примените переадресацию, как показано ниже:

```
books > bklist
```

то все ваши интерактивные запросы будут переадресованы в файл `bklist`. В результате в файл `bklist` будет помещен ненужный текст, и вы даже не сможете ознакомиться с вопросами, на которые вы, по идее, ответили.

Язык C, как и следовало ожидать, предлагает более мощные методы обмена данными с файлами. Он позволяет открыть файл из программы, а затем с помощью специальных функций ввода-вывода выполнять чтение и запись в этот файл. Однако прежде чем переходить к изучению этих методов, вкратце рассмотрим саму природу файла.

Что такое файл?

Файл представляет собой именованный раздел памяти, обычно расположенный на диске. Например, `stdio.h` можно представлять себе как имя файла, содержащего некоторую полезную информацию. Однако для операционной системы файл является более сложным объектом. Крупный файл, например, может быть храниться в нескольких фрагментах, расположенных в различных частях памяти, либо он может содержать дополнительные данные, которые позволяют операционной системе определять, к какой категории этот файл относится. В то же время, все это заботы операционной системы, а отнюдь не ваши (разумеется, если только вы не разработчик операционных систем). Вас интересует только то, как представляются файлы в программе на языке C.

C рассматривает файл как непрерывную последовательность байтов, каждый из которых может быть прочитан индивидуально. Это соответствует структуре файла в операционной среде Unix, из которой язык C берет свое начало. Поскольку другие операционные среды не соответствуют в точности этой модели, стандарт ANSI C предлагает два способа представления файлов: текстовое представление и двоичное представление.

Текстовое и двоичное представление файлов

Двумя установленными стандартом ANSI представлениями файла является *двоичное* и *текстовое*. В двоичном представлении (режиме) каждый байт файла доступен из программы. В текстовом представлении то, что программа видит, может отличаться от того, что хранится в файле. В текстовом режиме представление в локальной среде таких элементов, как конец строки, отображается на соответствующее представление в языке C при чтении файла. Аналогично, текстовые файлы MS-DOS представляют конец строки в виде комбинации символов возврата каретки и конца строки: `\r\n`. Текстовые файлы в системе Macintosh представляют конец строки одним символом возврата каретки, то есть `\r`. Программы на языке C представляют конец строки с помощью просто `\n`. По этой причине, когда программа на языке C использует текстовое представление текстового файла MS-DOS, она преобразует `\r\n` в `\n` при чтении файла и `\n` в `\r\n` при записи в файл. Когда программа на C использует текстовое представление текстового файла Macintosh, она преобразует комбинацию символов `\r` в `\n` при чтении файла и `\n` в `\r` при записи в файл.

При работе с текстовыми файлами MS-DOS вы не ограничены одним лишь текстовым представлением. Вы можете воспользоваться двоичным представлением того же файла. Если вы так и поступите, ваша программа увидит в файле как комбинации символов `\r`, так и символов `\n`, при этом никакого отображения одной на другую не происходит (рис. 13.1).

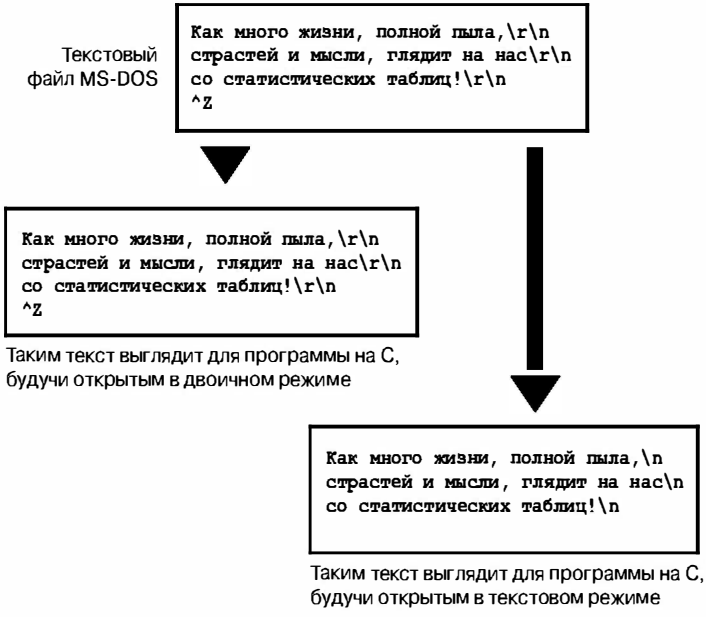


Рис. 13.1. *Двоичное и текстовое представления файлов*

Система MS-DOS различает текстовые и двоичные *файлы*, в то время как C поддерживает использование как текстового, так и двоичного *представлений*. Обычно текстовое представление используется для текстовых файлов, а двоичное — для двоичных. Тем не менее, можно использовать любое из указанных представлений для любого типа файлов, хотя текстовое представление двоичного файла — не очень удачный выбор.

И хотя стандарт ANSI C предусматривает как двоичное, так и текстовое представление, эти представления могут быть реализованы идентично. Например, поскольку Unix использует всего лишь одну файловую структуру, в приложениях Unix оба представления не отличаются друг от друга.

Уровни ввода-вывода

Помимо возможности выбора представления файла в большинстве случаев доступен выбор одного из двух возможных уровней ввода-вывода (то есть одного из двух уровней управления доступом к файлам). *Низкоуровневый ввод-вывод* использует базовые функции ввода-вывода, предоставляемые операционной системой. *Стандартный высокоуровневый ввод-вывод* предполагает применение стандартного пакета библиотечных функций C и определений из заголовочного файла `stdio.h`. Стандарт ANSI C поддерживает только стандартный пакет ввода-вывода, поскольку в данном случае нельзя гарантировать, что все операционные системы могут быть представлены одной и той же низкоуровневой моделью ввода-вывода. Поскольку ANSI C обеспечивает переносимость стандартной модели ввода-вывода, сосредоточим все внимание на ней.

Стандартные файлы

Программы на C автоматически открывают три файла, которые называются *стандартным вводом*, *стандартным выводом* и *стандартным выводом ошибок*. По умолчанию стандартный ввод представляет собой обычное устройство ввода в вашей системе, как правило, клавиатуру. По умолчанию стандартный вывод и стандартный вывод ошибок – это обычное устройство вывода вашей системы, такое как экран монитора.

Стандартный ввод, естественно, обеспечивает ввод для вашей программы. Это файл, который читается с помощью функций `getchar()`, `gets()` и `scanf()`. Стандартный вывод – то место, куда направляется обычный вывод программы. Он используется функциями `putchar()`, `puts()` и `printf()`. Перенаправление, как уже должно быть известно из главы 8, позволяет распознавать другие файлы как стандартный ввод и стандартный вывод. Назначение файла стандартного вывода ошибок заключается в том, чтобы предоставить место, логически отличное от других, для хранения сообщений об ошибках. Например, если вы используете перенаправление для пересылки вывода в файл вместо того, чтобы отображать его на экране, выходные данные, направляемые на стандартный вывод ошибок, все равно попадают на экран. Это удобно, поскольку если бы сообщения об ошибках пересылались в файл, вы бы их не увидели до тех пор, пока не просмотрели бы файл.

Стандартный ввод–вывод

Стандартный пакет ввода-вывода наряду со свойством переносимости обладает еще двумя преимуществами по сравнению с низкоуровневым вводом-выводом.

Во-первых, в нем доступно множество специализированных функций, решающих разнообразные задачи ввода-вывода. Например, функция `printf()` преобразует различные формы данных в строковый тип, который годится для вывода на терминалах.

Во-вторых, и ввод, и вывод *буферизованы*. Другими словами, информация передается большими порциями (обычно по 512 байтов и более за раз), чем один байт за один раз. Например, когда программа читает файл, некоторая порция данных считывается в буфер, представляющий промежуточную область памяти. Такая буферизации существенно увеличивает скорость передачи данных. Затем программа может исследовать отдельные байты в этом буфере.

Буферизация выполняется незаметно для пользователя, благодаря чему возникает иллюзия посимвольного доступа. (Вы можете также выполнять буферизацию низкоуровневого ввода-вывода, но при этом значительную часть работы придется выполнить самому.) В листинге 13.1 показана программа, которая демонстрирует использование стандартного ввода-вывода для чтения файла и подсчета количества символов в файле. Мы обсудим свойства программы из листинга 13.1 в нескольких последующих разделах. (Эта программа принимает аргументы командной строки. Если вы пользователь Windows, вы можете выполнить эту программу в окне MS-DOS по завершении процесса компиляции. Если вы пользователь Macintosh, вам потребуется `console.h` и функция `ccommand()`, как описано в главе 11 и в документации по Code Warrior. В качестве альтернативы можно изменить программу так, чтобы для ввода имени файла применялись функции `puts()` и `gets()`, а не аргументы командной строки.)

Листинг 13.1. Программа `count.c`

```
/* count.c -- использование стандартного ввода-вывода */
#include <stdio.h>
#include <stdlib.h>      // прототип ANSI C функции exit()
int main(int argc, char *argv[])
{
    int ch;              // место, куда помещается каждый символ в том виде,
                         // в каком он был прочитан
    FILE *fp;            // "указатель на файл"
    long count = 0;
    if (argc != 2)
    {
        printf("Использование: %s filename\n", argv[0]);
        exit(1);
    }
    if ((fp = fopen(argv[1], "r")) == NULL)
    {
        printf("Не удается открыть %s\n", argv[1]);
        exit(1);
    }
   while ((ch = getc(fp)) != EOF)
    {
        putc(ch,stdout);     // то же, что и putchar(ch);
        count++;
    }
    fclose(fp);
    printf("Файл %s содержит %ld символов\n", argv[1], count);
    return 0;
}
```

Проверка наличия аргументов командной строки

Прежде всего, программа, представленная в листинге 13.1, проверяет значение `argc`, выясняя, указан ли аргумент командной строки. Если такой аргумент не задан, программа выводит соответствующее сообщение и завершает работу. Строка `argv[0]` содержит имя программы. Явное использование `argv[0]` вместо имени программы приводит к автоматическому изменению сообщения об ошибке, если вы измените имя исполняемого файла. Это свойство полезно в таких операционных системах, как Unix, которые позволяют использовать множество имен для одного и того же файла. Однако будьте осторожны, поскольку некоторые операционные системы, например, предшествующие MS-DOS 3.0, не распознают `argv[0]`, следовательно, данную возможность нельзя назвать полностью переносимой.

Функция `exit()` вызывает завершение программы и закрытие всех открытых файлов. Аргумент функции `exit()` передается некоторым операционным системам, в числе которых Unix, Linux и MS-DOS, где он может использоваться другими программами. Традиционное соглашение заключается в том, что `0` возвращается в случае успешного завершения, а ненулевое значение — в случае аварийного завершения. Различные выходные значения могут быть использованы для обозначения различных

причин аварийного завершения программы, и это стало установившейся практикой программирования в операционных системах Unix и DOS. В то же время, не во всех операционных системах принят один и тот же диапазон возможных возвращаемых значений. В силу этого стандарт ANSI C рекомендует использовать ограниченный минимальный диапазон. В частности, стандарт требует, чтобы значение `0` или макрос `EXIT_SUCCESS` применялись при успешном завершении программы, а макрос `EXIT_FAILURE` служил для указания причины неудачного завершения. Эти макросы, наряду с прототипом функции `exit()`, можно найти в заголовочном файле `stdlib.h`. В данной книге соблюдаются общепринятые правила использования целочисленных выходных значений, в то же время для достижения максимальной переносимости применяются макросы `EXIT_SUCCESS` и `EXIT_FAILURE`.

По условиям стандарта ANSI C действие оператора `return` в исходном вызове функции `main()` приводит к тому же результату, что и вызов функции `exit()`. По этой причине, в функции `main()` оператор

```
return 0;
```

который использовался на протяжении всей этой книги, эквивалентен оператору:

```
exit(0);
```

Однако следует обратить внимание на определяющую фразу "исходный вызов". Если вы используете функцию `main()` в рекурсивной программе, функция `exit()` так или иначе, завершит ее работу, однако оператор `return` передает управление на предыдущий уровень рекурсии, пока не будет достигнут исходный уровень. Затем `return` завершает выполнение программы. Другое различие между оператором `return` и функцией `exit()` состоит в том, что `exit()` завершает выполнение программы, даже будучи вызванной из функции, отличной от `main()`.

Функция `fopen()`

Далее рассматриваемая программа использует функцию `fopen()`, чтобы открыть файл. Эта функция объявлена в заголовочном файле `stdio.h`. Ее первым аргументом является имя файла, который необходимо открыть; точнее, это адрес строки, содержащей имя файла. Второй аргумент — это строка, определяющая режим, в котором файл должен быть открыт. Как следует из табл. 13.1, библиотека C поддерживает несколько таких режимов.

Для таких операционных систем, как Unix и Linux, поддерживающих только один тип файла, режимы, обозначенные буквой `b` эквивалентны соответствующим режимам, в обозначении которых буква `b` отсутствует.

Внимание!

Если вы используете для существующих файлов какой-либо из режимов, в обозначении которого присутствует буква `"w"`, содержимое файла усекается так, что ваша программа начнет работу с пустым файлом.

После того, как программа успешно откроет файл, функция `fopen()` возвратит *указатель на файл*, который другие функции ввода-вывода могут затем использовать для ссылок на этот файл. Указатель на файл (в рассматриваемом примере это `fp`) имеет тип "указатель на `FILE`"; при этом `FILE` — это производный тип, определенный в `stdio.h`.

Таблица 13.1. Строки, задающие режим выполнения для функции `fopen()`

Строка режима	*Описание*
`"r"`	Открыть текстовый файл для чтения.
`"w"`	Открыть текстовый файл для записи, усекая длину существующего файла до нуля, либо создавая файл, если он не существует.
`"a"`	Открыть текстовый файл для записи, добавляя данные в конец существующего файла, либо создавая файл, если он еще не существует.
`"r+"`	Открыть текстовый файл для обновления (то есть для чтения и записи).
`"w+"`	Открыть текстовый файл для обновления (то есть для чтения и записи), выполнив сначала усечение файла до нулевой длины, если он существует, либо создавая файл, если его еще нет.
`"a+"`	Открыть текстовый файл для обновления (то есть, для чтения и записи), добавляя данные в конец существующего файла, либо создавая файл, если он еще не существует, при этом можно читать весь файл, однако запись добавляется в конец файла.
`"rb"`, `"wb"`, `"ab"`, `ab+"`, `"a+b"`, `"wb+"`, `"w+b"`, `"ab+"`, `"a+b"`	Эти режимы подобны описанным выше, но с одним отличием — вместо текстового режима доступа используется двоичный режим.

Указатель `fp` не указывает на фактический файл. Вместо этого он указывает на пакет, содержащий информацию о файле, в том числе информацию о буфере, используемом в операциях ввода-вывода этого файла. Поскольку функции ввода-вывода из стандартной библиотеки используют буфер, они должны знать, где этот буфер находится. Они также должны знать, насколько заполнен буфер и какой файл используется. Это позволяет функциям заполнять или опустошать буферы по мере необходимости. Пакет данных, на которые указывает `fp`, содержит всю эту информацию. (Такой пакет данных является примером структуры C; эта тема будет обсуждаться в главе 14.)

Функция `fopen()` возвращает нулевой указатель (который также определен в заголовочном файле `stdio.h`), если она не может открыть файл. Программа прекращает работу, если указатель `fp` равен `NULL`. Выполнение функции `fopen()` может завершиться неудачно по причине переполнения диска, из-за того, что имя файла указано неправильно, из-за ограничений доступа или из-за сбоев оборудования, и это только небольшая часть возможных причин неудач подобного рода. Таким образом, необходим контроль неисправностей; даже минимальные меры по устранению ошибок могут дать хорошие результаты.

Функции `getc()` и `putc()`

Функции `getc()` и `putc()` работают почти так же, как и функции `getchar()` и `putchar()`. Различие состоит в том, что вы должны указать новым функциям, какой файл следует использовать. Приведенный ниже многократно использованный нами оператор означает “получить символ из стандартного ввода”:

```
ch = getchar();
```

В то же время этот оператор означает "получить символ из файла, на который указывает fp":

```
ch = getc(fp);
```

Аналогично, следующий оператор означает "поместить символ ch в файл, на который ссылается указатель fpout типа FILE":

```
putc(ch, fpout);
```

В списке аргументов функции putc() символ задается первым, а затем идет указатель на файл.

В листинге 13.1 stdout используется в качестве второго аргумента функции putc(). В заголовочном файле stdio.h этот аргумент определен как файловый указатель, ассоциированный со стандартным выводом, таким образом, выражение putc(ch,stdout) эквивалентно putchar(ch). По сути дела, последняя функция обычно определена как первая. Аналогично, функция getchar() определена как getc(), использующая стандартный ввод.

У вас, по-видимому, возник вопрос, почему в этом примере используется функция putc() вместо putchar()? Одна из причин этого связана с необходимостью ознакомления с функцией putc(). Другая причина состоит в том, что вы легко можете изменить эту программу так, что она сможет пересылать выходные данные в файл, используя для этой цели аргумент, отличный от stdout.

Признак конца файла

Программа, читающая данные из файла, должна остановиться, как только достигнет конца файла. Как может программа сообщить о том, что она достигла конца файла? Функция getc() возвращает специальное значение EOF, когда предпринимается попытка прочитать символ и при этом обнаруживается, что достигнут конец файла. Следовательно, программа на C обнаруживает, что она достигла конца файла, только после того, как попытается выполнить чтение символов за концом файла. (Это не соответствует поведению некоторых языков, которые используют специальную функцию для проверки на наличие признака конца файла *перед* собственно чтением.)

Чтобы избежать проблемы чтения пустого файла, при файловом вводе необходимо использовать цикл с входной проверкой условия (не цикл do while). Ввиду конструктивных особенностей функции getc() (и других функций ввода языка C), программа должна выполнить чтение до входа в тело цикла. Следовательно, показанная ниже конструкция вполне подойдет:

```
// правильная конструкция #1
int ch;                    // значение типа int для хранения символа EOF
FILE * fp;
fp = fopen("wacky.txt", "r");
ch = getc(fp);             // получить исходный ввод
while (ch != EOF)
{
    putchar(ch);           // выполнение ввода
    ch = getc(fp);         // получить следующий ввод
}
```

Этот код можно ужать до следующей конструкции:

```
// правильная конструкция  #2
int ch;
FILE * fp;
fp = fopen("wacky.txt", "r");
while (( ch = getc(fp)) != EOF)
{
    putchar(ch);            // выполнение ввода
}
```

Поскольку оператор ввода является частью условия проверки цикла `while`, он выполняется до того, как программа войдет в тело цикла. Конструкций показанного ниже типа следует избегать:

```
// неправильна конструкция (две проблемы)
int ch;
FILE * fp;
fp = fopen("wacky.txt", "r");
while (ch != EOF)          // первое использование неопределенного
                           // значения ch
{
   ch = getc(fp);          // получить ввод
   putchar(ch);            // выполнение ввода
}
```

Одна из проблем заключается в том, что когда переменная `ch` в первый раз сравнивается с признаком `EOF`, ее значение не определено. Еще одна проблема состоит в том, что если функция `getc()` возвращает `EOF`, то цикл пытается обработать `EOF`, как если бы он был допустимым символом. Эти недостатки можно исправить. Например, вы можете инициализировать переменную `ch` некоторым фиктивным значением и поместить оператор `if` внутрь цикла, но какой в этом смысл, если существуют правильные программные структуры?

Эти меры предосторожности касаются и других функций ввода. Они также возвращают сигнал ошибки (`EOF` либо пустой указатель) по достижении конца файла.

Функция `fclose()`

Функция `fclose(fp)` закрывает файл, идентифицированный с помощью `fp`, очищая при этом буферы в случае необходимости. В более ответственных программах вы должны убедиться в том, что файл успешно закрыт. Функция `fclose()` возвращает значение `0`, если операция закрытия файла завершилась успешно, и `EOF` в противном случае:

```
if (fclose(fp) != 0)
   printf("Ошибка при закрытии файла %s\n", argv[1]);
```

Выполнение функции `fclose()` может завершиться неудачно, если, например, жесткий диск заполнен, гибкий диск изъят из привода или произошла ошибка ввода-вывода.

Указатели на стандартные файлы

Заголовочный файл stdio.h связывает три указателя на файлы с стремя стандартными файлами, которые автоматически открываются программами на C:

Стандартный файл	*Указатель файла*	*Обычное устройство*
Стандартный ввод	stdin	Клавиатура
Стандартный вывод	stdout	Экран
Стандартные ошибки	stderr	Экран

Все эти указатели имеют тип указателя на FILE, следовательно, они могут использоваться как аргументы стандартных функций ввода-вывода, таким указателем был fp в рассмотренном примере. Теперь обратимся к примеру, в рамках которого создается новый файл и производится запись в этот файл.

Простая программа сжатия файлов

Рассматриваемая программа копирует избранные данные из одного файла в другой. Она одновременно открывает два файла, используя режим "r" для одного и режим "w" для другого. Программа (показанная в листинге 13.2) уменьшает содержимое первого файла путем безжалостного удаления каждого первого и второго символов, оставляя только каждый третий символ. В завершение она помещает сжатый текст во второй файл. Имя второго файла представляет собой прежнее имя с расширением .red (что означает “reduced”, то есть “сжатый”). Использование аргументов командной строки, открытие одновременно более одного файла и расширение имени файла путем добавления в его конец символов .red в общем случае представляют собой довольно полезную технологию. Эта конкретная форма сжатия файла имеет ограниченное применение, но, как вы сможете убедиться в дальнейшем, оно существует. (Опять-таки, модификация этой программы для использования стандартных методов ввода-вывода вместо аргументов командной строки для указания имен файлов, не представляет особых трудностей.)

Листинг 13.2. Программа reducto.c

```
// reducto.c -- сжимает ваши файлы до одной трети первоначальных размеров!
#include <stdio.h>
#include <stdlib.h>                 // для функции exit()
#include <string.h>                 // для функций strcpy(), strcat()
#define LEN 40
int main(int argc, char *argv[])
{
    FILE *in, *out;                 // объявление двух указателей на FILE
    int ch;
    char name[LEN];                 // хранилище для имени выходного файла
    int count = 0;

// проверка аргументов командной строки
    if (argc < 2)
    {
        fprintf(stderr, "Использование: %s filename\n", argv[0]);
        exit(1);
    }
```

```
// настройка ввода
    if ((in = fopen(argv[1], "r")) == NULL)
    {
        fprintf(stderr, "Не удается открыть файл \"%s\"\n",
                argv[1]);
        exit(2);
    }
// настройка вывода
    strncpy(name,argv[1], LEN - 5);        // копирование имени файла
    name[LEN - 5] = '\0';
    strcat(name,".red");                   // добавить суффикса .red
    if ((out = fopen(name, "w")) == NULL)
    {                                      // открыть файл для записи
        fprintf(stderr,"Не могу построить выходной файл.\n");
        exit(3);
    }
// копирование данных
    while ((ch = getc(in)) != EOF)
        if (count++ % 3 == 0)
            putc(ch, out);               // печатать каждый третий символ
// очистка
   if (fclose(in) != 0 || fclose(out) != 0)
       fprintf(stderr,"Ошибка при закрытии файла\n");
   return 0;
}
```

Исполняемый файл называется `reducto`. Мы применим его к файлу под названием `eddy`, который содержит следующую единственную строку:

```
So even Eddy came oven ready.
```

Команда имеет такой вид:

```
reducto eddy
```

Выходные данные записывались в файл с именем `eddy.red`. Эта программа ничего не выводит на экран, однако отображение содержимого файла `eddy.red` позволяет обнаружить следующее:

```
Send money
```

Этот пример служит иллюстрацией нескольких методов программирования. Рассмотрим теперь некоторые из них.

Функция `fprintf()` во многом подобна функции `printf()`, за исключением того, что она требует в качестве первого аргумента указатель на файл. Мы использовали указатель `stderr` для пересылки сообщений об ошибках в стандартный вывод ошибок; в среде языка C это обычная практика программирования.

Чтобы сформировать новое имя для выходного файла, программа использует функцию `strncpy()` для копирования имени `eddy` в массив `name`. Аргумент `LEN - 5` оставляет место для суффикса `.red` и завершающего нулевого символа. Нулевой символ не копируется, если строка `argv[2]` длиннее, чем `LEN - 5`, по этой причине на всякий случай добавляется нулевой символ. Первый нулевой символ в массиве после вызова

функции strncpy() перезаписан точкой из расширения .red, когда функция strcat() добавляет эту строку, в результате в рассматриваемом случае получаем eddy.red. Мы также провели проверку того, что программе удалось открыть файл с таким именем. Это обстоятельство имеет особую важность в некоторых операционных системах, поскольку имя файла, подобное, скажем, strange.c.red, может оказаться недопустимым. Например, вы не можете добавлять расширение к расширению в среде DOS. (Правильный подход в системе MS-DOS состоит в том, чтобы заменить любое расширение на .red, так что преобразованная версия strange.c будет выглядеть как strange.red. Вы могли бы, например, воспользоваться функцией strchr(), чтобы определить местонахождение точки в имени, если таковая имеется, и копировать только часть строки, расположенную перед точкой.)

В данной программе открываются два файла одновременно, по этой причине объявлены два указателя на FILE. Обратите внимание, что каждый из этих файлов открывается и закрывается независимо от другого. Существует ограничение на количество одновременно открытых файлов. Ограничение зависит от типа системы и приложения, они обычно принимают значения в диапазоне от 10 до 20. Вы можете использовать один и тот же указатель файла применительно к различным файлам при условии, что эти файлы не открыты в один и тот же момент.

Функции ввода-вывода: fprintf(), fscanf(), fgets() и fputs()

Для каждой функции ввода-вывода, использованной в предыдущих главах, существует файловая функция ввода-вывода. Основное различие между ними состоит в том, что необходимо с помощью указателя на FILE сообщать новым функциям, с каким файлом нужно работать. Подобно getc() и putc(), эти функции требуют, чтобы вы определили для них файл посредством указателя на FILE, такого как, например, stdout, либо чтобы вы использовали возвращаемое значение функции fopen().

Функции fprintf() и fscanf()

Функции ввода-вывода fprintf() и fscanf() работают примерно так же, как и функции printf() и scanf(), отличаясь от них только существованием дополнительного первого аргумента для идентификации соответствующего файла. Ранее вы уже использовали функцию fprintf(). Листинг 13.3 служит иллюстрацией применения этих функций файлового ввода-вывода, а также функции rewind().

Листинг 13.3. Программа addaword.c

```
/* addaword.c -- использование функций fprintf(), fscanf() и rewind() */
#include <stdio.h>
#include <stdlib.h>
#define MAX 40

int main(void)
{
    FILE *fp;
    char words[MAX];
```

```
    if ((fp = fopen("wordy", "a+")) == NULL)
    {
       fprintf(stdout,"Не удается открыть файл \"words\".\n");
       exit(1);
    }
    puts("Введите слова, которые нужно включить в файл; нажмите клавишу Enter");
    puts("в начале строки для завершения программы.");
    while (gets(words) != NULL && words[0] != '\0')
       fprintf(fp, "%s ", words);
    puts("Содержимое файла:");
    rewind(fp);                          /* вернуться в начало файла */
    while (fscanf(fp, "%s",words) == 1)
       puts(words);
    if (fclose(fp) != 0)
        fprintf(stderr,"Ошибка при закрытии файла\n");
    return 0;
}
```

Эта программа позволяет добавлять слова в файл. Используя режим "a+", данная программа может читать и записывать в файл. Когда программа запускается в первый раз, она создает файл и обеспечивает возможность помещать в него слова. При использовании программы в следующий раз, она позволит добавлять (дописывать в конец) слова к предшествующему содержимому. Режим добавления позволяет добавлять материал в конец файла, а режим "a+" позволяет читать весь файл. Команда `rewind()` перемещает в начало файла, так что заключительный цикл `while` может распечатать содержимое файла. Обратите внимание, что функция `rewind()` принимает в качестве аргумента указатель на файл. Если вы введете пустую строку, функция `gets()` поместит нулевой символ в первый элемент массива. Программа использует этот факт для завершения цикла.

Ниже представлены результаты выполнения программы в среде DOS:

```
C>addaword
Введите слова, которые нужно включить в файл; нажмите клавишу Enter в
начале строки для завершения программы.
Опытный программист[enter]
[enter]
Содержимое файла:
Опытный
программист
C>addaword
Введите слова, которые нужно включить в файл; нажмите клавишу Enter в
начале строки для завершения программы.
очаровал[enter]
публику[enter]
[enter]
Содержимое файла:
Опытный
программист
очаровал
публику
```

Нетрудно заметить, что функции fprintf() и fscanf() работают аналогично функциям printf() и scanf(). В отличие от putc(), функции fprintf() и fscanf() принимают указатель на FILE в качестве первого, а не последнего аргумента.

Функции fgets() и fputs()

Вы сталкивались с функцией fgets() в главе 11. В отличие от одного аргумента функции gets(), fgets() принимает три аргумента. Первый аргумент, как и в случае gets(), представляет собой адрес (тип char *), по которому нужно сохранять ввод. Второй аргумент — целое число, определяющее максимальный размер входной строки. Заключительный аргумент — это указатель на читаемый файл:

```
fgets(buf, MAX, fp);
```

В данном случае buf — имя массива значений типа char, MAX — максимальный размер строки, а fp — указатель на FILE. Функция fgets() читает входные данные до первого символа новой строки, пока не будет прочитано количество символов, на единицу меньшее верхнего предела, либо пока не будет найден признак конца файла; функция fgets() затем добавляет в строку завершающий нулевой символ. Поэтому внешний предел представляет собой максимальное количество символов плюс нулевой символ. Если функция fgets() считывает целую строку, прежде чем будет достигнуто предельное количество символов, она добавит символ новой строки непосредственно перед нулевым символом, тем самым помечая конец строки. В этом она отличается от функции gets(), которая читает символ новой строки, но отбрасывает его за ненадобностью.

Как и gets(), функция fgets() возвращает значение NULL, когда встречает признак EOF. Вы можете воспользоваться этим обстоятельством для проверки конца файла. В противном случае она возвращает адрес, который ей передан.

Функция fputs() принимает два аргумента: первый — адрес строки и второй — указатель на файл. Она записывает строку, обнаруженную в указанной ячейке, в заданный файл. В отличие от puts(), функция fputs() при выводе не добавляет символ новой строки. Вызов функции выглядит следующим образом:

```
fputs(buf, fp);
```

В рассматриваемом случае buf является адресом строки, а fp идентифицирует целевой файл.

Поскольку функция fgets() сохраняет символ новой строки, а функция fputs() не добавляет этот символ, они достаточно хорошо работают в тандеме. В листинге 13.4 показана программа эхо-вывода, в которой используются обе эти функции.

Листинг 13.4. Программа parrot.c

```
/* parrot.c -- использование функций fgets() и fputs() */
#include <stdio.h>
#define MAXLINE 20
int main(void)
{
    char line[MAXLINE];
    while (fgets(line, MAXLINE, stdin) != NULL && line[0] != '\n')
       fputs(line, stdout);
    return 0;
}
```

Когда вы нажимаете клавишу Enter в начале строки, функция fgets() читает символ новой строки и помещает его в первый элемент строки массива. Этот факт используется для завершения цикла ввода. Появление признака конца файла также прерывает его выполнение. (В листинге 13.3 выполняется проверка на символ '\0' вместо '\n', поскольку функция gets() игнорирует символ новой строки.) Ниже показан пример выполнения этой программы. Вы не заметили ничего странного?

```
Князь тишины
Князь тишины
торжественно вошел в сырой и темный зал.
торжественно вошел в сырой и темный зал.
[enter]
```

Программа работает без сбоев. Это может показаться удивительным, так как вторая вводимая строка содержит 40 символов, а массив строк — только 20 элементов, в том числе и символ новой строки! Что произошло? Когда функция fgets() читает вторую строку, она читает всего лишь первые 19 символов, до второго пробела с начала строки включительно. Они копируются в строку, которую печатает функция fputs(). Поскольку функция fgets() не достигла конца строки, строка не содержит символа новой строки, следовательно, fputs() не печатает символ новой строки. Третий вызов функции fgets() возобновляет считывание там, где оно было завершено вторым вызовом. При этом она считывает следующие 19 символов в массив line, начиная со слова *в*. Этот блок заменяет предыдущее содержимое массива line и, в свою очередь, распечатывается в той же строке, что и предыдущий вывод. Вспомните, что последний вывод не содержал символа новой строки. Короче говоря, функция fgets() читает вторую строку порциями по 19 символов, а функция fputs() выводит данные такими же порциями.

Эта программа также прекращает ввод, если в строке содержится в точности 19 символов. В таком случае функция fgets() завершает считывание после ввода 19 символов, следовательно, следующее обращение к функции fgets() начинается с символа новой строки, находящегося в конце строки. Этот символ новой строки становится первым прочитанным символом, тем самым, прерывая цикл. Таким образом, даже если программа каждый раз принимает на вход один и тот же образец входных данных, она не работает правильно во всех случаях. Вы должны использовать массив памяти, достаточно большой, чтобы принять все строки, в противном случае потребуется воспользоваться более простым методом чтения по одному символу за раз.

Вас, по-видимому, интересует, почему программа не распечатала первые 19 символов второй строки непосредственно после их ввода с клавиатуры. Именно в этот момент вступает в силу буферизация. Вторая строка не была отображена на экран до тех пор, пока не был получен символ новой строки.

Комментарий: функции gets() и fgets()

Поскольку fgets() может применяться для предотвращения переполнения памяти, эта функции в большей степени подходит для серьезного программирования, чем gets(). Так как она не считывает символ новой строки и в связи с тем, что функция puts() добавляет символ новой строки в конец вывода, с функцией fgets() должна использоваться fputs(), а не puts(). Во всех других случаях один символ новой строки при вводе вызывает появление двух таких символов в выводе.

Шесть функций ввода-вывода, которые мы только что рассмотрели, предоставляют вам достаточные инструментальные средства, чтобы читать и записывать текстовые файлы. До сих пор мы использовали их только для последовательного доступа, то есть для обработки в порядке следования. Далее мы рассмотрим произвольный доступ, другими словами, доступ в том порядке, какой вам нужен.

Произвольный доступ: функции `fseek()` и `ftell()`

Функция `fseek()` предоставляет возможность рассматривать файл как массив и осуществлять доступ непосредственно к конкретному байту файла, открытого с помощью функции `fopen()`. Чтобы ознакомиться с тем, как она работает, напишем программу (см. листинг 13.5), которая отображает файл в обратном порядке. Из ранее рассмотренных примеров она использует аргумент командной сроки для получения имени файла, который она будет читать. Обратите внимание, что функция `fseek()` получает три аргумента и возвращает значение типа `int`. Функция `ftell()` возвращает текущую позицию в файле в виде значения типа `long`.

Листинг 13.5. Программа `reverse.c`

```
/* reverse.c -- отображение файла в обратном порядке */
#include <stdio.h>
#include <stdlib.h>
#define CNTL_Z '\032'       /* маркер конца файла в текстовых файлах DOS */
#define SLEN 50
int main(void)
{
    char file[SLEN];
    char ch;
    FILE *fp;
    long count, last;

    puts("Введите имя файла для обработки:");
    gets(file);

    if ((fp = fopen(file,"rb")) == NULL)
    {                       /* режим только чтения и двоичный режим */
        printf("Программа reverse не может открыть %s\n", file);
        exit(1);
    }

    fseek(fp, 0L, SEEK_END);            /* перейти в конец файла */
    last = ftell(fp);

    for (count = 1L; count <= last; count++)
    {
        fseek(fp, -count, SEEK_END);    /* вернуться */
        ch = getc(fp);

    /* для DOS, работает в Unix */
        if (ch != CNTL_Z && ch != '\r')
            putchar(ch);
```

```
    /* для Macintosh */
    /* if (ch == '\r')
           putchar('\n');
       else
          putchar(ch); */
    }
    putchar('\n');
    fclose(fp);
    return 0;
}
```

Ниже приводятся результаты выполнения этой программы применительно к простому файлу:

```
Введите имя файла для обработки:
Cluv
.C ni eno naht ylevol erom margorp a
ees reven llahs I taht kniht I
```

На заметку!

Если вы выполняете программу в среде командной строки, то эта команда ожидает, что файл с указанным именем находится в том же каталоге (или папке), что и сама исполняемая программа. Если программа выполняете в среде IDE (Integrated Development Environment — интегрированная среда разработки), то, к какому каталогу обращается программа, зависит от реализации. Например, Microsoft Visual C++ обращается к каталогу, содержащему исходных код, в то же время Metrowerks CodeWarrior — к каталогу, содержащему исполняемый файл.

Теперь нам нужно обсудить три темы: как работают функции `fseek()` и `ftell()`, как используется двоичный поток и как сделать программу переносимой.

Как работают функции `fseek()` и `ftell()`

Первым из трех аргументов функции `fseek()` идет указатель на `FILE` для файла, в которм будет выполняться поиск. К моменту вызова функции `fopen()` этот файл должен быть открыт.

Второй аргумент функции `fseek()` называется *смещением*. Этот аргумент показывает, насколько необходимо переместиться относительно отправной точки (см. приведенный ниже список отправных точек режимов). Этим аргументом должно быть значение типа `long`. Оно может быть положительным (движение вперед), отрицательным (движение назад) или нулевым (оставаться на месте) значением.

Третий аргумент задает режим, идентифицирующий отправную точку. В условиях стандарта ANSI заголовочный файл `stdio.h` определяет следующие именованные константы для этого режима:

Режим	*Измеряет смещение от*
`SEEK_SET`	начала файла
`SEEK_CUR`	текущей позиции
`SEEK_END`	конца файла

Прежние реализации могут не иметь таких определений и вместо них для представления этих режимов использовать числовые значения 0L, 1L и 2L, соответственно. Напомним, что суффикс L обозначает тип значений long. Возможно также, что эта реализация может использовать константы, определения которых содержатся в другом заголовочном файле. Когда возникают какие-то сомнения, обращайтесь к руководству пользователя или к соответствующему онлайновому справочнику.

Ниже показано несколько примеров вызова функций, в которых fp является указателем файла:

```
fseek(fp, 0L, SEEK_SET);   // перейти в начало файла
fseek(fp, 10L, SEEK_SET);  // сместиться на 10 байтов от начала файла
fseek(fp, 2L, SEEK_CUR);   // сместиться на 2 байта от текущей позиции
fseek(fp, 0L, SEEK_END);   // перейти в конец файла
fseek(fp, -10L, SEEK_END); // сместиться на 10 байтов от конца файла к началу
```

Возможны некоторые ограничения на такие вызовы; вы вернемся к этой теме чуть позже.

Значение, возвращаемое функцией fseek() равно 0, если все в порядке, и -1, если имела место ошибка, такая как выход за границы файла.

Функция ftell() имеет тип long, она возвращает текущую позицию в файле. По условиям ANSI она объявлена в заголовочном файле stdio.h. Первоначальная реализация функции ftell() в Unix определяла позицию символа в файле, возвращая количество байтов от начала файла, при этом первый байт получал номер 0 и так далее. В языке ANSI C это определение применимо к файлам, открытым в двоичном режиме, но не обязательно к файлам, открытым в текстовом режиме. Это одна из причин того, что программа, показанная в листинге 13.5, использует двоичный режим.

Теперь мы можем изучить базовые элементы программы из листинга 13.5. Прежде всего, оператор

```
fseek(fp, 0L, SEEK_END);
```

определяет позицию со смещением в 0 байтов от конца файла. Иначе говоря, он определяет позицию в конце файла. Далее, оператор

```
last = ftell(fp);
```

помещает в переменную last количество байтов от начала до конца указанного файла.

Далее идет следующий цикл:

```
for (count = 1L; count <= last; count++)
{
   fseek(fp, -count, SEEK_END);          /* вернуться назад */
      ch = getc(fp);
}
```

Первое выполнение цикла выводит программу на первый символ перед концом файла (то есть на завершающий символ файла). Затем программа печатает этот символ. Следующий цикл выводит программу на предпоследний символ файла, который она печатает. Этот процесс продолжается до тех пор, пока программа не выйдет на первый символ файла и не распечатает его.

Сравнение двоичного и текстового режимов

Мы разрабатывали программу, представленную в листинге 13.5, с таким расчетом, чтобы она работала в средах Unix и MS-DOS. В системе Unix имеется только один формат, поэтому никакие настройки в этом случае не нужны. Однако система MS-DOS требует дополнительного внимания. Многие редакторы MS-DOS отмечают конец текстового файла посредством символа Ctrl+Z. Когда такой файл открывается в текстовом режиме, язык C трактует этот символ как признак конца файла. Однако когда тот же файл открывается в двоичном режиме, символ Ctrl+Z рассматривается как обычный символ файла, а фактический признак конца файла появляется позже. Он может появиться сразу после символа Ctrl+Z, либо файл может быть дополнен нулевыми символами, количество которых позволяет получить размер файла, кратный, скажем, 256. Нулевые символы в среде DOS не печатаются, поэтому мы включили код, не позволяющий печатать символ Ctrl+Z.

О другом отличии уже говорилось выше: система MS-DOS представляет символ новой строки текстового файла в виде комбинации `\r\n`. Программа на языке C, открывающая тот же файл в текстовом режиме, "видит" комбинацию `\r\n` как просто `\n`, однако, используя двоичный режим, программа видит оба символа. По этой причине был предусмотрен код, который подавляет печать символа `\r`. (Для текстовых файлов в системе Macintosh необходим другой код, поскольку они в качестве маркера конца строки используют `\r`. В листинге 13.5 версия для системы Macintosh взята в комментарий.)

Поскольку текстовый файл Unix обычно не содержит ни символа Ctrl+Z, ни `\r`, этот дополнительный программный код не затрагивает большей части текстовых файлов в среде Unix.

Функция `ftell()` может работать по-разному в текстовом и двоичном режимах. Во многих системах имеются текстовые форматы, заметно отличающиеся от модели Unix, в которых отсчет байтов от начала файла не дает сколько-нибудь осмысленное значение. Стандарт ANSI C утверждает, что в текстовом режиме функция `ftell()` возвращает значение, которое может быть использовано как второй аргумент функции `fseek()`. Например, в среде MS-DOS функция `ftell()` может возвратить итоговое количество байтов, при котором комбинация `\r\n` рассматривается как один байт.

Переносимость

В идеальном случае функции `fseek()` и `ftell()` должны соответствовать модели Unix. В то же время, различия реальных систем делает это в некоторых случаях невозможным. По этой причине ANSI предусматривает заниженные ожидания от использования этих функций. Ниже описаны некоторые из ограничений.

- В двоичном режиме реализации не обязательно должны поддерживать режим `SEEK_END`. Поэтому нет никаких гарантий того, что программа из листинга 13.5 окажется переносимой. Тем не менее, этот листинг демонстрирует альтернативный метод поиска конца файла. Поскольку альтернативный метод последовательно считывает весь файл до конца, то это гораздо медленнее, чем сразу перейти в конец файла. Директивы условной компиляции препроцессора C, обсуждаемые в главе 16, обеспечивают более систематизированный способ альтернативного кодирования.

- В текстовом режиме гарантированно работающими являются только следующие вызовы функции `fseek()`:

Функция	*Результат вызова*
`fseek(file, 0L, SEEK_SET)`	Перейти в начало файла.
`fseek(file, 0L, SEEK_CUR)`	Оставаться в текущей позиции.
`fseek(file, 0L, SEEK_END)`	Перейти в конец файла.
`fseek(file, ftell-pos, SEEK_SET)`	Перейти в позицию `ftell-pos` от начала файла; `ftell-pos` — значение, возвращаемое функцией `ftell()`.

К счастью, многие широко используемые операционные среды позволяют в полной мере воспользоваться возможностями этих функций.

Функции `fgetpos()` и `fsetpos()`

Одна потенциальная проблема, связанная с функциями `fseek()` и `ftell()`, заключается в том, что они ограничивают размер файла до значений, которые могут быть представлены типом `long`. Возможно, двух миллиардов байтов покажется более чем достаточно, тем не менее, постоянно расширяющиеся возможности запоминающих устройств позволяют работать с файлами и больших размеров. Стандарт ANSI C вводит две новых функции позиционирования, которые ориентированы на работу с фалами крупных размеров. Вместо того чтобы использовать значения типа `long` для представления позиции, применяется новый тип, получивший имя `fpos_t` (для типа, представляющего позицию конкретного байта). Тип `fpos_t` не является фундаментальным, скорее, он определяется через другие типы. Переменная или объект данных типа `fpos_t` может определять позицию внутри файла, он не может быть массивом, что, впрочем, характерно его природе. Реализации могут предложить собственные типы, соответствующие потребностям той или иной платформы; такие типы могут быть, например, реализованы в виде структур.

Стандарт ANSI C регламентирует использование типа `fpos_t`. Функция `fgetpos()` имеет следующий прототип:

```
int fgetpos(FILE * restrict stream, fpos_t * restrict pos);
```

В результате вызова функция `fgetpos()` помещает значение `fpos_t` в ячейку, указываемую `pos`; это значение описывает позицию внутри файла. Функция возвращает ноль в случае успешного выполнения и ненулевое значение в случае неудачи.

Функция `fsetpos()` имеет следующий прототип:

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

В результате вызова функция использует значение типа `fpos_t`, содержащееся в ячейке, адрес которой хранится в переменной `pos`, чтобы сформировать указатель относительно начала файла на ячейку, на которую указывает это значение. Функция возвращает ноль в случае успешного выполнения и ненулевое значение в случае неудачи. Значение `fpos_t` должно было быть получено в предыдущем вызове функции `fgetpos()`.

За кулисами стандартного ввода-вывода

Теперь, когда вы ознакомились с некоторыми особенностями стандартного пакета ввода-вывода, рассмотрим концептуальную его модель, чтобы знать, как работает стандартный ввод-вывод.

Обычно первым шагом в использовании стандартного ввода-вывода является вызов функции `fopen()` для открытия конкретного файла. (С другой стороны, файлы `stdin`, `stdout` и `stderr` открываются автоматически.) Функция `fopen()` не только открывает файл, но и организует буфер (два буфера для режимов чтения-записи) и создает структуру данных, содержащую информацию о файле и о буфере. Кроме того, функция `fopen()` возвращает указатель на эту структуру, так что другие функции знают, где ее искать. Предположим, что это значение присвоено переменной типа указатель с именем `fp`. Говорят, что функция `fopen()` "открывает поток". Если файл открывается в текстовом режиме, вы получаете текстовый поток, и если в двоичном режиме — то двоичный поток.

Структура данных обычно включает индикатор позиции в файле, чтобы можно было определить текущую позицию в потоке. Она также содержит индикаторы ошибок и конца файла, указатель на начало буфера, идентификатор файла и счетчик количества байтов, фактически скопированных в буфер.

А теперь сконцентрируем все внимание на проблеме файлового ввода. Обычно следующий шаг заключается в вызове одной из функций ввода, объявленных в заголовочном файле `stdio.h`, таких как `fscanf()`, `getc()` или `fgets()`. Вызов одной из этих функций приводит к тому, что порция данных копируется из файла в буфер. Размер буфера зависит от реализации, но обычно он занимает 512 байтов или кратное этому числу, например, 4 096 или 16 384. (По мере того, как объемы жестких дисков и памяти компьютера возрастают, выбираемые размеры буферов также имеют тенденцию роста.) В дополнение к заполнению буфера первоначальный вызов функции устанавливает значения в структуре, на которую нацелен указатель `fp`. В частности, устанавливается текущая позиция в потоке и количество байтов, которые копируются в буфер. Обычно текущая позиция начинается с байта 0.

После того, как рассматриваемая структура данных и буфер будут инициализированы, функция ввода считывает запрашиваемые данные из буфера. По мере того, как она это делает, индикатор позиции файла устанавливается так, что он указывает на символ, следующий за последним считанным символом. Поскольку все функции ввода семейства `stdio.h` используют один и тот же буфер, обращение к любой из этих функций возобновляется там, где предыдущий вызов любой из этих функций завершил свою работу.

Когда функция ввода обнаруживает, что она прочитала все символы из буфера, она требует, чтобы следующая порция данных, равная размеру буфера, была скопирована в буфер из файла. Таким способом функции ввода могут читать все содержимое файла до самого конца. После того, как функция прочитает последний символ финальной порции данных, она устанавливает индикатор конца файла в `true`. Следующий вызов функции ввода возвращает маркер `EOF`.

Аналогичным образом функции вывода выполняют запись в буфер. Как только буфер оказывается заполненным, данные копируются в файл.

Другие стандартные функции ввода-вывода

Стандартная библиотека ANSI содержит более трех десятков функций, образующих семейство стандартных функций ввода-вывода. И хотя мы не сможем рассмотреть их все здесь, тем не менее, кратко опишем еще несколько таких функций, чтобы дать более четкое представление о том, что имеется в вашем распоряжении. Мы составим список всех этих функций в соответствии с их прототипами в ANSI C, что позволит изучить их аргументы и возвращаемые значения. Все функции, которые рассматриваются в данной главе, за исключением `setvbuf()`, доступны в реализациях, предшествующих ANSI. В разделе V справочника (приложение Б) представлен список содержимого пакета функций ввода-вывода ANSI C.

Функция `int ungetc(int c, FILE *fp)`

Функция `int ungetc()` заталкивает символ, заданный переменной `c`, обратно во входной поток. Если вы заталкиваете символ во входной поток, он будет читаться при следующем обращении к стандартной функции ввода, как показано на рис. 13.2. Предположим, например, что вам нужна функция, которая читает все символы до появления двоеточия, но не включая его. Вы можете воспользоваться функцией `getchar()` или `getc()` для чтения всех символов, пока не будет прочитано двоеточие, после чего вызвать `ungetc()`, чтобы поместить двоеточие обратно во входной поток. Стандарт ANSI C обеспечивает только одно заталкивание во входной поток за один раз. Если реализация позволяет затолкнуть сразу несколько символов в виде последовательности, функции ввода прочитают их в обратном порядке.

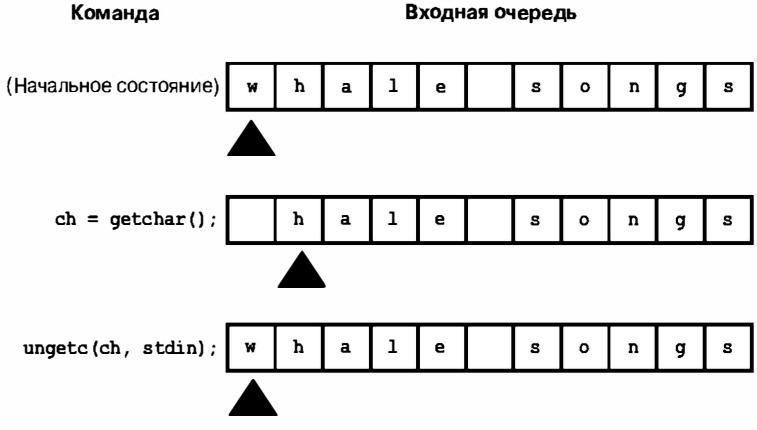


Рис. 13.2. *Функция* `ungetc()`

Функция `int fflush()`

Прототип функции `fflush()` имеет следующий вид:

```
int fflush(FILE *fp);
```

Вызов функции `fflush()` приводит к тому, что все несохраненные данные, содержащиеся в буфере, пересылаются в выходной файл, идентифицируемый указателем `fp`.

Этот процесс называется *очисткой буфера*. Если fp — пустой указатель, очищаются все выходные буферы. Результат использования функции fflush() применительно к входному потоку не определен. Вы можете использовать ее применительно к обновляемому потоку (любой из режимов чтения-записи), при условии, что самая последняя операция, использовавшая поток, не была операцией ввода.

Функция int setvbuf()

Прототип функции setvbuf() имеет следующий тип:

```
int setvbuf(FILE * restrict fp, char * restrict buf, int mode, size_t size);
```

Функция setvbuf() устанавливает альтернативный буфер, предназначенный для использования стандартными функциями ввода-вывода. Она вызывается после того, как файл уже открыт, и перед тем, как была выполнена какая-либо операция применительно к потоку. Указатель fp идентифицирует поток, а buf указывает на память, которая предназначена для использования. Если значение указателя не равно NULL, вы должны создать буфер. Например, вы можете построить массив из 1024 значений типа char и передать адрес этого массива. В то же время, если вы используете NULL в качестве значения указателя buf, эта функция самостоятельно распределит память под буфер. Переменная size сообщает функции setvbuf() размер этого массива. (Тип size_t — это производная целочисленная переменная; см. главу 5.) Режим можно выбрать из следующих вариантов: _IOFBF означает полную буферизацию (при заполнении буфер очищается), _IOLBF — построчную буферизацию (буфер очищается, как только в него будет записан символ новой строки) и _IONBF — отсутствие какой-либо буферизации. Функция возвращает ноль при успешном завершении и ненулевое значение в противном случае.

Предположим, что ваша программа работает с объектами данных, которые хранятся в памяти и имеют размер, скажем, 3 000 байтов каждый. Вы могли бы использовать функцию setvbuf() для создания буфера, размер которого соответствует размеру этого объекта данных.

Двоичный ввод-вывод: fread() и fwrite()

Следующими в этом списке идут функции fread() и fwrite(), однако сначала выясним некоторые вопросы. Стандартные функции ввода-вывода, которыми вы пользовались до сих пор, ориентированы на работу с текстом, имея дело с символами и строками. А что делать, если нужно сохранить данные в файле? Разумеется, можно воспользоваться функцией fprintf() и форматом %f, чтобы сохранить значение с плавающей точкой, но в этом случае вы сохраняете его как строку. Например, последовательность операторов

```
double num = 1./3.;
fprintf(fp,"%f", num);
```

сохраняет значение num как строку из восьми символов: 0.333333. Использование спецификатора %.2f позволяет сохранить его в виде последовательности из четырех символов: 0.33. Использование спецификатора %.12f дает возможность сохранить его в виде последовательности из 14 символов: 0.333333333333. Смена спецификаторов приводит к изменению объема пространства памяти, необходимого для хранения

конкретного значения. После того, как значение num было сохранено в формате 0.33, утрачивается возможность восстановить полную точность этого значения при чтении файла. В общем случае функция fprintf() преобразует числовое значение в строки, и при этом возможно изменение этого значения.

Наиболее точный и последовательный способ сохранения чисел заключается в том, чтобы сохранять их в той конфигурации битов, в какой их использует программа. В силу этого, значение типа double должно храниться в ячейке памяти размером double. Когда данные хранятся в файле в том же представлении, в каком их использует программа, мы говорим, что данные хранятся в *двоичной форме*. Перевод из числовой формы в строковую в этом случае не производится. В случае стандартного ввода-вывода функции fread() и fwrite() предоставляют такую услугу (рис. 13.3).

По сути дела, все данные хранятся в двоичной форме. Даже символы хранятся с использованием двоичного представления кода символа. В то же время, в случае, когда все данные файла интерпретируются как коды символов, мы говорим, что файл содержит текстовые данные.

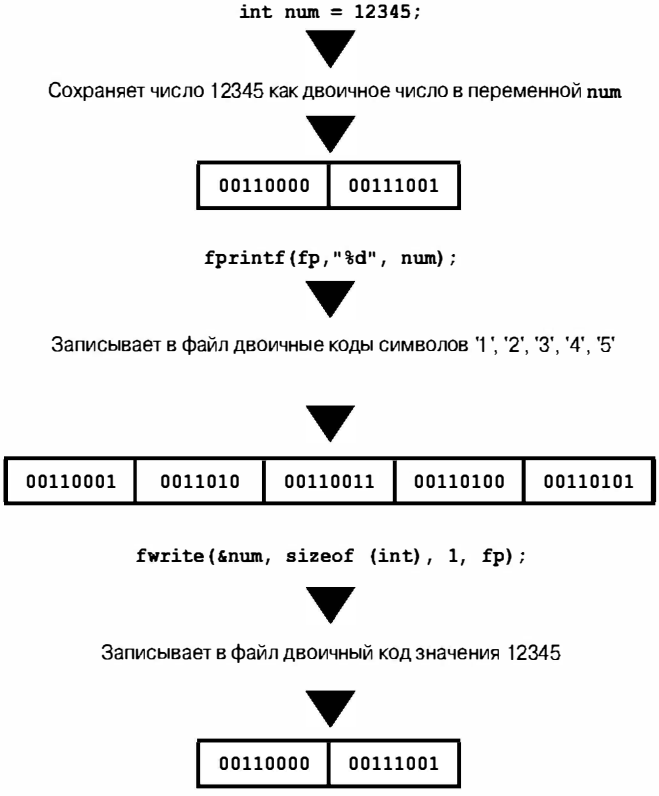


Рис. 13.3. *Двоичный и текстовый вывод*

Если некоторые или все данные интерпретируются как числовые, представленные в двоичной форме, мы говорим, что файл содержит двоичные данные. (Кроме того, двоичными также являются файлы, в которых данные представляют собой команды на машинном языке.)

Применение терминов *двоичный* и *текстовый* может привести к путанице. Стандарт ANSI C распознает два режима открытия файлов: двоичный и текстовый. Многие операционные системы распознают два формата файлов: двоичный и текстовый. Эти понятия логически связаны, но не идентичны. Вы можете открыть в двоичном режиме файл текстового формата. Вы можете сохранить текст в файле двоичного формата. Вы можете воспользоваться функцией `getc()` для копирования файлов, содержащих двоичные данные. Однако в общем случае для хранения двоичных данных в файле двоичного формата следует использовать двоичный режим. Аналогично, текстовые данные чаще всего используются из текстовых файлов, открытых в текстовом режиме. (Файлы, которые генерируют текстовые процессоры, как правило, являются двоичными, поскольку они содержат много нетекстовой информации, описывающей шрифты и форматирование.)

Функция `size_t fwrite()`

Прототип функции `fwrite()` показан ниже:

```
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb,
              FILE * restrict fp);
```

Функция `fwrite()` записывает двоичные данные в файл. Тип `size_t` определен через стандартные типы языка C. Этот тип возвращает операция `sizeof`. Обычно это тип `unsigned int`, в то же время в конкретной реализации C он может быть определен по-другому. Указатель `ptr` — это адрес порции данных, которую нужно записать в файл. Наряду с этим, значение переменной `size` представляет размер в байтах порции данных, записываемых в файл, а `nmemb` — количество таких порций. Как обычно, `fp` идентифицирует файл, в который выполнятся запись. Например, чтобы сохранить объект данных (например, массив) размером в 256 байтов, вы можете выполнить следующие операторы:

```
char buffer[256];
fwrite(buffer, 256, 1, fp);
```

Такой вызов функции переписывает одну порцию данных размером в 256 байтов из буфера в файл. Либо, чтобы сохранить массив из 10 значений типа `double`, вы должны выполнить следующие операторы:

```
double earnings[10];
fwrite(earnings, sizeof (double), 10, fp);
```

Этот вызов функции записывает данные из массива `earnings` в файл посредством 10 порций данных, каждая из которых имеет размер типа `double`.

Вы, должно быть, обратили внимание на странное объявление `const void * restrict ptr` в прототипе функции `fwrite()`. Одна из проблем, связанных с использованием функции `fwrite()`, состоит в том, что ее первый аргумент не имеет фиксированного типа. Например, в первом примере был использован аргумент `buffer`, имеющий тип указателя на `char`, а во втором примере применялся аргумент `earnings`,

имеющий тип указателя на double. В контексте прототипов функций ANSI C эти фактические аргументы приводятся к типу указателя на void, который действует как своего рода "ловушка" для всех типов указателей. (В реализациях языка C, предшествовавших ANSI, для этого аргумента использовался тип char *, и в силу этого обстоятельства вы должны приводить актуальные аргументы к этому типу.)

Функция fwrite() возвращает количество успешно записанных элементов. Обычно оно равно значению nmemb, однако оно может быть меньше этого значения, если при записи произошла ошибка.

Функция size_t fread()

Прототип функции fread() имеет следующий вид:

```
size_t fread(void * restrict ptr, size_t size, size_t nmemb,
             FILE * restrict fp);
```

Функция fread() принимает тот же набор аргументов, что и fwrite(). В данном случае ptr является адресом области памяти, куда считываются данные из файла, а fp идентифицирует считываемый файл. Эту функцию следует использовать для чтения данных, которые были записаны в файл с помощью функции fwrite(). Например, чтобы прочитать массив из 10 значений типа double, сохраненных в предыдущем примере, воспользуйтесь следующим кодом:

```
double earnings[10];
fread(earnings, sizeof (double), 10, fp);
```

Этот вызов копирует 10 значений типа double в массив earnings.

Функция fread() возвращает количество успешно прочитанных элементов. Обычно оно равно значению nmemb, но оно может быть и меньше этого значения, если происходит ошибка при чтении или если был достигнут конец файла.

Функции int feof(FILE *fp) и int ferror(FILE *fp)

Когда стандартные функции ввода возвращают маркер EOF, это обычно означает, что они достигли конца файла. В то же время он может также указывать на возникновение ошибки при чтении. Функции feof() и ferror() позволяют проводить различия между двумя этими возможностями. Функция feof() возвращает ненулевое значение, если последний вызов функции ввода обнаружил маркер конца ввода, и нулевое значение во всех других случаях. Функция ferror() возвращает ненулевое значение, если произошла ошибка при чтении или при записи, в противном случае она возвращает нуль.

Пример использования функций fread() и fwrite()

Попробуем использовать некоторые из этих функций в программе, которая добавляет содержимое файлов из заданного списка в конец другого файла. Одна из задач заключается в передаче информации из файла программе. Это можно реализовать в интерактивном режиме или с использованием аргументов командной строки.

Мы выбираем первый подход, который предполагает выполнение следующих действий:

- Запрос имени файла назначения и его открытие.
- Использование цикла для запроса исходных файлов.
- Поочередное открытие каждого исходного файла в режиме чтения и добавление его содержимого в конец файла назначения.

Чтобы проиллюстрировать работу функции `setvbuf()`, мы воспользуемся ею для определения различных размеров буфера. На следующей стадии уточнения проверяется, открыт ли файл назначения. Мы выполним следующие действия:

1. Открытие файла назначения в режиме добавления.
2. Если это сделать нельзя, завершить работу программы.
3. Установить буфер размером 1024 байтов для этого файла.
4. Если этого сделать нельзя, завершить работу программы.

Аналогично, мы уточняем ту часть программы, которая осуществляет копирование, выполняя следующие действия для каждого файла:

- Если это тот же файл, что и файл назначения, пропустите его и переходите к следующему файлу.
- Если файл не может быть открыт в режиме чтения, пропустите его и переходите к следующему файлу.
- Добавьте содержимое файла в файл назначения.

Чтобы попрактиковаться, воспользоваться для копирования функциями `fread()` и `fwrite()`. Результат показан в листинге 13.6.

Листинг 13.6. Программа `append.c`

```
/* append.c -- добавление содержимого файлов в конец другого файла */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFSIZE 1024
#define SLEN 81
void append(FILE *source, FILE *dest);

int main(void)
{
   FILE *fa, *fs;          // fa для файла назначения, fs для исходного файла
   int files = 0;          // количество добавляемых файлов
   char file_app[SLEN];    // имя файла назначения
   char file_src[SLEN];    // имя исходного файла

   puts("Введите имя файла назначения:");
   gets(file_app);
   if ((fa = fopen(file_app, "a")) == NULL)
   {
       fprintf(stderr, "Не удается открыть файл %s\n", file_app);
       exit(2);
   }
```

```
    if (setvbuf(fa, NULL, _IOFBF, BUFSIZE) != 0)
    {
        fputs("Не удается создать буфер вывода\n", stderr);
        exit(3);
    }
    puts("Введите имя первого исходного файла (или пустую строку для завершения):");
    while (gets(file_src) && file_src[0] != '\0')
    {
        if (strcmp(file_src, file_app) == 0)
            fputs("Нельзя добавлять файл в конец самого себя\n",stderr);
        else if ((fs = fopen(file_src, "r")) == NULL)
            fprintf(stderr, "Не удается открыть файл %s\n", file_src);
        else
        {
            if (setvbuf(fs, NULL, _IOFBF, BUFSIZE) != 0)
            {
                fputs("Не удается создать буфер ввода\n",stderr);
                continue;
            }
            append(fs, fa);
            if (ferror(fs) != 0)
                fprintf(stderr,"Ошибка во время чтения файла %s.\n",
                        file_src);
            if (ferror(fa) != 0)
                fprintf(stderr,"Ошибка во время записи в файл %s.\n",
                        file_app);
            fclose(fs);
            files++;
            printf("Файл %s добавлен.\n", file_src);
            puts("Введите имя следующего файла (или пустую строку для
завершения):");
        }
    }
    printf("Готово. Добавлено %d файлов.\n", files);
    fclose(fa);
    return 0;
}
void append(FILE *source, FILE *dest)
{
    size_t bytes;
    static char temp[BUFSIZE];          // распределить один раз
    while ((bytes = fread(temp,sizeof(char),BUFSIZE,source)) > 0)
        fwrite(temp, sizeof (char), bytes, dest);
}
```

Показанный ниже код создает буфер размером 1024 байта, предназначенный для работы с файлом назначения:

```
    if (setvbuf(fa, NULL, _IOFBF, BUFSIZE) != 0)
    {
       fputs("Не удается создать буфер вывода\n", stderr);
       exit(3);
    }
```

Если функция setvbuf() не может создать буфер, она возвращает ненулевое значение, после чего выполнение программы прекращается. Такой программный код устанавливает буфер размером 1024 байтов для копируемого на текущий момент файла. Использование значения NULL в качестве второго аргумента функции setvbuf(), предлагает этой функции самостоятельно выделить память под буфер.

Следующий программный код не позволяет добавлять содержимое файла в конец самого себя:

```
if (strcmp(file_src, file_app) == 0)
    fputs("Нельзя добавлять файл в конец самого себя\n",stderr);
```

Аргумент file_app представляет имя файла назначения, а аргумент file_src — имя файла, который обрабатывается в текущий момент.

Функция append() выполняет копирование. Вместо того чтобы копировать по одному байту за раз, она использует функции fread() и fwrite() для копирования 1024 байтов за один раз:

```
void append(source, dest)
FILE *source, *dest;
{
   size_t bytes;
   static char temp[BUFSIZE];          // распределить один раз
   while ((bytes = fread(temp,sizeof(char),BUFSIZE,source)) > 0)
      fwrite(temp, sizeof (char), bytes, dest);
}
```

Поскольку файл, заданный переменной dest, отрыт в режиме добавления, содержимое каждого исходного файла добавляется в конец файла один за другим. Обратите внимание на то, что массив temp имеет статическую продолжительность хранения (это значит, что память под него выделяется во время компиляции, а не при вызове функции append()) и область видимости в пределах блока (это значит, что переменная является приватной для данной функции).

В этом примере используются текстовые файлы; при указании режимов "ab" и "rb" программа сможет работать и с двоичными файлами.

Произвольный доступ с двоичным вводом-выводом

Произвольный доступ чаще всего примеряется к двоичным файлам, записанным с использованием двоичного ввода-вывода. Рассмотрим один короткий пример. Программа, представленная в листинге 13.7, создает файл чисел типа double, после чего предоставляет доступ к его содержимому.

Листинг 13.7. Программа randbin.c

```
/* randbin.c -- произвольный доступ, двоичный ввод-вывод */
#include <stdio.h>
#include <stdlib.h>
#define ARSIZE 1000

int main()
{
    double numbers[ARSIZE];
    double value;
```

```
    const char * file = "numbers.dat";
    int i;
    long pos;
    FILE *iofile;

    // создать набор значений типа double
    for(i = 0; i < ARSIZE; i++)
        numbers[i] = 100.0 * i + 1.0 / (i + 1);

    // попытка открыть файл
    if ((iofile = fopen(file, "wb")) == NULL)
    {
        fprintf(stderr, "Не удается открыть %s для вывода.\n", file);
        exit(1);
    }

    // записать массив в двоичном формате в файл
    fwrite(numbers, sizeof (double), ARSIZE, iofile);
    fclose(iofile);
    if ((iofile = fopen(file, "rb")) == NULL)
    {
        fprintf(stderr,
            "Не удается открыть %s для произвольного доступа.\n", file);
       exit(1);
    }

    // чтение избранных элементов из файла
    printf("Введите индекс из диапазона от 0 до %d.\n", ARSIZE - 1);
    scanf("%d", &i);
    while (i >= 0 && i < ARSIZE)
    {
        pos = (long) i * sizeof(double);         // вычислить смещение
        fseek(iofile, pos, SEEK_SET);            // перейти в эту позицию
        fread(&value, sizeof (double), 1, iofile);
        printf("В этом случае значение равно %f.\n", value);
        printf("Введите следующий индекс (или значение за пределами
диапазоны для завершения):\n");
        scanf("%d", &i);
    }

    // завершение программы
    fclose(iofile);
    puts("Всего доброго!");

    return 0;
}
```

Прежде всего, программа создает массив и помещает в него некоторые значения. Затем она создает файл с именем `numbers.dat` в двоичном режиме и использует функцию `fwrite()` для копирования содержимого массива в этот файл. 64-разрядный шаблон значения типа `double` копируется из памяти в массив. Вы не сможете прочитать двоичный файл в текстовом редакторе, поскольку эти значения не переводятся в строки. В то же время, каждое значение хранится в файле точно в том же порядке, в каком она хранилась в памяти, следовательно, в подобном случае потеря точности не-

возможна. Более того, каждое значение занимает 64 разряда памяти, благодаря чему нетрудно вычислить местонахождение каждого значения.

Во второй части программы файл открывается для чтения, а у пользователя запрашивается индекс значения. Умножение значения индекса на количество байтов, отведенное под значение типа `double`, позволяет определить местонахождение ячейки в файле. Затем программа вызывает функцию `fseek()` для перехода в эту ячейку, и функцию `fread()` для чтения значения, хранящегося в этой ячейке. Следует, однако, отметить, что в этом случае спецификаторы формата не используются. Взамен функция `fread()` копирует 8 байтов, начиная с указанной ячейки, в ячейку памяти, на которую указывает `&value`. Затем программа использует функцию `printf()` для отображения значений. Ниже показан пример результатов выполнения этой программы:

```
Введите индекс из диапазона от 0 до 999.
500
В этом случае значение равно 50000.001996.
Введите следующий индекс (или значение за пределами диапазоны для
завершения):
900
В этом случае значение равно 90000.001110.
Введите следующий индекс (или значение за пределами диапазоны для
завершения):
0
В этом случае значение равно 1.000000.
Введите следующий индекс (или значение за пределами диапазоны для
завершения):
-1
Всего доброго!
```

Ключевые понятия

Программа на языке C рассматривает ввод как некоторый поток данных. Источником этого потока может быть файл, устройство ввода (подобное, например, клавиатуре) или даже вывод в другую программу. Аналогично, программа на C трактует вывод как поток байтов, местом назначения которых может быть файл, устройство отображения и тому подобное.

Как C интерпретирует входной или выходной поток байтов, зависит от того, какую функцию ввода-вывода вы используете. Программа может читать и сохранять байты без изменений, с другой стороны, она может интерпретировать байты как символы, которые, в свою очередь, можно интерпретировать как обычный текст, либо текстовое представление чисел. Аналогично, при выводе функции, которые вы используете в своих программах, определяют, передаются ли двоичные значения без изменений либо же преобразуются в текст или текстовое представление чисел. Если имеются числовые данные, которые необходимо сохранить, а затем использовать без потери точности их представления, воспользуйтесь двоичным режимом и функциями `fread()` и `fwrite()`. Если вы сохраняете текстовую информацию и хотите создать файл, который легко просматривается с помощью обычных текстовых редакторов, воспользуйтесь текстовым режимом и такими функциями, как `getc()` и `fprintf()`.

Чтобы получить доступ к файлу, потребуется создать указатель на файл (типа `FILE *`) и связать его с конкретным именем файла. Последующий программный код для работы с этим файлом использует этот указатель, но не имя файла.

Важно понимать, как в C воспринимается понятие конца файла. Обычно программа, выполняющая чтение файла, читает ввод в цикле до тех пор, пока не достигнет конца файла. Функции ввода языка C не обнаруживают конца файла до тех пор, пока не предпримут попытку чтения символов, идущих за концом файла. Это означает, что проверка конца файла должна производиться непосредственно *после* попытки чтения. Вы можете использовать в качестве примера две модели ввода файлов, которые названы правильными в разделе "Признак конца файла" данной главы.

Резюме

Чтение и запись в файлы выполняются в большинстве программ на C. Большинство реализаций языка C предлагают для этих целей средства как низкоуровневого ввода-вывода, так и стандартного высокоуровневого ввода-вывода. Поскольку библиотека ANSI C включает стандартные средства ввода-вывода, но не средства низкоуровневого ввода-вывода, стандартный пакет обладает большей переносимостью.

Стандартный пакет ввода-вывода автоматически создает буферы для ввода и вывода, которые ускоряют передачу данных. Функция `fopen()` открывает файл для стандартного ввода-вывода и создает структуры данных, предназначенные для хранения информации о файле и буфере. Функция `fopen()` возвращает указатель на эту структуру данных, а сам указатель используется другими функциями для идентификации файла, предназначенного для обработки. Функции `feof()` и `ferror()` сообщают о причинах неудачного завершения операций ввода-вывода.

Язык C рассматривает ввод как поток байтов. Если вы используете функцию `fread()`, то C трактует ввод как двоичные значения, которые должны быть помещены в то место памяти, которое вы укажете. Если вы используете функции `fscanf()`, `getc()`, `fgets()` или любые другие логически связанные с ними функции, C рассматривает каждый байт как код конкретного символа. Функции `fscanf()` и `scanf()` затем пытаются преобразовать коды символов в другие типы, в соответствии со спецификаторами формата. Например, спецификатор `%f` преобразует входное значение 23 в значение с плавающей точкой, спецификатор `%d` преобразует тот же ввод в целочисленное значение, а спецификатор `%s` сохранит эту величину в виде строки. Семейство функций `getc()` и `fgets()` оставляют ввод в виде символьных кодов и сохраняют его либо в переменных типа `char` как отдельные символы, либо в виде массивов значений `char` как строки. Аналогично, функция `fwrite()` помещает двоичные данные непосредственно в выходной поток, в то время как другие функции вывода преобразуют несимвольные данные в символьные представления, и только затем помещают их в поток вывода.

Стандарт ANSI C предусматривает два режима открытия файла: двоичный и текстовый. Когда файл открывается в двоичном режиме, появляется возможность побайтового чтения файла. Когда файл открывается в текстовом режиме, его содержимое может быть отображено из системного представления текста в представление языка C. Для операционных систем Unix и Linux эти два режима идентичны.

Функции ввода getc(), fgets(), fscanf() и fread() обычно читают файл последовательно, начиная с начала файла. В то же время, функции fseek() и ftell() предоставляют программе возможность перемещаться в любую позицию в файле, обеспечивая тем самым произвольный доступ. Функции fgetpos() и fsetpos() распространяют эту возможность на файлы больших размеров. Произвольный доступ работает в двоичном режиме лучше, нежели в текстовом.

Вопросы для самоконтроля

1. Найдите ошибки в следующей программе:

```
int main(void)
{
    int * fp;
    int k;

    fp = fopen("gelatin");
    for (k = 0; k < 30; k++)
       puts(fp, "Кто-то не особо умный поглощает желатин.");
    fclose("gelatin");
    return 0;
}
```

2. Что делает приведенная ниже программа? (Пользователи Macintosh могут предположить, что программа корректно использует заголовочный файл console.h и функцию ccommand().)

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int main(int argc, char *argv[])
{
    int ch;
    FILE *fp;

    if (argc < 2)
         exit(2);
    if ( (fp = fopen(argv[1], "r")) == NULL)
         exit(1);
    while ( (ch= getc(fp)) != EOF )
        if( isdigit(ch) )
            putchar(ch);
    fclose (fp);

    return 0;
}
```

3. Предположим, что в программе присутствуют следующие операторы:

```
#include <stdio.h>
FILE * fp1,* fp2;
char ch;

fp1 = fopen("terky", "r");
fp2 = fopen("jerky", "w");
```

Кроме того, предположим, что оба файла были успешно открыты. Добавьте недостающие аргументы в следующие вызовы функций:

а. `ch = getc();`

б. `fprintf( ,"%c\n", );`

в. `putc( , );`

г. `fclose();` `/* закрыть файл terky */`

4. Напишите программу, которая не принимает аргументов командной строки или принимает один такой аргумент. Если используется один аргумент, он интерпретируется как имя файла. Если нет ни одного аргумента, то должен быть использован стандартный ввод (`stdin`). Предположим, что ввод целиком состоит из чисел с плавающей точкой. Ваша программа должна вычислить и отобразить на экране среднее арифметическое вводимых чисел.
5. Напишите программу, которая принимает два аргумента командной строки. Первый аргумент представляет собой символ, а второй — имя файла. Эта программа должна печатать только те строки из файла, которые содержат заданный символ.

На заметку!

Строки файла идентифицируются символом новой строки `'\n'`. Предположим, что ни одна из строк по длине не превышает 256 символов. Возможно, потребуется использовать функцию `fgets()`.

6. В чем состоит различие между двоичными и текстовыми файлами с одной стороны, и двоичными потоками и текстовыми потоками — с другой?
7. а. В чем различие между сохранением числа `8238201` с помощью функции `fprintf()` и сохранением этого числа с помощью функции `fwrite()`?

   б. В чем различие между сохранением символа *S* с помощью функции `putc()` и сохранением этого символа с помощью функции `fwrite()`?
8. Чем отличаются друг от друга следующие операторы?

```
printf("Здравствуйте, %s\n", name);
fprintf(stdout, "Здравствуйте, %s\n", name);
fprintf(stderr, "Здравствуйте, %s\n", name);
```

9. Режимы `"a+"`, `"r+"` и `"w+"` открывают файл для чтения и записи. Какой из этих режимов наилучшим образом подходит для изменения существующего содержимого файла?

Упражнения по программированию

1. Внесите изменения в программу, представленную в листинге 13.1, чтобы она запрашивала ввод имени файла и читала ответ пользователя, а не использовала для этого аргументы командной строки.
2. Напишите программу копирования файлов, которая принимает имена исходного файла и файла копии из командной строки. По возможности используйте стандартный ввод-вывод и двоичный режим.

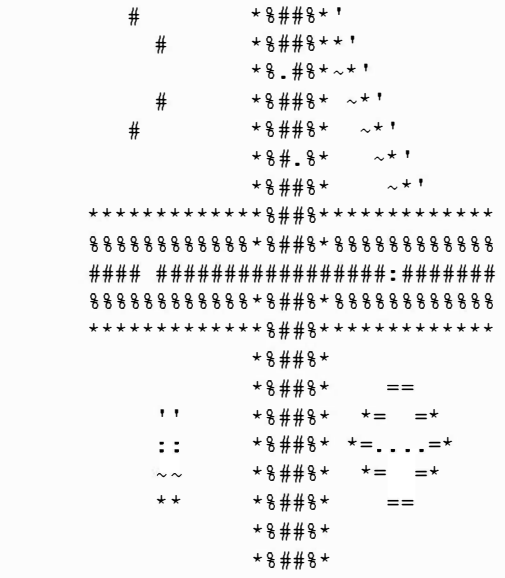
3. Напишите программу копирования файлов, которая приглашает пользователя ввести имя текстового файла, выступающего в роли исходного, и имя выходного файла. Эта программа должна использовать функцию `toupper()` из заголовочного файла `ctype.h` для перевода текста в верхний регистр во время его записи в выходной файл. Используйте стандартный ввод-вывод и текстовый режим.
4. Напишите программу, которая последовательно отображает на экране все файлы, список которых представлен в командной строке. Используйте `argc` для управления циклом.
5. Внесите изменения в программу, представленную в листинге 13.6, чтобы она могла использовать интерфейс командной строки вместо интерактивного интерфейса.
6. Программы, использующие аргументы программной строки, опираются на способности пользователя помнить, как их следует правильно применять. Перепишите программу их листинга 13.2 таким образом, чтобы вместо использования аргументов командной строки она запрашивала у пользователя ввод необходимой информации.
7. Напишите программу, которая открывает два файла. Вы можете получить имена файлов либо через командную строку, либо выдав приглашение на ввод этих аргументов.
   a. Сделайте так, чтобы эта программа печатала первую строку первого файла, первую строку второго файла, вторую строку первого файла, вторую строку второго файла и так далее, пока не будет напечатана последняя строка более длинного файла (по количеству строк).
   б. Модифицируйте программу таким образом, чтобы строки, имеющие один и тот же номер, печатались в одной строке.
8. Напишите программу, которая принимает в качестве аргументов командной строки некоторый символ и ноль или более имен файлов. Если за символом не следуют никакие аргументы, программа должна читать стандартный ввод. В противном случае она должны последовательно открыть каждый файл и сообщить, сколько раз указанный символ появляется в каждом файле. Имя файла и сам символ должны быть указаны вместе с каждым подсчетом. Включите в программу средства контроля ошибок с тем, чтобы проверить корректность количества аргументов и возможность открытия файлов. Если файл не может быть открыт, программа должны сообщить об этом факте и перейти к следующему файлу.
9. Внесите в программу, представленную в листинге 13.3, такие изменения, чтобы каждое слово было пронумеровано в том порядке, в каком оно включалось в список, начиная с 1. Позаботьтесь о том, чтобы при втором запуске программы на выполнение новая нумерация слов начиналась с того места, где была закончена предыдущая нумерация.
10. Напишите программу, которая открывает текстовый файл, имя которого может быть получено в интерактивном режиме. Организуйте цикл, который запрашивает пользователя позицию в файле. Затем программа должна распечатать часть файла, начинающуюся в этой позиции и продолжающуюся до следующего символа новой строки. Выполнение входного цикла пользователя должно прекращаться в случае ввода нечислового символа.

11. Напишите программу, которая принимает два аргумента командной строки. Первый аргумент — это строка, а второй аргумент — имя файла. Затем программа выполняет поиск файла, распечатывая все строки, содержащие заданную. Поскольку эта задача ориентирована на строки, а не на символы, воспользуйтесь функцией `fgets()` вместо `getc()`. Кроме того используйтесь функцию `strstr()` (кратко описанную в упражнении 7 из главы 11) из стандартной библиотеки C для поиска строки, содержащей заданную.

12. Создайте текстовый файл, содержащий 20 строк по 30 целочисленных значений. Эти значения входят в диапазон от 0 до 9 и отделяются друг от друга пробелами. Файл является цифровым представлением изображения, при этом значения от 0 до 9 показывают возрастание уровня темных тонов. Напишите программу, которая считывает содержимое файла в массив значений типа `int` размерности 20 на 30. При грубом подходе к преобразованию этого цифрового представления в изображение, программа использует значения в массиве для инициализации массива значений типа `char` размерностью 20 на 31, причем нулевое значение соответствует символу пробела, значение 1 — символу точки и так далее; каждый символ с бо́льшим значением представляет символ, занимающий большее пространство. Например, вы можете использовать символ # для представления значения 9. Последний символ (31-й) в каждой строке должен быть нулевым, делая его массивом из 20 строк. Программа должна отобразить полученное в результате изображение (то есть распечатать строки), а также сохранить результат в текстовом файле. Например, предположим, начальные данные выглядят следующим образом:

```
0 0 9 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 9 0 0 0 0 0 0 0 5 8 9 9 8 5 5 2 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 1 9 8 5 4 5 2 0 0 0 0 0 0 0 0 0
0 0 0 0 9 0 0 0 0 0 0 0 5 8 9 9 8 5 0 4 5 2 0 0 0 0 0 0 0 0
0 0 9 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 4 5 2 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 1 8 5 0 0 0 4 5 2 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 4 5 2 0 0 0 0 0
5 5 5 5 5 5 5 5 5 5 5 5 5 8 9 9 8 5 5 5 5 5 5 5 5 5 5 5 5 5
8 8 8 8 8 8 8 8 8 8 8 5 8 9 9 8 5 8 8 8 8 8 8 8 8 8 8 8 8 8
9 9 9 9 0 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 3 9 9 9 9 9 9 9 9
8 8 8 8 8 8 8 8 8 8 8 5 8 9 9 8 5 8 8 8 8 8 8 8 8 8 8 8 8 8
5 5 5 5 5 5 5 5 5 5 5 5 5 8 9 9 8 5 5 5 5 5 5 5 5 5 5 5 5 5
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 6 6 0 0 0 0 0 0
0 0 0 0 2 2 0 0 0 0 0 0 5 8 9 9 8 5 0 0 5 6 0 0 6 5 0 0 0 0
0 0 0 0 3 3 0 0 0 0 0 0 5 8 9 9 8 5 0 5 6 1 1 1 1 6 5 0 0 0
0 0 0 0 4 4 0 0 0 0 0 0 5 8 9 9 8 5 0 0 5 6 0 0 6 5 0 0 0 0
0 0 0 0 5 5 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 6 6 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 0 0 0 0 0 0 0
```

В условиях конкретного выбора выходных символов, вывод будет иметь такой вид:



13. Цифровые изображения, особенно те, которые переданы по радиоканалу космическим летательным аппаратом, могут подвергаться кратковременным импульсным помехам. Добавьте в программу из упражнения 12 функцию, подавляющую эти помехи. Она должна сравнивать каждое значение с соседними значениями, расположенными слева и справа, снизу и сверху. Если такое значение отличается более чем на 1 от своих соседей, замените его средним значением соседних величин. Программа должна округлить среднюю величину до ближайшего целого значения. Обратите внимание, что точки на границах имели менее четырех соседей, поэтому они требуют специальной обработки.

ГЛАВА 14

Структуры и другие формы данных

В этой главе:

- Ключевые слова: `struct`, `union`, `typedef`
- Операции: `.` `->`
- Структуры в языке C и создание шаблонов и переменных типа структур
- Доступ к членам структуры и написание функций поддержки структур
- Средство `typedef` языка C
- Объединения и указатели на функции

Одним из наиболее важных действий при разработке программы является выбор подходящего способа представления данных. Во многих случаях простых переменных или даже массивов оказывается недостаточно. Язык C позволяет расширить возможности представления данных с помощью *переменных типа структуры*. Структура в C является достаточно гибким средством в своей базовой форме, она способна представлять разнообразные данные, при этом она позволяет изобретать новые формы. Если вам знакомы "записи" языка Pascal, вам будет легко работать со структурами. Если нет, данная глава послужит введением в структуры C. Рассмотрим конкретный пример, который покажет, для чего могут понадобиться структуры, а также как они создаются и используются.

Учебная задача: создание каталога книг

Гвен Глен желает распечатать каталог своих книг. При этом ей необходима самая разнообразная информация, касающаяся книги: название, автор, издатель, дата установления авторского права, количество страниц и стоимость книги. Некоторые из этих элементов, такие как название книг, могут храниться в массивах строк. Другие элементы требуют массива значений типа `int` или `float`. При наличии семи различных переменных следить за всеми данными крайне затруднительно, особенно если учесть, что Гвен желает печатать несколько каталогов книг: один должен быть отсортирован по названиям, другой — по авторам, третий — по цене и так далее. Самым лучшим решением является использование одного массива, в котором каждый элемент содержит всю информацию об одной книге.

Кроме того, Гвен нужна форма данных, которая может содержать как строки, так и числа, при этом оба эти вида информации должны храниться отдельно. Структура C отвечает этим требованиям. Чтобы посмотреть, как создать такую структуру и как она работает, имеет смысл начать с упрощенного примера. Во-первых, ограничимся только названием, автором и текущей стоимостью книги. Во-вторых, пока будет считать, что каталог создается только для одной книги. Не стоит беспокоиться по поводу последнего ограничения, так как вскоре программа будет расширена.

Рассмотрим программу, представленную в листинге 14.1, и результаты ее выполнения. Далее приводится описание основных ее особенностей.

Листинг 14.1. Программа `book.c`

```
/* book.c -- каталог для одной книги */
#include <stdio.h>
#define MAXTITL 41          /* максимальная длина названия + 1 */
#define MAXAUTL 31          /* максимальная длина имени автора + 1 */

struct book {               /* шаблон структуры: дескриптором является book */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};                          /* конец шаблона структуры */

int main(void)
{
    struct book library;    /* объявить библиотеку как переменную типа book */

    printf("Введите название книги.\n");
    gets(library.title);    /* доступ к разделу названия книги */
    printf("Теперь введите ФИО автора.\n");
    gets(library.author);
    printf("Теперь введите цену.\n");
    scanf("%f", &library.value);
    printf("%s от %s: $%.2f\n",library.title,
         library.author, library.value);
    printf("%s: \"%s\" ($%.2f)\n", library.author,
         library.title, library.value);
    printf("Готово.\n");

    return 0;
}
```

Вот как выглядят результаты выполнения программы:

```
Введите название книги.
Chicken of the Alps
Теперь введите ФИО автора.
Bismo Lapoult
Теперь введите цену.
14.95
Chicken of the Alps от Bismo Lapoult: $14.95
Bismo Lapoult: "Chicken of the Alps" ($14.95)
```

Структура, созданная в листинге 14.1, состоит из трех частей (называемых *элементами* или *полями*) — одна для хранения названия книги, другая для хранения имени автора и третья для хранения цены. Вы должны научиться выполнять следующие действия:

- Разработать формат или компоновку структуры.
- Объявлять переменную, соответствующую данной компоновке.
- Обеспечить доступ к индивидуальным компонентам переменной типа структуры.

Объявление структуры

Объявление структуры представляет собой общее описание, показывающие, из каких частей состоит структура. Объявление структуры в данном примере имеет следующий вид:

```
struct book {
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};
```

Это объявление описывает структуру, образованную из двух символьных массивов и одной переменной типа `float`. Оно не создает фактического объекта данных, зато оно описывает, что именно является составляющими частями этого объекта. (Иногда мы будем называть объявление структуры *шаблоном*, потому что оно показывает, как и в каком виде будут храниться данные. Возможно, вы что-либо слышали о шаблонах в языке C++, однако в данном случае имеет место другое, более амбициозное использование этого слова.) Обратимся к подробностям. Первым идет ключевое слово `struct`. Оно определяет, что все, что следует за этим словом, представляет собой структуру. Далее следует необязательный дескриптор, а именно `book`, так называемая сокращенная метка, которую можно использовать для ссылки на эту структуру. Таким образом, далее мы будем пользоваться следующим объявлением:

```
struct book library;
```

Оно объявляет `library` как переменную типа структуры, использующую схему структуры `book`.

Следующим в объявлении структуры идет список элементов структуры, заключенных в одинарные кавычки. Каждый элемент описан собственным объявлением, которое оканчивается точкой с запятой. Например, раздел названия книги представляет собой массив значений типа `char` из `MAXTITL` элементов. Элементом структуры может быть любой тип данных C, в том числе и тип, включающий другие структуры!

Точка с запятой, проставленная после закрывающей круглой скобки, завершает определение шаблона структуры. Вы можете вынести это объявление за пределы любой функции (внешнее определение) либо поместить внутрь определения функции.

Если объявление структуры находится внутри функции, ее дескриптор может применяться только внутри функции. Если использовано внешнее определение, оно доступно всем функциям, которые следуют за этим объявлением в файле. Например, во второй функции можно было бы указать следующее объявление:

```
struct book dickens;
```

в результате чего в этой функции появилась бы переменная dickens, которая имела бы ту же форму, что и структура book.

Имя дескриптора не обязательно указывать, в то же время это имя должно использоваться при создании структуры, как это делали мы, когда шаблон структуры определялся в одном месте программы, а фактические переменные — в другом месте. Мы вернемся к этому вопросу позже, после того как рассмотрим, как выполняется определение переменных типа структуры.

Объявление переменной типа структуры

Понятие *структура* используется в двух значениях. Одно из этих значений — "схема структуры", именно ее мы только что обсуждали. Схема структуры сообщает компилятору, *как* представлять данные, но она не может заставить компилятор *выделять* пространство памяти под эти данные. Следующий шаг предусматривает создание переменной типа структуры, и в этом состоит второе значение слова *структура*. Строка программы, создающая переменную типа структуры, имеет вид:

```
struct book library;
```

Обнаружив эту команду, компилятор создает переменную library. Используя шаблон book, компилятор распределяет память под массив из MAXTITL элементов типа char, под массив из MAXAUTL элементов типа char и под переменную типа float. Эта память объединена в единую область под именем library (рис. 14.1). (В следующем разделе будет показано, как при необходимости можно разделить эту область.)

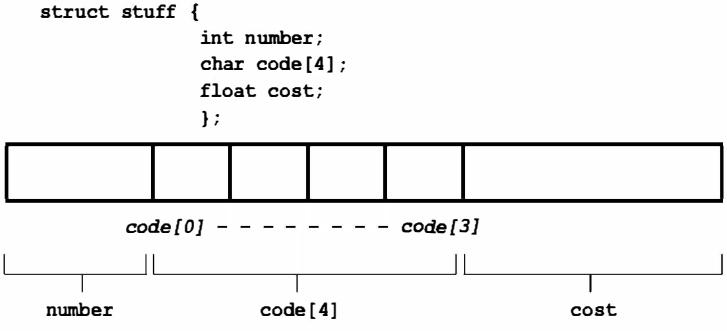


Рис. 14.1. *Распределение памяти под структуру*

При объявлении переменной типа структуры, шаблон struct book играет ту же роль, какую играют ключевые слова int или float в более простых объявлениях. Например, вы можете объявить две переменных типа struct book и даже указатель на этот вид структуры:

```
struct book doyle, panshin, * ptbook;
```

Каждая переменная doyle и panshin типа структуры будет иметь три части: title, author и value. Указатель ptbook может указывать на элементы структуры doyle, panshin или на любую другую структуру book. По сути дела, объявление структуры book создает новый тип с именем struct book.

С точки зрения компьютера объявление

```
struct book library;
```

является сокращенным вариантом объявления

```
struct book {
    char title[MAXTITL];
    char author[AXAUTL];
    float value;
} library;              /* объявление с именем переменной */
```

Другими словами, объявление структуры и объявление переменной типа структуры можно объединить в одну операцию. Такое объединение объявлений делает излишним использование дескриптора:

```
struct {                /* дескриптор не используется */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
} library;
```

Тем не менее, дескриптор понадобится, если шаблон структуры должен быть применен более одного раза, либо если используется альтернативный вариант объявления с помощью `typedef`, о котором речь пойдет далее в этой главе.

Еще один аспект, связанный с определением переменной типа структуры, пока еще не демонстрировался. Речь идет об инициализации. Перейдем к ее изучению.

Инициализация структур

Вы уже знаете, как выполняется инициализация переменных и массивов:

```
int count = 0;
int fibo[7] = {0,1,1,2,3,5,8};
```

Можно ли таким же способом инициализировать и структуру? Да, можно. Чтобы инициализировать структуру (любой класс памяти в стандарте ANSI C, но за исключением автоматических переменных в реализациях языка C, предшествующих ANSI C), вы используете синтаксис, аналогичный применяемому для инициализации массивов:

```
struct book library = {
    "The Pirate and the Devious Damsel",
    "Renee Vivotte",
    1.95
};
```

Короче говоря, используйте список инициализаторов, разделенных запятыми, заключенный в фигурные скобки. Каждый инициализатор должен соответствовать по типу инициализируемому элементу структуры. По этой причине вы можете инициализировать элемент `title` строкой, а элемент `value` — числовым значением. Чтобы сделать эти связи более очевидными, мы выделим каждому элементу отдельную строку инициализации, в то же время компилятору вполне достаточно, чтобы инициализаторы были отделены друг от друга запятыми.

Инициализация структуры и продолжительность хранения

В главе 12 говорилось, что если вы инициализируете переменную со статической продолжительностью хранения (например, статическое внешнее связывание, статическое внутреннее связывание или статическая продолжительность хранения без связывания), следует употреблять константные значения. Это относится также и к структурам. Если вы инициализируете структуру со статической продолжительностью хранения, значение в списке инициализаторов должны быть константными выражениями. Если продолжительность хранения автоматическая, значения в списке инициализаторов не обязательно должны быть константами.

Доступ к элементам структуры

Структура во многих отношениях подобна "супермассиву", в котором один элемент может быть значением типа `char`, второй элемент — `float`, а следующий — вообще массив значений типа `int`. Вы можете получить доступ к отдельным элементам массива с помощью индекса. Однако как получить доступ к отдельным элементам структуры? Для этой цели служит операция точки (`.`) или, как ее еще называют, операция принадлежности структуре. Например, `library.value` — это элемент `value` структуры `library`. Конструкция `library.value` используется подобно любой другой переменной типа `float`. Аналогично, конструкция `library.title` используется точно так же, как массив значений типа `char`. По этой причине в приведенной выше программе присутствуют такие выражения, как

```
gets(library.title);
```

и

```
scanf("%f", &library.value);
```

По сути дела, `.title`, `.author` и `.value` выступают в роли подиндексов в структуре `book`.

Обратите внимание, что хотя `library` и является структурой, тем не менее, `library.value` имеет тип `float` и используется как любой другой тип `float`. Например, `scanf("%f",...)` требует адреса ячейки `float`, и именно таким адресом является `&library.float`. Операция точки имеет более высокий приоритет, чем операция `&`, это выражение равнозначно выражению `&(library.float)`.

Если существует вторая переменная типа структуры того же типа, можно воспользоваться тем же методом:

```
struct book bill, newt;

gets(bill.title);
gets(newt.title);
```

Конструкция `.title` относится к первому элементу структуры `book`. Теперь обратите внимание на то, как исходная программа выводит содержимое структуры `library` в двух различных форматах. Этот пример может служить иллюстрацией той свободы, какую вы имеете при использовании элементов структуры.

Выделенные инициализаторы структур

Стандарт C99 предоставляет выделенные инициализаторы для структур. Соответствующий синтаксис подобен синтаксису выделенных инициализаторов для массивов. В то же время выделенные инициализаторы для структур при идентификации конкретных элементов используют операцию точки и имена элементов вместо квадратных скобок и индексов. Например, чтобы инициализировать только элемент `value` структуры `book`, вы должны сделать следующее:

```
struct book surprise = { .value = 10.99};
```

Выделенные инициализаторы можно указывать в любом порядке:

```
struct book gift = { .value = 25.99,
                     .author = "James Broadfool",
                     .title = "Rue for the Toad"};
```

Как и в случае с массивами, обычный инициализатор, указанный за выделенным, присваивает значение элементу, следующему за выделенным элементом. Наряду с этим, последнее значение, присвоенное конкретному элементу, является значением, которое он получает. Например, рассмотрим следующее объявление:

```
struct book gift= { .value = 18.90,
                    .author = "Philionna Pestle",
                     0.25};
```

Значение `0.25` присваивается элементу `value`, поскольку именно это значение идет в списке непосредственно за элементом `author` в объявлении структуры. Новое значение `0.25` затирает старое значение `18.90`, присвоенное ранее. Теперь, когда в вашем распоряжении появились эти базовые средства, вы можете существенно расширить кругозор и рассмотреть несколько новых разновидностей структур. Вы изучите массивы структур, структуры структур, указатели на структуры и функции обработки структур.

Массивы структур

Расширим программу создания каталога книг, чтобы она поддерживала более одной книги. Ясно, что каждая книга может быть описана одной переменной типа структуры `book`. Чтобы описать две книги, необходимо использовать две таких переменных и так далее. Для поддержки нескольких книг потребуется массив таких структур, и эта возможность реализована в программе, представленной в листинге 14.2. (Если вы работаете с Borland C/C++, ознакомьтесь с врезкой “Borland C и плавающая запятая”.)

Структуры и память

Программа `manybook.c` использует массив из 100 структур. Поскольку массивы являются объектами автоматического класса памяти, соответствующая информация обычно размещается в стеке. Такой крупный массив требует большую область памяти, что может стать источником проблем. Если появляется сообщение об ошибке во время выполнения программы, которое, возможно, уведомляет о недостаточно большом размере стека или о переполнении стека, ваш компилятор, скорее всего, использует стек с размером по умолчанию, который оказался недостаточным в условиях рассматриваемого примера. Для исправления ситуации можно с помощью опций компилятора установить стек размером 10000, чтобы можно было выполнять обработку данного массива

структур, также можно сделать массив статическим или внешним (в этом случае он не будет размещаться в стеке) либо уменьшить его размер, скажем, до 16. Почему мы начали с проблемы небольшого стека? Да потому, что вы должны иметь представление об этой потенциальной проблеме, дабы уметь решать ее, когда с ней доведется столкнуться на практике.

Листинг 14.2. Программа `manybook.c`

```
/* manybook.c -- каталог для нескольких книг */
#include <stdio.h>
#define MAXTITL 40
#define MAXAUTL 40
#define MAXBKS 100                    /* максимальное количество книг */
struct book {                         /* шаблон book */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};
int main(void)
{
    struct book library[MAXBKS];      /* массив структур типа book */
    int count = 0;
    int index;
    printf("Введите название книги.\n");
    printf("Нажмите [enter] в начале строки для выхода из программы.\n");
    while (count < MAXBKS && gets(library[count].title) != NULL
                          && library[count].title[0] != '\0')
    {
       printf("Теперь введите ФИО автора.\n");
       gets(library[count].author);
       printf("Теперь введите цену книги.\n");
       scanf("%f", &library[count++].value);
       while (getchar() != '\n')
           continue;                   /* очистить входную строку */
       if (count < MAXBKS)
       printf("Введите название следующей книги.\n");
    }
    if (count > 0)
    {
        printf("Каталог ваших книг:\n");
        for (index = 0; index < count; index++)
        printf("%s от %s: $%.2f\n", library[index].title,
            library[index].author, library[index].value);
    }
    else
          printf("Вообще нет книг? Очень плохо.\n");
    return 0;
}
```

Borland C и плавающая запятая

Ранние версии компиляторов языка Borland C пытались делать программы более компактными путем использования сокращенных версий функции `scanf()` в тех случаях, когда программа не использовала значений с плавающей запятой. Тем не менее, компиляторы (до версии Borland C/C++ 3.1 для DOS включительно, но не версия Borland C/C++ 4.0) могут быть введены в заблуждение, если массив структур содержит только значения с плавающей запятой, как в случае программы, представленной на листинге 14.2. В результате вы получите подобного рода сообщения:

```
scanf : floating point formats not linked
Abnormal program termination
```

Одно из решений этой проблемы заключается во включению в вашу программу следующего кода:

```
#include <math.h>
double dummy = sin(0.0);
```

Этот программный код заставляет компилятор загрузить версию функции `scanf()`, предназначенную для работы со значениями с плавающей запятой.

Приводим результаты выполнения этой учебной программы:

```
Введите название книги.
Нажмите [enter] в начале строки для выхода из программы.
My Life as a Budgie
Теперь введите ФИО автора.
Mack Zackles
Теперь введите цену книги.
12.95
Введите название следующей книги.
   ...ввод информации о других книгах...
Каталог ваших книг:
My Life as a Budgie от Mack Zackles: $12.95
Thought and Unthought Rethought от Kindra Schlagmeyer: $43.50
The Business of a Bee от Salome Deschamps: $14.99
The CEO Power Diet от Buster Downsize: $19.25
C++ Primer Plus от Stephen Prata: $40.00
Under a Tofu Moon от Angus Bull: $15.97
Coping with Coping от Dr. Rubin Thonkwacker: $0.00
Delicate Frivolity от Neda McFey: $29.99
Murder Wore a Bikini от Mickey Splats: $18.95
A History of Buvania, Volume 4, от Prince Nikoli Buvan: $50.00
Mastering Your Digital Watch, 2nd Edition, от Miklos Mysz: $18.95
A Foregone Confusion от Phalty Reasoner: $5.99
Outsourcing Government: Selection vs. Election от Ima Pundit: $33.33
```

Прежде всего, рассмотрим, как объявлять массив структур и как получать доступ к его отдельным элементам. Затем проанализируем два аспекта программы.

Объявление массива структур

Объявление массива структур подобно объявлению любого другого вида массива. Ниже показан пример:

```
struct book library[MAXBKS];
```

Этот оператор объявляет `library` как массив из `MAXBKS` элементов. Каждый элемент этого массива является структурой типа `book`. Следовательно, `library[0]` — также структура типа `book`, `library[1]` — структура типа `book` и так далее. Рисунок 14.2 поможет более отчетливо представлять, как устроена структура. Само имя `library` не есть имя структуры, оно представляет собой имя массива, элементы которого — структуры `struct book`.

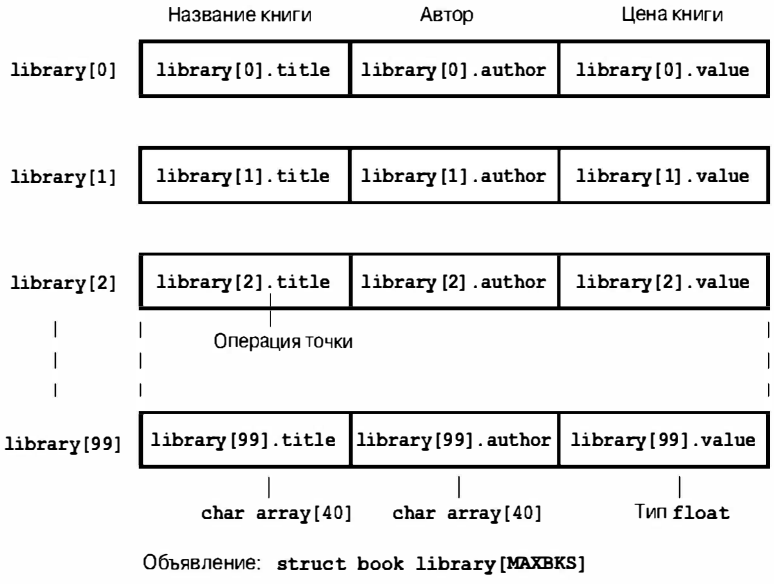


Рис. 14.2. *Массив структур*

Идентификация элементов массива структур

Для идентификации элементов массива структур применяются те же правила, которые использовались применительно к индивидуальным структурам: за именем структуры должна следовать операция точки, а затем имя элемента.

Рассмотрим следующий пример:

```
library[0].value   /* значение, ассоциированное с первым
                      элементом массива */
library[4].title   /* название книги, ассоциированное с пятым
                      элементом массива */
```

Обратите внимание, что индекс массива дописан к `library`, а не в конец имени:

```
library.value[2]       // Неправильно
library[2].value       // Правильно
```

Причина, по которой используется конструкция `library[2].value`, состоит в том, что `library[2]` является именем переменной типа структуры, точно так же как `library[1]` — именем еще одной переменной типа структуры.

Кстати, что, по вашему мнению, представляет собой следующее выражение?

```
library[2].title[4]
```

Это пятый символ в названии книги (часть `title[4]`), описанной третьей структурой (часть `library[2]`). В рассматриваемом примере это будет символ *B*. Данный пример показывает, что индексы, которые находятся справа от операции точки, применяются к отдельным элементам, в то же время индексы, которые находятся слева от операции точки, относятся к массиву структур.

В конечном итоге получается следующая последовательность:

```
library               // массив структур book
library[2]            // элемент массива, следовательно, структура book
library[2].title      // массив значений типа char (элемент title
                      //  структуры library[2])
library[2].title[4]   // значение типа char в элементе title
```

А теперь завершим рассмотрение нашей программы.

Анализ программы

Основное отличие этой программы от первой заключается в том, что в новой версии присутствует цикл, обеспечивающий возможность чтения сразу нескольких входных записей. Цикл начинается со следующего условия `while`:

```
while (count < MAXBKS && gets(library[count].title) != NULL
                      && library[count].title[0] != '\0')
```

Выражение `gets(library[count].title)` читает строку названия книги; это выражение принимает значение `NULL`, если функция `gets()` пытается прочитать символ, следующий за маркером конца файла. Выражение `library[count].title[0] != '\0'` проверяет, не является ли первый символ строки нулевым (другими словами, строка пуста). Если пользователь нажмет клавишу <Enter> в начале строки, будет введена пустая строка, и цикл завершается. В программу также встроен механизм проверки, позволяющий следить за тем, чтобы количество введенных записей не превышало максимально допустимый размер массива.

Далее в программе идут следующие строки:

```
while (getchar() != '\n')
   continue;                    /* очистить входную строку */
```

Как уже говорилось в предыдущих главах, этот код служит заменой функции `scanf()`, игнорирующей пробелы и символы новой строки. Когда вы отвечаете на запрос ввода цены книги, вы вводите с клавиатуры что-нибудь наподобие:

```
12.50[enter]
```

Этот оператор передает следующую последовательность символов:

```
12.50\n
```

Функция `scanf()` собирает символы `1`, `2`, `.`, `5` и `0`, оставляя `\n` и ожидая появления следующего оператора чтения. При отсутствии соответствующего кода следующий

оператор чтения, в данном случае gets(library[count].title), читает этот оставшийся символ новой строки и интерпретирует его как пустую строку, предполагая, что послан сигнал выхода из программы. Код, который мы включили, "поглощает" символы до тех пор, пока не обнаружит символ новой строки и не избавится от него. Он ничего не делает с этими символами, а лишь только извлекает их из входной очереди. Все это создает условия для применения функции gets().

Теперь вернемся к исследованию структур.

Вложенные структуры

Иногда удобно вложить одну структуру в другую. Например, некая Шейла Пироски создает структуру с данными о ее друзьях. Одним из элементов структуры является, естественно, имя друга. Однако само имя может быть представлено структурой, в которой отдельные элементы отводятся под имя и фамилию. В листинге 14.3 показаны результаты работы Шейлы.

Листинг 14.3. Программа `friend.c`

```
// friend.c -- пример вложенной структуры
#include <stdio.h>
#define LEN 20
const char * msgs[5] =
{
    "  Благодарю вас за чудесно проведенный вечер, ",
    "Вы однозначно продемонстрировали, что ",
    "являет собою особый тип мужчины. Мы обязательно должны встретиться",
    "за восхитительным ужином с ",
    " и весело провести время."
};
struct names {                  // первая структура
    char first[LEN];
    char last[LEN];
};
struct guy {                    // вторая структура
    struct names handle;        // вложенная структура
    char favfood[LEN];
    char job[LEN];
    float income;
};
int main(void)
{
    struct guy fellow = {       // инициализация переменной
        { "Стивен", "Кинг" },
          "запеченными омарами",
          "персональный тренер",
           58112.00
    };

    printf("Дорогой %s, \n\n", fellow.handle.first);
    printf("%s%s.\n", msgs[0], fellow.handle.first);
    printf("%s%s\n", msgs[1], fellow.job);
```

```
    printf("%s\n", msgs[2]);
    printf("%s%s%s", msgs[3], fellow.favfood, msgs[4]);
    if (fellow.income > 150000.0)
        puts("!!");
    else if (fellow.income > 75000.0)
        puts("!");
    else
        puts(".");
    printf("\n%40s%s\n", " ", "До скорой встречи,");
    printf("%40s%s\n", " ", "Шейла");
    return 0;
}
```

Ниже представлены выходные данные этой программы:

```
Дорогой Стивен,

  Благодарю вас за чудесно проведенный вечер, Стивен.
Вы однозначно продемонстрировали, что персональный тренер
являет собою особый тип мужчины. Мы обязательно должны встретиться
за восхитительным ужином с запеченными омарами и весело провести время.

                                        До скорой встречи,
                                        Шейла
```

Во-первых, обратите внимание на описание вложенной структуры в объявлении структуры. Она объявляется очень просто, подобно объявлению, скажем, переменной типа `int`:

```
struct names handle;
```

Это объявление говорит о том, что `handle` — переменная типа `struct names`. Разумеется, файл также должен включать объявление структуры `names`.

Во-вторых, посмотрите, как выполняется доступ к элементу вложенной структуры; вы просто дважды используете операцию точки:

```
printf("Дорогой %s!\n", fellow.handle.first);
```

Эта конструкция интерпретируется следующим образом, если рассматривать ее слева направо:

```
(fellow.handle).first
```

Другими словами, сначала нужно найти структуру `fellow`, затем элемент `handle` структуры `fellow`, а затем элемент `first` вложенной структуры.

Указатели на структуры

Любители указателей будут рады узнать о возможности использования указателей на структуры. Существуют, по меньшей мере, три причины, оправдывающие применение указателей на структуры. Во-первых, в силу того, что манипулировать указателями на массивы значительно проще, чем самими массивами (например, при решении задачи сортировки), иметь дело с указателями на структуры зачастую эффективнее, чем с самими структурами. Во-вторых, в некоторых более ранних реализациях языка C структура не может передаваться как аргумент функции, в то время как указатель на

структуру — может. В-третьих, многие замысловатые представления данных используют структуры, содержащие указатели на другие структуры.

Следующий краткий пример (листинг 14.4) показывает, как определить указатель на структуру и как его использовать для доступа к ее элементам.

Листинг 14.4. Программа `friends.c`

```
/* friends.c --использование указателя на структуру */
#include <stdio.h>
#define LEN 20

struct names {
    char first[LEN];
    char last[LEN];
};

struct guy {
    struct names handle;
    char favfood[LEN];
    char job[LEN];
    float income;
};

int main(void)
{
    struct guy fellow[2] = {
       {{ "Стивен", "Кинг" },
        "запеченными омарами",
        "персональный тренер",
        58112.00
       },
       {{ "Родни", "Стюарт" },
        "рыбным фрикасе",
        "редактор таблоида",
        232400.00
       }
    };
    struct guy * him;          /* указатель на структуру */
    printf("адрес #1: %p #2: %p\n", &fellow[0], &fellow[1]);
    him = &fellow[0];          /* говорит указателю, на что указывать */
    printf("указатель #1: %p #2: %p\n", him, him + 1);
    printf("him->income равно $%.2f: (*him).income равно $%.2f\n",
         him->income, (*him).income);
    him++;                     /* указатель на следующую структуру */
    printf("him->favfood равно %s: him->handle.last равно %s\n",
         him->favfood, him->handle.last);
    return 0;
}
```

Выходные данные выглядят следующим образом:

```
адрес #1: 0x0012fea4 #2: 0x0012fef8
указатель #1: 0x0012fea4 #2: 0x0012fef8
him->income равно $58112.00: (*him).income равно $58112.00
him->favfood равно рыбным фрикасе: him->handle.last равно Стюарт
```

Сначала рассмотрим, как создать указатель на структуру guy. Затем покажем, как описать отдельные элементы структур, используя этот указатель.

Объявление и инициализация указателя на структуру

Объявление указателя на структуру не вызывает никаких трудностей:

```
struct guy * him;
```

Первым идет ключевое слово struct, далее следует дескриптор структуры guy, за ним — звездочка (*) и, наконец, имя указателя. Тот же синтаксис применяется и при объявлении других рассмотренных ранее указателей.

Это объявление не приводит к созданию новой структуры, в то же время оно позволяет использовать указатель him с тем, чтобы он указывал на любую существующую структуру типа guy. Например, если barney является структурой типа guy, вы можете иметь следующий оператор:

```
him = &barney;
```

В отличие от массивов, имя структуры не является ее адресом — вы должны использовать операцию &.

В рассматриваемом примере fellow — это массив структур, это означает, что fellow[0] представляет собой структуру, следовательно, этот код инициализирует указатель him так, что в конечном итоге он указывает на структуру fellow[0]:

```
him = &fellow[0];
```

Две первые выходные строки показывают, что эта операция присваивания выполнена успешно. Сравнивая две эти строки, вы убеждаетесь в том, что указатель him указывает на структуру fellow[0], а him + 1 — на структуру fellow[1]. Обратите внимание, что добавление 1 к указателю him добавляет к адресу значение 84. В шестнадцатеричной системе счисления, ef8 – ea4 = 54 (шестнадцатеричное) = 84 (десятичное), поскольку каждая структура guy занимает 84 байта памяти: names.first занимает 20 байтов, names.last — 20, favfood — 20, job — 20 и income — 4, столько, сколько занимает тип float в нашей системе. В некоторых системах размер структуры может быть больше, чем сумма всех ее частей. Это объясняется тем, что выравнивание, осуществляемое системой, может привести к появлению неиспользованных участков памяти, например, система должна размещать каждый элемент структуры по четному адресу или по адресу, кратному четырем. Такие структуры могут содержать в себе так называемые промежутки.

Доступ к элементам структуры через указатели

Указатель him указывает на структуру fellow[0]. Как использовать указатель him, чтобы получить значение того или иного элемента структуры fellow[0]? Третья выходная строка демонстрирует два метода.

Первый, наиболее распространенный метод, использует новую операцию ->. Знак этой операции формируется путем ввода с клавиатуры дефиса (-), за которым следует символ “больше” (>).

Приведенный ниже пример позволяет уяснить действие этой операции:

```
him->income равно fellow[0].income, если him == &fellow[0]
```

Другими словами, указатель на структуру, за которым следует операция ->, дает тот же результат, что и имя структуры, за которым следует операция точки. (Выражения him.income использовать нельзя, поскольку him не является именем структуры.)

Важно отметить, что хотя him — указатель, тем не менее, him->income — это элемент структуры, на который он указывает. Следовательно, в данном случае him->income представляет собой переменную типа float.

Второй метод определения значения элемента структуры вытекает из следующей последовательности: если him == &fellow[0], то *him == fellow[0], поскольку & и * являются обратными операциями. Отсюда после соответствующей подстановки получаем следующее выражение:

```
fellow[0].income == (*him).income
```

Круглые скобки здесь необходимы, поскольку операция точки имеет более высокий приоритет, чем операция *. Короче говоря, если him есть указатель на структуру типа guy с именем barney, имеем следующую последовательность эквивалентных выражений:

```
barney.income == (*him).income == him->income
  // предполагается, что him == &barney
```

Теперь рассмотрим, как взаимодействуют структуры и функции.

Взаимодействие функций и структур

Вспомните, что аргументы функции передают ей значения. Каждое значение может быть числом типа int, float, ASCII-кодом символа или адресом. Структура сложнее, чем одиночное значение, поэтому не удивительно, что более ранние реализации языка C не позволяют использовать структуру в качестве аргумента функции. Это ограничение снято в более новых реализациях, и уже ANSI C позволяет указывать структуры в качестве аргументов функций. В силу этого современные реализации языка C оставляют вам на выбор передавать в качестве аргументов сами структуры или же передавать указатели на эти структуры, либо если вас интересует только определенная часть структуры, вы можете передавать элементы структур.

Далее мы рассмотрим все три метода. Начнем с передачи частей структуры в качестве аргументов.

Передача элементов структуры

Если элемент структуры имеет тип данных с единственным значением (то есть int или один из его производных типов, char, float, double либо указатель), его можно передавать в качестве аргумента функции, которая принимает данный конкретный тип. Простейшая банковская программа, код которой показан в листинге 14.5, снимающая определенную сумму с обычного банковского счета клиента и добавляющая ее на сберегательный счет, служит наглядной иллюстрацией сказанному выше.

Листинг 14.5. Программа `funds1.c`

```
/* funds1.c -- передача элементов структуры в качестве аргументов */
#include <stdio.h>
#define FUNDLEN 50

struct funds {
    char bank[FUNDLEN];
    double bankfund;
    char save[FUNDLEN];
    double savefund;
};

double sum(double, double);

int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        3024.72,
        "Lucky's Savings and Loan",
        9237.11
    };
    printf("Сумма на счету у Стэна составляет $%.2f.\n",
           sum(stan.bankfund, stan.savefund) );
   return 0;
}
/* суммирование двух чисел типа double */
double sum(double x, double y)
{
    return(x + y);
}
```

Приводим результаты выполнения этой программы:

```
Сумма на счету у Стэна составляет $12261.83.
```

Итак, программа работает. Следует отметить, что функция `sum()` либо не знает, либо ей безразлично, являются ли фактические аргументы элементами структуры, единственное, чего она требует — чтобы они имели тип `double`.

Разумеется, если вы хотите, чтобы вызванная функция оказывала воздействия на значения элементов в вызывающей функции, вы можете передать адрес этого элемента:

```
modify(&stan.bankfund);
```

Это будет функция, которая корректирует сумму на банковском счету клиента Стэна. Следующий метод передачи информации о структуре предусматривает уведомление вызываемой функции о том, что она имеет дело со структурой.

Использование адреса структуры

Мы будем решать ту же задачу, что и раньше, но на этот раз в качестве аргумента воспользуемся структурой. Поскольку функция должна работать со структурой `funds`, она также должна воспользоваться объявлением этой структуры. В листинге 14.6 представлена соответствующая программа.

Листинг 14.6. Программа `funds2.c`

```
/* funds2.c -- передача указателя на структуру */
#include <stdio.h>
#define FUNDLEN 50

struct funds {
    char bank[FUNDLEN];
    double bankfund;
    char save[FUNDLEN];
    double savefund;
};

double sum(const struct funds *);       /* аргумент является указателем */

int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        3024.72,
        "Lucky's Savings and Loan",
        9237.11
    };
    printf("Сумма на счету у Стэна составляет $%.2f.\n", sum(&stan));
    return 0;
}

double sum(const struct funds * money)
{
    return(money->bankfund + money->savefund);
}
```

Выполнение этой программы дает тот же результат:

```
Сумма на счету у Стэна составляет $12261.83.
```

Функция `sum()` использует указатель (`money`) на структуру `funds` в качестве своего единственного аргумента. Передача адреса `&stan` функции приводит к тому, что указатель `money` теперь указывает на структуру `stan`. Затем с помощью операции `->` получаются значения `stan.bankfund` и `stan.savefund`. Поскольку функция не меняет значения, на которое ссылается указатель, она объявляет `money` указателем на `const`.

Эта функция также имеет доступ к именам организации, хотя она их не использует. Обратите внимание, что для получения адреса структуры вы должны использовать операцию `&`. В отличие от имени массива имя структуры не является синонимом ее адреса.

Передача структуры в качестве аргумента

Для компиляторов, которые позволяют передавать структуры в качестве аргументов, последний пример потребуется переписать так, как показано в листинге 14.7.

Листинг 14.7. Программа `funds3.c`

```
/* funds3.c -- передача структуры */
#include <stdio.h>
#define FUNDLEN 50
```

```
struct funds {
    char bank[FUNDLEN];
    double bankfund;
    char save[FUNDLEN];
    double savefund;
};
double sum(struct funds moolah);   /* аргумент есть структура */
int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        3024.72,
        "Lucky's Savings and Loan",
        9237.11
    };
    printf("Сумма на счету у Стэна составляет $%.2f.\n", sum(stan));
    return 0;
}
double sum(struct funds moolah)
{
    return(moolah.bankfund + moolah.savefund);
}
```

И снова получаем прежний результат:

```
Сумма на счету у Стэна составляет $12261.83.
```

Мы заменили `money`, указатель на `struct funds`, на `moolah`, переменную `struct funds`. Когда вызывается функция `sum()`, создается автоматическая переменная с именем `moolah` в соответствии с шаблоном `funds`. Затем выполняется инициализация элементов этой структуры таким образом, что они получают значения соответствующих элементов структуры `stan`, становясь их копиями. В силу этого вычисления выполняются с использованием копии исходной структуры, в то время как предыдущая программа (та, в которой применяется указатель) использует саму исходную структуру. Поскольку `moolah` — структура, рассматриваемая программа использует конструкцию `moolah.bankfund`, но не `moolah->bankfund`. С другой стороны, в программе из листинга 14.6 использована конструкция `money->bankfund`, так как `money` — указатель, но не структура.

Дальнейший анализ свойств структур

Современные версии языка C позволяют выполнять присваивание одной структуры другой; что-либо подобное делать с массивами нельзя. То есть, если элементы данных `n_data` и `o_data` — структуры одного и того же типа, вы имеете возможность выполнить следующее действие:

```
o_data = n_data; // присваивание одной структуры другой
```

Это приводит к тому, что каждому элементу структуры `n_data` присваивается значение соответствующего элемента структуры `o_data`. Это возможно и в тех случаях, когда в качестве элемента выступает массив.

Наряду с этим вы можете инициализировать одну структуру другой структурой того же типа:

```
struct names right_field = {"Джеймс", "Бонд"};
struct names captain = right_field;    // инициализация одной структуры
                                       // значениями другой структуры
```

В современных версиях языка C, в том числе и ANSI C, структуры не только можно передавать функции в качестве аргументов, но и возвращать в качестве возвращаемого значения. Использование структур в качестве аргументов функции позволяет передавать функции информацию о структуре; использование функций для возврата структур позволяет передавать информацию из вызываемой функции в вызывающую. Указатели на структуры допускают также двусторонний обмен данными, следовательно, вы часто можете применять любой из этих двух подходов для решения задач программирования. Рассмотрим еще один набор примеров, иллюстрирующих эти два подхода.

Чтобы сравнить эти два подхода, напишем простую программу, которая работает со структурами, используя с этой целью указатели, затем мы переделаем ее таким образом, чтобы в ней выполнялась передача и возврат структур. Сама программа запрашивает ваше имя и фамилию и сообщает, из скольких букв они состоят. Этот проект не требует применения структур, в то же время он позволяет ознакомиться с их работой.

В листинге 14.8 представлен вариант программы с использованием указателей.

Листинг 14.8. Программа `names1.c`

```
/* names1.c -- использует указатели на структуры */
#include <stdio.h>
#include <string.h>

struct namect {
    char fname[20];
    char lname[20];
    int letters;
};

void getinfo(struct namect *);
void makeinfo(struct namect *);
void showinfo(const struct namect *);

int main(void)
{
    struct namect person;

    getinfo(&person);
    makeinfo(&person);
    showinfo(&person);
    return 0;
}

void getinfo (struct namect * pst)
{
    printf("Введите свое имя.\n");
    gets(pst->fname);
    printf("Введите свою фамилию.\n");
    gets(pst->lname);
}
```

```
void makeinfo (struct namect * pst)
{
    pst->letters = strlen(pst->fname) +
                   strlen(pst->lname);
}
void showinfo (const struct namect * pst)
{
    printf("%s %s, ваше имя и фамилия содержат %d букв.\n",
        pst->fname, pst->lname, pst->letters);
}
```

Компиляция и выполнение программы позволяет получить следующие результаты:

```
Введите свое имя.
Васисуалий
Введите свою фамилию.
Лоханкин
Васисуалий Лоханкин, ваше имя и фамилия содержат 18 букв.
```

Работа, которую выполняет данная программа, распределяется между тремя функциями, которые вызываются из главной функции `main()`. Во всех случаях адрес структуры `person` передается функции.

Функция `getinfo()` передает хранящуюся в ней информацию функции `main()`. В частности, она получает имена от пользователя и помещает их в структуру `person`, используя с этой целью ссылающийся на нее указатель `pst`. Напомним, что выражение `pst->lname` означает элемент структуры `lname`, на которую указывает `pst`. Это делает выражение `pst->lname` эквивалентным имени массива значений типа `char`, следовательно, подходящим аргументом для функции `gets()`. Отметим, что хотя функция `getinfo()` снабжает информацией главную программу, она не использует для этого механизм возврата, следовательно, ее типом является `void`.

Функция `makeinfo()` выполняет двустороннюю передачу информации. Используя указатель на структуру `person`, она отыскивает имя и фамилию, хранящиеся в этой структуре. Она использует функцию `strlen()` из библиотеки C для вычисления количества букв в имени и фамилии, а затем использует адрес структуры `person`, чтобы сохранить полученную сумму. И в этом случае типом функции является `void`. И, наконец, функция `showinfo()` использует указатель для поиска информации, которую нужно вывести на печать. Поскольку эта функция не меняет содержимого массива, она объявляет указатель как `const`.

Во всех этих операциях использовалась только одна переменная типа структуры, а именно, `person`, и каждая из функций использует адрес этой структуры для доступа к ней. Одна из функций передает вызывающей программе информацию, которая содержалась в ней самой, другая функция принимает информацию из вызывающей программы для себя, а третья делает и то и другое.

Теперь посмотрим, как запрограммировать ту же задачу, используя в качестве аргументов структуры и возвращаемые значения. Прежде всего, чтобы передать саму структуру, используйте в качестве аргумента `person`, но не `&person`. В этом случае соответствующим формальным аргументом будет объявленный тип `struct namect`, но не указатель на этот тип. Во-вторых, чтобы предоставить значения структуры функции `main()`, вы можете возвратить ей всю структуру. В листинге 14.9 показана версия программы без указателей.

Листинг 14.9. Программа names2.c

```
/* names2.c -- передает и возвращает структуры */
#include <stdio.h>
#include <string.h>

struct namect {
    char fname[20];
    char lname[20];
    int letters;
};

struct namect getinfo(void);
struct namect makeinfo(struct namect);
void showinfo(struct namect);

int main(void)
{
    struct namect person;
    person = getinfo();
    person = makeinfo(person);
    showinfo(person);

    return 0;
}

struct namect getinfo(void)
{
    struct namect temp;
    printf("Введите свое имя.\n");
    gets(temp.fname);
    printf("Введите свою фамилию.\n");
    gets(temp.lname);

    return temp;
}

struct namect makeinfo(struct namect info)
{
    info.letters = strlen(info.fname) + strlen(info.lname);

    return info;
}

void showinfo(struct namect info)
{
    printf("%s %s, ваше имя и фамилия содержат %d букв.\n",
        info.fname, info.lname, info.letters);
}
```

Эта версия программы дает тот же результат, что и предыдущая, однако она работает несколько по-другому. Каждая из описанных выше трех функций создает собственную копию структуры `person`, таким образом, эта программа использует четыре разные структуры вместо всего лишь одной.

Рассмотрим, например, функцию `makeinfo()`. В первой программе был передан адрес структуры `person`, и указанная функция оперировала фактическими значениями

этой структуры. Во второй версии этой программы создается новая структура под именем info. Значения, которые хранятся в структуре person, копируются в структуру info, а функция имеет дело с копией. По этой причине, когда подсчитывается количество букв, оно сохраняется в структуре info, но не в структуре person. В то же время механизм возврата вносит соответствующие исправления.

Строка функции makeinfo()

```
return info;
```

образует комбинацию со строкой функции main()

```
person = makeinfo(person);
```

которая позволяет копировать значения из структуры info в структуру person. Обратите внимание на то, что функция makeinfo() должна иметь объявленный тип struct namect, поскольку она возвращает структуру.

Структуры или указатели на структуры?

Предположим, что вам надо написать функцию, манипулирующую структурами. Будете ли вы пользоваться указателями на структуру или воспользуетесь структурами в качестве аргументов и возвращаемых значений? У каждого из этих подходов имеются свои сильные и слабые стороны.

Метод, использующий указатели в качестве аргументов, обладает двумя достоинствами: он работает как в условиях более ранних версий языка C, так и в современных реализациях языка C, к тому же он характеризуется высоким быстродействием; вы передаете функции всего лишь один адрес. Его недостатком является то, что в условиях этого метода ваши данные становятся более уязвимыми. Некоторые операции, выполняемые вызываемой функцией, неизбежно затрагивают данные в исходной структуре. Тем не менее, наличие в стандарте ANSI C спецификатора const позволяет решить эту проблему. Например, если в функцию showinfo() поместить код, который изменяет какой-либо из элементов структуры, компилятор обнаруживает и истолковывает это как ошибку.

Одно из преимуществ передачи структур в качестве аргументов состоит в том, что эта функция имеет дело с копией исходных данных, что намного безопаснее, чем работать непосредственно с исходными данными. Наряду с этим более понятным становится и стиль программирования. Предположим, что вы определили следующую структуру:

```
struct vector {double x; double y;};
```

Вектору ans присваивается сумма векторов a и b. Вы можете написать функцию, передающую и возвращающую структуру, которая может использоваться следующим образом:

```
struct vector ans, a, b;
struct vector sum_vect(vector, vector);
...
ans = sum_vect(a,b);
```

Для инженеров эта версия выглядит более естественной, чем версия с использованием указателя, которая может иметь примерно такой вид:

```
struct vector ans, a, b;
void sum_vect(const vector *, const vector *, vector *);
...
sum_vect(&a, &b, &ans);
```

Кроме того, в версии с указателями пользователь должен помнить, каким аргументом должен быть представлен адрес суммы, первым или последним.

Два основных неудобства, связанных с передачей структур в качестве аргументов функций, заключаются в том, что более ранние реализации языка C не понимают такой код, а также тот факт, что в этом случае неэкономно расходуется процессорное время и память. Особо расточительной является передача крупных структур функциям, которые используют один или два элемента такой структуры. В данном случае передача указателя или передача только нужных элементов как индивидуальных аргументов имеет больше смысла. Как правило, программисты используют указатели на структуры в качестве аргументов функции из соображений эффективности, задавая спецификатор `const`, когда необходимо защитить данные от несанкционированных изменений. Передача структур по значению чаще всего применяется к структурам небольших размеров.

Символьные массивы или указатели на символы в структурах

В рассмотренных выше примерах для хранения строк в структуре применялись символьные массивы. Вас, должно быть, интересует, нельзя ли вместо них использовать указатели на значения `char`? Например, в листинге 14.3 присутствует следующее объявление:

```
#define LEN 20
struct names {
   char first[LEN];
   char last[LEN];
};
```

Можно ли вместо этого использовать следующий код:

```
struct pnames {
   char * first;
   char * last;
};
```

Ответ на этот вопрос такой: да, можно, однако при этом могут возникнуть проблемы, если вы не осознаете всех негативных последствий подобного подхода. Рассмотрим следующий программный код:

```
struct names veep = {"Talia", "Summers"};
struct pnames treas = {"Brad", "Fallingjaw"};
printf("%s and %s\n", veep.first, treas.first);
```

Этот код допустим, и он работает, однако рассмотрим, где эти строки хранятся. Для переменной `veep` структуры `names` строки хранятся внутри структуры; эта структура выделяет в сумме 40 байтов для хранения двух имен. Однако для переменной `treas` структуры `pnames` строки хранятся там, где компилятор сохраняет константы.

Данная структура содержит всего лишь два адреса, которые в нашей системе занимают 8 байтов. В частности, структура pnames не выделяет память для хранения строк. Она может использоваться только теми строками, память под которые была выделена где-нибудь в другом месте, например, строковыми константами и строками из массивов. Короче говоря, указатели в структуре pnames должны использоваться только для управления строками, которые были созданы, и им была выделена память в другом месте программы.

Давайте выясним, в каких случаях эти ограничения вызывают затруднения. Рассмотрим следующий код:

```
struct names accountant;
struct pnames attorney;
puts("Введите фамилию вашего бухгалтера:");
scanf("%s", accountant.last);
puts("Введите фамилию вашего адвоката:");
scanf("%s", attorney.last);        /* именно здесь и таится опасность */
```

С точки зрения синтаксиса этот программный код безупречен. Но где запоминаются входные данные? Что касается бухгалтера, то его фамилия сохраняется в последнем элементе переменной accountant; в этой структуре предусмотрен массив для хранения строки. Что касается фамилии адвоката, то функция scanf() получает указание поместить эту строку по адресу, заданному элементом attorney.last. Поскольку данная переменная не инициализирована, этот адрес может иметь произвольное значение, и программа может поместить фамилию в каком угодно месте. Если вам повезет, то программа будет работать, по крайней мере, некоторое время, либо выполнение данной программы завершится аварийно. По сути, если программа работает, то вам совершенно не повезло, поскольку в ней притаилась катастрофическая ошибка, а вы об этом даже не подозреваете.

Таким образом, если вы хотите, чтобы структура сохраняла строки, воспользуйтесь элементами типа символьного массива. Хранение указателей на значения char иногда применяется, но такой подход сопряжен с непредсказуемыми последствиями.

Структура, указатели и функция malloc()

Одним из случаев, когда использование указателей в структуре при работе со строками целесообразно, является вызов функции malloc() для распределения памяти и применение указателей для хранения адресов. Этот подход обладает тем преимуществом, что вы можете с помощью malloc() выделить такое пространство памяти, которое необходимо для размещения конкретной строки. Вы можете запросить 4 байта для сохранения строки "Joe" и 18 байтов для строки "Rasolofomasoandro". Нетрудно настроить листинг 14.9 на такой подход. Два основных изменения состоят в том, что определение структуры настраивается на использование указателей вместо массивов с последующим применением новой версии функции getinfo(). Новое определение структуры принимает следующий вид:

```
struct namect {
   char * fname;        // использование указателей вместо массивов
   char * lname;
   int letters;
};
```

Новая версия функции getinfo() читает входные данные во временный массив, использует функцию malloc() для распределения пространства памяти и копирует в него строку. Она делает это для каждого имени:

```
void getinfo (struct namect * pst)
{
    char temp[81];
    printf("Введите свое имя.\n");
    gets(temp);
    // распределение памяти для хранения имени
    pst->fname = (char *) malloc(strlen(temp) + 1);
    // скопировать имя в выделенную память
    strcpy(pst->fname, temp);
    printf("Введите свою фамилию.\n");
    gets(temp);
    pst->lname = (char *) malloc(strlen(temp) + 1);
    strcpy(pst->lname, temp);
}
```

Вы должны четко понимать, что эти две строки не хранятся в структуре. Они хранятся в том участке памяти, которым управляет функция malloc(). В то же время адреса этих двух строк хранятся в структуре, именно адресами обычно и манипулируют функции, предназначенные для работы со строками. В то же время нет необходимости что-либо менять в других функциях, используемых в рассматриваемой программе. Однако, как следует из главы 12, вы должны согласовывать вызовы функции malloc() с вызовами функции free(), с этой целью в программу включена новая функция под именем cleanup(), которая освобождает память, когда программа прекратит ее использование. Вы можете ознакомиться с этой новой функцией и остальной частью программы в листинге 14.10.

Листинг 14.10. Программа names3.c

```
// names3.c -- использование указателей и функции malloc()
#include <stdio.h>
#include <string.h>               // для функций strcpy(), strlen()
#include <stdlib.h>               // для функций malloc(), free()

struct namect {
    char * fname;                 // использование указателей
    char * lname;
    int letters;
};

void getinfo(struct namect *);    // распределение памяти
void makeinfo(struct namect *);
void showinfo(const struct namect *);
void cleanup(struct namect *);    // освобождение памяти, когда она не нужна

int main(void)
{
    struct namect person;
    getinfo(&person);
    makeinfo(&person);
    showinfo(&person);
```

```
    cleanup(&person);
    return 0;
}
void getinfo (struct namect * pst)
{
    char temp[81];
    printf("Введите свое имя.\n");
    gets(temp);
    // распределение памяти для хранения имени
    pst->fname = (char *) malloc(strlen(temp) + 1);
    // копирование имени в распределенную память
    strcpy(pst->fname, temp);
    printf("Введите свою фамилию.\n");
    gets(temp);
    pst->lname = (char *) malloc(strlen(temp) + 1);
    strcpy(pst->lname, temp);
}
void makeinfo (struct namect * pst)
{
    pst->letters = strlen(pst->fname) +
                   strlen(pst->lname);
}
void showinfo (const struct namect * pst)
{
    printf("%s %s, ваше имя и фамилия содержат %d букв.\n",
    pst->fname, pst->lname, pst->letters);
}
void cleanup(struct namect * pst)
{
    free(pst->fname);
    free(pst->lname);
}
```

Ниже показан пример выходных данных этой программы:

```
Введите свое имя.
Васисуалий
Введите свою фамилию.
Лоханкин
Васисуалий Лоханкин, ваше имя и фамилия содержат 18 букв.
```

Составные литералы и структуры (C99)

Новые составные литералы, введенные в употребление стандартом C99, можно использовать как в структурах, так и в массивах. Составные литералы пригодны для создания структуры с последующим ее передачей в качестве аргумента функции либо для ее присваивания другой структуре. Синтаксис составного литерала предусматривает список инициализаторов, которому предшествует имя типа в круглых скобках.

Например, приведенная ниже конструкция представляет собой составной литерал типа `struct book`:

```
(struct book) {"Идиот", "Федор Достоевский", 6.99}
```

В листинге 14.11 показан пример использования составных литералов, которые представляют два альтернативных значения переменной типа структуры. (На момент написания книги только несколько компиляторов поддерживали это средство, однако со временем эта проблема должна быть решена.)

Листинг 14.11. Программа `complit.c`

```
/* complit.c -- составные литералы */
#include <stdio.h>
#define MAXTITL 41
#define MAXAUTL 31

struct book {           // шаблон структуры: дескриптором является book
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};

int main(void)
{
    struct book readfirst;
    int score;

    printf("Введите рейтинг: ");
    scanf("%d",&score);
    if(score >= 84)
        readfirst = (struct book) {"Преступление и наказания",
                                   "Федор Достоевский",
                                   9.99};
    else
          readfirst = (struct book) {"Красивая шляпа мистера Баунси",
                                     "Фред Уинсом",
                                     5.99};
    printf("Присвоенные вами рейтинги:\n");
    printf("%s by %s: $%.2f\n",readfirst.title,
          readfirst.author, readfirst.value);

    return 0;
}
```

Составные литералы можно использовать как аргументы функций. Если функция ожидает структуру, вы можете передать ей составной литерал в качестве фактического параметра:

```
struct rect {double x; double y;};
double rect_area(struct rect r){return r.x * r.y;}
...
double area;
area = rect_area( (struct rect) {10.5, 20.0});
```

Это приводит к тому, что переменной `area` присваивается значение `210.0`.

Если функция ожидает адрес, вы можете передать адрес составного литерала:

```
struct rect {double x; double y;};
double rect_areap(struct rect * rp){return rp->x * rp->y;}
...
double area;
area = rect_areap( &(struct rect) {10.5, 20.0});
```

Это приводит к тому, что переменной `area` присваивается значение `210.0`.

Сложные литералы, использующиеся за пределами каких-либо функций, имеют статическую продолжительность хранения. Те же синтаксические правила действуют в отношении составных литералов, рассматриваемых как хранилища для обычных списков инициализаторов. Это означает, например, что вы можете использовать выделенные инициализаторы применительно к составным литералам.

Элементы типа гибких массивов (C99)

Стандарт C99 вводит в употребление новое средство, получившее название *элемента типа гибкого массива*. Оно позволяет объявить структуру, в которой последний элемент является массивом со специальными свойствами. Одно из специальных свойств заключается в том, что такой массив не существует, или, по меньшей мере, появляется не сразу. Второе специальное свойство состоит в том, что при наличии соответствующего программного кода вы можете пользоваться элементом типа гибкого массива, как если бы он существовал и имел такое количество элементов, какое вам нужно. Возможно, это звучит несколько странно, потому для большей ясности рассмотрим все действия по созданию и использованию структуры с элементом типа гибкого массива.

Прежде всего, ниже представлены правила по созданию элемента типа гибкого массива:

- Элемент типа гибкого массива должен быть последним элементом структуры.
- В структуре должен содержаться, по меньшей мере, один элемент.
- Гибкий массив объявляется как обычный массив за исключением того, что его квадратные скобки пусты.

Ниже показан пример, иллюстрирующий эти правила:

```
struct flex
{
    int count;
    double average;
    double scores[];          // элемент гибкого массива
};
```

Если вы объявили переменную типа `struct flex`, вы не можете использовать элемент `scores`, поскольку для него нигде не зарезервирована память. Фактически, от вас не требуется всякий раз объявлять переменные тип `struct flex`. Вместо этого предполагается, что вы объявите *указатель* на тип `struct flex`, а затем воспользуетесь функцией `malloc()` для распределения памяти, достаточной для размещения обычного содержимого переменной типа `struct flex` *плюс* любое дополнительное пространство, которое вы хотите выделить для размещения элемента типа гибкого массива.

Например, предположим, что вы хотите, чтобы элемент scores представлял массив из пяти значений типа double. В этом случае потребуется сделать следующее:

```
struct flex * pf;             // объявить указатель
// запросить пространство памяти для размещения структуры и массива
pf = malloc(sizeof(struct flex) + 5 * sizeof(double));
```

Теперь вы располагаете порцией памяти, достаточной для того, чтобы сохранить значения count, average и массив, содержащий пять значений типа double. Вы можете воспользоваться указателем pf для доступа к элементам этой структуры:

```
pf->count = 5;                // установить элемент count
pf->scores[2] = 18.5;         // доступ к элементу массива,
                              // который является элементом структуры
```

Листинг 14.12 представляет собой дальнейшее развитие этого примера, позволяя гибкому массиву представлять пять значений в одном случае и девять значений — в другом. Он также демонстрирует, как следует писать функцию для обработки структуры с элементом типа гибкого массива. (Элементы типа гибкого массива получили в настоящее время более широкую поддержку, нежели составные литералы структуры.)

Листинг 14.12. Программа flexmemb.c

```
// flexmemb.c -- элемент типа гибкого массива
#include <stdio.h>
#include <stdlib.h>

struct flex
{
    int count;
    double average;
    double scores[];          // элемент типа гибкого массива
};

void showFlex(const struct flex * p);

int main(void)
{
    struct flex * pf1, *pf2;
    int n = 5;
    int i;
    int tot = 0;

    // распределение памяти для структуры плюс массив
    pf1 = malloc(sizeof(struct flex) + n * sizeof(double));
    pf1->count = n;
    for (i = 0; i < n; i++)
    {
        pf1->scores[i] = 20.0 - i;
        tot += pf1->scores[i];
    }
    pf1->average = tot / n;
    showFlex(pf1);

    n = 9;
    tot = 0;
```

```
    pf2 = malloc(sizeof(struct flex) + n * sizeof(double));
    pf2->count = n;
    for (i = 0; i < n; i++)
    {
        pf2->scores[i] = 20.0 - i/2.0;
        tot += pf2->scores[i];
    }
    pf2->average = tot / n;
    showFlex(pf2);
    free(pf1);
    free(pf2);
    return 0;
}
void showFlex(const struct flex * p)
{
    int i;
    printf("Рейтинги: ");
    for (i = 0; i < p->count; i++)
        printf("%g ", p->scores[i]);
    printf("\nСреднее значение: %g\n", p->average);
}
```

Ниже показаны результаты выполнения данной программы:

```
Рейтинги: 20 19 18 17 16
Среднее значение: 18
Рейтинги: 20 19.5 19 18.5 18 17.5 17 16.5 16
Среднее значение: 17
```

Функции, использующие массив структур

Предположим, что имеются массивы структур, которые необходимо обработать с помощью той или иной функции. Имя массива — это синоним его адреса, и в силу этого оно может быть передано функции. Опять-таки, функция нуждается в доступе к шаблону структуры. Чтобы продемонстрировать, как это средство работает, в листинге 14.13 банковская программа расширена с целью обслуживания двух человек, по этой причине в ней используется массив, содержащий две структуры `funds`.

Листинг 14.13. Программа `funds4.c`

```
/* funds4.c -- передача функции массива структур */
#include <stdio.h>
#define FUNDLEN 50
#define N 2
struct funds {
    char bank[FUNDLEN];
    double bankfund;
    char save[FUNDLEN];
    double savefund;
};
```

```
double sum(const struct funds money[], int n);
int main(void)
{
    struct funds jones[N] = {
        {
             "Garlic-Melon Bank",
             3024.72,
             "Lucky's Savings and Loan",
             9237.11
        },
        {
             "Honest Jack's Bank",
             3534.28,
             "Party Time Savings",
             3203.89
        }
    };
    printf("Семейство Джонсов имеет на счету общую сумму в $%.2f.\n",
           sum(jones,N));
    return 0;
}
double sum(const struct funds money[], int n)
{
    double total;
    int i;
    for (i = 0, total = 0; i < n; i++)
        total += money[i].bankfund + money[i].savefund;
    return(total);
}
```

В результате выполнения программы получаем следующие результаты:

```
Семейство Джонсов имеет на счету общую сумму в $19000.00.
```

(Какая ровная сумма! Можно подумать, что цифры были специально подобраны.)

Имя массива `jones` является и адресом массива. В частности, это адрес первого элемента массива, которым является структура `jones[0]`. Поэтому первоначально указатель `money` задается следующим выражением:

```
money = &jones[0];
```

Поскольку указатель `money` указывает на первый элемент массива `jones`, `money[0]` — это другое имя первого элемента массива. Аналогично, `money[1]` — второй элемент. Каждый элемент представляет собой структуру `funds`, таким образом, каждый из них может использовать операцию точки (`.`) для доступа к элементам структур.

Отметим ключевые моменты:

- Вы можете использовать имя массива для передачи в функцию адреса первой структуры массива.
- Вы можете использовать запись с квадратными скобками для доступа к последующим структурам массива.

Обратите внимание на то, что вызов функции

```
sum(&jones[0], N)
```

приведет к тому же результату, что и использование имени массива, поскольку как `jones`, так и `&jones[0]` — один и тот же адрес. Имя массива представляет собой косвенный способ передачи адреса структуры.

- Поскольку функция `sum()` не должна изменять исходные данные, эта функция использует квалификатор `const` стандарта ANSI C.

Сохранение содержимого структур в файле

Поскольку структуры могут содержать самую разнообразную информацию, они остаются важным инструментальным средством построения баз данных. Например, вы можете воспользоваться структурой для хранения информации, имеющей отношение к служащим или к автомобильным запчастям. Вы неизбежно захотите иметь возможность сохранять эту информацию в файле и извлекать ее из файла. Файл базы данных содержит произвольное количество таких объектов данных. Полный набор информации, сохраняемой в структуре, называется *записью*, а отдельные элементы структуры — *полями*. Рассмотрим эти понятия более подробно.

Наиболее очевидный способ сохранения записи, возможно, является и наименее эффективным; он предусматривает использование функции `fprintf()`. В качестве примера вспомним структуру `book`, определенную в листинге 14.1:

```
#define MAXTITL  40
#define MAXAUTL  40
struct book {
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};
```

Если `pbooks` идентифицирует файловый поток, вы можете сохранить эту информацию в переменной типа `struct book` с именем `primer` с помощью следующего оператора:

```
fprintf(pbooks, "%s %s %.2f\n", primer.title,
        primer.author, primer.value);
```

Такой подход становится громоздким уже для структур, имеющих, скажем, 30 элементов. Наряду с этим, в условиях данного подхода возникает проблема извлечения, поскольку программе необходим какой-то способ, уведомляющий ее о том, где одно поле кончается, а другое начинается. Эту проблему можно решить за счет использования формата с полями фиксированного размера (например, `"%39s%39s%8.2f"`), тем не менее, громоздкость остается.

Более приемлемое решение предусматривает использование функций `fread()` и `fwrite()` для считывания и записи элементов данных типа структуры. Напомним, что эти функции выполняют чтение и запись, используя то же двоичное представление, что и программа.

Например:

```
fwrite(&primer, sizeof (struct book), 1, pbooks);
```

переходит к начальному адресу структуры `primer` и копирует все байты этой структуры в файл, ассоциированный с `pbooks`. Выражение `sizeof (struct book)` сообщает функции, какой величины блок следует копировать, при этом `1` означает, что должен быть скопирован только один блок. Функция `fread()` с теми же аргументами копирует порцию данных размером со структуру из файла в область памяти, на которую указывает `&primer`. Короче говоря, эти функции выполняют одноразовые операции чтения-записи применительно ко всей записи, но не применительно к конкретному полю.

Пример сохранения структуры

Чтобы продемонстрировать, как эти функции могут использоваться в программе, мы внесли некоторые изменения в листинг 14.2, благодаря чему названия книг сохраняются в файле с именем `book.dat`. Если этот файл уже существует, программа отображает его текущее содержимое и обеспечивает возможность добавления в него данных. В листинге 14.14 представлена новая версия этой программы. (Если вы работаете с Borland C/C++, ознакомьтесь с врезкой "Borland C и плавающая запятая".)

Листинг 14.14. Программа `booksave.c`

```
/* booksave.c -- сохранение содержимого структуры в файле */
#include <stdio.h>
#include <stdlib.h>
#define MAXTITL   40
#define MAXAUTL   40
#define MAXBKS   10                    /* максимальное количество книг */
struct book {                          /* создание шаблона book */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};
int main(void)
{
    struct book library[MAXBKS];       /* массив структур */
    int count = 0;
    int index, filecount;
    FILE * pbooks;
    int size = sizeof (struct book);
    if ((pbooks = fopen("book.dat", "a+b")) == NULL)
    {
        fputs("Не удается открыть файл book.dat\n",stderr);
        exit(1);
    }
    rewind(pbooks);                    /* переход в начало файла */
    while (count < MAXBKS && fread(&library[count], size,
                1, pbooks) == 1)
    {
```

```
        if (count == 0)
            puts("Текущее содержимое файла book.dat:");
        printf("%s by %s: $%.2f\n",library[count].title,
            library[count].author, library[count].value);
        count++;
    }
    filecount = count;
    if (count == MAXBKS)
    {
        fputs("Файл book.dat заполнен.", stderr);
        exit(2);
    }
    puts("Введите названия новых книг.");
    puts("Нажмите [enter] в начале строки для выхода из программы.");
    while (count < MAXBKS && gets(library[count].title) != NULL
                          && library[count].title[0] != '\0')
    {
        puts("Теперь введите имя автора.");
        gets(library[count].author);
        puts("Теперь введите цену книги.");
        scanf("%f", &library[count++].value);
        while (getchar() != '\n')
            continue;                          /* очистить входную строку */
        if (count < MAXBKS)
            puts("Введите название следующей книги.");
    }
    if (count > 0)
    {
        puts("Каталог ваших книг:");
        for (index = 0; index < count; index++)
            printf("%s by %s: $%.2f\n",library[index].title,
                   library[index].author, library[index].value);
        fwrite(&library[filecount], size, count - filecount,
               pbooks);
    }
    else
           puts("Вообще нет книг? Очень плохо.\n");
    puts("Всего доброго.\n");
    fclose(pbooks);
    return 0;
}
```

Сначала рассмотрим результаты нескольких запусков на выполнение данной программы, а затем проанализируем ее основные особенности.

```
% booksave
Введите названия новых книг.
Нажмите [enter] в начале строки для выхода из программы.
Metric Merriment
Теперь введите имя автора.
Polly Poetica
```

```
Теперь введите цену книги.
18.99
Введите название следующей книги.
Deadly Farce
Теперь введите имя автора.
Dudley Forse
Теперь введите цену книги.
15.99
Введите название следующей книги.
[enter]
Каталог ваших книг:
Metric Merriment by Polly Poetica: $18.99
Deadly Farce by Dudley Forse: $15.99
Всего доброго.
% booksave
Текущее содержимое файла book.dat:
Metric Merriment by Polly Poetica: $18.99
Deadly Farce by Dudley Forse: $15.99
Введите названия новых книг.
The Third Jar
Теперь введите имя автора.
Nellie Nostrum
Теперь введите цену книги.
22.99
Введите название следующей книги.
[enter]
Каталог ваших книг:
Metric Merriment by Polly Poetica: $18.99
Deadly Farce by Dudley Forse: $15.99
The Third Jar by Nellie Nostrum: $22.99
Всего доброго.
%
```

При следующем выполнении программы `booksave.c` все три книги будут отображены как текущие записи файла `book.dat`.

Анализ программы

Во-первых, для открытия файла используется режим `"a+b"`. Часть `a+` позволяет программе читать весь файл и добавлять данные в конец файла. Часть `b` представляет собой сигнал ANSI, уведомляющий, что программа использует двоичный файловый формат. Для систем Unix, которые не воспринимают `b`, вы можете опустить его, поскольку Unix в любом случае поддерживает только одну форму файлов. В условиях других реализаций языка C, предшествующих стандарту ANSI, вам, возможно, понадобится локальный эквивалент использования `b`.

Двоичный режим был выбран, поскольку функции `fread()` и `fwrite()` предназначены для работы с двоичными файлами. Правда, содержимое некоторых структур представляет собой текст, однако элемент `value` структуры таковым не является. Если вы используете текстовый редактор для просмотра файла `book.dat`, текстовая часть будет отображаться правильно, в то время как числовую часть невозможно будет прочитать, более того, она может стать причиной сбоев вашего текстового редактора.

Команда `rewind()` обеспечивает установку указателя позиции в файле в начало этого файла, то есть файл находится в состоянии готовности к первому чтению.

Начальный цикл `while` считывает одну структуру за раз в массив структур, этот процесс прекращается в случае заполнения массива или в случае исчерпания файла. Переменная `filecount` фиксирует количество прочитанных структур.

Следующий цикл `while` запрашивает ввод пользователя и принимает его. Как и в случае листинга 14.2, этот цикл прекращается в момент заполнения массива или в том случае, когда пользователь нажимает клавишу <Enter> в начале строки. Обратите внимание, что переменная `count` при запуске цикла имеет значение, полученное по окончании предыдущего цикла. Это необходимо для добавления новых записей в конец массива.

Затем цикл `for` печатает данные, полученные как из файла, так и от пользователя. Поскольку файл был открыт в режиме добавления, новые записи добавляются к существующему содержимому файла.

Мы могли бы использовать цикл для разового добавления структур в конец файла. Однако мы приняли решение воспользоваться возможностями функции `fwrite()` по записи более одного блока за раз.

Выражение `count - filecount` принимает значение количества названий новых книг, а вызов функции `fwrite()` позволяет записать такое количество блоков размером со структуру в файл. Выражение `&library[filecount]` — это адрес первой новой структуры в массиве, следовательно, копирование начнется с этой точки.

Этот пример, по-видимому, является простейшим способом записи структур в файл и их поиска, в то же время при этом может иметь место пустая трата пространства памяти, так как возможно запоминание неиспользованных частей структур. Размер этой структуры определяется как `2 × 40 × sizeof (char) + sizeof (float)`, что в сумме дает 84 байта в нашей системе. Ни одна из записей фактически не требует такого пространства. В то же время, один и тот же размер каждой порции данных существенно облегчает задачу поиска данных.

Другой подход предполагает использование записей различных размеров. Чтобы облегчить считывание этих записей из файла, каждая запись должна начинаться с числового поля, определяющего размер записи. Это немного сложнее, чем то, что мы делали раньше. Обычно этот метод предусматривает использование “связных структур”, которые мы будем изучать далее, и динамического распределения памяти, которое рассматривается в главе 16.

Структуры: что дальше?

Прежде чем завершить наши исследования структур, следует коснуться одного из наиболее важных применений структур, а именно — создания новых форм данных. Пользователям компьютера для конкретных задач необходимы намного более эффективные формы данных, чем простые массивы и обыкновенные структуры, которые были представлены выше. Эти формы получили собственные названия, среди которых очереди, двоичные деревья, кучи, хеш-таблицы и графы. Многие формы были построены на базе связных структур. Обычно каждая структура содержит один или два элемента данных плюс один или два указателя на другие структуры того же типа. Эти указатели связывают одну структуру с другой и образуют путь, позволяющий выполнить проход по всему дереву структур.

Например, на рис. 14.3 показано структура бинарного дерева, в котором каждая отдельная структура (или узел) связана с двумя структурами уровнем ниже.

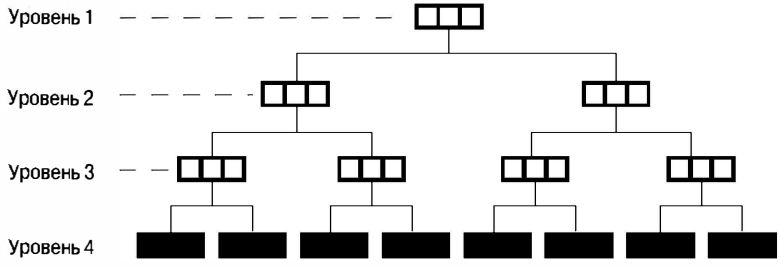


Рис. 14.3. *Структура бинарного дерева*

Является ли иерархическая, или *древовидная*, структура, показанная на рис. 14.3, более эффективной, чем массив? Рассмотрим случай дерева с 10 уровнями узлов. Оно имеет $2^{10} - 1$, или 1023, узла, в которых можно хранить до 1023 слов. Если эти слова упорядочены в соответствии с некоторым смысловым планом, то вы можете начать с наивысшего уровня и найти нужное слово максимум за девять перемещений по мере того, как поиск переходит с одного уровня на следующий. Если эти слова хранились бы в массиве, то для того, чтобы найти нужное слово, вам в худшем случае пришлось просмотреть все 1023 элементов.

Если вас интересуют современные идеи в области структур данных, подобные этой, вы можете обратиться к различным публикациям по компьютерным наукам. Что касается структур в языке C, то вы можете создавать и использовать практически любую форму, описанную в этих статьях. Наряду с этим в главе 17 исследуются некоторые из таких расширенных форм.

На этом мы завершаем обзор структур в данной главе, в то же время в главе 17 будут представлены примеры связанных структур. Далее мы рассмотрим три других средства манипуляции данными в языке C: объединения, перечисления и `typedef`.

Объединения: краткое знакомство

Объединение (union) — это тип, который позволяет хранить различные типы данных в одном и том же пространстве памяти (но не одновременно). Типичным видом объединения может служить таблица, предназначенная для хранения смеси типов в некотором порядке, которые не являются ни регулярными, ни известными заранее. Используя массив объединений, вы можете создать массив элементов одного размера, каждый из которых может содержать различные типы данных. Объединения образуются во многом подобно структурам. Существуют шаблоны объединений и переменные типа объединения. Они могут быть определены с помощью одного или двух действий, причем в последнем случае используется дескриптор объединения. Ниже показан пример шаблона объединения с дескриптором:

```
union hold {
   int digit;
   double bigfl;
   char letter;
};
```

Структура с таким объявлением способна хранить значения типа int, double и char *одновременно*, тогда как объединение может хранить значение типа int *или* double *или* char.

Вот пример определения трех переменных объединения типа hold:

```
union hold fit;            // переменная объединения типа hold
union hold save[10];       // массив из 10 переменных объединения
union hold * pu;           // указатель на переменную типа hold
```

Первое объявление создает единственную переменную fit. Компилятор выделяет пространство памяти, достаточное для того, чтобы хранить наибольшую из описанных вариантов. В этом случае наибольшим из внесенных в список вариантов является тип double, который в нашей системе требует 64 разряда, или 8 байтов. Второе объявление создает массив с именем save с 10 элементами, каждый из которых имеет размер в 8 байтов. Третье объявление создает указатель, который может содержать адрес объединения hold.

Возможна инициализация объединения. Поскольку объединение содержит только одно значение, его правила инициализации отличаются от правил инициализации структуры. В частности, вы можете выбрать один из трех вариантов: инициализировать объединение другим объединением такого же типа, инициализировать первый элемент объединения, либо в условиях действия стандарта C99, воспользоваться выделенным инициализатором:

```
union hold valA;
valA.letter = 'R';
union hold valB = valA;  // инициализация одного объединения другим
union hold valC = {88};  // инициализация числового элемента объединения
union hold valD = {.bigfl = 118.2};   // выделенный инициализатор
```

Объединение можно использовать следующим образом:

```
fit.digit = 23;   // 23 хранится в переменной fit; использовано 2 байта
fit.bigfl = 2.0;  // Значение 23 затерто, хранится 2.0; использовано 8 байтов
fit.letter = 'h'; // Значение 2.0 затерто, хранится h; использован 1 байт
```

Операция точки показывает, какой тип данных используется в текущий момент. За один раз запоминается только одно значение. Нельзя одновременно хранить значение типа char и значение типа int, даже если для этого имеется достаточно пространства. Следить за тем, какие на текущий момент хранятся в объединении, входит в обязанности программиста.

Вы можете воспользоваться операцией -> с указателем на объединение точно так же, как вы используете эту операцию применительно к указателям на структуры:

```
pu = &fit;
x = pu->digit;                    // то же, что и x = fit.digit
```

Ниже показано, как *не* следует поступать:

```
fit.letter = 'A';
flnum = 3.02*fit.bigfl;           // ОШИБКА ОШИБКА ОШИБКА
```

Эта последовательность ошибочна, поскольку запоминается значение типа char, однако в следующей строке предполагается, что значение переменной fit имеет тип double.

Тем не менее, иногда бывает полезно использовать один элемент для размещения значений в объединении, и другой элемент — для просмотра содержимого объединения. В листинге 15.4 следующей главы представлен соответствующий пример.

Другим местом применения объединений является структура, в которой сохраняемая информация зависит от одного из ее элементов. Например, предположим, что в вашем распоряжении имеется структура, представляющая автомобиль. Если автомобиль принадлежит пользователю, вы хотите, чтобы в структуре был элемент, описывающий его владельца. Если автомобиль взят на прокат, вы хотите, чтобы в структуре был элемент, описывающий лизинговую компанию. Затем можно воспользоваться следующими строками:

```
struct owner {
   char socsecurity[12];
   ...
};
struct leasecompany {
   char name[40];
   char headquarters[40];
   ...
};
union data {
   struct owner owncar;
   struct leasecompany leasecar;
};
struct car_data {
   char make[15];
   int status;                          /* 0 = владение, 1 = аренда */
   union data ownerinfo;
   ...
};
```

Предположим, что `flits` — это структура `car_data`. Тогда если элемент `flits.status` имеет значение `0`, программа может использовать значение `flits.ownerinfo.owncar.socsecurity`.

С другой стороны, если элемент `flits.status` равен `1`, программа может использовать значение `flits.ownerinfo.leasecar.name`.

Сводка: операции структур и объединений

Операция принадлежности:

.

Комментарии общего характера:

Эта операция используется с именем структуры или объединения с целью обозначения конкретного элемента этой структуры или объединения. Если `name` — имя структуры, а `member` описан шаблоном структуры, приведенное ниже выражение определяет этот элемент структуры:

`name.member`

Типом элемента `name.member` является тип, определенный для `member`. Операция принадлежности также может использоваться и для объединений.

Пример:

```
struct {
   int code;
   float cost;
} item;
item.code = 1265;
```

Последний оператор присваивает значение элементу `code` структуры `item`.

Косвенная операция принадлежности:

```
->
```

Комментарии общего характера:

Эта операция используется с указателем на структуру или объединение с целью идентификации конкретного элемента этой структуры или объединения. Предположим, что `ptrstr` является указателем на структуру, а `member` — элементом, описанным в шаблоне структуры. В этом случае оператор

```
ptrstr->member
```

идентифицирует этот элемент шаблона структуры. Косвенная операция принадлежности аналогично может использоваться применительно к объединениям.

Пример:

```
struct {
       int code;
       float cost;
} item, * ptrst;
ptrst = &item;
ptrst->code = 3451;
```

Последний оператор присваивает значение типа `int` элементу `code` структуры `item`. Приведенные ниже три выражения эквивалентны:

```
ptrst->code   item.code   (*ptrst).code
```

Перечислимые типы

Перечислимый (*enumerated*) *тип* служит для объявления символических имен, представляющих целочисленные константы. Воспользовавшись ключевым словом `enum`, вы можете создать новый "тип" и описать, какие значения он может принимать.

(По сути дела, константы `enum` имеют тип `int`; следовательно, они могут применяться всякий раз, когда должен использоваться тип `int`.) Назначение перечислимых типов заключается в том, чтобы повысить удобочитаемость программы. Синтаксис в этом случае аналогичен синтаксису, который используется для описания структур. Например, вы можете подготовить следующие объявления:

```
enum spectrum {red, orange, yellow, green, blue, violet};
enum spectrum color;
```

Первое объявление устанавливает `spectrum` как имя дескриптора, который позволяет использовать `enum spectrum` в качестве имени типа. Второе объявление делает `color` переменной этого типа. Идентификаторы, заключенные в фигурные скобки, перечисляют возможные значения, которые может принимать переменная `spectrum`. Возможными значениями `color` являются `red`, `orange`, `yellow` и так далее. Затем можно использовать операторы следующего вида:

```
int c;
color = blue;
if (color == yellow)
   ...;
for (color = red; color <= violet; coloc++)
   ...;
```

И хотя перечислимые константы имеют тип int, перечислимые переменные не так жестко привязаны к целочисленному типу данных, поскольку этот тип может содержать перечислимые константы. Например, перечислимые константы перечислимой переменной spectrum имеют диапазон 0–5, благодаря чему компилятор может выбрать тип unsigned char для представления переменной color.

Между прочим, некоторые свойства перечислений в языке C не переносятся в C++. Например, C позволяет применять операцию ++ к перечислимой переменной, а язык C++ не позволяет. Таким образом, если вы рассчитываете на то, что ваш код когда-нибудь будет переписан на C++, вы должны объявить в предыдущем примере color как тип int. После этого код будет работать как в программах на C, так и на C++.

Константы типа enum

Что собой представляют blue и red? С технической точки зрения они представляют собой константы типа int. Например, имея предыдущее объявление перечислимого типа, можно попытаться выполнить следующий оператор:

```
printf("red = %d, orange = %d\n", red, orange);
```

В результате получается следующий вывод:

```
red = 0, orange = 1
```

Получилось так, что red стала именованной константой, представляющей целочисленное значение 0.

Аналогично, другие идентификаторы являются именованными константами, представляющими целые числа в пределах от 1 до 5. Вы можете использовать перечислимые константы в любом месте, в котором допускаются целочисленные константы. Например, вы можете использовать их как размерности в объявлениях массивов или как метки в операторе swicth.

Значения по умолчанию

По умолчанию константам в перечислимом списке присваиваются целые значения 0, 1, 2 и так далее. В силу этого объявление

```
enum kids {nippy, slats, skippy, nina, liz};
```

приводит к тому, что nina получает значение 3.

Присваиваемые значения

Можно выбрать целые значения, которые должны получить константы. Для этого достаточно включить нужные значения в объявление:

```
enum levels {low = 100, medium = 500, high = 2000};
```

Если вы назначаете конкретное значение одной из констант, но не всем следующим константам, все следующие константы будут перенумерованы последовательно. Например, предположим, имеется следующее объявление:

```
enum feline {cat, lynx = 10, puma, tiger};
```

В этом случае cat получает значение 0 по умолчанию, lynx, puma и tiger, соответственно, получают значения 10, 11 и 12.

Использование ключевого слова enum

Напомним, что назначение перечислимых типов состоит в том, чтобы повысить удобочитаемость программы. Если вы имеете дело с цветами, использование названий red (красный) и blue (голубой) намного информативнее, нежели значений 0 и 1. Обратите внимание, что перечислимые типы предназначены для внутреннего использования. Если вы хотите ввести значение orange для переменной color, вы должны ввести 1, но не слово orange, либо вы можете прочитать строку "orange" и затем преобразовать ее в значение orange.

В силу того, что перечислимый тип является целочисленным типом, переменные типа enum могут использоваться в выражениях подобно целочисленным переменным. Они удобны в использовании в качестве меток в операторе case.

Программа, показанная в листинге 14.15, представляет собой небольшой пример использования типа enum. Пример основан на схеме установки значений по умолчанию, когда red получает значение 0, которое делает его индексом для указателя на строку "red".

Листинг 14.15. Программа enum.c

```
/* enum.c -- использование перечислимых значений */
#include <stdio.h>
#include <string.h>                    // для функции strcmp()
#include <stdbool.h>                   // требование стандарта C99
enum spectrum {red, orange, yellow, green, blue, violet};
const char * colors[] = {"красный", "оранжевый", "желтый",
                         "зеленый", "голубой", "фиолетовый"};
#define LEN 30
int main(void)
{
   char choice[LEN];
   enum spectrum color;
   bool color_is_found = false;
   puts("Введите цвет (или пустую строку для выхода из программы):");
   while (gets(choice) != NULL && choice[0] != '\0')
   {
       for (color = red; color <= violet; color++)
       {
           if (strcmp(choice, colors[color]) == 0)
           {
               color_is_found = true;
               break;
           }
       }
```

```
      if (color_is_found)
         switch(color)
         {
                case red : puts("Розы красные.");
                           break;
                case orange : puts("Маки оранжевые.");
                           break;
                case yellow : puts("Подсолнухи желтые.");
                           break;
                case green : puts("Трава зеленая.");
                           break;
                case blue : puts("Колокольчики голубые.");
                           break;
                case violet : puts("Фиалки фиолетовые.");
                           break;
         }
      else
         printf("Мне не известен %s цвет.\n", choice);
      color_is_found = false;
      puts("Введите следующий цвет (или пустую строку для выхода из программы):");
   }
   puts("Всего доброго!");
   return 0;
}
```

Программа выходит из цикла for, если входная строка совпадает с одной из строк, на которую указывает один из элементов массива colors. Если цикл находит соответствующий цвет, то программа использует значение перечислимой переменной для сопоставления с перечислимыми константами, которые указаны в качестве меток в операторе switch.

Ниже показаны результаты выполнения этой демонстрационной программы:

```
Введите цвет (или пустую строку для выхода из программы):
голубой
Колокольчики голубые.
Введите следующий цвет (или пустую строку для выхода из программы):
оранжевый
Маки оранжевые.
Введите следующий цвет (или пустую строку для выхода из программы):
пурпурный
Мне не известен пурпурный цвет.
Введите следующий цвет (или пустую строку для выхода из программы):
Всего доброго!
```

Совместно используемые пространства имен

Термин *пространство имен* (namespace) в языке C применяется для идентификации тех частей программы, в которых это имя распознается. Область видимости является частью этой концепции: две переменные, имеющие одно и то же имя в различных областях видимости, не вступают в конфликт друг с другом. Существуют категории про-

странств имен. Дескрипторы структур, дескрипторы объединений и дескрипторы перечислений в конкретной области видимости совместно используют (разделяют) одно и то же пространство имен, и это пространство имен отличается от пространства, используемого обычными переменными. Это означает, что вы можете выбирать одно и то же имя для одной переменной и одного дескриптора в одной и той же области видимости, и при этом не возникнет ошибка. В то же время вы не можете объявить два дескриптора с одним и тем же именем или две одноименных переменных в одной и той же области видимости. Например, следующие объявления не вызывают конфликта в C:

```
struct rect { double x; double y; };
int rect;                                    // конфликта в C нет
```

В то же время использование одного идентификатора двумя разными путями может привести к путанице, кроме того, C++ не допускает этого, поскольку размещает дескрипторы и переменные в одном и том же пространстве имен.

Оператор typedef: краткое знакомство

Оператор typedef представляет собой усовершенствованное средство манипулирования данными, которое позволяет создавать собственные имена для конкретных типов. В этом отношении он подобен директиве #define, но с тремя отличиями:

- В отличие от директивы #define, оператор typedef ограничивается присвоением символических имен только типам, но не значениям.
- Интерпретация оператора typedef выполняется компилятором, а не препроцессором.
- В рамках своей области действия оператор typedef является более гибким средством, нежели директива #define.

Рассмотрим, как работает typedef. Предположим, что вы хотите использовать термин BYTE для однобайтовых чисел. Вы просто объявляете BYTE, как если бы это была переменная типа char, при этом данному определению должно предшествовать ключевое слово typedef:

```
typedef unsigned char BYTE;
```

После этого вы можете использовать слово BYTE для определения переменных:

```
BYTE x, y[10], * z;
```

Область видимости этого определения зависит от местоположения оператора typedef. Если определение находится внутри функции, областью видимости будет локальная область, ограниченная пределами этой функции. Если определение находится вне функции, область видимости глобальная.

Для определений typedef довольно часто применяются буквы верхнего регистра, чтобы напомнить пользователю о том, что имя типа, по сути, является символической аббревиатурой, тем не менее, разрешено использовать и буквы нижнего регистра, например:

```
typedef unsigned char byte;
```

Имена, указываемые в typedef, подчиняются тем же правилам, которые регламентируют создание допустимых имен переменных.

Создание имени для существующего типа может показаться средством отнюдь не первой необходимости, тем не менее, оно может приносить определенную пользу. Что касается предыдущего примера, то указание типа BYTE вместо unsigned char позволяет документировать ваше намерение использовать переменные типа BYTE для представления чисел, а не кодов символов. Например, ранее мы упоминали тип size_t, который представляет тип, возвращаемый оператором sizeof, и тип time_t, который представляет тип, возвращаемый функцией time(). Стандарт C утверждает, что оператор sizeof и функция time() возвращают целочисленные типы, и в то же время оставляет на усмотрение реализации, каковыми должны быть эти типы. Причина такой неопределенности заключается в том, что комитет ANSI придерживается того мнения, что ни один выбор, скорее всего, не будет наилучшим для всех компьютерных платформ. В результате был создан новый тип, такой как time_t, а конкретным реализациям языка было разрешено использовать оператор typedef применительно к некоторым специальным типам. Таким образом, был предложен обобщенный прототип:

```
time_t time(time_t *);
```

В одной системе тип time_t может быть unsigned int; в другой системе — unsigned long. Если вы включаете в программу заголовочный файл time.h, ваша программа имеет доступ к соответствующему определению, и вы имеете возможность объявлять в коде переменные типа time_t.

Некоторые свойства оператора typedef можно продублировать с помощью директивы #define.

Например, определение

```
#define BYTE unsigned char
```

заставляет препроцессор заменять BYTE на unsigned char. Ниже показан пример, когда такое дублирование посредством #define невозможно:

```
typedef char * STRING;
```

Без ключевого слова typedef этот пример идентифицирует саму STRING как указатель на значение типа char. Вместе с ключевым словом объявление делает STRING идентификатором указателя на значение типа char.

В связи с этим,

```
STRING name, sign;
```

означает

```
char * name, * sign;
```

Предположим, что вместо этого вы воспользовались объявлением:

```
#define STRING char *
```

Тогда оператор

```
STRING name, sign;
```

преобразуется к следующему оператору:

```
char * name, sign;
```

В этом случае только name может быть указателем.

Вы также можете применить оператор typedef к структурам:

```
typedef struct complex {
        float real;
        float imag;
} COMPLEX;
```

Теперь для представления комплексных чисел можно использовать тип COMPLEX вместо структуры с именем complex. Одна из целей применения оператора typedef состоит в том, чтобы создать удобные, распознаваемые имена типов, которые часто используются. Например, многие программисты предпочитают использовать имя типа STRING или его эквивалент, как в приведенном выше примере.

Вы можете опустить дескриптор, когда применяете typedef к имени типа структуры:

```
typedef struct {double x; double y;} rect;
```

Предположим, что вы используете оператор typedef следующим образом:

```
rect r1 = {3.0, 6.0};
rect r2;
```

Эти программные коды преобразуются в приведенные ниже операторы:

```
struct {double x; double y;} r1= {3.0, 6.0};
struct {double x; double y;} r2;
r2 = r1;
```

Если две структуры объявлены без дескриптора, но с одинаковыми элементами (имена элементов и типов совпадают), то в C эти две структуры рассматриваются как имеющие один и тот же тип, таким образом, присваивание r2 значения r1 является допустимой операцией.

Вторая цель применения typedef заключается в том, что имена, присвоенные typedef, часто используются для именования сложных типов. Например, описание

```
typedef char (* FRPTC ()) [5];
```

делает FRPTC объявлением типа, которым является функция, возвращающая указатель на массив из пяти элементов типа char. (См. обсуждение фиктивных объявлений в следующем разделе.)

При использовании оператора typedef всегда имейте в виду, что он не создает новых типов, а всего лишь предоставляет удобные для использования метки типов. Это значит, например, что объявленные нами переменные типа STRING могут использоваться в качестве аргументов функций, ожидающих тип указателя на char.

Предлагая структуры, объединения и оператор typedef, язык C обеспечивает вас инструментальными средствами эффективного и переносимого манипулирования данными.

Фиктивные объявления

Язык C позволяет создавать сложные формы данных. Несмотря на то что мы является приверженцами максимально простых форм, все же имеет смысл раскрыть все доступные возможности. Когда вы формулируете то или иное объявление, имя (или идентификатор) может быть модифицировано за счет добавления соответствующего модификатора.

Модификатор	*Описание*
*	Обозначает указатель.
()	Обозначает функцию.
[]	Обозначает массив.

Язык C позволяет использовать одновременно нескольких модификаторов, а это в свою очередь дает возможность создавать множество типов, как показано в следующих примерах:

```
int board[8][8];     // массив массивов значений int
int ** ptr;          // указатель на указатель на int
int * risks[10];     // 10-элементный массив указателей на int
int (* rusks)[10];   // указатель на массив из 10 значений int
int * oof[3][4];     // массив размером 3 x 4 указателей на значения int
int (* uuf)[3][4];   // указатель на массив размером 3 x 4 значений int
int (* uof[3])[4];   // 3-элементный массив указателей на 4-элементные
                     // массивы значений типа int
```

Искусство распутывания таких объявлений состоит в определении порядка, в котором необходимо применять модификаторы. Следующие правила позволяют успешно решить эту задачу.

1. Скобки `[]`, которые обозначают массив, и скобки `()`, обозначающие функцию, обладают одним и тем же приоритетом. Этот приоритет выше, чем приоритет операции разыменования `*`, которая означает, что приведенное ниже объявление делает `risks` массивом указателей, а не указателем на массив:

   ```
   int * risks[10];
   ```

2. Для скобок `[]` и `()` справедлива ассоциативность слева направо. Следующее объявление делает `goods` массивом 12 массивов по 50 значений типа `int` каждый:

   ```
   int goods[12][50];
   ```

3. Как скобки `[]`, так и скобки `()` имеют один и тот же приоритет, но поскольку для них выполняется закон ассоциативности слева направо, в следующем ниже объявлении `*` и `rusks` группируются, а затем к ним применяются квадратные скобки. Это означает, что представленное далее объявление превращает `rusks` в указатель на массив из 10 значений типа `int`:

   ```
   int (* rusks)[10];
   ```

Применим эти правила к следующему объявлению:

```
int * oof[3][4];
```

Конструкция `[3]` имеет более высокий приоритет, чем операция разыменования `*`, и поскольку действует правило ассоциативности слева направо, она имеет более высокий приоритет, чем `[4]`. Отсюда следует, что `oof` — это массив из трех элементов. Следующим по порядку идет `[4]`, таким образом, элементы массива `oof` — массивы из четырех элементов. Операция `*` говорит о том, что эти элементы являются указателями. Картину завершает значение типа `int`: `oof` представляет собой трехэлементный массив четырехэлементных массивов указателей на `int`, или, короче, массив указате-

лей на значения типа int размерности 3×4. Под 12 указателей резервируется необходимая память.

Теперь рассмотрим представленное ниже объявление:

```
int (* uuf)[3][4];
```

Круглые скобки приводят к тому, что модификатор * получает первый приоритет, благодаря чему uuf становится указателем на массив значений типа int размерности 3×4. Память выделяется только под один указатель.

Эти правила позволяют иметь дело со следующими типами:

```
char * fump();            // функция, возвращающая указатель на тип char
char (* frump)();         // указатель на функцию, возвращающую тип char
char (* flump[3])();      // массив из 3 указателей на функцию, которая
                          // возвращает тип char
```

Оператором typedef можно воспользоваться для создания последовательности родственных типов данных:

```
typdef int arr5[5];
typedef arr5 * p_arr5;
typedef p_arr5 arrp10[10];
arr5 togs;   // togs - массив из 5 значений типа int
p_arr5 p2;   // p2 - указатель на массив из 5 значений int
arrp10 ap;   // ap - массив 10 указателей на массив из 5 значений типа int
```

После того, как вы научитесь создавать все эти структуры, вы сможете объявлять самые замысловатые и причудливые формы. А что касается конкретных их применений, то мы отсылаем вас к специальной литературе.

Функции и указатели

Как показали проведенные выше обсуждения, можно объявлять указатели на функции. Вас, по-видимому, заинтересует вопрос, есть ли какая-либо польза от всех этих сложных объявлений. Как правило, указатель функции используется в качестве аргумента другой функции, указывая ей, какую функцию необходимо использовать. Например, сортировка массива предусматривает сравнение двух элементов для определения, какой из них должен быть первым. Если элементы являются числовыми, можно воспользоваться операцией >. В общем случае элементом может быть строка или структура, требующая вызова специальной функции для выполнения сравнения. Функция qsort() из библиотеки C предназначена для работы с массивами любой природы, если вы уведомите ее о том, какую функцию применять для сравнения элементов. С этой целью она принимает указатель на функцию в качестве одного из своих аргументов. Затем функция qsort() использует эту функцию для сортировки типов, то есть, является ли тип целым числом, строкой или структурой.

Остановимся подробнее на указателях на функции. Прежде всего, что они собой представляют? Указатель, например, значение типа int, содержит адрес ячейки памяти, в которой можно сохранить значение типа int. Функции также имеют адреса в силу того, что реализация функции на машинном языке состоит из программного кода, загруженного в память. Указатель на функцию может содержать адрес, соответствующий началу кода функции.

Далее, когда вы объявляете указатель на данные, вы должны объявить тип данных, на которые он указывает. При объявлении указателя на функцию вы должны объявить тип функции, на которую ссылается указатель. Чтобы определить тип функции, вы задаете тип возвращаемого значения этой функции и типы параметров функции. Например, рассмотрим следующий прототип:

```
void ToUpper(char *);   // преобразование строки к верхнему регистру
```

Тип функции `ToUpper()` определяется как "функция с параметром `char *` и возвращаемым типом `void`". Чтобы объявить указатель `pf` на этот тип данных, воспользуйтесь следующим кодом:

```
void (*pf)(char *);     // pf - указатель на функцию
```

Читая это объявление, вы видите, что первая пара круглых скобок связывает операцию `*` с `pf`, а это означает, что `pf` является указателем на функцию. В результате выражение `(*pf)` становится функцией, при этом `(char *)` выступает в качестве списка параметров функции, а `void` — возвращаемым типом. Возможно, простейший способ создания такого объявления связан с учетом того факта, что оно заменяет имя функции `ToUpper` выражением `(*pf)`. Следовательно, если вы хотите объявить указатель на некоторый конкретный тип функции, вы можете объявить функцию этого типа, а затем заменить имя функции выражением вида `(*pf)` и в результате получить объявление указателя на функцию. Как было отмечено выше, первые круглые скобки необходимы для того, чтобы придать операции необходимый приоритет. Если их опустить, мы получим совсем не то, что нужно:

```
void *pf(char *);       // pf - функция, возвращающая указатель
```

Совет

Чтобы объявить указатель на конкретный тип функции, потребуется сначала объявить функцию нужного типа, а затем заменить имя функции выражением вида `(*pf)`; при этом `pf` становится указателем на функцию выбранного типа.

После получения указателя на функцию ему можно присвоить адрес функции соответствующего типа. В этом контексте для представления адреса функции может использоваться ее *имя*:

```
void ToUpper(char *);
void ToLower(char *);
int round(double);
void (*pf)(char *);
pf = ToUpper;           // допустимо, ToUpper - адрес функции
pf = ToLower;           // допустимо, ToLower - адрес функции
pf = round;             // недопустимо, round - неподходящий тип функции
pf = ToLower();         // недопустимо, ToLower() не является адресом
```

Последняя операция присваивания также недопустима, поскольку вы не можете использовать функцию `void` в операторе присваивания. Обратите внимание, что указатель `pf` может указывать на любую функцию, которая принимает аргумент `char *` и имеет `void` в качестве возвращаемого типа, но не на функции с другими характеристиками.

Подобно тому, как можно использовать указатель на данные с целью доступа к данным, вы можете применять указатель на функцию для доступа к самой функции. Как ни странно, но существуют два логически несовместимых синтаксических правила, позволяющие сделать это, как показано в следующих примерах:

```
void ToUpper(char *);
void ToLower(char *);
void (*pf)(char *);
char mis[] = "Nina Metier";
pf = ToUpper;
(*pf)(mis);           // применить ToUpper к массиву mis (синтаксис 1)
pf = ToLower;
pf(mis);              // применить ToLower к массиву mis (синтаксис 2)
```

Каждый из показанных подходов логически обоснован. Суть первого подхода заключается в следующем: поскольку `pf` указывает на функцию `ToUpper`, `*pf` является функцией `ToUpper`, следовательно, выражение `(*pf)(mis)` равносильно `ToUpper(mis)`. Достаточно проанализировать объявления `ToUpper` и `pf`, чтобы прийти к выводу, что `ToUpper` и `(*pf)` эквивалентны. Второй поход состоит в следующем: поскольку имя функции представляет собой указатель, вы можете использовать указатель и имя функций поочередно, следовательно, `pf(mis)` — то же, что и `ToLower(mis)`. Достаточно взглянуть на оператор присваивания для `pf`, чтобы заметить, что `pf` и `ToLower` эквивалентны. Исторически сложилось так, что разработчики языка C и операционной системы Unix в Bell Labs были сторонниками первого подхода, а разработчики расширений Unix в университете Беркли отдали предпочтение второму подходу. Компилятор K&R C не допускает второй формы, в то же время, чтобы сохранить совместимость с существующими программными кодами, стандарт ANSI C трактует обе формы как эквивалентные.

Одним из наиболее часто встречающихся применений указателей на данные является их использование в качестве аргументов функций. Например, рассмотрим следующий прототип функции:

```
void show(void (* fp)(char *), char * str);
```

И хотя этот оператор выглядит сложным для восприятия, тем не менее, он объявляет два параметра, `fp` и `str`. Параметр `fp` представляет собой указатель на функцию, а `str` — указатель на данные. Если конкретно, то `fp` указывает на функцию, которая принимает параметр `char *` и имеет возвращаемый тип `void`, а `str` указывает на значение типа `char`. Следовательно, в условиях представленных выше объявлений вы можете выполнять вызовы функций, подобные показанным ниже:

```
show(ToLower, mis); /* show() использует функцию ToLower(): fp = ToLower */
show(pf, mis);      /* show() использует функцию,
                       на которую указывает pf: fp = pf */
```

Каким образом функция `show()` использует указатели на функции, которые ей передаются? Она использует либо синтаксис `fp()`, либо синтаксис `(*fp)()` для обращения к функции:

```
void show(void (* fp)(char *), char * str)
{
   (*fp)(str);      /* применить выбранную функцию к str */
   puts(str);       /* вывести результат */
}
```

Например, в данном случае функция show() сначала преобразует строку str путем применения к ней функции, на которую указывает fp, после чего она воспроизводит на экране преобразованную строку.

Кстати, функции с возвращаемыми значениями могут использоваться двумя различными способами в качестве аргументов других функций. Например, рассмотрим следующие операторы:

```
function1(sqrt);       /* передает адрес функции sqrt */
function2(sqrt(4.0));  /* передает значение, возвращаемое функцией sqrt */
```

Первый оператор передает адрес функции sqrt(), и предположительно функция function1() будет использовать указанную выше функцию в своем программном коде. Второй оператор сначала вызывает функцию sqrt(), производит с ее помощью необходимые вычисления, а затем передает возвращаемое значение (в рассматриваемом случае это 2.0) функции function2().

Для демонстрации основных идей программа, показанная в листинге 14.16, использует функцию show() вместе с множеством других функций, выполняющих преобразования, в качестве аргументов. Кроме того, в листинге 14.16 представлены эффективные способы построения и организации меню.

Листинг 14.16. Программа func_ptr.c

```
// func_ptr.c -- использование указателей на функции
#include <stdio.h>
#include <string.h>
#include <ctype.h>

char showmenu(void);
void eatline(void);              // читать до конца строки
void show(void (* fp)(char *), char * str);
void ToUpper(char *);            // преобразовать строку к верхнему регистру
void ToLower(char *);            // преобразовать строку к нижнему регистру
void Transpose(char *);          // перестановка регистров
void Dummy(char *);              // строка остается неизменной

int main(void)
{
    char line[81];
    char copy[81];
    char choice;
    void (*pfun)(char *);        // указывает на функцию, которая использует
                                 // значение типа char * в качестве аргумента
                                 // и не имеет возвращаемого значения
    puts("Введите строку (пустая строка - выход из программы):");
    while (gets(line) != NULL && line[0] != '\0')
    {
        while ((choice = showmenu()) != 'n')
        {
            switch (choice )        // установка указателя
            {
                case 'u' : pfun = ToUpper; break;
                case 'l' : pfun = ToLower; break;
                case 't' : pfun = Transpose; break;
```

```
                case 'o' : pfun = Dummy; break;
            }
            strcpy(copy, line);    // копия для функции show()
            show(pfun, copy);      // используется избранная функция
        }
        puts("Введите строку (пустая строка - выход из программы):");
    }
    puts("Всего доброго!");
    return 0;
}
char showmenu(void)
{
    char ans;
    puts("Введите номер выбранного элемента меню:");
    puts("u) нижний регистр         l) верхний регистр");
    puts("t) перестановка регистров o) исходный регистр");
    puts("n) следующая строка");
    ans = getchar();               // ввод ответа
    ans = tolower(ans);            // перевод в нижний регистр
    eatline();                     // удаление оставшейся части строки
    while (strchr("ulton", ans) == NULL)
    {
        puts("Пожалуйста, введите a u, l, t, o или n:");
        ans = tolower(getchar());
        eatline();
    }
    return ans;
}
void eatline(void)
{
    while (getchar() != '\n')
        continue;
}
void ToUpper(char * str)
{
    while (*str)
    {
        *str = toupper(*str);
        str++;
    }
}
void ToLower(char * str)
{
    while (*str)
    {
        *str = tolower(*str);
        str++;
    }
}
```

```
void Transpose(char * str)
{
    while (*str)
    {
        if (islower(*str))
            *str = toupper(*str);
        else if (isupper(*str))
             *str = tolower(*str);
        str++;
    }
}
void Dummy(char * str)
{
                                // оставляет строку неизменной
}
void show(void (* fp)(char *), char * str)
{
    (*fp)(str);                 // применяет избранные функции к строке str
    puts(str);                  // отображает результат
}
```

Ниже показаны выходные данные, полученные в результате выполнения этой программы:

```
Введите строку (пустая строка - выход из программы):
Does C make you feel loopy?
Введите номер выбранного элемента меню:
u) верхний регистр        l) нижний регистр
t) перестановка регистров o) исходный регистр
n) следующая строка
t
dOES c MAKE YOU FEEL LOOPY?
Введите номер выбранного элемента меню:
u) верхний регистр        l) нижний регистр
t) перестановка регистров o) исходный регистр
n) следующая строка
l
does c make you feel loopy?
Введите номер выбранного элемента меню:
u) верхний регистр        l) нижний регистр
t) перестановка регистров o) исходный регистр
n) следующая строка
n
Введите строку (пустая строка - выход из программы):
Всего доброго!
```

Следует отметить, что функции `ToUpper()`, `ToLower()`, `Transpose()` и `Dummy()` имеют один и тот же тип, следовательно, все они могут быть присвоены указателю `pfun`. Эта программа использует указатель `pfun` как аргумент функции `show()`, но вы можете также передавать любое из имен этих четырех функций непосредственно в аргументе, например, `show(Transpose, copy)`.

В ситуациях подобного рода можно воспользоваться typedef. Например, программа могла выполнить следующие операторы:

```
typedef void (*V_FP_CHARP)(char *);
void show (V_FP_CHARP fp, char *);
V_FP_CHARP pfun;
```

Если вы не боитесь риска, можете объявить и инициализировать такой массив указателей:

```
V_FP_CHARP arpf[4] = {ToUpper, ToLower, Transpose, Dummy};
```

Если затем модифицировать функцию showmenu() таким образом, чтобы она имела тип int и возвращала 0, если пользователь вводит символ u, 1, если l, 2, если t, и так далее, то можно заменить цикл, содержащий оператор switch, следующим кодом:

```
index = showmenu();
while (index >= 0 && index <= 3)
{
    strcpy(copy, line);            /* получить копию функции show() */
    show(arpf[index], copy);       /* использовать выбранную функцию */
    index = showmenu();
}
```

Невозможно иметь массив функций, однако можно иметь массив указателей на функции.

К этому моменту вы увидели все способы использования имени функции: в определении, в объявлении, в вызове и в качестве указателя на функцию (рис. 14.4).

Имя функции содержится в объявлении прототипа:	`int comp(int x, int y);`
Имя функции используется в вызове функции:	`status = comp(q,r);`
Имя функции используется в определении функции:	`int comp(int x, int y);` `{ ...`
Имя функции используется как указатель в присваивании:	`pfunct = comp;`
Имя функции используется как указатель аргумента:	`slowsort(arr,n,comp);`

Рис. 14.4. *Применение имени функции*

Что касается поддержки меню, то функция showmenu() позволяет делать это несколькими методами. Во-первых, программные коды

```
ans = getchar();              // ввод ответа
ans = tolower(ans);           // перевод в нижний регистр
```

и

```
ans = tolower(getchar());
```

демонстрируют два способа преобразования данных, введенных пользователем, к одному регистру, дабы не пришлось проверять на равенство 'u' и 'U' и так далее.

Функция eatline() отбрасывает за ненадобностью оставшуюся часть введенной строки. Это полезно по двум причинам.

Во-первых, пользователь сначала вводит с клавиатуры букву, после чего нажимает клавишу <Enter>, в результате чего появляется символ новой строки. Если не избавиться от этого символа, он будет рассматриваться как следующий ответ.

Во-вторых, предположим, что вместо буквы `u` пользователь вводит слово *uppercase* полностью. Без функции `eatline()` программа рассматривает каждый символ слова *uppercase* как отдельный ответ. Благодаря функции `eatline()`, программа обрабатывает символ `u` и отбрасывает остальную часть строки.

Далее, функция `showmenu()` разрабатывалась с таким расчетом, чтобы возвращать только один вариант выбора в программу. Для облегчения этой задачи программа использует стандартную библиотечную функцию `strchr()` из заголовочного файла `string.h`:

```
while (strchr("ulton", ans) == NULL)
```

Эта функция ищет первое вхождение символа `ans` в строке `"ulton"` и возвращает указатель на него. Если она не находит этот символ, она возвращает нулевой указатель. По этой причине существует более удобная версия цикла `while`:

```
while (ans != 'u' && ans != 'l' && ans != 't' && ans != 'o' && ans != 'n')
```

Чем больше вариантов приходится проверять, тем более эффективным становится использование функции `strchr()`.

Ключевые понятия

Информация, которая необходима для решения задачи по программированию, зачастую выходит далеко за рамки какого-то одного числа или их списка. Программа может иметь дело с некоторой сущностью или коллекцией сущностей, обладающих множеством свойств. Например, вы можете представлять клиента посредством его имени и фамилии, адреса, номера телефона и другой информации. Либо вы можете описывать DVD-диск, указывая его название, поставщика, длительность, стоимость и прочие данные. Структура C позволяет накапливать всю эту информацию в одном элементе. Это очень полезно при организации программы. Вместо того чтобы хранить информацию в переменных, разбросанных по разным частям программы, вы можете хранить всю необходимую информацию в одном месте.

При проектировании структуры часто бывает удобным разработать пакет функций и в дальнейшем пользоваться только им. Например, вместо того, чтобы писать набор операторов `printf()` каждый раз, когда необходимо отобразить на экране содержимое той или иной структуры, вы можете написать функцию отображения, которая принимает данную структуру в качестве аргумента. Поскольку вся информация содержится в структуре, достаточно будет одного аргумента. Если вы должны распределить эту информацию по отдельным переменным, вы должны использовать отдельный аргумент для каждой конкретной части. Наряду с этим, если вы, скажем, добавляете элемент в структуру, вы должны внести соответствующие изменения в функции, но при этом не потребуется изменять вызовы функций, что является большим удобством в плане изменения программы.

Объявление объединения во многом подобно объявлению структуры. Однако элементы объединения совместно используют одно и то же пространство памяти, и только один элемент может содержать данные в конкретный момент времени. По сути де-

ла, объединение позволяет создать переменную, которая может содержать одно значение, но более чем одного типа.

Средство enum предлагает возможность определения символических констант, а средство typedef — возможность для создания нового идентификатора для базового или производного типа.

Указатели на функции обеспечивают средство уведомления конкретной функции о том, какие функции она должна использовать.

Резюме

Структуры языка C служат средством сохранения нескольких элементов данных, обычно разных типов, в одном и том же объекте данных. Для идентификации шаблона и объявления переменных данного типа можно воспользоваться дескрипторами. Операция принадлежности (.) позволяет получить доступ к отдельным элементам структуры через использование меток шаблона структуры.

Если у вас есть указатель на конкретную структуру, для доступа к отдельным элементам структуры вы можете использовать указатель и косвенную операцию принадлежности (->) вместо имени и операции точки. Получить адрес структуры можно с помощью операции &. В отличие от массивов, имя структуры не может служить адресом структуры.

По традиции функции, ориентированные на работу со структурами, используют указатели на структуры в качестве аргументов. Современная версия C допускает передачу структур в качестве аргументов, использование структур в качестве возвращаемых значений и выполнение операций присваивания над структурами одного и того же типа.

Объединения имеют тот же синтаксис, что и структуры. Тем не менее, в случае структур элементы используют одно и то же пространство памяти. Вместо того, чтобы одновременно хранить несколько типов данных в виде структуры, объединение хранит один тип данных из списка возможных значений. Другими словами, структура может хранить, например, значение типа int, значение типа double и значение типа char, а соответствующее объединение может содержать либо int либо double либо char.

Перечислимые типы позволяют создавать группы символических целочисленных констант (перечислимых констант) и объявлять связанные с ними перечислимые типы.

Операция typedef позволяет устанавливать альтернативные имена или сокращенные представления стандартных типов языка C.

Имя функции служит ее адресом. Такие адреса могут передаваться функциям, которые в дальнейшем используют функции, заданные этими адресами. Если pf является указателем на функцию, которому был присвоен адрес конкретной функции, вы можете вызвать эту функцию двумя способами:

```
#include <math.h>       /* объявление функции double sin(double) */
...
double (*pdf)(double);
double x;

pdf = sin;
x = (*pdf)(1.2);        // вызывает функцию sin(1.2)
x = pdf(1.2);           // также вызывает функцию sin(1.2)
```

Вопросы для самоконтроля

1. Какая ошибка допущена в следующем шаблоне?

```
structure {
       char itable;
       int num[20];
       char * togs
}
```

2. Ниже приведен фрагмент программы. Что он выводит на печать?

```
#include <stdio.h>
struct house {
    float sqft;
    int rooms;
    int stories;
    char address[40];
};
int main(void)
{
   struct house fruzt = {1560.0, 6, 1, "22 Spiffo Road"};
   struct house *sign;
   sign = &fruzt;
   printf("%d %d\n", fruzt.rooms, sign->stories);
   printf("%s \n", fruzt.address);
   printf("%c %c\n", sign->address[3], fruzt.address[4]);
   return 0;
}
```

3. Разработайте шаблон структуры, которая содержит название месяца, трехбуквенную аббревиатуру месяца, количество дней в месяце и номер месяца.

4. Определите массив из 12 структур вида, описанного в вопросе 3, и инициализируйте ее для года, отличного от високосного.

5. Напишите функцию, которая, получив номер месяца, возвращает общее количество дней с начала года до окончания указанного месяца. Примите, что шаблон структуры, описанный в вопросе 3, и соответствующий массив таких структур объявлены за пределами функции.

6. а. С помощью typedef объявите 10-элементный массив указанных структур. Затем, воспользовавшись операцией присваивания применительно к отдельным элементам структуры (эквиваленту строки), сделайте так, чтобы третий элемент этого массива описывал объектив Remarkatar с фокусным расстоянием 500 мм и диафрагмой f/2.0.

```
typedef struct lens {      /* дескриптор структуры lens */
   float foclen;           /* фокусное расстояние в миллиметрах */
   float fstop;            /* диафрагма */
   char brand[30];         /* марка производителя */
} LENS;
```

б. Повторите часть а), но при этом воспользуйтесь списком инициализации с выделенным инициализатором в объявлении, но не используйте отдельные операторы присваивания для каждого элемента.

7. Рассмотрим следующий фрагмент программного кода:

```
struct name {
        char first[20];
        char last[20];
};
struct bem {
        int limbs;
        struct name title;
        char type[30];
};
struct bem * pb;
struct bem deb = {
        6,
        {"Berbnazel", "Gwolkapwolk"},
         "Arcturan"
};
pb = &deb;
```

а. Что распечатают следующие операторы?

```
printf("%d\n", deb.limbs);
printf("%s\n", pb->type);
printf("%s\n", pb->type + 2);
```

б. Как вы можете представить "Gwolkapwolk" в форме записи, используемой для структур (два способа)?

в. Напишите функцию, которая принимает адрес структуры bem в качестве своего аргумента и печатает содержимое этой структуры в форме, представленной ниже (предполагается, что шаблон структуры определен в заголовочном файле starfolk.h):

Berbnazel Gwolkapwolk is a 6-limbed Arcturan.

8. Рассмотрим следующие объявления:

```
struct fullname {
                char fname[20];
                char lname[20];
                };
struct bard     {
                struct fullname name;
                int born;
                int died;
                };
struct bard willie;
struct bard *pt = &willie;
```

а. Идентифицируйте элемент born структуры willie с помощью идентификатора willie.

б. Идентифицируйте элемент born структуры willie с помощью идентификатора pt.

в. С помощью функции scanf() прочитайте значение элемента born, используя идентификатор willie.

г. С помощью функции scanf() прочитайте значение элемента born, используя идентификатор pt.

д. С помощью функции scanf() прочитайте значение элемента lname структуры name, используя идентификатор willie.

е. С помощью функции scanf() прочитайте значение элемента lname структуры name, используя идентификатор pt.

ж. Создайте идентификатор для третьей буквы фамилии индивидуума, описанного переменной willie.

з. Напишите выражение, представляющее общее количество букв в фамилии и имени индивидуума, описанного переменной willie.

9. Определите шаблон структуры, способной хранить следующие данные: марка автомобиля, его мощность в лошадиных силах, рейтинг в смысле экологии, колесная база и год выпуска. Используйте в качестве дескриптора шаблона имя car.

10. Предположим, что имеется следующая структура:

```
struct gas {
        float distance;
        float gals;
        float mpg;
};
```

а. Напишите функцию, которая принимает в качестве аргумента struct gas. Предположим, что передаваемая структура содержит информацию distance и gals. Эта функция должна вычислять правильное значение элемента mpg и возвращать заполненную структуру.

б. Напишите функцию, которая принимает в качестве аргумента адрес struct gas. Предположим, что передаваемая структура содержит информацию distance и gals. Эта функция должна вычислять правильное значение элемента mpg и присваивать ее соответствующему элементу.

11. Объявите перечисление с дескриптором, которое устанавливает перечислимые константы no, yes и maybe, представленные, соответственно, как 0, 1 и 2.

12. Объявите указатель на функцию, который возвращает указатель на char и принимает в качестве аргументов указатель на char и значение типа char.

13. Объявите четыре функции и инициализируйте массив указателей, указывающих на эти функции. Каждая функция должна принимать два аргумента типа double и возвращать значение типа double.

Упражнения по программированию

1. Переделайте задание из вопроса 3 таким образом, чтобы аргумент был представлен названием месяца, а не его номером. (Можете воспользоваться функцией `strcmp()`.)
2. Напишите программу, которая приглашает пользователя ввести день, месяц и год. Месяц может быть представлен порядковым номером, названием или аббревиатурой. Затем программа должна возвратить общее количество дней, истекших с начала года по указанный день включительно.
3. Измените программу, представленную в листинге 14.2, таким образом, чтобы она сначала распечатывала описания книг в том порядке, в каком они вводились, затем в алфавитном порядке названий книг и, наконец, в порядке возрастания их цен.
4. Напишите программу, которая создает шаблон структуры с двумя элементами в соответствии со следующими критериями:
   а. Первым элементом является номер карточки социального страхования. Второй элемент — это структура, состоящая из трех элементов. Ее первый элемент содержит имя, второй элемент — фамилию и третий элемент — отчество. Создайте и инициализируйте массив из пяти таких структур. Программа должна распечатывать данные в следующем формате:

      ```
      Петруччо, Айвен В. -- 302039823
      ```

      Печататься должна только начальная буква отчества, за которой следует точка. Разумеется, если этот элемент пуст, не печатается ни инициал, ни точка. Напишите функцию, которая выполняет такую печать, передайте рассматриваемую структуру этой функции.
   б. Измените задание пункта а) таким образом, чтобы вместо адреса передавалось значение этой структуры.
5. Напишите программу, которая соответствует следующим требованиям:
   а. Дайте внешнее определение шаблона структуры `name` с двумя элементами: строка, предназначенная для хранения фамилии, и строка, предназначенная для хранения имени.
   б. Дайте внешнее определение шаблона структуры `student` с тремя элементами: структура `name`, массив `grade` для хранения трех оценок в виде чисел с плавающей запятой, и переменная для хранения среднего значения этих трех оценок.
   в. Напишите функцию `main()`, в которой объявляется массив структур `student` размерности `CSIZE` (в рассматриваемом случае `CSIZE = 4`) и инициализируются элементы `name` именами по вашему выбору. Используйте функции для выполнения заданий, описанных в частях г), д), е) и ж).
   г. В интерактивном режиме введите оценки для каждого студента, запрашивая у пользователя ввод ФИО студента и его оценок. Поместите оценки в массив `grade` соответствующей структуры. Требуемый цикл можно реализовать в функции `main()` или в любой другой функции по вашему усмотрению.

д. Вычислите среднюю оценку по каждой структуре и присвойте ее соответствующему элементу структуры.

е. Распечатайте информацию из каждой структуры.

ж. Распечатайте среднее значение по классу для каждого числового элемента структуры.

6. Текстовый файл содержит информацию о команде по софтболу. В каждой строке данные упорядочены следующим образом:

```
4 Джесси Джойбат 5 2 1 1
```

Первым элементом является номер игрока, обычно это число из диапазона 0–18. Второй элемент — это имя игрока, а третий — его фамилия. Каждое имя состоит из одного слова. Следующий элемент показывает, сколько раз игрок принимал мяч, за которым следует количество нанесенных игроком ударов, проходов и засчитанных пробежек. Файл может содержать результаты более чем одной игры, следовательно, для одного и того же игрока могут существовать несколько строк. Напишите программу, которая сохраняет соответствующие данные в массиве структур. Структура должна состоять из элементов, в которых представлены фамилия и имя, число набранных очков, проходов и засчитанных пробежек, а также средний результат (эти значения вычисляются позже). В качестве индекса массива можете использовать номер игрока. Программа должна выполнять чтение до появления маркера конца файла, при этом она должны накапливать итоговые результаты по каждому игроку.

Статистика бейсбола достаточно сложна. Например, проход или взятие базы в результате ошибки не расценивается так же высоко, как тот же результат, полученный за счет меткого удара, в то же время он позволяет получить засчитанную пробежку. Однако задача этой программы состоит в том, чтобы читать и обрабатывать файлы данных в соответствии с приведенным ниже описанием, независимо от того, насколько реалистичными являются данные.

Простейший способ достижения программой этой цели предусматривает инициализацию содержимого структуры нулями, считывание файла данных во временные переменные с последующим их добавлением к содержимому соответствующей структуры. После того, как программа завершит чтение файла, она должна вычислить среднее значение заработанных очков каждым игроком и запомнить их в соответствующем элементе структуры. Среднее значение заработанных очков вычисляется путем деления накопленного числа ударов, выполненных игроком, на число выходов на ударные позиции, вычисление должно выполняться с использованием арифметики плавающей запятой. Затем программа должна отобразить накапливаемые данные по каждому игроку наряду со строкой, в которой содержатся суммарные статистические данные по всей команде.

7. Внесите изменения в листинг 14.14 с тем, чтобы каждая запись считывалась из файла с последующим ее отображением, чтобы вы имели возможность удалить запись или модифицировать ее содержимое. Если вы удаляете запись, используйте освобожденную позицию массива для чтения следующей записи. Чтобы обеспечить изменение существующего содержимого, вы должны выбрать режим `"r+b"` вместо режима `"a+b"`, и уделять больше внимания установке указателя в файле, чтобы добавляемые записи не затирали существующие записи.

Проще всего внести все изменения в данные, сохраняемые в памяти, а затем записать всю полученную информацию в файл.

8. Самолетный парк авиакомпании Colossus Airlines включает один самолет с количеством сидячих мест, равным 12. Он выполняет один рейс ежедневно. Напишите программу резервирования авиабилетов, обладающую следующими возможностями:
   а. Программа использует массив из 12 структур. Каждая структура содержит номер места в самолете, специальный маркер, который показывает, зарезервировано ли данное место, фамилию и имя пассажира, для которого это место выделено.
   б. Программа отображает следующее меню:

```
Для выбора функции введите ее буквенное обозначение:
a) Показать количество свободных мест
b) Показать список пустых свободных мест
c) Показать алфавитный список занятых мест
d) Назначить пассажиру место
e) Удалить назначение места
f) Выйти из программы
```

   в. Программа выполняет действия, соответствующие пунктам меню. Позиции меню `d` и `e` требуют ввода дополнительных данных, каждая позиция меню должна обеспечить пользователю возможность отказаться от выбранного варианта.
   г. По завершении выполнения конкретной функции программа воспроизводит меню на экране, исключение составляет позиция `f`.
   д. Данные в промежутке между сеансами выполнения программы хранятся в файле. При повторном запуске программа сначала загружает данные из файла, если таковые имеются.

9. Авиакомпания Colossus Airlines (см. упражнение 8) приобретает второй самолет (с тем же количеством посадочных мест) и расширяет свою коммерческую деятельность до четырех рейсов ежедневно (рейсы с номерами 102, 311, 444 и 519). Внесите в программу соответствующие изменения, чтобы она могла обслуживать четыре рейса. Она должна предложить меню верхнего уровня, позволяющее пассажиру выбрать подходящий рейс и возможность выхода из программы. После выбора рейса на экран должно выводиться меню, аналогичное описанному в упражнении 8. Кроме того, потребуется добавить один новый элемент — подтверждение назначения места. Наряду с этим, позиция выхода из программы должна быть заменена на выход в меню верхнего уровня. Каждое отображение должно показывать, какой рейс подвергается обработке на текущий момент. Кроме того, функция отображения распределения мест в самолетах должна показывать состояние подтверждения.

10. Напишите программу, которая реализует меню с использованием массива указателей на функции. Например, выбор из меню пункта `a` должен активизировать функцию, на которую указывает первый элемент массива.

11. Напишите функцию с именем `transform()`, которая принимает четыре аргумента: имя исходного массива, содержащего исходные данные типа `double`, имя целевого массива типа `double`, значение типа `int`, представляющее количество элементов массива, и имя функции (или, что фактически одно и то же, указатель на функцию). Функция `transform()` должна применять указанную функцию к каждому элементу исходного массива и помещать возвращаемое значение в целевой массив. Например, вызов

    ```
    transform(source, target, 100, sin);
    ```

    должен присвоить элементу `target[0]` значение `sin(source[0])` и так далее для 100 элементов целевого массива. Протестируйте функцию в программе, которая вызывает функцию `transform()` четыре раза, используя в качестве аргументов две функции из библиотеки `math.h` и две подходящих функции, написанные вами.

ГЛАВА 15

Операции с разрядами

В этой главе:

- Операции: ~, &, |, ^, >>, <<, &=, |=, ^=, >>=, <<=
- Двоичная, восьмеричная и шестнадцатеричная системы счисления (обзор)
- Два средства языка С для обработки отдельных разрядов значения: поразрядные операции и разрядные поля

Язык С позволяет управлять отдельными разрядами (битами) значения переменной. Может возникнуть вопрос, для чего это нужно. Не сомневайтесь, что иногда такая возможность необходима или, по крайней мере, удобна. Примером может служить управление некоторым устройством, что часто связано с передачей нескольких битов, причем каждый из них имеет определенный смысл. Кроме того, информация о файлах в операционной системе обычно хранится в виде специальных разрядов, указывающих на определенные элементы. Многие операции сжатия и шифрования связаны с управлением отдельными разрядами. Языки высокого уровня обычно не обеспечивают такого уровня детализации. Способность совмещать возможности языка высокого уровня с операциями на уровне, обычно резервируемом для языка ассемблера, делает язык С предпочтительным выбором для написания драйверов устройств и встроенного кода.

В этой главе мы изучим возможности языка С по управлению разрядами, но сначала ознакомимся с понятиями разряда (бита) и байта, а также с различными системами счисления.

Двоичные числа, биты и байты

Основанием привычной системы записи чисел служит значение 10. Например, в числе 2157 в позиции тысяч стоит цифра 2, в позиции сотен — 1, в позиции десятков — 5, а в позиции единиц — 7. Это означает, что число 2157 можно рассматривать следующим образом:

```
2 × 1000 + 1 × 100 + 5 × 10 + 7 × 1
```

Принимая во внимание, что 1000 — это 10 в кубе, 100 — это десять в квадрате, 10 — 10 в первой степени, а 1 — это 10 (как и любое другое положительное число) в нулевой степени, число 2157 можно записать так:

$$2 \times 10^3 + 1 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$$

Поскольку привычная система записи чисел основана на степенях значения 10, принято говорить, что число 2157 записано по *основанию* 10.

Люди пользуются десятичной системой счисления потому, что у них на руках 10 пальцев. Тогда будем считать, что у бита только два пальца, поскольку для него возможны два значения — 0 и 1 (сброшенное и установленное состояние). Поэтому для компьютера естественной является двоичная система счисления. Для записи чисел он использует степени двойки, а не десятки. Записанные по основанию 2 числа называют *двоичными*. Число 2 играет такую же роль для двоичной системы, что и число 10 для десятичной системы. Например, двоичная запись 1101 означает следующее:

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

В десятичной записи это можно выразить следующим образом:

$$1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 13$$

В двоичной системе можно выразить любое целое число (при достаточном количестве разрядов) комбинацией нулей и единиц. Эта система очень удобна для цифровых вычислительных систем, у которых информация выражается в виде комбинаций включенных и выключенных состояний, что можно сопоставить с единицами и нулями. Рассмотрим применение двоичной системы для записи однобайтовых целочисленных значений.

Двоичная запись целочисленных значений

Обычно байт содержит восемь разрядов (битов). Напомним, что в языке C термин *байт* используется для обозначения размера (разрядности) хранения набора символов. Поэтому в языке C байт может содержать 8, 9, 16 и другое количество разрядов. Однако в характеристиках модулей памяти и систем передачи данных предполагается, что байт содержит восемь разрядов. Для простоты в этой главе под байтом подразумевается восьмиразрядное значение. Можно считать, что разряды байта пронумерованы справа налево числами от 0 до 7. Седьмой разряд называется *старшим*, а нулевой разряд — *младшим*. Каждый номер разряда соответствует определенной степени числа 2. Это представление байта иллюстрируется на рис. 15.1.

Рис. 15.1. *Номера и значения разрядов*

Здесь значение 128 представляет собой седьмую степень двойки и так далее. Байт имеет наибольшее значение, когда все его разряды установлены (имеют значение 1): 11111111. Значение данного двоичного числа определяется следующим образом:

```
128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
```

Наименьшему значению соответствует комбинация 00000000, которая представляет собой просто ноль. Байт может хранить числа от 0 до 255, что составляет 256 возможных значений. Программа может интерпретировать комбинацию разрядов иначе и применять байт для хранения чисел от –128 до 127, что также составляет 256 возможных значений. Например, тип `unsigned char` обычно характеризуется использованием байта для представления чисел в диапазоне от 0 до 255, а тип `signed char` — для представления чисел в диапазоне от –128 до 127.

Целочисленные значения со знаком

Представление целочисленных значений со знаком определяется оборудованием, а не языком C. Вероятно, проще всего представлять числа со знаком путем резервирования одного разряда, например старшего, для обозначения знака. В однобайтовом значении для представления самого числа останется 7 разрядов. В таком *знакоразмерном* представлении комбинация 10000001 будет соответствовать числу –1, а комбинация 00000001 — числу 1. Тогда диапазон представляемых значений будет составлять от –127 до +127.

Один из недостатков такого подхода состоит в возможности двоякого представления нуля: +0 и –0. Это сбивает с толку. Кроме того, для представления одного и того же значения используются две комбинации разрядов.

Метод *дополнения до двух* позволяет устранить этот недостаток, и сейчас он распространен наиболее широко. Рассмотрим его применительно к однобайтовому значению. Значения от 0 до 127 представляются семью последними разрядами, старший разряд сброшен (имеет значение 0). Здесь нет различий со знакоразмерным представлением. Значение 1 старшего разряда означает, что число отрицательное. Различие состоит в определении значения (модуля) отрицательного числа. Для этого нужно вычесть данную комбинацию разрядов из 9-разрядной комбинации 100000000 (двоичного представления 256). Предположим, вычисляемая комбинация равна 10000000. Применительно к типу `unsigned char` она соответствует числу 128. Применительно к типу `signed char` значение отрицательно (седьмой разряд равен 1), а его модуль равен 100000000 – 10000000, или 100000000 (128). Таким образом, число равно –128. (В знакоразмерном представлении оно бы соответствовало значению –0.) Аналогично, комбинация 10000001 соответствует значению –127, а комбинация 11111111 — значению -1. Данный метод представляет числа в диапазоне от –128 до 127.

Простейший способ смены знака двоичного числа, определяемого методом дополнения до двух, состоит в инверсии каждого разряда (нулей в единицы и наоборот) с последующим добавлением единицы. Например, значение 1 соответствует комбинации 00000001, а значение –1 соответствует выражению 11111110 + 1 или 11111111, как было показано выше.

Метод *дополнения до единицы* позволяет сделать число отрицательным за счет инверсии каждого разряда комбинации. Например, комбинация 000000001 соответствует значению 1, а комбинация 11111110 — значению –1. Этот метод также позволяет

представить величину –0: 11111111. Диапазон представляемых чисел (размером в 1 байт) составляет от –127 до +127.

Двоичное представление чисел с плавающей точкой

Числа с плавающей точкой хранятся с разбиением на две части: двоичную дробь и двоичную экспоненту. Рассмотрим, как это осуществляется.

Двоичные дроби

Десятичная дробь 0.527 представляет собой сумму

```
5/10 + 2/100 + 7/1000
```

Здесь знаменатели представляют возрастающие степени числа 10. В двоичной дроби знаменатели представляют степени двойки. Таким образом, двоичная дробь .101 может быть записана как сумма

```
1/2 + 0/4 + 1/8
```

В десятичной записи это имеет вид:

```
0.50 + 0.00 + 0.125
```

или 625.

Многие дроби, такие как 1/3, не могут быть точно представлены в десятичной записи. Подобно этому, многие дроби невозможно точно представить и в двоичной записи. На самом деле, точно могут быть представлены лишь комбинации составляющих, которые кратны числу 1/2 в некоторой целой степени. Например, дроби 3/4 и 7/8 можно точно записать в двоичном представлении, тогда как для дробей 1/3 и 2/5 это невозможно.

Представление чисел с плавающей точкой

Представление числа с плавающей точкой предусматривает выделение некоторого количества (в зависимости от применяемой вычислительной системы) разрядов для хранения двоичной дроби. Остальные разряды представляют экспоненту. Числовое значение определяется как произведение двоичной дроби на 2 в степени, выражаемой экспонентой. Например, чтобы умножить число с плавающей точкой на 4, нужно удвоить экспоненту и оставить двоичную дробь без изменений. Умножение на число, не являющееся степенью двух, влечет изменение двоичной дроби и, если необходимо, экспоненты.

Другие основания систем счисления

Специалисты по вычислительной технике часто используют системы счисления по основаниям 8 и 16. Поскольку числа 8 и 16 являются степенями 2, эти системы счисления более тесно связаны с двоичной системой компьютера по сравнению с десятичной системой.

Восьмеричная система счисления

В основании этой системы лежит число 8. Каждое знакоместо восьмеричного числа соответствует определенной степени восьми. Для записи используются цифры от 0 до 7. Например, восьмеричное число 451 (в языке C записывается как 0451) представляет следующее десятичное значение:

$4 \times 8^2 + 5 \times 8^1 + 1 \times 8^0 = 297$ (по основанию 10)

Каждая цифра восьмеричного числа соответствует трем двоичным цифрам, как показано в табл. 15.1. Это упрощает перевод чисел из одной системы в другую. Например, восьмеричное число 0377 соответствует двоичному числу 11111111. Здесь цифра 3 заменяется комбинацией 011, в которой лидирующий ноль опускается, а каждая цифра 7 заменяется комбинацией 111. Единственное неудобство состоит в том, что трехзначное восьмеричное число в двоичной форме может занимать до 9 разрядов. Поэтому восьмеричное значение, превышающее 0377, требует более одного байта памяти. Обратите внимание, что внутренние нули не опускаются: числу 0173 соответствует комбинация 01 111 011, а не 01 111 11.

Таблица 15.1. Двоичные эквиваленты восьмеричных цифр

Восьмеричная цифра	*Двоичный эквивалент*
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Шестнадцатеричная система счисления

В этой системе используются степени числа 16 и цифры от 0 до 15. Для представления цифр, соответствующих десятичным значениям от 10 до 15, используются буквы от A до F. Например, шестнадцатеричное число A3F (в языке C записывается как 0xA3F) представляет значение

$10 \times 16^2 + 3 \times 16^1 + 15 \times 16^0 = 2623$ (по основанию 10)

поскольку цифра A представляет значение 10, а F — значение 15. Язык C допускает использование букв нижнего и верхнего регистра для обозначения шестнадцатеричных цифр. Таким образом, число 2623 можно записать также в виде 0xa3f.

Каждая цифра шестнадцатеричного числа соответствует 4-значному двоичному числу. Поэтому две шестнадцатеричных цифры соответствуют одному восьмиразрядному байту. Первая цифра представляет 4 старших разряда, а вторая цифра — 4 младших разряда. Это делает шестнадцатеричное представление очень удобным для записи значений байтов. Соответствие между шестнадцатеричными цифрами и двоичны-

ми числами показано в табл. 15.2. Например, шестнадцатеричное число 0xC2 соответствует комбинации разрядов 11000010. Выполняя обратное преобразование, комбинацию 11010101 представим в виде 1101 0101, что может быть записано как 0xD5.

Таблица 15.2. Десятичные, шестнадцатеричные числа и их двоичные эквиваленты

Десятичное число	*Шестнадцатеричная цифра*	*Двоичный эквивалент*
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Теперь, когда мы ознакомились с понятием разрядов и байтов, рассмотрим, какие операции с ними возможны в языке C. Существует два средства управления разрядами. Первое представляет собой набор поразрядных операций, а второе — форму *полей* данных, которая предоставляет доступ к разрядам значения типа `int`. Эти средства рассматриваются в последующих разделах.

Поразрядные операции

В языке C существует два вида поразрядных операций: логические операции и операции сдвига. В последующих примерах запись значений будет выполняться в двоичной системе, чтобы результаты операций с разрядами были нагляднее. В реальной программе применяются целочисленные переменные или константы в обычной форме записи. Например, вместо записи `00011001` будет применяться запись `25`, `031` или `0x19`. В наших примерах будут использоваться 8-разрядные числа. Разряды нумеруются слева направо от 0 до 7.

Поразрядные логические операции

Четыре логических поразрядных операции работают с целочисленными данными, включая тип char. Название *поразрядные* связано с тем, что операции выполняются над каждым разрядом независимо от разрядов слева или справа. Не путайте их с обычными логическими операциями (&&, || и !), которые управляют значениями в целом.

Дополнение до единицы или поразрядное отрицание: ~

Унарная операция ~ преобразовывает все единицы в нули и все нули в единицы, как показано в следующем примере:

```
~(10011010)    // выражение

(01100101)     // результат
```

Предположим, переменной val типа unsigned char присвоено значение 2. В двоичном выражении оно имеет вид 00000010. Тогда выражению ~val будет соответствовать значение 11111101 или 253. Обратите внимание, что операция не изменяет значения переменной val подобно тому, как не изменяет значение val выражение 3 * val. Оно по-прежнему равно 2. Вместо этого создается новое значение, которое может быть использовано в другом выражении либо присвоено другой переменной:

```
newval = ~val;

printf("%d", ~val);
```

Чтобы значение val изменить на ~val, можно воспользоваться простым оператором присваивания:

```
val = ~val;
```

Поразрядная операция "И": &

Двоичная операция & создает новое значение за счет выполнения поразрядного сравнения двух операндов. Для каждой позиции результирующий разряд будет иметь значение 1 только в случае, когда соответствующие разряды обоих операндов имеют значение 1. (Применительно к логическим значениям можно сказать, что результат будет истинным только в случае, когда оба двоичных операнда имеют истинные значения.) Таким образом, в результате вычисления выражения

```
(10010011) & (00111101)    // выражение
```

получается следующее значение:

```
(00010001)                 // результат
```

Такой результат обусловлен тем, что только нулевой и четвертый разряды обоих операндов имеют значение 1.

В языке C также существует комбинированная операция "И"-присваивание: &=. Оператор

```
val &= 0377;
```

дает тот же результат, что и следующее выражение:

```
val = val & 0377;
```

Поразрядная операция "ИЛИ": |

Двоичная операция | создает новое значение за счет выполнения поразрядного сравнения двух операндов. Для каждой позиции результирующий разряд будет иметь значение 1, если любой из соответствующих разрядов обоих операндов равен 1. (Применительно к логическим значениям можно сказать, что результат будет истинным в случае, когда хотя бы один двоичный операнд имеет истинное значение.) Таким образом, в результате вычисления выражения

```
(10010011) | (00111101)   // выражение
```

получается следующее значение:

```
(10111111)              // результат
```

Это связано с тем, что все разряды, кроме шестого, хотя бы в одном из операндов имеют значение 1.

В языке C также существует комбинированная операция "ИЛИ"-присваивание: |=. Оператор

```
val |= 0377;
```

дает тот же результат, что и следующее выражение:

```
val = val | 0377;
```

Поразрядное "исключающее ИЛИ": ^

Двоичная операция ^ выполняет поразрядное сравнение двух операндов. Для каждой позиции результирующий разряд будет иметь значение 1, если один из соответствующих разрядов какого-либо операнда (но не обеих сразу) равен 1. (Применительно к логическим значениям можно сказать, что результат будет истинным в случае, когда хотя бы один двоичный операнд имеет истинное значение.) Таким образом, в результате вычисления выражения

```
(10010011) ^ (00111101)   // выражение
```

получается следующее значение:

```
(10101110)              // результат
```

Обратите внимание, что поскольку у обоих операндов нулевой разряд имеет значение 1, результирующий нулевой разряд получает значение 0.

В языке C также существует комбинированная операция "исключающее ИЛИ"-присваивание: ^=. Оператор

```
val ^= 0377;
```

дает тот же результат, что и следующее выражение:

```
val = val ^ 0377;
```

Область применения: маски

Поразрядная операция "И" часто используется с маской. *Маска* представляет собой некоторую комбинацию разрядов. Чтобы понять смысл названия "маска" рассмотрим, что произойдет при попытке применения маски (операции &) к некоторой величине.

Предположим, определена символическая константа MASK со значением 2 (что соответствует двоичному коду 00000010), у которой только первый разряд имеет ненулевое значение. Тогда оператор

```
flags = flags & MASK;
```

установит для всех разрядов переменной flags, кроме первого, значение 0, поскольку применение операции "И" к любой паре разрядов, из которых один нулевой, дает значение 0. Первый разряд переменной flags остается без изменений. (Если бы в маске соответствующий разряд был равен 0, результат был бы также нулевым.) Этот процесс называется "использованием маски", поскольку нули переменной MASK скрывают соответствующие разряды переменной flags.

Разряды маски с нулевыми значениями можно считать аналогом непрозрачных ячеек реальной маски, а разряды со значениями 1 — прозрачными ячейками. Выражение flags & MASK можно сравнить с наложением маски на комбинацию разрядов переменной flags. Видимы только те разряды, которые находятся под включенными (с 1) разрядами переменной MASK (рис. 15.2).

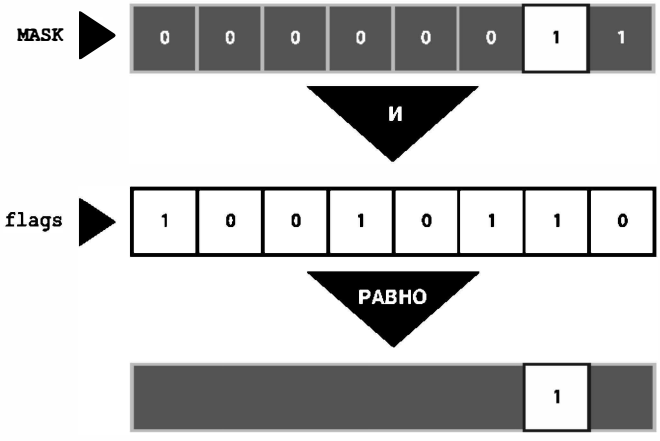


Рис. 15.2. *Наглядное представление маски*

Для сокращения кода можно воспользоваться операцией "И"-присваивание:

```
flags &= MASK;
```

Ниже показан один из распространенных вариантов применения этой операции:

```
ch &= 0xff;      /* или ch &= 0377; /*
```

Значение 0xff соответствует двоичному числу 11111111, или восьмеричному числу 0377. Эта маска оставляет восемь заключительных разрядов переменной ch без изменений, а остальные отключает (обнуляет). Независимо от разрядности исходной переменной ch (8, 16 или более разрядов), результирующее значение усекается до величины, которая соответствует одному обычному байту.

Область применения: включение разрядов

Иногда требуется включить отдельные разряды значения, оставив остальные без изменений. Например, компьютер управляет устройствами, отправляя в порты определенные значения. Для включения, скажем, динамика, необходимо включить первый разряд, а остальные оставить без изменений. Для этого можно воспользоваться поразрядной операцией "ИЛИ".

Предположим, у переменной MASK включен только первый разряд. Тогда оператор

```
flags = flags | MASK;
```

включает первый разряд переменной flags и оставляет все остальные разряды без изменений. Дело в том, что комбинирование посредством операции | любого разряда с нулевым значением оставляет разряд без изменений, а комбинирование со значением 1 включает этот разряд. Для сокращенной записи можно применять поразрядную операцию "ИЛИ"-присваивание:

```
flags |= MASK;
```

Разряды переменной falgs, которым соответствуют включенные разряды переменной MASK, будут включены, а остальные останутся без изменений.

Область применения: отключение разрядов

Иногда возникает необходимость не включать, а отключать определенные разряды, оставляя остальные без изменений. Предположим, требуется отключить первый разряд переменной flags. И снова у переменной MASK включен только первый разряд. Можно воспользоваться следующим выражением:

```
flags = flags & ~MASK;
```

Поскольку у переменной MASK все разряды, кроме первого, отключены, выражение ~MASK дает значение, в котором все разряды, кроме первого, включены. Комбинирование любого разряда с включенным разрядом с помощью операции & оставляет первый разряд без изменений. Поэтому все разряды переменной flags, кроме первого, остаются без изменений. Комбинирование любого разряда с отключенным разрядом с использованием операции & отключает первый разряд независимо от его исходного значения.

Ниже представлен сокращенный вариант приведенного выше выражения:

```
flags &= ~MASK;
```

Область применения: переключение разрядов

Переключение разряда означает его отключение, если он включен, и включение, если отключен. Для переключения разрядов можно воспользоваться поразрядной операцией исключающего "ИЛИ". Если обозначить значение разряда буквой b, то выражение 1 ^ b будет равно 0, когда b = 1, и равно 1, когда b = 0. Кроме того, выражение 0 ^ b равно b независимо от значения b. Следовательно, в результате комбинирования некоторого значения с маской с помощью операции ^ разряды, соответствующие единицам маски, переключатся, а разряды, соответствующие нулям маски, останутся без изменений. Чтобы переключить первый разряд переменной flag, можно выполнить одно из следующих действий:

```
flag = flag ^ MASK;
flag ^= MASK;
```

Область применения: проверка значения разряда

Мы рассмотрели методы изменения значений разрядов. Предположим, вместо этого нужно проверить значение разряда. Например, требуется узнать, включен ли первый разряд переменной flag. Простое сравнение переменных flag и MASK здесь не подходит:

```
if (flag == MASK)
  puts("В яблочко!);          /* это не даст нужного результата */
```

Даже если первый разряд переменной flag включен, значение какого-то другого разряда может сделать результат сравнения ложным. Поэтому, чтобы выполнить сравнение только для первого разряда, необходимо применить маску ко всем остальным разрядам переменной flag:

```
if ((flag & MASK) = = MASK)
   puts ("В яблочко!");
```

Поразрядные операции имеют более низкий приоритет по сравнению с операцией ==, поэтому выражение flag & MASK должно быть заключено в скобки.

Чтобы данные охватывались полностью, разрядная маска должна иметь размер не меньший, чем у маскируемого значения.

Поразрядные операции сдвига

Теперь рассмотрим операции сдвига, применяемые в языке C. Эти операции сдвигают разряды влево или вправо. Для большей наглядности здесь мы также будем применять двоичную запись чисел.

Сдвиг влево: <<

Операция сдвига влево (<<) сдвигает разряды левого операнда влево на количество позиций, указываемое правым операндом. Освобождаемые позиции заполняются нулями. Разряды, выходящие за пределы значения левого операнда, теряются. В следующем примере каждый разряд сдвигается на две позиции влево:

```
(10001010) << 2      // выражение
(00101000)           // результат
```

В результате операции появилось новое значение, а сами операнды не изменились. Предположим, что переменная stonk имеет значение 1. Выражение stonk<<2 дает значение 4, но переменная stonk по-прежнему равна 1. Чтобы изменить значение переменной, можно воспользоваться операцией сдвига влево с присваиванием (<<=). Эта операция сдвигает разряды переменной влево на количество позиций, указываемое правым операндом. Вот пример:

```
int stonk = 1;
int onkoo;
onkoo = stonk << 2;  /* переменной onkoo присваивается значение 4 */
stonk <<= 2;         /* значение stonk увеличивается до 4 */
```

Сдвиг вправо: >>

Операция сдвига вправо (>>) сдвигает разряды левого операнда вправо на количество позиций, указываемое правым операндом. Выходящие за правую границу разряды теряются. Для типов данных без знака (unsigned) освобождаемые слева позиции заполняются нулями. Для знаковых типов данных результат зависит от используемой системы. Освобождаемые позиции могут заполняться нулями либо копиями знакового (первого слева) разряда:

```
(10001010) >> 2 // выражение с участием значения со знаком
(00100010)      // результат для некоторых систем
(10001010) >> 2 // выражение с участием значения со знаком
(11100010)      // результат для других систем
```

Для значения без знака результат будет следующим:

```
(10001010) >> 2 // выражение с участием значения без знака
(00100010)      // результат для всех систем
```

Каждый разряд перемещается на две позиции вправо, а освобождаемые позиции заполняются нулями.

Операция сдвига вправо с присваиванием (>>=) сдвигает вправо разряды левого операнда на указанное количество позиций, как показано в следующем примере:

```
int sweet = 16;
int ooosw;
ooosw = sweet >> 3; /* ooosw = 2, sweet по-прежнему 16 */
sweet >>=3;         /* переменная sweet стала равной 2 */
```

Область применения: поразрядные операции сдвига

Поразрядные операции сдвига могут служить удобным и эффективным (в зависимости от оборудования) средством выполнения операций умножения и деления на числа, представляющие собой степени двойки:

`number << n` Умножает `number` на 2 в степени `n`.

`number >> n` Делит `number` на 2 в степени `n`, если значение `number` неотрицательно.

Эти операции сдвига аналогичны смещению десятичной точки при умножении или делении на 10.

Операции сдвига также могут применяться для извлечения групп разрядов из крупных элементов данных. Предположим, что для представления цвета используется значение типа `unsigned long`. При этом младший байт содержит интенсивность красной составляющей, следующий байт — интенсивность зеленой составляющей, а третий байт — интенсивность синей составляющей цвета. Необходимо сохранить интенсивность каждой составляющей в собственной переменной типа `unsigned char`. Для этого можно написать примерно такой код:

```
#define BYTE_MASK 0xff
unsigned long color = 0x002a162f;
unsigned char blue, green, red;
red = color & BYTE_MASK;
green = (color >> 8) & BYTE_MASK;
blue = (color >> 16) & BYTE_MASK;
```

Здесь операция сдвига вправо используется для перемещения 8-разрядного значения цвета в младший байт. Затем с помощью маски значение младшего байта присваивается некоторой переменной.

Пример программы

В главе 9 для написания программы преобразования чисел в двоичное представление применялась рекурсия. Теперь мы решим ту же задачу с помощью поразрядных операций. Программа, показанная в листинге 15.1, выполняет чтение вводимого с клавиатуры целочисленного значения, затем передает его и адрес строки функции с именем `itobs()` (конечно, для строки двоичного представления целого числа). Эта функция использует поразрядные операции для определения правильной комбинации нулей и единиц, помещаемых в строку.

Листинг 15.1. Программа `binbit.c`

```
/* binbit.c -- использование поразрядных операций для отображения двоичных
чисел */
#include <stdio.h>
char * itobs(int, char *);
void show_bstr(const char *);

int main(void)
{
  char bin_str[8 * sizeof(int) + 1];
  int number;

  puts("Введите целые числа и просмотрите их двоичные представления.");
  puts("Нечисловой ввод завершает программу.");
  while (scanf("%d", &number) == 1)
  {
    itobs(number,bin_str);
    printf("%d представляется как ", number);
    show_bstr(bin_str);
    putchar('\n');
  }
  puts("Программа завершена.");

  return 0;
}

char * itobs(int n, char * ps)
{
  int i;
  static int size = 8 * sizeof(int);

  for (i = size - 1; i >= 0; i--, n >>= 1)
     ps[i] = (01 & n) + '0';
  ps[size] = '\0';

  return ps;
}

/* отображение двоичной строки блоками по 4 элемента */
void show_bstr(const char * str)
```

```
{
  int i = 0;
  while (str[i]) /* для символов, отличных от нулевого */
  {
     putchar(str[i]);
     if(++i % 4 == 0 && str[i])
        putchar(' ');
  }
}
```

Программа в листинге 15.1 написана в предположении, что в данной системе байт содержит восемь разрядов. Поэтому выражение `8 * sizeof(int)` определяет количество разрядов в значении типа `int`. Массив `bin_str` содержит на один элемент больше этой величины, что позволяет включить в него завершающий нулевой символ.

Функция `itobs()` возвращает передаваемый ей адрес, что позволяет ее использовать, например, в качестве аргумента функции `printf()`. На первой итерации цикла `for` функция вычисляет выражение `01 & n`. Операнд `01` представляет собой восьмеричное представление маски, у которой все разряды, кроме нулевого, отключены. Поэтому выражение `01 & n` равно значению последнего разряда числа `n`. Оно может быть нулем или единицей, но в массив нужно записать *символ* `'0'` или *символ* `'1'`. Преобразование завершается добавлением ASCII-кода `'0'`. Результат помещается в позицию предпоследнего элемента массива. (Последний элемент зарезервирован для нулевого символа.)

Кстати, вместо выражения `01 & n` можно указать и `1 & n`. Использование восьмеричной единицы вместо десятичной более стильно для операций с разрядами.

Затем в цикле выполняются операции `i--` и `n >>= 1`. Первая операция осуществляет переход к предыдущему элементу массива, а вторая сдвигает разряды числа `n` на одну позицию вправо. На следующей итерации цикла определяется значение текущего первого справа разряда. Соответствующий цифровой символ присваивается элементу массива, который предшествует последней цифре. Таким образом, функция заполняет массив слева направо.

Для отображения строки результата можно воспользоваться функцией `printf()` или `puts()`. Однако в листинге 15.1 определена функция `show_bstr()`, которая разбивает данные на группы по четыре разряда, чтобы строка легче читалась.

Ниже показан пример выполнения программы:

```
Введите целые числа и просмотрите их двоичные представления.
Нечисловой ввод завершает программу.
7
7 представляется как 0000 0000 0000 0000 0000 0000 0000 0111
2006
2006 представляется как 0000 0000 0000 0000 0000 0111 1101 0110
-1
-1 представляется как 1111 1111 1111 1111 1111 1111 1111 1111
32123
32123 представляется как 0000 0000 0000 0000 0111 1101 0111 1011
q
Программа завершена.
```

Еще один пример программы

Рассмотрим еще один пример. На этот раз ставится задача написать функцию, которая инвертирует n последних разрядов значения. При этом число n и изменяемое значение являются аргументами функции.

Операция ~ инвертирует все разряды, а не какую-либо выбранную часть. Однако операция ^ (исключающее "ИЛИ"), как уже говорилось, может применяться для переключения отдельных разрядов. Предположим, создается маска, где n последних разрядов включены, а остальные отключены. Тогда применение операции ^ к этой маске и значению переключает, или *инвертирует*, n последних разрядов, оставляя остальные разряды без изменений. Этот подход используется в следующем фрагменте кода:

```
int invert_end(int num, int bits)
{
   int mask = 0;
   int bitval = 1;

   while (bits-- > 0)
   {
      mask |= bitval;
      bitval <<= 1;
   }
   return num ^ mask;
}
```

Маска создается в цикле while. Сначала все разряды маски отключаются. Во время первой итерации цикла включается нулевой разряд, а затем увеличивается до 2 значение переменной bitval. Другими словами, нулевой разряд отключается, а первый разряд включается. Во время следующей итерации включается первый разряд переменной mask и так далее. Наконец, операция num ^ mask дает требуемый результат.

Для проверки работы функции можно ввести ее в предыдущую программу, как показано в листинге 15.2.

Листинг 15.2. Программа `invert4.c`

```
/* invert4.c -- использование поразрядных операций для отображения
двоичного представления чисел */
#include <stdio.h>
char * itobs(int, char *);
void show_bstr(const char *);
int invert_end(int num, int bits);

int main(void)
{
  char bin_str[8 * sizeof(int) + 1];
  int number;

  puts("Введите целые числа и просмотрите их двоичные представления.");
  puts("Нечисловой ввод завершает программу.");
  while (scanf("%d", &number) == 1)
  {
    itobs(number,bin_str);
    printf("%d представляется как\n", number);
```

```
    show_bstr(bin_str);
    putchar('\n');
    number = invert_end(number, 4);
    printf("Результат инвертирования последних 4 разрядов:\n");
    show_bstr(itobs(number,bin_str));
    putchar('\n');
  }
  puts("Программа завершена.");
  return 0;
}
char * itobs(int n, char * ps)
{
  int i;
  static int size = 8 * sizeof(int);
  for (i = size - 1; i >= 0; i--, n >>= 1)
    ps[i] = (01 & n) + '0';
  ps[size] = '\0';
  return ps;
}
/* отображение двоичных строк блоками по 4 элемента */
void show_bstr(const char * str)
{
  int i = 0;
  while (str[i]) /* для символов, отличных от нулевого */
  {
    putchar(str[i]);
    if(++i % 4 == 0 && str[i])
       putchar(' ');
  }
}
int invert_end(int num, int bits)
{
  int mask = 0;
  int bitval = 1;
  while (bits-- > 0)
  {
    mask |= bitval;
    bitval <<= 1;
  }
  return num ^ mask;
}
```

Ниже представлен пример выполнения программы:

```
Введите целые числа и просмотрите их двоичные представления.
Нечисловой ввод завершает программу.
7
7 представляется как
0000 0000 0000 0000 0000 0000 0000 0111
```

```
Результат инвертирования последних 4 разрядов:
0000 0000 0000 0000 0000 0000 0000 1000
12541
12541 представляется как
0000 0000 0000 0000 0011 0000 1111 1101
Результат инвертирования последних 4 разрядов:
0000 0000 0000 0000 0011 0000 1111 0010
q
Программа завершена.
```

Разрядные поля

Второй метод управления разрядами состоит в использовании *разрядного поля*, которое представляет собой просто последовательную цепочку разрядов в рамках значения типа `signed int` или `unsigned int`. (Стандарт C99 также допускает применение разрядных полей типа `_Bool`.) Разрядное поле создается путем объявления структуры, которая помечает каждое поле и определяет его размер. Например, следующее объявление устанавливает четыре 1-разрядных поля:

```
struct {
   unsigned int autfd : 1;
   unsigned int bldfc : 1;
   unsigned int undln : 1;
   unsigned int itals : 1;
} prnt;
```

Это определение создает структуру `prnt`, которая содержит четыре 1-разрядных поля. Теперь для присвоения значений отдельным полям можно воспользоваться обычной операцией принадлежности к структуре:

```
prnt.itals = 0;
prnt.undln = 1;
```

Поскольку каждое поле представляет собой единственный разряд, в операциях присвоения можно использовать только значения `1` и `0`. Переменная `prnt` хранится в ячейке памяти, размерность которой соответствует типу `int`. Однако в нашем примере используются лишь четыре разряда.

Структуры с разрядными полями служат удобным средством отслеживания настроек. Многие настройки, такие как полужирное или курсивное начертание шрифта, могут определяться указанием одной опции из двух — включено или отключено (истина и ложь). Нет необходимости использовать переменную целиком там, где достаточно одного разряда. Структура с разрядными полями позволяет хранить несколько параметров в одном элементе.

Иногда настройка предусматривает более двух опций, поэтому для представления всех вариантов одного разряда недостаточно. Это не вызывает затруднений, поскольку размеры полей не ограничены одним разрядом. Структуру можно определить и таким образом:

```
struct {
   unsigned int code1 : 2;
   unsigned int code2 : 2;
   unsigned int code3 : 8;
} prcode;
```

В этом фрагменте создаются два 2-разрядных поля и одно 8-разрядное. Теперь возможны следующие операции присваивания:

```
prcode.code1 = 0;
prcode.code2 = 3;
prcode.code3 = 102;
```

Нужно просто следить, чтобы значение не превышало размерность поля.

А что, если общее количество объявленный разрядов превысит размер для типа `unsigned int`? Тогда будет использоваться следующая область хранения `unsigned int`. Отдельное поле не должно накладываться на границу между двумя последовательными областями `unsigned int`. Компилятор автоматически сдвигает перекрывающее границу определение поля таким образом, чтобы поле умещалось в пределы области `unsigned int`. Когда это происходит, в первой области `unsigned int` остается неименованный промежуток.

Структуру полей можно заполнить неименованными промежутками путем указания размеров неименованных полей. Если указано неименованное поле размером в 0 разрядов, следующее поле будет связано с последующей областью хранения целочисленных значений:

```
struct {
   unsigned int field1 : 1;
   unsigned int        : 2;
   unsigned int field2 : 1;
   unsigned int        : 0;
   unsigned int field3 : 1;
} stuff;
```

Здесь между полями `stuff.field1` и `stuff.field2` указан 2-разрядный промежуток, а данные поля `stuff.field3` будут храниться в следующей ячейке типа `int`.

Одна из важных характеристик системы определяет порядок, в котором поля помещаются в область хранения типа `int`. В некоторых системах поля помещаются слева направо, в других — справа налево. Кроме того, системы различаются местонахождением границ между полями. По ряду причин разрядные поля не обладают высокой степенью переносимости (то есть совместимостью с другими платформами). Обычно они используются для целей, не требующих переносимости, таких как помещение данных в форму, используемую строго определенным устройством.

Пример использования разрядных полей

Разрядные поля часто служат средством компактного хранения данных. Предположим, нужно представить свойства выводимого на экран окна. Не будем применять сложную графику и примем для окна следующий набор свойств:

- Окно может быть прозрачным или непрозрачным.
- Цвет фона выбирается из следующей палитры: черный, красный, зеленый, желтый, синий, пурпурный, голубой и белый.
- Рамка может быть скрыта или отображена.
- Цвет рамки выбирается из той же палитры, что и цвет фона.
- Для рамки применяются три стиля линии: сплошная, пунктирная и штриховая.

Для каждого свойства можно было бы использовать отдельную переменную или полноразмерный элемент структуры, однако это связано с неэкономным расходом разрядов. Например, для указания прозрачности или непрозрачности окна достаточно одного разряда. То же касается свойства отображения или сокрытия рамки. Восемь возможных значений цвета могут быть представлены 3-разрядным элементом, а 2-х разрядного элемента более чем достаточно для представления трех стилей линии рамки. Таким образом, десяти разрядов достаточно для представления всех пяти свойств окна.

Один из вариантов представления информации предусматривает использование заполнителей, чтобы поместить связанную с фоном окна информацию в один байт, а связанную с рамкой — во второй. Это реализовано в следующем объявлении структуры box_props:

```
struct box_props {
   unsigned int opaque       : 1;
   unsigned int fill_color   : 3;
   unsigned int              : 4;
   unsigned int show_border  : 1;
   unsigned int border_color : 3;
   unsigned int border_style : 2;
   unsigned int              : 2;
};
```

В результате применения заполнителей размер структуры увеличивается до 16 разрядов. В противном случае было бы достаточно 10 разрядов. Однако следует учитывать, что в языке C для структур с разрядными полями в качестве базового элемента размещения данных используется тип unsigned int. Поэтому, даже если структура содержит единственный элемент, который представляет 1-разрядное поле, размер структуры будет соответствовать типу unsigned int, что для рассматриваемой системы составляет 32 разряда.

Для элемента opaque можно использовать значение 1, обозначающее непрозрачность окна, и значение 0, определяющее свойство прозрачности. То же применимо к элементу show_border. Для цветов можно применить простое RGB-представление (красный–зеленый–синий). Это основные цвета спектра. В мониторе применяется смешанное свечение красных, зеленых и синих пикселей для воспроизведения различных цветов. В первых моделях мониторов каждый пиксель мог иметь только включенное или выключенное состояние. Поэтому для представления интенсивности каждой из трех составляющих спектра было достаточно одного разряда. Обычно левый разряд представлял интенсивность синего, средний — интенсивность зеленого, а правый — красного цвета. В табл. 15.3 показаны восемь возможных комбинаций. Они могут служить значениями элементов fill_color и border_color. Наконец, значения 0, 1 и 2 могут представлять сплошной, пунктирный и штриховой тип линий, определяемый элементом border_style.

В листинге 15.3 в качестве простого примера используется структура box_props. Директивы #define служат для создания символических констант, представляющих возможные значения элементов. Обратите внимание, что основные цвета представлены включением единственного разряда. Остальные цвета могут представляться комбинацией основных цветов. Например, пурпурный цвет создается включением разрядов синего и красного цвета, поэтому его можно представить комбинацией BLUE | RED.

Таблица 15.3. Простое представление цветов

Комбинация разрядов	*Десятичный эквивалент*	*Цвет*
000	0	Черный
001	1	Красный
010	2	Зеленый
011	3	Желтый
100	4	Синий
101	5	Пурпурный
110	6	Голубой
111	7	Белый

Листинг 15.3. Программа `fields.c`

```
/* fields.c -- определение и использование полей */
#include <stdio.h>
/* свойство непрозрачности и отображения */
#define YES 1
#define NO 0
/* стили линии */
#define SOLID 0
#define DOTTED 1
#define DASHED 2
/* основные цвета */
#define BLUE 4
#define GREEN 2
#define RED 1
/* смешанные цвета */
#define BLACK 0
#define YELLOW (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN (GREEN | BLUE)
#define WHITE (RED | GREEN | BLUE)
const char * colors[8] = {"черный", "красный", "зеленый", "желтый",
                          "синий", "пурпурный", "голубой", "белый"};
struct box_props {
  unsigned int opaque       : 1;
  unsigned int fill_color   : 3;
  unsigned int              : 4;
  unsigned int show_border  : 1;
  unsigned int border_color : 3;
  unsigned int border_style : 2;
  unsigned int              : 2;
};
void show_settings(const struct box_props * pb);
int main(void)
```

```
{
  /* создание и инициализация структуры box_props */
  struct box_props box = {YES, YELLOW , YES, GREEN, DASHED};

  printf("Исходные параметры окна:\n");
  show_settings(&box);

  box.opaque = NO;
  box.fill_color = WHITE;
  box.border_color = MAGENTA;
  box.border_style = SOLID;
  printf("\nИзмененные параметры окна:\n");
  show_settings(&box);

  return 0;
}
void show_settings(const struct box_props * pb)
{
  printf("Фон %s.\n",
           pb->opaque == YES? "непрозрачный": "прозрачный");
  printf("Цвет фона %s.\n", colors[pb->fill_color]);
  printf("Рамка %s.\n",
           pb->show_border == YES? "видимая" : "невидимая");
  printf("Цвет рамки %s.\n", colors[pb->border_color]);
  printf ("Стиль линии рамки ");
  switch(pb->border_style)
  {
    case SOLID : printf("сплошной.\n"); break;
    case DOTTED : printf("пунктирный.\n"); break;
    case DASHED : printf(штриховой.\n"); break;
    default : printf("неизвестный.\n");
  }
}
```

Выходные данные программы имеют следующий вид:

```
Исходные параметры окна:
Фон непрозрачный.
Цвет фона желтый.
Рамка видимая.
Цвет рамки зеленый.
Стиль линии рамки штриховой.

Измененные параметры окна:
Фон прозрачный.
Цвет фона белый.
Рамка видимая.
Цвет рамки пурпурный.
Стиль линии рамки сплошной.
```

Отметим несколько моментов. Во-первых, можно инициализировать структуру разрядных полей с использованием обычного для структур синтаксиса:

```
struct box_props box = {YES, YELLOW , YES, GREEN, DASHED};
```

Аналогично присваиваются значения элементам разрядных полей:

```
box.fill_color = WHITE;
```

Кроме того, элемент разрядного поля может служить выражением для операции switch. Он даже может играть роль индекса массива:

```
printf("Цвет фона %s.\n", colors[pb->fill_color]);
```

Обратите внимание, что массив colors может быть определен таким образом, чтобы каждое значение индекса соответствовало строковому представлению названия цвета. Для этого значение индекса должно выражать числовое значение цвета. Например, индекс 1 соответствует строке "красный", а константа RED перечисления имеет значение 1.

Разрядные поля и поразрядные операции

Разрядные поля и поразрядные операции служат двумя альтернативными подходами к решению однотипных задач программирования. Другими словами, часто можно применять любой из подходов. В предыдущем примере для хранения информации об окне использовалась структура, разрядность которой соответствует типу unsigned int. Те же данные можно было сохранять и в переменной типа unsigned int. Затем для доступа к отдельным данным вместо синтаксиса принадлежности к структуре можно было применить поразрядные операции. Обычно такая методика не очень удобна. Рассмотрим пример, в котором используются обе методики. (Здесь преследуется цель показать различия методик, а не внушить мысль о целесообразности одновременного применения обеих подходов!)

В качестве средства комбинирования методик, основанных на структурах и на поразрядных операциях, можно воспользоваться объединением. Исходя из существующего объявления типа данных struct box_props, можно объявить следующее объединение:

```
union Views /* просмотр данных в виде структуры или типа unsigned short */
{
  struct box_props st_view;
  unsigned int ui_view;
};
```

В некоторых системах переменная типа unsigned int и структура типа box_props должны занимать 16 разрядов памяти. В других системах, таких как используемая в нашем примере, эти данные занимают 32 разряда. В любом случае, для просмотра содержимого памяти в виде структуры можно воспользоваться элементом st_view, а для просмотра того же блока памяти в виде данных типа unsigned int — элементом ui_view. Какие разрядные поля структуры соответствуют отдельным разрядам переменной типа unsigned int? Это зависит от реализации языка и аппаратных средств. Для системы на базе IBM PC и версии Microsoft Visual C/C++ 7.1 структуры загружаются в память в следующем порядке: сначала младшие, а затем старшие разряды байта. Другими словами, первое разрядное поле структуры соответствует нулевому разряду поля. (В целях простоты на рис. 15.3 этот принцип иллюстрируется для 16-разрядного элемента.)

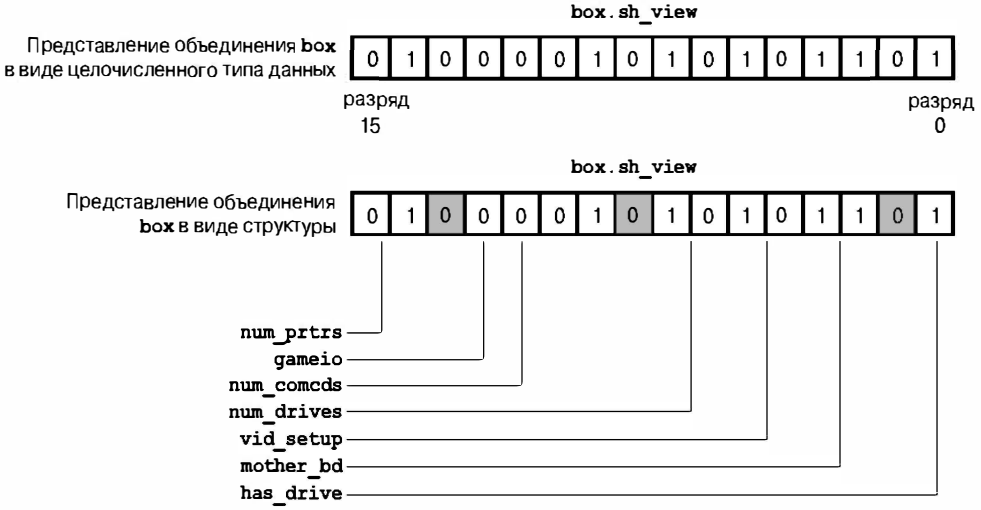


Рис. 15.3. *Представление объединения в виде целочисленного типа и в виде структуры*

В листинге 15.4 объединение `Views` применяется для сравнения методик, основанных на разрядных полях и поразрядных операциях. Здесь `box` — это объединение типа `Views`, поэтому `box.st_view` — это структура типа `box_props`, использующая разрядные поля, а `box.ui_view` — те же данные, представленные в переменной типа `unsigned int`. Напомним, что у объединения может быть инициализирован первый элемент, поэтому присвоенные значения соответствуют представлению структуры. Программа отображает свойства окна с помощью функции, реализующей просмотр содержимого структуры, а также с помощью функции, реализующей просмотр содержимого переменной типа `unsigned int`. Каждый подход обеспечивает доступ к данным, но при этом используются различные приемы. Кроме того, программа вызывает определенную ранее в этой главе функцию `itobs()` для отображения данных в виде строки двоичных цифр, чтобы более наглядно представить, какие разряды включены, а какие отключены.

Листинг 15.4. Программа `dualview.c`

```
/* dualview.c -- разрядные поля и поразрядные операции */
#include <stdio.h>
/* КОНСТАНТЫ РАЗРЯДНЫХ ПОЛЕЙ */
/* свойства прозрачности и видимости */
#define YES     1
#define NO      0
/* стили линии     */
#define SOLID   0
#define DOTTED  1
#define DASHED  2
/* основные цвета  */
#define BLUE    4
#define GREEN   2
#define RED     1
```

```
/* смешанные цвета */
#define BLACK   0
#define YELLOW (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN (GREEN | BLUE)
#define WHITE (RED | GREEN | BLUE)
/* разрядные константы */
#define OPAQUE        0x1
#define FILL_BLUE     0x8
#define FILL_GREEN    0x4
#define FILL_RED      0x2
#define FILL_MASK     0xE
#define BORDER        0x100
#define BORDER_BLUE   0x800
#define BORDER_GREEN  0x400
#define BORDER_RED    0x200
#define BORDER_MASK   0xE00
#define B_SOLID       0
#define B_DOTTED      0x1000
#define B_DASHED      0x2000
#define STYLE_MASK    0x3000
const char * colors[8] = {"черный", "красный", "зеленый", "желтый",
                          "синий", "пурпурный", "голубой", "белый"};
struct box_props {
  unsigned int opaque       : 1;
  unsigned int fill_color   : 3;
  unsigned int              : 4;
  unsigned int show_border  : 1;
  unsigned int border_color : 3;
  unsigned int border_style : 2;
  unsigned int              : 2;
};
union Views    /* просмотр данных в виде структуры или переменной
                  типа unsigned short */
{
  struct box_props st_view;
  unsigned int ui_view;
};
void show_settings(const struct box_props * pb);
void show_settings1(unsigned short);
char * itobs(unsigned int n, char * ps);
int main(void)
{
 /* создание объекта Views, инициализация структуры box типа View */
  union Views box = {{YES, YELLOW , YES, GREEN, DASHED}};
  char bin_str[8 * sizeof(unsigned int) + 1];
  printf("Исходные параметры окна:\n");
  show_settings(&box.st_view);
  printf("\nПросмотр параметров окна с помощью переменной типа unsigned int:\n");
  show_settings1(box.ui_view);
```

```
  printf("комбинация разрядов: %s\n",
      itobs(box.ui_view,bin_str));
  box.ui_view &= ~FILL_MASK; /* отключение разрядов цвета фона */
  box.ui_view |= (FILL_BLUE | FILL_GREEN); /* переустановка цвета фона */
  box.ui_view ^= OPAQUE; /* переключение свойства прозрачности */
  box.ui_view |= BORDER_RED; /* неправильный подход */
  box.ui_view &= ~STYLE_MASK; /* очистка разрядов стиля */
  box.ui_view |= B_DOTTED; /* установка пунктирного стиля*/
  printf("\nИзмененные параметры окна:\n");
  show_settings(&box.st_view);
  printf("\n Просмотр параметров окна с помощью переменной типа unsigned int:\n");
  show_settings1(box.ui_view);
  printf("комбинация разрядов: %s\n",
  itobs(box.ui_view,bin_str));
  return 0;
}
void show_settings(const struct box_props * pb)
{
  printf("Фон %s.\n",
      pb->opaque == YES? "непрозрачный": "прозрачный");
  printf("Цвет фона %s.\n", colors[pb->fill_color]);
  printf("Рамка %s.\n",
          pb->show_border == YES? "видима" : "невидима");
  printf("Цвет рамки %s.\n", colors[pb->border_color]);
  printf ("Стиль линии рамки ");
  switch(pb->border_style)
  {
    case SOLID : printf("сплошной.\n"); break;
    case DOTTED : printf("пунктирный.\n"); break;
    case DASHED : printf("штриховой.\n"); break;
    default : printf("неизвестный тип.\n");
  }
}
void show_settings1(unsigned short us)
{
  printf("Фон %s.\n",
      us & OPAQUE == OPAQUE? "непрозрачный": "прозрачный");
  printf("Цвет фона %s.\n",
      colors[(us >> 1) & 07]);
  printf("Рамка %s.\n",
      us & BORDER == BORDER? "видимая" : "невидимая");
  printf ("Стиль линии рамки ");
  switch(us & STYLE_MASK)
  {
    case B_SOLID : printf("сплошной.\n"); break;
    case B_DOTTED : printf("пунктирный.\n"); break;
    case B_DASHED : printf("штриховой.\n"); break;
    default : printf("неизвестный тип.\n");
  }
```

```
  printf("Цвет рамки %s.\n",
      colors[(us >> 9) & 07]);
}

/* преобразование типа int в строку двоичных цифр */
char * itobs(unsigned int n, char * ps)
{
  int i;
  static int size = 8 * sizeof(unsigned int);

  for (i = size - 1; i >= 0; i--, n >>= 1)
     ps[i] = (01 & n) + '0';
  ps[size] = '\0';

  return ps;
}
```

Ниже показаны выходные данные, полученные в результате выполнения этой программы.

```
Исходные параметры окна:
Фон непрозрачный.
Цвет фона желтый.
Рамка видимая.
Цвет рамки зеленый.
Стиль линии рамки штриховой.

Просмотр параметров окна с помощью переменной типа unsigned int:
Фон непрозрачный.
Цвет фона желтый.
Рамка видимая.
Стиль линии рамки штриховой.
Цвет рамки зеленый.
комбинация разрядов: 00000000000000000010010100000111
Измененные параметры окна:
Фон прозрачный.
Цвет фона голубой.
Рамка видимая.
Цвет рамки желтый.
Стиль линии рамки пунктирный.

Просмотр параметров окна с помощью переменной типа unsigned int:
Фон прозрачный.
Цвет фона голубой.
Рамка невидимая.
Стиль линии рамки пунктирный.
Цвет рамки желтый.
комбинация разрядов: 00000000000000000001011100001100
```

Имеет смысл обсудить несколько моментов. Одно из отличий двух методик просмотра данных состоит в том, что методу с использованием поразрядных операций необходима информация о позициях разрядов. Например, для представления синего цвета в программе используется константа `BLUE`. Она имеет числовое значение `4`. Однако, поскольку данные размещены в структуре, в действительности для представления синего цвета фона используется третий разряд (не забывайте, что нумерация разрядов начина-

ется с нуля, как показано на рис. 15.1), а для представления синего цвета рамки используется 11-й разряд. Поэтому в программе определены дополнительные константы:

```
#define FILL_BLUE    0x8
#define BORDER_BLUE  0x800
```

Здесь `0x8` — значение комбинации, когда только третий разряд включен, а `0x800` — значение комбинации, когда включен только 11-й разряд. Первую константу можно использовать для установки синего цвета фона окна, а вторую — для установки синего цвета рамки. Шестнадцатеричная система записи упрощает определение разрядов, задействованных в параметре. Напомним, что каждая шестнадцатеричная цифра представляет четыре разряда. Таким образом, значение `0x800` соответствует комбинации разрядов `0x8`, но с тем отличием, что восемь разрядов переведены из отключенного состояния во включенное. Для эквивалентных чисел в десятичном исчислении (2048 и 8) это сходство величин не так очевидно.

Для получения значений, которые представляют собой степень 2, можно использовать операцию левого сдвига. Например, последние операции `#define` можно изменить следующим образом:

```
#define FILL_BLUE    1<<3
#define BORDER_BLUE  1<<11
```

Здесь второй операнд указывает степень числа 2. Таким образом, значение `0x8` равно 2^3, а значение `0x800` равно 2^{11}. Аналогично, выражение `1<<n` представляет собой значение, у которого включен только `n`-й разряд. Выражения, такие как `1<<11`, являются константами и вычисляются во время компиляции.

Для создания символьных констант вместо директивы `#define` можно воспользоваться перечислением. Ниже показан пример такого перечисления:

```
enum { OPAQUE = 0x1, FILL_BLUE = 0x8, FILL_GREEN = 0x4, FILL_RED = 0x2,
  FILL_MASK = 0xE, BORDER = 0x100, BORDER_BLUE = 0x800,
  BORDER_GREEN = 0x400, BORDER_RED = 0x200, BORDER_MASK = 0xE00,
  B_DOTTED = 0x1000, B_DASHED = 0x2000, STYLE_MASK = 0x3000};
```

Если создание переменных в перечислении не планируется, использовать дескриптор в объявлении не нужно.

Обратите внимание на то, что использовать поразрядные операции для изменения параметров сложнее. Для примера попробуем установить голубой цвет фона окна. При этом недостаточно просто включить разряды, соответствующие синему и зеленому цвету:

```
box.ui_view |= (FILL_BLUE | FILL_GREEN);  /* переустановка цвета фона */
```

Дело в том, что цвет определяется и состоянием "красного" разряда. Если он включен (например, чтобы получить желтый оттенок), в результате выполнения этой строки кода цвет станет белым. Проще всего до установки новых значений отключить все разряды, отвечающие за цвет. Именно поэтому программа содержит следующий код:

```
box.ui_view &= ~FILL_MASK;     /* отключение разрядов цвета фона */
box.ui_view |= (FILL_BLUE | FILL_GREEN); /* переустановка цвета  фона */
```

Для демонстрации ситуации, когда предварительное обнуление отвечающих за цвет разрядов не выполнено, программа содержит следующую строку:

```
box.ui_view |= BORDER_RED;     /* неправильный подход */
```

Поскольку разряд BORDER_GREEN был ранее включен, в результате получится комбинация BORDER_GREEN | BORDER_RED, что соответствует желтому цвету.

В подобных случаях применять разрядные поля проще:

```
box.st_view.fill_color = CYAN;    /*применение разрядного поля */
```

Предварительное обнуление разрядов не требуется. Кроме того, элементы разрядных полей допускают применение одних и тех же значений цвета для рамки и фона окна. В случае использования поразрядных операций эти значения должны быть различны (поскольку они должны учитывать позиции разрядов).

Теперь сравним два следующих оператора вывода на печать:

```
printf("Цвет рамки %s.\n", colors[pb->border_color]);
printf("Цвет рамки %s.\n", colors[(us >> 9) & 07]);
```

В первом операторе выражение pb->border_color имеет значение из диапазона 0–7, поэтому его можно использовать в качестве индекса массива colors. Получить ту же информацию с помощью поразрядных операций сложнее. Одно из решений состоит в использовании операции ui >> 9 для сдвига разрядов цвета рамки в крайне правое положение (это разряды 0–2), а затем скомбинировать полученное значение с маской 07, в результате чего все разряды, кроме трех правых, будут отключены. Полученное значение будет лежать в диапазоне 0–7 и может использоваться в качестве индекса массива colors.

Внимание!

Соответствие между разрядными полями и позициями разрядов зависит от применяемой платформы.

Например, при выполнении программы из листинга 15.4 в системе Macintosh будут получены следующие выходные данные:

```
Исходные параметры окна:
Фон непрозрачный.
Цвет фона желтый.
Рамка видимая.
Цвет рамки зеленый.
Стиль линии рамки штриховой.

Просмотр параметров окна с помощью переменной типа unsigned int:
Фон прозрачный.
Цвет фона черный.
Рамка невидимая.
Стиль линии рамки сплошной.
Цвет рамки черный.
комбинация разрядов: 10110000101010000000000000000000

Измененные параметры окна:
Фон прозрачный.
Цвет фона желтый.
Рамка видимая.
Цвет рамки зеленый.
Стиль линии рамки штриховой.
```

```
Просмотр параметров окна с помощью переменной типа unsigned int:
Фон непрозрачный.
Цвет фона голубой.
Рамка видимая.
Стиль линии рамки пунктирный.
Цвет рамки красный.
комбинация разрядов: 10110000101010000001001000001101
```

Здесь изменения затрагивают те же самые разряды, но в системе Macintosh загрузка структур в память выполняется по-другому. В частности, первое разрядное поле загружается в позицию старшего, а не младшего разряда. Поэтому представление структуры отразится на первых 16 разрядах (которые следуют не в том порядке, что на платформе PC). При этом представление данных типа `unsigned int` затрагивает последние 16 разрядов. По этой причине допущение, которое программа из листинга 15.4 делает о позициях разрядов, для Macintosh неприменимо. Использование поразрядных операций для изменения параметров прозрачности и цвета фона привело к переустановке не тех разрядов.

Ключевые понятия

Одна из особенностей, которая отличает C от большинства языков высокого уровня, связана с возможностью доступа к отдельным разрядам целочисленного значения. Это часто играет важную роль во взаимодействии с оборудованием и операционными системами.

Язык C обладает двумя основными средствами операций с разрядами. Одно из них представляет собой семейство поразрядных операций, а другое состоит в возможности создания структур с разрядными полями.

Обычно, но не всегда, программы, использующие эти средства, предназначены для определенных аппаратных платформ или операционных систем и не обладают свойством переносимости.

Резюме

Вычислительные системы тесно связаны с двоичной системой счисления, поскольку нули и единицы могут представлять отключенное и включенное состояние разрядов памяти или регистров. Хотя язык C не допускает написания чисел в двоичной форме, он распознает восьмеричные и шестнадцатеричные системы записи. Подобно тому, как двоичная цифра представляет один разряд, восьмеричная цифра представляет три разряда, а шестнадцатеричная — четыре разряда. Эта взаимосвязь позволяет сравнительно просто преобразовывать двоичные числа в восьмеричную или шестнадцатеричную форму.

В языке C существуют несколько поразрядных операций. Это название связано со способностью управлять каждым разрядом значения по отдельности. Поразрядная операция отрицания (`~`) инвертирует каждый разряд операнда (единицы заменяются нулями и наоборот). Поразрядная операция “И” (`&`) формирует значение из двух операндов. Каждый разряд значения включается, если включены соответствующие разряды обоих операндов. В противном случае разряд отключается. Поразрядная операция “ИЛИ” (`|`) также формирует значение из двух операндов. Разряд значения вклю-

чен, если соответствующий разряд хотя бы одного из операндов включен. В противном случае разряд отключен. Поразрядная операция исключающего "ИЛИ" (^) действует аналогично с тем отличием, что результирующий разряд включается только в случае, когда включен соответствующий разряд лишь одного из операндов.

Помимо этого существуют операции сдвига влево (<<) и вправо (>>). Каждая из них создает значение, формируемое путем сдвига разрядов (влево или вправо) левого операнда на количество позиций, указываемое правым операндом. При операции сдвига влево освобождаемые разряды устанавливаются в 0. При операции сдвига вправо освобождаемые разряды устанавливаются в 0 для значений без знака (`unsigned`). Для значений со знаком (`signed`) поведение операции сдвига вправо зависит от используемой системы.

Для обращения к отдельным разрядам группы или значения можно использовать разрядные поля. При этом результат операций с разрядами будет зависеть от используемой системы.

Поразрядные операции служат средством взаимодействия программ на C в оборудованием, поэтому они чаще всего применяются для определенных систем.

Вопросы для самоконтроля

1. Преобразуйте следующие десятичные значения в двоичную форму:
   а. 3
   б. 13
   в. 59
   г. 119
2. Преобразуйте следующие двоичные значения в десятичную, восьмеричную и шестнадцатеричную форму:
   а. 00010101
   б. 01010101
   в. 01001100
   г. 10011101
3. Вычислите следующие выражения исходя из допущения, что каждое значение имеет 8 разрядов:
   а. `~3`
   б. `3 & 6`
   в. `3 | 6`
   г. `1 | 6`
   д. `3 ^ 6`
   е. `7 >> 1`
   ж. `7 << 2`
4. Вычислите следующие выражения исходя из допущения, что каждое значение имеет 8 разрядов:
   а. `~0`
   б. `!0`

в. `2 & 4`

г. `2 && 4`

д. `2 | 4`

е. `2 || 4`

ж. `5 << 3`

5. Поскольку в ASCII-коде используются только 7 последних разрядов, иногда остальные разряды необходимо маскировать. Как будет выглядеть подходящая маска в двоичной форме? В десятичной? В восьмеричной? В шестнадцатеричной?

6. В листинге 15.2 следующий код

```
while (bits-- > 0)
{
   mask |= bitval;
   bitval <<= 1;
}
```

можно заменить следующим фрагментом:

```
while (bits-- > 0)
{
   mask += bitval;
   bitval *= 2;
}
```

Программа будет по-прежнему работоспособной. Означает ли это, что операции `*= 2` и `<<= 1` эквивалентны? Можно ли то же сказать об операциях `|=` и `+=`?

7. а. Компьютер Tinkerbell содержит в специальном байте информацию, касающуюся оборудования. Этот байт может быть прочитан программой. Он содержит следующую информацию:

Разряды	*Значение*
0–1	Количество дисководов 1.44 Мбайт.
2	Не используется.
3–4	Количество приводов чтения компакт-дисков.
5	Не используется.
6–7	Количество жестких дисков.

Подобно платформе IBM PC, компьютер Tinkerbell заполняет разрядные поля структуры справа налево. Создайте шаблон разрядных полей, пригодный для хранения информации.

б. Компьютер Klinkerbell, ближайший аналог Tinkerbell, заполняет разрядные поля структур слева направо. Создайте соответствующий шаблон для системы Klinkerbell.

Упражнения по программированию

1. Напишите функцию, которая преобразует двоичную строку в числовое значение. Другими словами, если выполнить следующий код:

   ```
   char * pbin = "01001001";
   ```

   можно передать переменную `pbin` в качестве аргумента функции, которая должна вернуть значение `25` типа `int`.

2. Напишите программу, которая выполняет чтение двух двоичных строк как аргументов командной строки и выводит результаты применения операции `~` к каждому числу, а также результаты применения операций `&`, `|` и `^` к паре чисел. Отобразите результаты в виде двоичных строк.

3. Напишите функцию, которая принимает аргумент типа `int` и возвращает количество включенных разрядов в его значении. Протестируйте функцию в программе.

4. Напишите функцию, которая принимает два аргумента типа `int`: значение и позицию разряда. Если разряд в этой позиции включен, функция возвращает 1, в противном случае — 0. Протестируйте функцию в программе.

5. Напишите функцию, которая циклически сдвигает разряды значения типа `unsigned int` на указанное количество позиций влево. Например, функция `rotate_1(x,4)` перемещает разряды значения `x` на четыре позиции влево. При этом утраченные слева разряды воспроизводятся в правой части комбинации. Другими словами, вытесненный старший разряд помещается в позицию младшего разряда. Протестируйте функцию в программе.

6. Разработайте структуру разрядных полей, которая содержит следующую информацию:

   Идентификатор шрифта: число от 0 до 255

   Размер шрифта: число от 0 до 127

   Выравнивание: число от 0 до 2, представляющее опции выравнивания влево, по центру и вправо

   Полужирный: отключен (0) или включен (1)

   Курсив: отключен (0) или включен (1)

   Подчеркнутый: отключен (0) или включен (1)

   Воспользуйтесь этой структурой в программе, которая отображает параметры шрифта и дает пользователю возможность менять параметры с помощью циклического меню. Ниже приводится пример выполнения программы:

```
   ИД     РАЗМЕР     ВЫРАВНИВАНИЕ        Ж         К         Ч
    1         12        влево            откл.     откл.     откл.

f) изменить шрифт     s) изменить размер      a) изменить выравнивание
b) полужирный         i) курсив               u) подчеркнутый
q) выйти
s
Введите размер шрифта (0-127): 36
```

```
   ИД    РАЗМЕР    ВЫРАВНИВАНИЕ       Ж        К        Ч
    1        36       влево          откл.    откл.    откл.

f) изменить шрифт    s) изменить размер     a) изменить выравнивание
b) полужирный        i) курсив              u) подчеркнутый
q) выйти
a
Выберите выравнивание:
l) влево               c) по центру             r) вправо
r
   ИД    РАЗМЕР    ВЫРАВНИВАНИЕ       Ж        К        Ч
    1        36       вправо         откл.    откл.    откл.

f) изменить шрифт    s) изменить размер     a) изменить выравнивание
b) полужирный        i) курсив              u) подчеркнутый
q) выйти
i
   ИД    РАЗМЕР    ВЫРАВНИВАНИЕ       Ж        К        Ч
    1        36       вправо         откл.    вкл.     откл.

f) изменить шрифт    s) изменить размер     a) изменить выравнивание
b) полужирный        i) курсив              u) подчеркнутый
q) выйти
q
Программа завершена.
```

Чтобы обеспечить преобразование вводимых значений идентификатора и размера шрифта в значения из указанного диапазона, программа должна использовать операцию `&` и необходимые маски.

7. Напишите программу, которая выполняет действия, описанные в предыдущем упражнении, однако для хранения параметров шрифта должна использоваться переменная типа `unsigned long`. Для управления информацией вместо элементов структуры применяйте поразрядные операции.

ГЛАВА 16

Препроцессор и библиотека языка C

В этой главе:

- Директивы препроцессора: `#define`, `#include`, `#ifdef`, `#else`, `#endif`, `#ifndef`, `#if`, `#elif`, `#line`, `#error`, `#pragma`
- Функции: `sqrt()`, `atan()`, `atan2()`, `exit()`, `atexit()`, `assert()`, `memcpy()`, `memmove()`, `va_start()`, `va_arg()`, `va_copy()`, `va_end()`
- Дополнительные возможности препроцессора C
- Макросы-функции и условная компиляция
- Встраиваемые функции
- Библиотека C и ее некоторые удобные функции

Язык C построен на ключевых словах, выражениях, операторах, а также правилах их использования. Однако стандарт ANSI C выходит за рамки описания самого языка. Он кроме этого определяет функционирование препроцессора, устанавливает, какие функции формируют стандартную библиотеку C, и детально описывает работу этих функций. В этой главе изучается препроцессор и библиотека C. Начнем с препроцессора.

Препроцессор, как подсказывает его название, анализирует программу до ее компиляции. Следуя директивам, препроцессор заменяет символьные сокращения программы объектами, которые эти сокращения представляют. По запросу программы препроцессор может включать в нее другие файлы и выбирать код, который должен быть доступен компилятору. Препроцессор "не знает" правил языка C. В основном он преобразует один текст в другой. Это описание не дает точного представления о применении препроцессора, поэтому перейдем к конкретным примерам. Директивы `#define` и `#include` встречались уже неоднократно. Теперь можно объединить и расширить полученные сведения.

Первые шаги трансляции программы

До передачи управления препроцессору компилятор должен выполнить несколько этапов трансляции программы. Компилятор начинает работу с того, что устанавливает соответствие символов исходного кода с базовым набором символов. При этом обрабатываются многобайтные символы и триграфы — символьные расширения, кото-

рые привносят в язык C возможности работы с интернациональными шрифтами, установками и так далее. (Обзор этих расширений приводится в справочном разделе VII (приложение Б).)

Во вторую очередь компилятор обнаруживает все вхождения обратного слеша с последующим символом новой строки и удаляет их. В результате две физические строки, такие как

```
printf("That's wond\
erful!\n");
```

преобразуются в одну *логическую строку*:

```
printf("That's wonderful\n!");
```

Обратите внимание, что в этом контексте "символ новой строки" означает символ, сгенерированный нажатием клавиши <Enter> для перехода к новой строке файла исходного кода, но не символьное представление `\n`.

Это необходимо для подготовки к предварительной обработке (работе препроцессора), поскольку выражения препроцессора должны иметь длину в одну логическую строку, которая может содержать несколько физических строк.

Затем компилятор разбивает текст на последовательность лексем препроцессора, а также на последовательности пробелов и комментариев. (В общем случае, лексемы представляют собой группы, отделяемые друг от друга пробелами. Более подробно мы рассмотрим это понятие ниже.) Интересно отметить, что на этом этапе каждый комментарий заменяется символом пробела. Например, строка

```
int/* это не похоже на пробел*/fox;
```

примет вид:

```
int fox;
```

Кроме того, в некоторых реализациях компилятора последовательности пробельных символов (кроме символа новой строки) заменяются одним пробелом. Наконец, программа готова для этапа предварительной обработки, и препроцессор начинает поиск своих потенциальных директив, обозначаемых символом # в начале строки.

Именованные константы: #define

Подобно всем остальным директивам, директива препроцессора `#define` предусматривает наличие символа # в начале строки. Стандарт ANSI допускает наличие пробелов или символов табуляции перед знаком #. Кроме того, между символом # и остальной частью директивы может находиться пробел. Однако в прежних версиях C обычно требовалось, чтобы директива начиналась в крайней левой позиции строки, а пробелы между символом # и остальной частью директивы не допускались. Директива может находиться в любом месте исходного файла. Ее область действия распространяется от позиции вхождения директивы до конца файла. В рассмотренных ранее программах директивы часто использовались для определения символических, или *именованных*, констант. Однако, как нам предстоит убедиться, область применения директив этим не ограничивается. Листинг 16.1 демонстрирует некоторые возможности и свойства директивы `#define`.

Директива препроцессора выполняется до тех пор, пока после знака # не встретится символ новой строки. Другими словами, длина директивы ограничена одной строкой. Однако, как уже говорилось, комбинации обратного слеша и символа новой строки удаляются до начала работы препроцессора, поэтому директиву можно распространить на несколько физических строк. Тем не менее, эти строки будут образовывать одну логическую строку.

Листинг 16.1. Программа `preproc.c`

```
/* preproc.c -- простые примеры директив препроцессора */
#include <stdio.h>
#define TWO 2  /* при желании можно использовать комментарии   */
#define OW "Логика - последнее убежище лишенных\
воображения. - Оскар Уальд" /* обратный слеш переносит         */
                            /* определение на следующую строку */
#define FOUR TWO*TWO
#define PX printf("X = %d.\n", x)
#define FMT "X = %d.\n"
int main(void)
{
  int x = TWO;

  PX;
  x = FOUR;
  printf(FMT, x);
  printf("%s\n", OW);
  printf("TWO: OW\n");

  return 0;
}
```

Каждая строка с директивой #define (или логическая строка) состоит из трех частей. Первая часть представляет собой саму директиву #define. Вторая часть — выбранное программистом сокращение, называемое *макросом*. Некоторые макросы, как в приведенном выше примере, представляют значения. Они называются *макросами-объектами*. (В языке С еще существуют *макросы-функции*, о которых речь пойдет позже.) Имя макроса не должно содержать пробелов. На макросы распространяются правила именования переменных: допускаются только буквы, цифры и символ подчеркивания (_). Первый символ не должен быть цифрой. Третья часть (остаток строки) называется *списком замены* или *телом* (рис. 16.1).

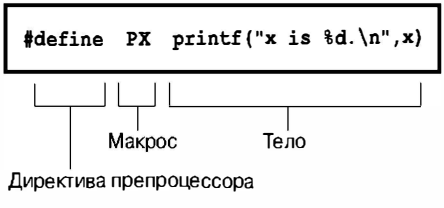


Рис. 16.1. *Части объектного макроопределения*

Когда препроцессор обнаруживает в программе экземпляр имени одного из макросов, он почти всегда заменяет его телом. (Как вскоре будет показано, из этого правила существует одно исключение.) Этот процесс перехода от макроса к подставляемому итоговому значению называется *макрорасширением*. Обратите внимание, что в строку `#define` можно включать стандартные комментарии. Как уже говорилось, до начала работы препроцессора каждый комментарий заменяется пробелом.

Выполним программу и посмотрим на выходные данные:

```
X = 2.
X = 4.
Логика - последнее убежище лишенных воображения. - Оскар Уальд
TWO: OW
```

Проанализируем работу программы. Оператор

```
int x = TWO;
```

преобразован следующим образом:

```
int x = 2;
```

поскольку значение `2` заменило идентификатор `TWO`. Оператор

```
PX;
```

заменен следующим оператором:

```
printf("X is %d.\n", x);
```

Это новый прием, поскольку до сих пор мы использовали макросы только для представления констант. Здесь показано, что макрос может представлять любую строку, даже целое выражение на языке C. Однако заметьте, что это строка-константа; имя `PX` определяет печать лишь переменной с именем `x`.

Следующая строка также демонстрирует новый прием. Может создаться впечатление, что имя `FOUR` заменено значением `4`, но на самом деле имел место следующий процесс. Строка

```
x = FOUR;
```

преобразована в выражение

```
x = TWO*TWO;
```

которое в свою очередь привело к следующему результату:

```
x = 2*2;
```

На этом процесс макрорасширения завершается. Умножение выполняется не на стадии работы препроцессора, а во время компиляции, поскольку компилятор C на этом этапе вычисляет все выражения-константы (выражения, которые содержат только константы). Препроцессор не выполняет вычислений, он лишь подставляет указанные директивами строки вместо имен макросов.

Обратите внимание, что макроопределение может включать другие макросы. (Некоторые компиляторы не поддерживают вложенность макросов.)

Выражение

```
printf (FMT, x);
```

приводится к виду:

```
printf("X = %d.\n",x);
```

поскольку имя FMT заменяется соответствующей строкой. Этот подход удобен в случаях, когда длинную управляющую строку приходится использовать несколько раз. Вместо этого можно было применить следующий оператор:

```
const char * fmt = "X = %d.\n";
```

а затем использовать имя fmt в качестве управляющей строки функции printf().

В следующей строке имя OW заменяется соответствующей строкой. Двойные кавычки делают строку замещения символьной строкой-константой. Компилятор сохранит ее в массиве, завершаемом нулевым символом. Таким образом, директива

```
#define HAL 'Z'
```

определяет символьную константу, а директива

```
#define HAP "Z"
```

определяет строку символов: Z\0.

В нашем примере символу перевода строки предшествует обратный слеш, чтобы распространить директиву на следующую строку:

```
#define OW "Логика - последнее убежище лишенных\
воображения. - Оскар Уальд"
```

Обратите внимание, что вторая строка выровнена влево. Если бы директива имела вид:

```
#define OW "Логика - последнее убежище лишенных\
   воображения. - Оскар Уальд"
```

то выходные данные программы выглядели бы так:

```
Логика - последнее убежище лишенных    воображения. - Оскар Уальд
```

Дело в том, что пробелы перед словом "воображения" считаются частью строки.

Обычно, когда препроцессор обнаруживает в программе один из макросов, он заменяет его текстовым эквивалентом. Если строка замещения содержит вложенные макросы, они также заменяются соответствующими строками. Единственное исключение составляет случай, когда макрос заключен в двойные кавычки. Поэтому строка

```
printf("TWO: OW");
```

выводит текст: TWO: OW буквально, а не преобразует его в строку:

```
2: Логика - последнее убежище лишенных воображения. - Оскар Уальд
```

Чтобы указанная выше строка была выведена на экран, следует применить показанный ниже код:

```
printf("%d: %s\n", TWO, OW);
```

Здесь имя макроса не заключено в двойные кавычки.

Когда необходимо использовать символические константы? Их целесообразно применять для обозначения числовых констант. Если в вычислениях применяется некоторое постоянное число, символическое имя делает понятнее его назначение. Если число представляет собой размер массива, символическое имя упрощает изменение этой величины и, соответственно, граничных условий выполнения циклов, которые этот массив обрабатывают. Предположим, число представляет собой системный код, такой как EOF (конец файла).

Символическое представление этого значения существенно повышает переносимость программы. Достаточно будет изменить одно-единственное определение EOF. Наглядность мнемонического обозначения, простота изменения величин и переносимость определяют целесообразность применения символических констант.

Однако ключевое слово const, поддерживаемое в современном языке C, служит более гибким средством создания констант. Оно позволяет создавать глобальные и локальные константы, числовые константы, массивы-константы и структуры-константы. С другой стороны, макросы-константы могут применяться для указания размеров стандартных массивов, а также инициализирующих значений для величин со спецификатором const.

```
#define LIMIT 20
const int LIM = 50;
static int data1[LIMIT];      // допустимо
static int data2[LIM];        // допустимо
const int LIM2 = 2 * LIMIT;   // допустимо
const int LIM3 = 2 * LIM;     // допустимо
```

Лексемы

С технической точки зрения тело макроса представляет собой строку *лексем*, а не строку символов. Обрабатываемые препроцессором С лексемы являются отдельными "словами" тела макроопределения. Они отделяются друг от друга пробелами. Например, определение

```
#define FOUR 2*2
```

содержит одну лексему — последовательность 2*2, а определение

```
#define SIX 2 * 3
```

содержит три лексемы: 2, * и 3.

Строки символов и строки лексем отличаются обработкой множественных пробелов. Рассмотрим следующее определение:

```
#define EIGHT 4   *   8
```

Препроцессор, который интерпретирует тело макроса как строку символов, заменяет имя EIGHT строкой 4   *   8.

Другими словами, дополнительные пробелы становятся частью заменяющего текста. Если же препроцессор интерпретирует тело макроса как строку лексем, имя EIGHT будет заменено тремя лексемами, разделенными одиночными пробелами: 4 * 8. Другими словами, интерпретация в качестве строки символов предполагает, что пробелы являются частью тела макроса, а интерпретация в качестве строки лексем предполагает, что пробелы служат разделителями лексем. На практике некоторые компиляторы С рассматривают тела макросов как строки, а не лексемы. Это различие имеет практическое значение только для более сложных вариантов применения по сравнению с рассмотренными здесь.

В частности, компилятор С реализует более сложный алгоритм обработки лексем по сравнению с препроцессором. Компилятор учитывает правила языка С. Для разделения лексем ему не всегда нужны пробелы. Например, компилятор С рассматривает выражение 2*2 как три лексемы, поскольку он распознает каждый символ 2 в качестве константы, а символ * в качестве операции.

Переопределение констант

Предположим, для константы LIMIT определено значение 20, а затем в том же файле для нее определяется значение 25. Этот процесс называется *переопределением константы*. Правила переопределения зависят от реализации языка. Некоторые компиляторы считают попытку переопределения ошибкой, если новое определение не совпадает со старым. Другие допускают переопределение, но могут выводить предупреждающее сообщение. Стандарт ANSI допускает переопределение только в случае, когда новое определение дублирует прежнее.

Совпадение определений означает, что их тела содержат одни и те же лексемы, следующие в одном и том же порядке. Поэтому приведенные ниже определения эквивалентны:

```
#define SIX 2 * 3
#define SIX 2   *  3
```

Оба определения содержат три одинаковых лексемы, дополнительные пробелы не считаются частью тела определения. Следующее определение считается отличающимся:

```
#define SIX 2*3
```

Оно содержит лишь одну лексему, а не три, поэтому не соответствует предыдущим определениям. Чтобы переопределить макрос, следует воспользоваться директивой #undef, которая будет рассмотрена позже.

Если требуется переопределить некоторые константы, для достижения цели иногда проще воспользоваться ключевым словом const и правилами определения области действия.

Использование аргументов в директиве #define

С помощью аргументов можно создавать *макросы-функции*, которые во многом действуют подобно обычным функциям. Аргументы макроса также заключаются в круглые скобки. Макросы-функции содержат один или несколько аргументов в скобках. Эти же аргументы присутствуют в выражении замены, как показано на рис. 16.2.

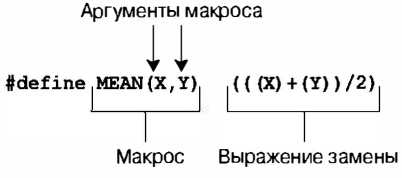


Рис. 16.2. *Составляющие макроса-функции*

Рассмотрим пример макроопределения:

```
#define SQUARE(X) X *X
```

Оно может использоваться в программе следующим образом:

```
z = SQUARE(2);
```

Это напоминает вызов функции, хотя поведение макроса не обязательно будет таким же. Листинг 16.2 демонстрирует использование этого и другого макроса. Некоторые примеры также указывают на возможные ошибки, поэтому читайте их внимательно.

Листинг 16.2. Программа mac_arg.c

```
/* mac_arg.c -- макросы с аргументами */
#include <stdio.h>
#define SQUARE(X) X*X
#define PR(X) printf("Результат: %d.\n", X)
int main(void)
{
  int x = 4;
  int z;
  printf("x = %d\n", x);
  z = SQUARE(x);
  printf("Вычисление SQUARE(x): ");
  PR(z);
  z = SQUARE(2);
  printf("Вычисление SQUARE(2): ");
  PR(z);
  printf("Вычисление SQUARE(x+2): ");
  PR(SQUARE(x+2));
  printf("Вычисление 100/SQUARE(2): ");
  PR(100/SQUARE(2));
  printf("x = %d.\n", x);
  printf("Вычисление SQUARE(++x): ");
  PR(SQUARE(++x));
  printf("После инкремента x = %x.\n", x);
  return 0;
}
```

Для макроса SQUARE существует следующее определение:

```
#define SQUARE(X) X*X
```

Здесь SQUARE — идентификатор макроса, X в выражении SQUARE(X) — аргумент макроса, а X*X — список замены. Каждое вхождение выражения SQUARE(X) в листинге 16.2 заменяется значением X*X. Отличие данного примера от предыдущих состоит в возможности использования в макросе любых символов помимо X. Символ X в макроопределении заменяется символом, который указывается при вызове макроса в программе. Таким образом, вызов SQUARE(2) заменяется значением 2*2, так что X действительно играет роль аргумента.

Однако, как вскоре будет показано, аргумент макроса не действует в точности так же, как аргумент функции. Ниже представлены выходные данные программы. Обратите внимание, что некоторые вычисления дают неожиданный результат. В действительности, ваш компилятор может вычислить выражение предпоследней строки с другим результатом:

```
x = 4
Вычисление SQUARE(x): Результат: 16.
Вычисление SQUARE(2): Результат: 4.
Вычисление SQUARE(x+2): Результат: 14.
Вычисление 100/SQUARE(2): Результат: 100.
x = 4.
Вычисление SQUARE(++x): Результат: 30.
После инкремента x = 6.
```

Первые две строки не таят в себе неожиданностей, но последующие результаты вычислений выглядят несколько странно. Аргумент x имеет значение 4. Можно предположить, что выражение SQUARE(x+2) эквивалентно значению 6*6 или 36, но выводится число 14, которое не является квадратом! Причина заключается в том, что, как уже говорилось, препроцессор не выполняет вычислений. Он просто заменяет строки. Препроцессор выполняет замену строки x+2 в соответствии с определением. Поэтому строка

```
x*x
```

принимает вид:

```
x+2*x+2
```

Единственной операцией умножения является 2*x. Если x равен 4, выражение вычисляется следующим образом:

```
4 + 2 * 4 + 2 = 4 + 8 + 2 = 14
```

Этот пример демонстрирует важное отличие вызова функции от вызова макроса. При вызове функции в нее передается значение аргумента в процессе выполнения программы. При вызове макроса лексема аргумента передается в программу до выполнения компиляции. Это другой процесс, который происходит в другое время. Можно ли исправить определение, чтобы вызов SQUARE(x+2) давал результат 36? Вполне. Достаточно ввести дополнительные скобки:

```
#define SQUARE(x) (x)*(x)
```

Теперь вызов SQUARE(x+2) даст результат (x+2)*(x+2), и операция умножения будет выполнена правильно.

Однако это не решает всех проблем. Рассмотрим события, которые предшествуют вычислению выражения в следующей строке:

```
100/SQUARE(2)
```

Это выражение преобразуется к виду:

```
100/2*2
```

В соответствии с правилами приоритета операций выражение вычисляется слева направо: (100/2)*2 или 50*2, что дает значение 100. Для исправления ошибки можно изменить определение макроса SQUARE(x) следующим образом:

```
#define SQUARE(x) (x*x)
```

Это приведет к результату 100/(2*2), что дает 100/4, или 25.

Следующее определение учитывает ошибки обоих примеров:

```
#define SQUARE(x) ((x)*(x))
```

Отсюда можно сделать вывод, что необходимо использовать достаточное количество скобок для обеспечения правильного порядка следования операций.

Но и эти меры предосторожности не спасают от ошибки в последнем примере:

```
SQUARE(++x)
```

Это выражение приводится к виду:

```
++x*++x
```

Здесь x инкрементируется дважды — до операции умножения и после нее:

```
++x*++x = 5*6 = 30
```

Поскольку приоритет операций здесь строго не определен, некоторые компиляторы дают результат 6*5. Другие компиляторы могут выполнять оба инкрементирования до операции умножения, что даст в результате 6*6. В любом случае x сначала имеет значение 4, а затем получает значение 6, хотя код выглядит так, будто инкремент выполняется всего лишь один раз.

Во избежание ошибок проще всего не использовать выражение ++x в качестве аргумента макроса. В общем случае не следует использовать в макросах операции инкремента и декремента. Обратите внимание, что выражение ++x применимо в качестве аргумента функции. Сначала будет вычислено значение 5, которое затем будет передано функции.

Создание строк из аргументов макроса: операция

Рассмотрим следующий макрос-функцию:

```
#define PSQR(X) printf("Квадрат X равен %d.\n", ((X)*(X)));
```

Предположим, программа содержит следующий вызов макроса:

```
PSQR(8);
```

Выходные данные программы:

```
Квадрат X равен 64.
```

Обратите внимание, что определение X в строке в двойных кавычках рассматривается как обычный текст, а не лексема, которую можно заменить.

Предположим, что требуется включить в строку аргумент макроса. Стандарт ANSI C это допускает. В строке замены макроса-функции символ # играет роль операции препроцессора, которая преобразует лексемы в строки. Предположим, x — это параметр макроса. Тогда #x — имя параметра, преобразованное в строку "x". Этот процесс называется *созданием строки*. Он демонстрируется в программе, представленной в листинге 16.3.

Листинг 16.3. Программа subst.c

```
/* subst.c -- замена в строке */
#include <stdio.h>
#define PSQR(x) printf("Квадрат " #x " равен %d.\n",((x)*(x)))
int main(void)
{
  int y = 5;
  PSQR(y);
  PSQR(2 + 4);

  return 0;
}
```

Выходные данные программы выглядят следующим образом:

```
Квадрат y равен 25.
Квадрат 2 + 4 равен 36.
```

При первом вызове макроса символы #x были заменены именем аргумента "y", а в результате второго вызова символы #x были заменены выражением "2 + 4". В соответствии со стандартом ANSI C эти строки были объединены с другими строками оператора printf(). Например, первый вызов макроса сгенерировал следующий оператор:

```
printf("Квадрат " "y" " равен %d.\n",((y)*(y)));
```

Затем три строки были объединены в одну:

```
"Квадрат y равен %d.\n"
```

Средство объединения препроцессора: операция

Подобно операции #, операция ## может быть применена в разделе замены макроса-функции. Кроме того, она может использоваться в разделе замены макроса-объекта. Операция ## объединяет две лексемы в одну. Предположим, существует такое определение:

```
#define XNAME(n) x ## n
```

Тогда вызов макроса

```
XNAME(4)
```

приведет к следующему результату:

```
x4
```

В листинге 16.4 этот и подобный ему макросы используются для "склеивания" лексем с помощью операции ##.

Листинг 16.4. Программа `glue.c`

```
// glue.c -- использование операции ##
#include <stdio.h>
#define XNAME(n) x ## n
#define PRINT_XN(n) printf("x" #n " = %d\n", x ## n);
int main(void)
{
  int XNAME(1) = 14;  // результат: int x1 = 14;
  int XNAME(2) = 20;  // результат: int x2 = 20;

  PRINT_XN(1);        // результат: printf("x1 = %d\n", x1);
  PRINT_XN(2);        // результат: printf("x2 = %d\n", x2);

  return 0;
}
```

Ниже показаны выходные данные этой программы:

```
x1 = 14
x2 = 20
```

Обратите внимание, что в макросе PRINT_XN() операция # используется для объединения строк, а операция ## — для объединения лексем в новый идентификатор.

Варьируемые макросы: ... и __VA_ARGS__

Некоторые функции, такие как printf(), могут принимать различное количество аргументов. Рассмотренный ранее заголовочный файл stdvar.h предоставляет средства создания определяемых пользователем функций с различным количеством аргументов. Стандарт C99 играет ту же роль для макросов. Термин "варьируемый" вошел в обиход для обозначения этой возможности, хотя он и не используется в стандарте. (Однако процесс расширения словаря C, который обусловил появление терминов *создание строк* и *варьируемый*, пока не привел к тому, что макросы и функции с фиксированным количеством аргументов называются *нормируемыми* макросами и *фиксируемыми* функциями.)

Дело в том, что последний аргумент в списке аргументов макроопределения может быть троеточием. В таком случае в разделе замены может использоваться предопределенный макрос __VA_ARGS__, который подставляется вместо троеточия. Для примера рассмотрим следующее определение:

```
#define PR(...) printf(__VA_ARGS__)
```

Предположим, далее программа содержит следующие вызовы макроса:

```
PR("Здравствуйте");
PR("вес = %d, доставка = $%.2f\n", wt, sp);
```

В случае первого вызова макрос __VA_ARGS__ подставляется вместо одного аргумента:

```
"Здравствуйте"
```

В случае второго вызова он заменяет три аргумента:

```
"вес = %d, доставка = $%.2f\n", wt, sp
```

Результат преобразования:

```
printf("Здравствуйте");
printf("вес = %d, доставка = $%.2f\n", wt, sp);
```

В листинге 16.5 представлен более сложный пример, в котором используются объединение строк и операция #.

Листинг 16.5. Программа variadic.c

```
// variadic.c -- варьируемые макросы
#include <stdio.h>
#include <math.h>
#define PR(X, ...) printf("Сообщение " #X ": " __VA_ARGS__)
int main(void)
{
  double x = 48;
  double y;

  y = sqrt(x);
  PR(1, "x = %g\n", x);
  PR(2, "x = %.2f, y = %.4f\n", x, y);

  return 0;
}
```

При первом вызове макроса X имеет значение 1, поэтому обозначение #X заменяется строкой "1". В результате выполняется следующее преобразование:

```
print("Сообщение " "1" ": " "x = %g\n", x);
```

Затем четыре строки объединяются, сводя вызов к следующему виду:

```
print("Сообщение 1: x = %g\n", x);
```

В конечном итоге имеем такие выходные данные программы:

```
Сообщение 1: x = 48
Сообщение 2: x = 48.00, y = 6.9282
```

Не забывайте, что троеточие должно быть последним аргументом макроса (показанный ниже макрос содержит ошибку):

```
#define WRONG(X, ..., Y) #X #__VA_ARGS__ #y
```

Макрос или функция?

Многие задачи могут решаться за счет использования макроса с аргументами либо функции. Что именно применить? Здесь нет простых и строгих правил, однако мы рассмотрим некоторые рекомендации.

Макросы несколько сложнее использовать по сравнению с обычными функциями, поскольку для них характерны побочные эффекты, и они требуют осмотрительности. Некоторые компиляторы ограничивают определение макроса одной строкой. Вероятно, лучше следовать этому ограничению даже в случае, когда компилятор его не накладывает.

Выбор между макросом и функцией связан с достижением компромисса между быстродействием и размером кода. Макрос генерирует встраиваемый код. Другими словами, в программу вводится оператор. Если макрос используется 20 раз, в программу вставляется 20 строк кода. Когда 20 раз используется функция, в программе все равно содержится лишь единственный экземпляр ее операторов. Это уменьшает размер кода. С другой стороны, управление программой должно переходить к фрагменту кода, содержащему функцию, а затем возвращаться к вызывающей процедуре. Этот процесс занимает больше времени, чем выполнение встраиваемого кода.

Преимущество макросов состоит в том, что для них не имеют значения типы переменных. (Дело в том, что они оперируют строками символов, а не реальными значениями.) Поэтому макрос SQUARE(x) может с одинаковым успехом использоваться для аргументов как типа int, так и типа float.

Стандарт С99 предоставляет третью альтернативу — встраиваемые функции. Они рассматриваются в этой главе позже.

Программисты обычно используют макросы для реализации простых функций, как показано в следующих примерах:

```
#define MAX(X,Y)    ((X) > (Y) ? (X) : (Y))
#define ABS(X)      ((X) < 0 ? -(X) : (X))
#define ISSIGN(X)   ((X) == '+' || (X) == '-' ? 1 : 0)
```

(Последний макрос возвращает значение 1, то есть истинное, если x представляет собой символ алгебраического знака.)

Ниже приводится несколько рекомендаций.

- Помните, что имя макроса не должно содержать пробелов. Однако пробелы допускаются в строке замены. Стандарт ANSI C допускает наличие пробелов в списке аргументов.
- Заключайте в скобки каждый аргумент, а также определение в целом. Это обеспечивает правильное разбиение по группам для выражений, подобных следующему:

```
forks = 2 * MAX(guests + 3, last);
```

- В именах макросов-функций используйте прописные буквы. Это правило не так широко распространено, как применение прописных букв в именах макросов-констант. Один из веских доводов в пользу написания имен в верхнем регистре состоит в том, что это напоминает о возможных побочных эффектах макросов.
- Если вы намерены использовать макрос вместо функции прежде всего для ускорения работы программы, сначала попытайтесь определить, даст ли это заметный эффект. Если макрос используется в программе один раз, значительное сокращение времени выполнения программы маловероятно. Намного больше для этой цели подходит макрос внутри вложенного цикла. Многие системы позволяют выполнять профилирование программы, которое помогает выявить фрагменты кода, требующие наибольшего времени выполнения.

Предположим, вы разработали несколько макросов-функций. Требуется ли их каждый раз набирать, чтобы использовать в новой программе? Нет, если воспользоваться директивой `#include`, которая рассматривается в следующем разделе.

Включение файлов: директива `#include`

Когда препроцессор встречает директиву `#include`, он выполняет поиск файла с указанным именем и включает его содержимое в текущий файл. Директива `#include` в файле исходного кода заменяется текстом включаемого файла. Это аналогично вводу содержимого включаемого файла в ту же позицию исходного файла. Существуют две разновидности директивы `#include`:

```
#include <stdio.h>          ←Имя файла в угловых скобках.
#include "mystuff.h"        ←Имя файла в двойных кавычках.
```

В системе Unix угловые скобки указывают препроцессору вести поиск файла в одном или нескольких стандартных системных каталогах. Двойные кавычки указывают, что сначала поиск должен выполняться в текущем каталоге (или другом каталоге, заданном в имени файла), а затем в стандартных каталогах:

```
#include <stdio.h>          ←Поиск в системных каталогах.
#include "hot.h"            ←Поиск в текущем рабочем каталоге.
#include "/usr/biff/p.h"    ←Поиск в каталоге /usr/biff.
```

Интегрированная среда разработки (IDE — Integrated Development Environment) также характеризуется наличием стандартного каталога или каталогов для заголовочных файлов системы. Многие среды IDE предоставляют опции меню для определения дополнительных каталогов, которые должны просматриваться в случае использования угловых скобок. Как и в системе Unix, наличие двойных кавычек означает, что сначала поиск должен выполняться в локальном каталоге. Однако этот каталог определяется компилятором. Некоторые компиляторы предусматривают поиск в каталоге, который содержит исходный код, другие — в текущем рабочем каталоге, а третьи — в каталоге, содержащем файл проекта.

Стандарт ANSI C не требует строгого соблюдения модели каталогов для размещения файлов, поскольку не все вычислительные системы организованы одинаково. Обычно принцип именования файлов зависит от системы, а использование угловых скобок и двойных кавычек — нет.

Для чего включаются файлы? Они содержат информацию, необходимую для компилятора. Например, файл `stdio.h` обычно содержит определения `EOF`, `NULL`, `getchar()` и `putchar()`. Два последних идентификатора определены как макросы-функции. Кроме того, файл содержит прототипы функций ввода-вывода языка C.

Суффикс `.h` традиционно используется для *заголовочных файлов*, содержащих информацию, которая помещается в заголовок программы. Заголовочные файлы часто содержат операторы препроцессора. Некоторые файлы, например, `stdio.h`, предоставляются системой, но разработчик и сам может создавать собственные заголовочные файлы.

Включение крупного заголовочного файла не всегда значительно увеличивает размер программы. В основном содержимое заголовочных файлов представляет собой информацию, используемую компилятором для создания исполняемого кода, а не добавляемый к коду материал.

Пример заголовочного файла

Предположим, разработчик создал структуру для хранения имени и фамилии некоторого лица, а также написал функции для использования этой структуры. Все необходимые объявления можно поместить в заголовочный файл. В листинге 16.6 представлен пример такого файла.

Листинг 16.6. Заголовочный файл `names_st.h`

```
// names_st.h -- заголовочный файл для структуры names_st
// константы
#define SLEN 32
// объявление структуры
struct names_st
{
  char first[SLEN];
  char last[SLEN];
};
// определения типов
typedef struct names_st names;
// прототипы функций
void get_names(names *);
void show_names(const names *);
```

Этот заголовочный файл содержит экземпляры многих типичных для таких файлов составляющих: директив #define, объявлений структур, операторов определения типов и прототипов функций. Обратите внимание, что ни один из этих элементов не является исполняемым кодом. Они представляют информацию, используемую компилятором при создании исполняемого кода.

Обычно исполняемый код размещается в файле исходного кода, а не в заголовочном файле. Примером может служить листинг 16.7, который содержит определения функций, соответствующих прототипам заголовочного файла. Данная программа содержит директиву включения заголовочного файла, который предоставляет компилятору информацию о типе данных names_st.

Листинг 16.7. Исходный файл name_st.c

```
// names_st.c -- определение функций names_st
#include <stdio.h>
#include "names_st.h"   // включение заголовочного файла
// определения функций
void get_names(names * pn)
{
  int i;
  printf("Пожалуйста, введите имя: ");
  fgets(pn->first, SLEN, stdin);
  i = 0;
  while (pn->first[i] != '\n' && pn->first[i] != '\0')
    i++;
  if (pn->first[i] == '\n')
    pn->first[i] = '\0';
  else
  while (getchar() != '\n')
    continue;
  printf("Пожалуйста, введите фамилию: ");
  fgets(pn->last, SLEN, stdin);
  i = 0;
  while (pn->last[i] != '\n' && pn->last[i] != '\0')
    i++;
  if (pn->last[i] == '\n')
    pn->last[i] = '\0';
  else
    while (getchar() != '\n')
      continue;
}
void show_names(const names * pn)
{
  printf("%s %s", pn->first, pn->last);
}
```

Функция get_names() вызывает функцию fgets(), чтобы избежать переполнения массивов, куда помещаются вводимые строки. Когда в сохраняемой строке встречается символ новой строки, функция заменяет его нулевым символом. При отсутствии

символа новой строки функция fgets() прекращает прием данных до достижения конца строки. Оставшиеся в текущей строке вводимые данные отбрасываются.

Листинг 16.8 содержит пример программы, в которой используются заголовочный и исходный файлы из предыдущих листингов.

Листинг 16.8. Программа useheader.c

```
// useheader.c -- использование структуры names_st
#include <stdio.h>
#include "names_st.h"
// компилировать вместе с файлом names_st.c
int main(void)
{
  names candidate;
  get_names(&candidate);
  printf("Добро пожаловать в программу, ");
  show_names(&candidate);
  printf(" !\n");
  return 0;
}
```

Пример выполнения программы:

```
Пожалуйста, введите имя: Наташа
Пожалуйста, введите фамилию: Карасева
Добро пожаловать в программу, Наташа Карасева!
```

Обратите внимание на следующие особенности программы:

- В обоих исходных файлах используется структура names_st, поэтому они содержат директиву включения заголовочного файла names_st.h.
- Необходимо скомпоновать и скомпилировать файлы исходного кода names_st.c и useheader.c.
- Объявления и подобные элементы содержатся в заголовочном файле names_st.h. Определения функций находятся в файле исходного кода names_st.c.

Область применения заголовочных файлов

Ознакомление с содержимым стандартных заголовочных файлов поможет дать представление о том, какого рода информация для них характерна. К наиболее типичному содержимому заголовочных файлов относятся:

- **Символические константы**. Например, типичный файл stdio.h определяет константы EOF, NULL и BUFSIZ (стандартный размер буфера ввода-вывода).
- **Макросы-функции**. Например, функция getchar() обычно определяется в виде getc(stdin), а getc() — в форме довольно сложного макроса. Заголовочный файл ctype.h, как правило, содержит макроопределения функций ctype.
- **Объявления функций**. Например, заголовочный файл string.h (в некоторых системах strings.h) содержит объявления семейства функций обработки строк.

Согласно стандарту ANSI C, эти объявления реализованы в форме прототипов функций.

- **Определения шаблонов структур**. Стандартные функции ввода-вывода используют структуру `FILE`, содержащую информацию о файле и связанным с ним буфером. Объявление этой структуры содержится в файле `stdio.h`.
- **Определения типов**. Напомним, что стандартные функции ввода-вывода используют аргумент типа указателя на `FILE`. Обычно файл `stdio.h` применяет директиву `#define` или `typedef`, чтобы имя `FILE` представляло указатель на структуру. Аналогичным образом, в заголовочных файлах определены типы данных `size_t` и `time_t`.

Многие программисты разрабатывают собственные стандартные заголовочные файлы, чтобы использовать их в своих программах. Это особенно целесообразно в случае, когда создано семейство взаимосвязанных функций и (или) структур.

Кроме того, заголовочные файлы можно использовать для объявления внешних переменных, к которым имеют общий доступ несколько файлов. Это целесообразно, например, когда создано семейство функций, совместно использующих переменную для сообщения о некотором состоянии, таком как ошибка. В этом случае можно определить переменную с внешним связыванием и областью действия файла в файле исходного кода, который содержит объявления функций:

```
int status = 0;     // переменная с областью действия
                    // в пределах файла исходного кода
```

Затем можно поместить ссылочное объявление в заголовочный файл, связанный с файлом исходного кода:

```
extern int status;  // в заголовочном файле
```

Этот код затем вводится в каждый файл, в который включается данный заголовочный файл. В результате переменная становится доступной файлам, использующим рассматриваемое семейство функций. Кроме того, за счет директивы включения это объявление входит в файл исходного кода функций. Наличие определяющего и ссылочного объявлений в одном файле допускается, если в них указан один и тот же тип данных.

Еще в заголовочный файл имеет смысл включать переменную или массив с областью действия в пределах файла, внутренним связыванием и спецификатором `const`. Спецификатор `const` предотвращает непреднамеренное изменение значения, а спецификатор `static` указывает, что каждый файл, включающий заголовок, принимает собственную копию констант. Это устраняет необходимость включения определяющего объявления в один файл и ссылочных объявлений в остальные файлы.

Директивы препроцессора `#include` и `#define` используются наиболее часто. Остальные директивы будут рассмотрены менее подробно.

Остальные директивы

Программистам иногда приходится создавать программы и библиотечные пакеты на языке C, которые предназначены для работы в различных системах. Коды программ для различных систем могут отличаться. Существует несколько директив пре-

процессора, помогающих программисту создавать код, который может переноситься из одной системы в другую за счет изменения значений макросов, определяемых с помощью #define. Директива #undef отменяет прежнее определение директивы #define. Директивы #if, #ifdef, #ifndef, #else, #elif и #endif позволяют указывать различные варианты кода, подлежащего компиляции. Директива #line дает возможность переустанавливать информацию о строках и файлах, директива #error служит для вывода сообщений об ошибках, а директива #pragma предоставляет инструкции компилятору.

Директива #undef

Директива #undef отменяет определение данной директивы #define. Предположим, имеется следующее определение:

```
#define LIMIT 400
```

Тогда директива

```
#undef LIMIT
```

удаляет это определение. Затем можно переопределить константу LIMIT, присвоив ей другое значение. Отмена определения константы LIMIT допустима даже в случае, если она не была предварительно определена. Если требуется использовать некоторое имя, но нет уверенности, что оно не было ранее определено, для надежности можно воспользоваться директивой #undef.

Определенность с точки зрения препроцессора

В отношении идентификаторов препроцессора применяются те же правила, что и для идентификаторов в С. Идентификатор должен содержать только буквы верхнего и нижнего регистра, цифры и символы подчеркивания. Когда препроцессор встречает в директиве идентификатор, он считает его определенным или неопределенным. Имеется в виду, что идентификатор *определен* посредством директивы препроцессора. Если идентификатор является именем макроса, созданного ранее директивой #define в том же файле, и к нему не применялась директива #undef, идентификатор определен. Если идентификатор представляет не макрос, а, например, переменную с областью действия в пределах файла, с точки зрения препроцессора он не определен.

Определенным может быть макрос-объект, а также пустой макрос или макрос-функция:

```
#define LIMIT 1000          // константа LIMIT определена
#define GOOD                // идентификатор GOOD определен
#define A(X) ((-(X))*(X)) // идентификатор A определен
int q;                      // q не макрос, поэтому не определен
#undef GOOD                 // идентификатор GOOD не определен
```

Обратите внимание, что область действия макроса #define начинается от позиции объявления и продолжается до применения к нему директивы #undef либо до конца файла, смотря что произойдет раньше. Кроме того, имейте в виду, что позиция директивы #define в файле будет зависеть от позиции директивы #include, если макрос включается в составе заголовочного файла.

Несколько предопределенных макросов, таких как __DATE__ и __FILE__ (о них речь пойдет ниже) всегда считаются определенными. Их определение не может быть отменено.

Условная компиляция

Остальные из упомянутых директив можно использовать для реализации условной компиляции. Другими словами, эти директивы могут указывать компилятору, принимать или игнорировать информационные блоки или код в соответствии с условиями.

Директивы #ifdef, #else и #endif

Для прояснения понятия условной компиляции воспользуемся кратким примером. Рассмотрим следующий код:

```
#ifdef MAVIS
  #include "horse.h" // выполняется, если имя MAVIS определено
  #define STABLES 5
#else
  #include "cow.h"   // выполняется, если MAVIS не определено
  #define STABLES 15
#endif
```

Здесь применены отступы, допускаемые новыми реализациями языка и стандартом ANSI. Для прежних реализаций может потребоваться выровнять все директивы или хотя бы символы # (см. следующий пример) влево:

```
#ifdef MAVIS
#   include "horse.h"      /* выполняется, если имя MAVIS определено */
#   define STABLES    5
#else
#   include "cow.h"       /* выполняется, если MAVIS не определено */
#   define STABLES   15
#endif
```

Директива #ifdef указывает, что, если следующий идентификатор (MAVIS) был определен препроцессором, необходимо выполнить все директивы и компилировать весь код до следующей директивы #else или #endif, смотря какая из них встретится раньше. Если присутствует директива #else, подлежит выполнению весь код между директивами #else и #endif, если идентификатор не определен.

Форма директив #ifdef–#else во многом подобна форме операторов if–else языка C. Основное отличие состоит в том, что препроцессор не распознает фигурные скобки ({}) как метод обозначения блока. Поэтому для выделения блоков директив применяются директивы #else (если присутствует) и #endif (обязательна). Эти условные структуры могут быть вложенными. Как показано в листинге 16.9, данные директивы могут обозначать и блоки операторов языка C.

Листинг 16.9. Программа `ifdef.c`

```
/* ifdef.c -- использование условной компиляции */
#include <stdio.h>
#define JUST_CHECKING
#define LIMIT 4
```

```
int main(void)
{
  int i;
  int total = 0;
  for (i = 1; i <= LIMIT; i++)
  {
    total += 2*i*i + 1;
#ifdef JUST_CHECKING
  printf("i=%d, текущая сумма = %d\n", i, total);
#endif
  }
  printf("Итоговая сумма = %d\n", total);
  return 0;
}
```

В результате компиляции и выполнения программы выводятся следующие данные:

```
i=1, текущая сумма = 3
i=2, текущая сумма = 12
i=3, текущая сумма = 31
i=4, текущая сумма = 64
Итоговая сумма = 64
```

Если опустить определение JUST_CHECKING (можно поместить его в комментарий или отменить определение с помощью директивы #undef), а затем повторно скомпилировать программу, будет выведена только последняя строка. Этим приемом можно пользоваться, например, для отладки программы. Если определить идентификатор JUST_CHECKING и включить его в директиву #ifdef, в процесс компиляции будет вовлечен программный код вывода на печать промежуточных значений для отладки. После отладки можно удалить определение и повторно скомпилировать программу. Если впоследствии снова потребуется вывод промежуточных значений, можно повторно вставить определение и избавить себя от необходимости повторно вводить все дополнительные операторы печати. Еще одна область применения состоит в использовании директивы #ifdef для выбора альтернативных блоков кода, приспособленных для различных реализаций языка С.

Директива #ifndef

Директива #ifndef, как и #ifdef, может применяться совместно с директивами #else и #endif. Директива #ifndef обрабатывает условие, когда следующий идентификатор *не* определен; она представляет собой инверсию директивы #ifdef. Эта директива часто применяется для определения констант при условии, что они не были определены ранее. Рассмотрим пример:

```
/* arrays.h */
#ifndef SIZE
   #define SIZE 100
#endif
```

(Прежние реализации языка С могут не допускать написания директивы #define с отступом.)

Обычно эта конструкция используется для предотвращения множественных определений одного и того же макроса в случае включения нескольких заголовочных файлов, из которых любой может содержать определение. В этом случае определение первого заголовочного файла становится активным, а определения других файлов игнорируются. Рассмотрим еще одну область применения. Предположим, в заголовок файла помещена строка

```
#include "arrays.h"
```

В результате константа SIZE определена со значением 100. Однако если поместить в заголовок файла строки

```
#define SIZE 10
#include "arrays.h"
```

константе SIZE будет присвоено значение 10. Это определение предшествует обработке файла arrays.h, поэтому строка #define SIZE 100 пропускается. Таким приемом можно воспользоваться, например, для тестирования программы с применением массива меньшего размера. Добившись нормальной работы программы, можно удалить оператор #define SIZE 10 и выполнить повторную компиляцию. В результате не придется изменять объявление массива в заголовочном файле.

Директива #ifndef часто применяется для исключения многократного включения одного и того же файла. Поэтому заголовочные файлы часто содержат следующие строки:

```
/* things.h */
#ifndef THINGS_H_
   #define THINGS_H_
   /* остальная часть включаемого файла */
#endif
```

Предположим, этот файл каким-то образом включен несколько раз. Когда процессор встречает первое включение данного файла, идентификатор THINGS_H_ не определен, поэтому программа определяет его и обрабатывает остальную часть файла. Когда препроцессор встречает следующую директиву включения того же файла, идентификатор THINGS_H_ уже определен, поэтому остальная часть файла пропускается.

Каким образом файл может включаться несколько раз? Одна из наиболее распространенных причин заключается в том, что многие включаемые файлы содержат директивы включения других файлов. Можно явно включить такой файл, что приведет к включению указанных в нем файлов. В чем проблема? Некоторые элементы заголовочных файлов, такие как объявления типов структур, могут встречаться в программе только один раз. Во избежание многократного включения файлов стандартные заголовочные файлы содержат директивы #ifndef. Одна из задач — убедиться, что проверяемый идентификатор не определен в другом месте. Обычно поставщики программного обеспечения решают ее за счет использования имени файла в качестве идентификатора. При этом буквы пишутся в верхнем регистре, точки заменяются символами подчеркивания и добавляется символ подчеркивания (или два) в качестве префикса и суффикса. Например, файл stdio.h может содержать код, подобный следующему:

```
#ifndef _STDIO_H
#define _STDIO_H
// содержимое файла
#endif
```

Можете пользоваться этим приемом. Однако не следует указывать символ подчеркивания в качестве префикса, поскольку стандарт рассматривает такое использование как зарезервированное. Это поможет избежать случайного определения макроса, который конфликтует с каким-либо идентификатором стандартного заголовочного файла. В листинге 16.10 директива `#ifndef` обеспечивает защиту от многократного включения заголовочного файла из листинга 16.6.

Листинг 16.10. Заголовочный файл `names_st.h`

```
// names_st.h -- защита от многократного включения
#ifndef NAMES_H_
#define NAMES_H_

// константы
#define SLEN 32

// объявления структур
struct names_st
{
  char first[SLEN];
  char last[SLEN];
};

// определения типов
typedef struct names_st names;

// прототипы функций
void get_names(names *);
void show_names(const names *);

#endif
```

Можно протестировать функционирование этого заголовочного файла совместно с программой из листинга 16.11. Программа будет работать правильно, но если в листинге 16.10 удалить защитную директиву `#ifndef`, возникнет ошибка компиляции.

Листинг 16.11. Программа `doubincl.c`

```
// doubincl.c -- двойное включение заголовочного файла
#include <stdio.h>
#include "names.h"
#include "names.h" // непреднамеренное повторное включение

int main()
{
  names winner = {"Иван", "Иванов"};
  printf("Победителем стал %s %s.\n", winner.first,
     winner.last);

  return 0;
}
```

Директивы #if и #elif

Директива #if во многом подобна обычному оператору if языка C. После нее следует выражение, представляющее целочисленную константу. Оно считается истинным, если имеет ненулевое значение. В нем могут использоваться логические операции и операции отношения:

```
#if SYS == 1
#include "ibm.h"
#endif
```

Для расширения комбинации if-else может применяться директива #elif (в устаревших реализациях языка она недоступна). Рассмотрим следующий пример:

```
#if SYS == 1
   #include "ibmpc.h"
#elif SYS == 2
   #include "vax.h"
#elif SYS == 3
   #include "mac.h"
#else
   #include "general.h"
#endif
```

Многие новые реализации языка предлагают второй способ проверки условия определенности имени. Вместо строки

```
#ifdef VAX
```

можно применять следующую форму записи:

```
#if defined (VAX)
```

Здесь defined — оператор препроцессора, который возвращает значение 1, если его аргумент определен директивой #defined. В противном случае возвращается значение 0. Преимущество новой формы состоит в том, что в нее можно включить директиву #elif. Исходя из этого, можно переписать предыдущий пример следующим образом:

```
#if defined (IBMPC)
   #include "ibmpc.h"
#elif defined (VAX)
   #include "vax.h"
#elif defined (MAC)
   #include "mac.h"
#else
   #include "general.h"
#endif
```

Если эти строки используются, скажем, в системе VAX, необходимо определить имя VAX в этом файле ранее с помощью такой директивы:

```
#define VAX
```

Одно из применений условной компиляции состоит в достижении переносимости программ. За счет изменения нескольких ключевых определений в начале файла можно устанавливать различные значения и включать определенные файлы для различных систем.

Предопределенные макросы

Стандарт C предусматривает несколько предопределенных макросов, перечисленных в табл. 16.1.

Таблица 16.1. Предопределенные макросы

Макрос	*Назначение*
__DATE__	Строка символов в форме "Ммм дд гггг", представляющая дату препроцессорной обработки.
__FILE__	Строка символов, представляющая имя текущего файла исходного кода.
__LINE__	Целочисленная константа, представляющая номер строки в текущем файле исходного кода.
__STDC__	Принимает значение 1 для указания, что реализация соответствует стандарту C.
__STDC_HOSTED__	Принимает значение 1 для среды со статусом хоста; в противном случае имеет значение 0.
__STDC_VERSION__	Для стандарта C99 принимает значение 199901L.
__TIME__	Время трансляции в форме "чч:мм:сс".

На момент написания этой главы в стандарт C99 включено обозначение __func__. Оно расширяет строковое представление имени функции, содержащей идентификатор. По этой причине идентификатор должен иметь область действия а пределах функции, в то время как макросы, по сути, имеют область действия в пределах файла. Таким образом, обозначение __func__ является предопределенным идентификатором языка C, а не предопределенным макросом.

Листинг 16.12 содержит несколько примеров использования предопределенных идентификаторов. Обратите внимание, что некоторые из них являются нововведениями стандарта C99, поэтому созданные до ввода этого стандарта компиляторы могут их не поддерживать.

Листинг 16.12. Программа `predef.c`

```
// predef.c -- предопределенные идентификаторы
#include <stdio.h>
void why_me();

int main()
{
  printf("Файл: %s.\n", __FILE__);
  printf("Дата: %s.\n", __DATE__);
  printf("Время: %s.\n", __TIME__);
  printf("Версия: %ld.\n", __STDC_VERSION__);
  printf("Это строка %d.\n", __LINE__);
  printf("Это функция %s\n", __func__);
  why_me();

  return 0;
}
```

```
void why_me()
{
  printf("Это функция %s\n", __func__);
  printf("Это строка %d.\n", __LINE__);
}
```

Ниже показан пример выполнения программы:

```
Файл: predef.c.
Дата: Feb 19 2006.
Время: 10:00:30.
Версия: 199901.
Это строка 11.
Это функция main
Это функция why_me
Это строка 21.
```

Директивы #line и #error

Директива #line позволяет переустанавливать нумерацию строк и имя файла, выводимые с помощью макросов __LINE__ и __FILE__. Директиву #line можно использовать следующим образом:

```
#line 1000        // присваивает текущему номеру строки значение 1000
#line 10 "cool.c" // присваивает номеру строки значение 10,
                  // а имени файла строку cool.c
```

Директива #error указывает препроцессору вывести сообщение об ошибке, которое включает текст, заданный ее параметром. Если возможно, процесс компиляции должен приостановиться. Эту директиву можно использовать так:

```
#if __STDC_VERSION__ != 199901L
   #error Несоответствие стандарту C99
#endif
```

Директива #pragma

У современных компиляторов существует несколько настроек, которые могут изменяться с помощью аргументов командной строки или через меню интегрированной среды разработки. Директива #pragma позволяет помещать инструкции компилятору в исходный код. Например, во время разработки стандарта C99 он назывался C9X, и компилятор использовал следующую директиву для активации поддержки этого стандарта:

```
#pragma c9x on
```

Вообще говоря, каждый компилятор содержит собственный набор директив-инструкций. Они могут применяться, например, для управления объемом памяти, выделяемым под автоматические переменные, установки тщательности проверки ошибок или включения нестандартных языковых средств. Стандарт C99 предусматривает три стандартных директивы-инструкции технического свойства, которые здесь не рассматриваются.

Кроме того, стандарт С99 поддерживает операцию препроцессора _Pragma. Она преобразует строку в обычную инструкцию компилятору. Например, операция

```
_Pragma("nonstandardtreatmenttypeB on")
```

является эквивалентом следующей директивы:

```
#pragma nonstandardtreatmenttypeB on
```

Поскольку эта операция не использует символ #, она может выступать в качестве части макрорасширения:

```
#define PRAGMA(X) _Pragma(#X)
#define LIMRG(X) PRAGMA(STDC CX_LIMITED_RANGE X)
```

Затем можно воспользоваться следующей строкой кода:

```
LIMRG ( ON )
```

Между прочим, следующее определение не действует, хотя это не очевидно:

```
#define LIMRG(X) _Pragma(STDC CX_LIMITED_RANGE #X)
```

Дело в том, что здесь применяется конкатенация строк, но компилятор не выполняет эту операцию до завершения работы препроцессора.

Оператор _Pragma выполняет всю работу по преобразованию строк. Другими словами, управляющие последовательности строки преобразуются в представляемые ими символы. Таким образом, операция

```
_Pragma("use_bool \"true \"false")
```

приводится к директиве

```
#pragma use_bool "true "false
```

Встраиваемые функции

Обычно вызов функции связан с замедлением работы программы. Это означает, что вызов, передача аргументов, переход к коду функции и возврат значения занимает некоторое время выполнения. Сокращение таких затрат времени является одной из причин использования макросов-функций. Стандарт С99 предлагает альтернативу — *встраиваемые* (или *подставляемые*) *функции*. Эту особенность стандарта можно охарактеризовать так: "превращение функции во встраиваемую предполагает наивысшую из возможных скорость ее работы. Эффективность этого метода зависит от реализации языка". Поэтому встраивание функции может сократить обычные затраты времени на ее вызов либо не дать никакого эффекта.

Метод состоит в использовании спецификатора inline в объявлении функции. Обычно встраиваемые функции определяются до их первого использования в файле, поэтому определение играет роль также и прототипа. Таким образом, код имеет примерно следующий вид:

```
#include <stdio.h>
inline void eatline() // определение-прототип встраиваемой функции
{
  while (getchar() != '\n')
    continue;
}
```

```
int main()
{
...
  eatline(); // вызов функции
...
}
```

Встретив объявление встраиваемой функции, компилятор может, например, заменить вызов `eatline()` телом функции. Другими словами, эффект может сводиться к замене фрагмента следующим кодом:

```
#include <stdio.h>
inline void eatline() // определение-прототип встраиваемой функции
{
  while (getchar() != '\n')
    continue;
}
int main()
{
...
  while (getchar() != '\n') // вместо вызова функции
    continue;
...
}
```

Поскольку для встраиваемой функции не выделяется отдельный блок кода, нельзя получить ее адрес. (На самом деле это возможно, но тогда компилятор сгенерирует обычную функцию.) Кроме того, встраиваемая функция может не отображаться в отладчике.

Код встраиваемой функции должен быть коротким. Если размер кода большой, затраты времени на вызов функции невелики по сравнению временем выполнения тела функции. В результате применение встраиваемой функции не даст существенной экономии времени.

Компилятору для выполнения оптимизации, связанной с использованием встраиваемой функции, необходим анализ определения функции. Это означает, что определение встраиваемой функции должно содержаться в том же файле, что и ее вызов. По этой причине встраиваемые функции обычно имеют внутреннее связывание. Следовательно, если программа состоит из нескольких файлов, каждый файл, вызывающий встраиваемую функцию, должен содержать ее определение. Для выполнения этого условия проще всего поместить определение встраиваемой функции в заголовочный файл, а затем включить этот файл во все файлы, использующие данную функцию. Встраиваемая функция является исключением из правила, которое не рекомендует помещать исполняемый код в заголовочный файл. Поскольку встраиваемая функция имеет внутреннее связывание, ее определение в нескольких файлах не вызывает ошибок.

Язык C допускает несколько способов применения встраиваемых функций в многофайловых программах. Обычно язык C допускает наличие только одного определения функции, но на встраиваемые функции это ограничение не распространяется. Поэтому самый прямой путь — поместить определение встраиваемой функции в каждый файл, где она используется. Для облегчения задачи можно поместить определение встраиваемой функции в заголовочный файл, а затем включить его директивой `#include` в файлы исходного кода, где эта функция используется.

В отличие от C++, язык C допускает комбинирование встраиваемых определений с внешними определениями (определениями функций с внешним связыванием). Для примера рассмотрим следующий фрагмент кода:

```
// файл file1a.c
...
inline double square(double);
double square(double x) { return x * x; }
int main()
{
  double q = square(1.3);
...
// файл file2a.c
...
extern double square(double);
double square(double x) { int y; y = x*x; return y; }
void spam(double v)
{
  double kv = square(v);
...
// файл file3a.c
...
extern double square(double);
void masp(double w)
{
  double kw = square(w);
...
```

Здесь файл `file1a.c` должен использовать встраиваемую версию функции `square()`, которая определена в файле `file1a.c`. Однако в файлах `file2a.c` и `file3a.c` используется внешнее определение функции из файла `file2a.c`.

Язык C даже позволяет помещать внешнее объявление в файл, содержащий встраиваемое определение:

```
//file1b.c - будьте осторожны!
...
extern double square(double); // объявление функции square()
                              // как внешней
inline double square(double); // объявление функции square()
                              // как встраиваемой
double square(double x) { return x * x; }
int main()
{
  double q = square(1.3) + square(1.5); // которая из функций square()?
...
// файл file2b.c
...
extern double square(double);
double square(double x) { int y; y = x*x; return y; }
...
```

В данном случае при вызове функции `square()` в файле `file1b.c` компилятор может использовать любое из ее определений. Здесь даже не обязательно будет прослеживаться какая-либо последовательность. Например, в предыдущем фрагменте кода для вызова `square(1.3)` могла использоваться встраиваемая версия функции, а для вызова `square(1.5)` — внешняя версия. Стандарт предупреждает, что не следует писать код так, чтобы работа программы зависела от выбираемой версии определения функции. Как уже говорилось, если любой файл с определением встраиваемой функции применяет код, извлекающий адрес функции (например, за счет передачи имени функции в качестве фактического аргумента), компилятор генерирует определение внешней функции.

Библиотека C

Изначально официальной библиотеки C не существовало. Впоследствии возник фактический стандарт, основанный на реализации C для системы Unix. Комитет ANSI C, в свою очередь, разработал официальную стандартную библиотеку, которая в значительной степени основана на фактическом стандарте. Учитывая распространение языка C, комитет решил пересмотреть библиотеку, чтобы она могла реализовываться в широком разнообразии систем.

Мы уже рассматривали некоторые функции ввода-вывода, управления символами и строками, которые входят в библиотеку. В этой главе представлен обзор еще нескольких функций. Но сначала изучим способы применения библиотеки.

Доступ к библиотеке C

Способ получения доступа к библиотеке C зависит от реализации языка. Поэтому нужно ознакомиться с тем, как обращаются к системе наиболее часто используемые операторы. Во-первых, библиотечные функции часто содержатся в нескольких различных местах. Например, функция `getchar()` обычно определена как макрос в файле `stdio.h`, но функция `strlen()` чаще всего содержится в библиотечном файле. Во-вторых, для разных систем существуют свои способы вызова этих функций. Они рассматриваются в последующих разделах.

Автоматический доступ

Во многих системах достаточно скомпилировать программу, и наиболее распространенные библиотечные функции автоматически станут доступными.

Имейте в виду, что для используемых функций необходимо объявлять их типы. Обычно это достигается включением определенного заголовочного файла. Руководства, описывающие библиотечные функции, указывают, какие файлы необходимо включить. Однако в некоторых прежних реализациях языка требуется вводить объявления функций самостоятельно. На этот случай руководство пользователя указывает тип функции. Кроме того, в приложении Б содержится описание библиотеки ANSI C. Функции сгруппированы по заголовочным файлам.

Раньше имена заголовочных файлов не были совместимыми в различных реализациях языка. В стандарте ANSI C библиотечные функции сгруппированы в семейства. Для каждого семейства существует заголовочный файл, содержащий прототипы функций.

Включение файлов

Если функция определена в качестве макроса, с помощью директивы `#include` можно включить файл, содержащий ее определение. Часто сходные макросы объединены в заголовочном файле с характерным именем. Например, многие системы, включая все системы на базе стандарта ANSI C, содержат заголовочный файл `ctype.h`. Он содержит несколько макросов, которые определяют характер символа, например, верхний регистр, цифра и тому подобное.

Включение библиотек

На определенном этапе компиляции либо компоновки программы может потребоваться указать опцию включения библиотеки. Даже если система автоматически просматривает стандартную библиотеку, в ней могут существовать другие библиотеки функций, которые используются реже. Эти библиотеки должны явно запрашиваться с помощью опций времени компиляции. Обратите внимание, что этот процесс отличается от включения заголовочного файла. Заголовочный файл предоставляет объявления или прототипы функций. Опция включения библиотеки указывает системе, где искать код функций. Конечно, мы не можем рассмотреть особенности всех систем, но материал главы поможет вам определиться с поиском необходимых библиотек.

Использование описаний библиотеки

Объем книги не позволяет представить полный обзор библиотеки, но мы рассмотрим некоторые характерные примеры. Для начала обратимся к документации. Может существовать несколько источников, содержащих описания функций. Иногда реализация языка сопровождается интерактивным руководством. Интегрированная среда разработки часто обладает интерактивной справочной документацией. Поставщики языка С иногда предоставляют отпечатанные руководства пользователя, содержащие описания библиотечных функций, либо компакт-диск с аналогичным материалом. Несколько издательств опубликовали справочные пособия по функциям библиотеки С. Одни из них имеют обобщенный характер, а другие ориентированы на определенные реализации языка. Кроме того, как уже говорилось, краткое описание функций содержится в приложении Б этой книги.

Для чтения документации важнее всего понимать описание заголовков функций. Стиль описания со временем меняется. Для примера рассмотрим описание функции `fread()` в старой документации для Unix:

```
#include <stdio.h>
fread(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;
```

Сначала указывается необходимый для включения файл. Для функции `fread()`, параметров `ptr`, `sizeof(*ptr)` и `nitems` не указаны типы. Раньше для этих элементов по умолчанию принимался тип `int`. Однако из контекста ясно, что `ptr` является указателем. (В ранних версиях С указатели обрабатывались как целочисленные значения.) Аргумент `stream` объявлен как указатель на объект `FILE`. Объявление создает впечатление, что в качестве второго аргумента используется операция `sizeof`. На самом деле указывается, что значение этого аргумента должно представлять размер объекта, на который указывает аргумент `ptr`. Скорее всего, именно операция `sizeof` и будет

использоваться при вызове функции, но с точки зрения синтаксиса допустимо любое значение типа int.

Впоследствии форма описания изменилась следующим образом:

```
#include <stdio.h>
int fread(ptr, size, nitems, stream;)
char *ptr;
int size, nitems;
FILE *stream;
```

Теперь все типы данных указаны явно, а параметр ptr определен как указатель на тип char.

Стандарт ANSI C90 предлагает следующее описание:

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Прежде всего, в нем применен новый формат прототипа. Кроме того, некоторые типы данных изменены. Тип size_t определен как целочисленное значение без знака, возвращаемое операцией sizeof. Обычно для него принят тип unsigned int или unsigned long. Файл stddef.h содержит определение типа или директиву #define для аргумента size_t. Это определение доступно и для нескольких других файлов, включая stdio.h, как правило, за счет включения файла stddef.h. Во многие функции, включая fread(), встроена операция sizeof как часть фактического аргумента. Тип size_t обеспечивает соответствие формального аргумента этому типичному варианту использования.

Кроме того, в стандарте ANSI C используется указатель на void как обобщенный указатель для случаев, когда могут применяться указатели на различные типы данных. Например, первый аргумент функции fread() может представлять указатель на массив значений типа double или на некоторую структуру. Если фактический аргумент представляет указатель, скажем, на массив из 20 значений типа double, а формальный аргумент является указателем на void, компилятор примет вариант для подходящего типа и не будет "жаловаться" на несоответствие типов.

Недавно стандарт C99 внедрил в описание функций новое ключевое слово restrict:

```
#include <stdio.h>
size_t fread(void * restrict ptr, size_t size,
             size_t nmemb, FILE * restrict stream);
```

Теперь перейдем к обзору некоторых специфических функций.

Библиотека математических функций

Библиотека математических функций содержит множество полезных функций такого рода. Их объявления или прототипы содержатся в заголовочном файле math.h. В табл. 16.2 перечислено несколько функций, объявленных в файле math.h. Обратите внимание, что все углы измеряются в радианах (один радиан составляет $180/\pi$, или 57,296 градуса). В разделе V справочника (приложение Б) представлен полный список функций, определенных в стандарте C99.

Таблица 16.2. Стандартные математические функции ANSI C

Прототип	*Описание*
`double acos(double x)`	Возвращает угол (от 0 до π радиан), косинус которого равен x.
`double asin(double x)`	Возвращает угол (от $-\pi/2$ до $\pi/2$ радиан), синус которого равен x.
`double atan(double x)`	Возвращает угол (от $-\pi/2$ до $\pi/2$ радиан), тангенс которого равен x.
`double atan2(double y, double x)`	Возвращает угол (от $-\pi$ до π радиан), тангенс которого равен y/x.
`double cos(double x)`	Возвращает косинус x (x в радианах).
`double sin(double x)`	Возвращает синус x (x в радианах).
`double tan(double x)`	Возвращает тангенс x (x в радианах).
`double exp(double x)`	Возвращает экспоненциальную функцию x (e^x).
`double log(double x)`	Возвращает натуральный логарифм x.
`double log10(double x)`	Возвращает логарифм x по основанию 10.
`double pow(double x, double y)`	Возвращает x в степени y.
`double sqrt(double x)`	Возвращает квадратный корень x.
`double ceil(double x)`	Возвращает наименьшее целое, которое не меньше x.
`double fabs(double x)`	Возвращает абсолютное значение x.
`double floor(double x)`	Возвращает наибольшее целое, которое не больше x.

Воспользуемся библиотекой математических функций для решения типичной задачи преобразования декартовых координат в полярные (модуль и угол). Предположим, на сетке проведена линия, протяженность которой составляет 4 единицы по горизонтали (значение x) и 3 единицы по вертикали (значение y). Каковы длина (модуль) и направление линии? По законам тригонометрии:

```
модуль = квадратный корень (x**2 + y**2)
угол = арктангенс (y/x)
```

Библиотека `math` содержит функцию извлечения квадратного корня и две функции вычисления арктангенса, что позволяет выразить решение на языке C. Функция извлечения квадратного корня, `sqrt()`, принимает аргумент типа `double` и возвращает его квадратный корень — также значение типа `double`.

Функция `atan()` принимает аргумент типа `double` и возвращает угол, тангенс которого равен этому аргументу. К сожалению, функция `atan()` не учитывает квадрант вектора. Например, если координаты x и y вектора равны `-5` и `-5`, функция `atan()` даст результат 45 градусов, поскольку (–5)/(–5) = 1. Тот же результат будет для вектора с координатами `5` и `5`. Другими словами, функция `atan()` не различает векторы, углы которых отличаются на 180 градусов. (На самом деле функция `atan()` выводит результат в радианах, а не градусах, о преобразовании этих единиц речь пойдет ниже.)

К счастью, библиотека C содержит и функцию `atan2()`. Она принимает два аргумента: значения x и y. Таким образом, функция способна анализировать знаки координат и правильно определять угол. Его значение также выводится в радианах.

Чтобы преобразовать радианы в градусы, нужно умножить значение на 180 и разделить на π. Для вычисления значения π можно воспользоваться выражением `4 * atan(1)`. Эти операции продемонстрированы в листинге 16.13. Кроме того, в примере используются структуры и оператор typedef.

Листинг 16.13. Программа `rect_pol.c`

```
/* rect_pol.c -- преобразует ортогональные координаты в полярные */
#include <stdio.h>
#include <math.h>

#define RAD_TO_DEG (180/(4 * atan(1)))

typedef struct polar_v {
  double magnitude;
  double angle;
} POLAR_V;

typedef struct rect_v {
  double x;
  double y;
} RECT_V;

POLAR_V rect_to_polar(RECT_V);

int main(void)
{
  RECT_V input;
  POLAR_V result;
  puts("Введите координаты x,y; введите q для выхода:");
  while (scanf("%lf %lf", &input.x, &input.y) == 2)
  {
    result = rect_to_polar(input);
    printf("модуль = %0.2f, угол = %0.2f\n",
           result.magnitude, result.angle);
  }
  puts("Программа завершена.");
  return 0;
}

POLAR_V rect_to_polar(RECT_V rv)
{
  POLAR_V pv;
  pv.magnitude = sqrt(rv.x * rv.x + rv.y * rv.y);
  if (pv.magnitude == 0)
    pv.angle = 0.0;
  else
    pv.angle = RAD_TO_DEG * atan2(rv.y, rv.x);
  return pv;
}
```

Пример выполнения программы:

```
Введите координаты x,y; введите q для выхода:
10 10
модуль = 14.14, угол = 45.00
```

```
-12 -5
модуль = 13.00, угол = -157.38
q
Программа завершена.
```

Если в процессе компиляции будет выведено сообщение, такое как

```
Undefined: _sqrt
Не определено: _sqrt
```

или

```
'sqrt': unresolved external
'sqrt': неразрешенный внешний идентификатор
```

либо нечто подобное, компилятор или компоновщик не смог обнаружить библиотеку `math`. Система Unix требует, чтобы компоновщику было указано искать библиотеку `math` с помощью флага `-lm`:

```
cc rect_pol.c -lm
```

Компилятор gnu в системе Linux действует аналогично:

```
gcc rect_pol.c -lm
```

Библиотека утилит общего назначения

Библиотека утилит общего назначения содержит множество функций, включая генератор случайных чисел, функции поиска и сортировки, преобразования и управления памятью. В главе 12 уже рассматривались функции `rand()`, `srand()`, `malloc()` и `free()`. В соответствии со стандартом ANSI C прототипы этих функций содержатся в заголовочном файле `stdlib.h`. В разделе V справочника (приложение Б) перечислены все функции данного семейства. Сейчас мы рассмотрим некоторые из них подробнее.

Функции `exit()` и `atexit()`

Функция `exit()` уже использовалась ранее в нескольких примерах. Кроме того, функция `exit()` вызывается автоматически после возврата из функции `main()`. Стандарт ANSI предусматривает две новых привлекательных возможности, которыми мы еще не пользовались. Наибольшее значение из них имеет возможность указания определенных функций, которые должны вызываться во время выполнения функции `exit()`. Функция `atexit()` обеспечивает эту возможность за счет регистрации функций, вызываемых при выполнении `exit()`. Функция `atexit()` принимает в качестве аргумента указатель функции. Упомянутые приемы демонстрируются в листинге 16.14.

Листинг 16.14. Программа `byebye.c`

```
/* byebye.c -- пример использования функции atexit() */
#include <stdio.h>
#include <stdlib.h>
void sign_off(void);
void too_bad(void);

int main(void)
{
```

```
  int n;
  atexit(sign_off); /* регистрация функции sign_off() */
  puts("Введите целое число:");
  if (scanf("%d",&n) != 1)
  {
    puts("Это не целое!");
    atexit(too_bad); /* регистрация функции too_bad() */
    exit(EXIT_FAILURE);
  }
  printf("%d - это число %s.\n", n, (n % 2 == 0)? "четное" : "нечетное");
  return 0;
}
void sign_off(void)
{
  puts("Завершение работы очередной замечательной программы от");
  puts("SeeSaw Software!");
}
void too_bad(void)
{
  puts("SeeSaw Software приносит вам соболезнования");
  puts("в связи со сбоем программы.");
}
```

Ниже показан пример выполнения программы:

```
Введите целое число:
212
212 - это число четное.
Завершение работы очередной замечательной программы от
SeeSaw Software!
```

Если программа выполняется в интегрированной среде разработки, две последних строки могут не отображаться.

Рассмотрим еще один пример выполнения программы:

```
Введите целое число:
чего?
Это не целое!
SeeSaw Software приносит вам соболезнования
в связи со сбоем программы.
Завершение работы очередной замечательной программы от
SeeSaw Software!
```

Если программа выполняется в интегрированной среде разработки, четыре последних строки могут не отображаться.

Обсудим два основных момента: аргументы функций `atexit()` и `exit()`.

Использование функции `atexit()`

Это функция, которая принимает указатели на функции! При обращении к ней достаточно передать адрес функции, которую следует вызвать во время вызова `exit()`. Поскольку имя функции, когда оно используется в качестве аргумента, играет

роль адреса, аргументами могут служить имена sign_off или too_bad. Затем atexit() регистрирует указываемую функцию в списке функций, выполняемых при вызове exit(). Стандарт ANSI гарантирует, что список может вместить не менее 32 функций. Каждая из них добавляется путем отдельного вызова функции atexit(). Когда доходит очередь до функции exit(), все эти функции выполняются. Причем первой выполняется функция, добавленная в список последней.

Обратите внимание, что в результате недопустимого ввода пользователя вызываются обе функции (sign_off() и too_bad()). Однако в случае ввода допустимого значения вызывается только функция sign_off(). Дело в том, что оператор if регистрирует функцию too_bad() только для случая недопустимого ввода. Кроме того, заметьте, что первой вызывается функция, которая была зарегистрирована последней.

Функции, регистрируемые atexit(), подобные sign_off() и too_bad(), должны иметь тип void и не содержать аргументов. Обычно они выполняют служебные задачи, такие как обновление файла мониторинга программы или переустановка переменных среды.

Обратите внимание, что функция sign_off() вызывается даже в случае, когда функция exit() не вызвана явно. Это объясняется тем, что exit() неявно вызывается после завершения функции main().

Использование функции exit()

Когда exit() выполнит функции, заданные с помощью atexit(), она предпринимает собственные шаги по наведению порядка. Функция очищает все потоки вывода, закрывает все открытые потоки, а также временные файлы, созданные путем вызова стандартной функции ввода-вывода tmpfile(). Затем функция exit() возвращает управление среде хоста и, если возможно, сообщает среде состояние завершения. Традиционно в программах для Unix используется значение 0 для указания успешного завершения и ненулевое значение для сообщения о сбое. Коды возврата в Unix применимы не ко всем системам, поэтому стандарт ANSI C определяет макрос EXIT_FAILURE, которым можно пользоваться в различных системах для указания ситуации сбоя. Аналогично, для индикации успешного завершения определен макрос EXIT_SUCCESS. Однако функция exit() также принимает в этом качестве значение 0. В соответствии со стандартом ANSI C использование функции exit() в нерекурсивной версии main() эквивалентно применению ключевого слова return. Однако exit() завершает программу и в тех случаях, когда применяется в функциях, отличных от main().

Функция qsort()

Метод "быстрой сортировки" является одним из наиболее эффективных алгоритмов сортировки, в особенности для крупных массивов. Он разработан К. А. Р. Хоаром (C. A. R. Hoare) в 1962 году и состоит в делении массивов на все более мелкие части вплоть до достижения уровня элементов. Сначала массив делится на две части. При этом каждое значение одной части меньше любого значения другой части. Упомянутый процесс продолжается до завершения полной сортировки массива.

В языке С алгоритм быстрой сортировки реализован функцией qsort(). Эта функция сортирует массив объектов данных. Она имеет следующий ANSI-прототип:

```
void qsort (void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
```

Первый аргумент представляет собой указатель на начало сортируемого массива. Стандарт ANSI C допускает приведение указателя на любые данные к типу указатель на void. Это позволяет первому фактическому аргументу функции qsort() ссылаться на массив элементов любого типа.

Второй аргумент представляет количество сортируемых элементов. Прототип преобразует это значение к типу size_t. Как уже несколько раз упоминалось, size_t является целочисленным типом данных, который возвращается операцией sizeof и определен в стандартных заголовочных файлах.

Поскольку функция qsort() преобразует свой первый аргумент в указатель типа void, она утрачивает информацию о размере каждого элемента массива. Для компенсации этого недостатка следует явно указать функции qsort() размер объекта данных. Именно для этого служит третий аргумент. Например, если выполняется сортировка массива типа double, в качестве третьего аргумента используется выражение sizeof(double).

Наконец, qsort() для определения порядка сортировки требует передачи указателя на функцию. Функция сравнения должна принимать два аргумента: указатели на два сравниваемых элемента. Она возвращает положительное целое число, если первый элемент должен следовать за вторым, ноль, если элементы одинаковы, и отрицательное целочисленное значение, если второй элемент должен следовать за первым. Функция qsort() использует функцию сравнения, передавая ей значения указателей, которые вычисляет на основе другой предоставляемой информации.

Форма функции сравнения задается последним аргументом прототипа функции qsort():

```
int (*compar)(const void *, const void *)
```

В прототипе определено, что последний аргумент представляет собой указатель на функцию, которая возвращает значение типа int и принимает два аргумента. Каждый из них является указателем на тип const void. Эти два указателя ссылаются на сравниваемые элементы.

Листинг 16.15 и последующий анализ иллюстрируют метод определения функции сравнения и применения функции qsort(). Программа создает массив случайных значений с плавающей запятой, а затем сортирует его.

Листинг 16.15. Программа qsorter.c

```
/* qsorter.c -- использование быстрой сортировки для упорядочения групп
чисел */
#include <stdio.h>
#include <stdlib.h>

#define NUM 40
void fillarray(double ar[], int n);
void showarray(const double ar[], int n);
int mycomp(const void * p1, const void * p2);

int main(void)
{
  double vals[NUM];
```

```
  fillarray(vals, NUM);
  puts("Список случайных чисел:");
  showarray(vals, NUM);
  qsort(vals, NUM, sizeof(double), mycomp);
  puts("\nСортированный список:");
  showarray(vals, NUM);
  return 0;
}
void fillarray(double ar[], int n)
{
  int index;
  for( index = 0; index < n; index++)
    ar[index] = (double)rand()/((double) rand() + 0.1);
}
void showarray(const double ar[], int n)
{
  int index;
  for( index = 0; index < n; index++)
  {
    printf("%9.4f ", ar[index]);
    if (index % 6 == 5)
      putchar('\n');
  }
  if (index % 6 != 0)
    putchar('\n');
}
/* сортировка по возрастанию */
int mycomp(const void * p1, const void * p2)
{
  /* для доступа к значениям нужны указатели на тип double */
  const double * a1 = (const double *) p1;
  const double * a2 = (const double *) p2;
  if (*a1 < *a2)
    return -1;
  else if (*a1 == *a2)
    return 0;
  else
    return 1;
}
```

Ниже показан пример выполнения программы:

```
Список случайных чисел:
   0.0022    0.2390    1.2191    0.3910    1.1021    0.2027
   1.3836   20.2872    0.2508    0.8880    2.2180   25.5033
   0.0236    0.9308    0.9911    0.2507    1.2802    0.0939
   0.9760    1.7218    1.2055    1.0326    3.7892    1.9636
   4.1137    0.9241    0.9971    1.5582    0.8955   35.3843
   4.0580   12.0467    0.0096    1.0110    0.8506    1.1530
   2.3614    1.5876    0.4825    6.8751
```

```
Сортированный список:
   0.0022     0.0096    0.0236    0.0939    0.2027    0.2390
   0.2507     0.2508    0.3910    0.4825    0.8506    0.8880
   0.8955     0.9241    0.9308    0.9760    0.9911    0.9971
   1.0110     1.0326    1.1021    1.1530    1.2055    1.2191
   1.2802     1.3836    1.5582    1.5876    1.7218    1.9636
   2.2180     2.3614    3.7892    4.0580    4.1137    6.8751
  12.0467    20.2872   25.5033   35.3843
```

Рассмотрим два ключевых момента: использование функции `qsort()` и определение функции `mycomp()`.

Использование функции `qsort()`

Функция `qsort()` выполняет сортировку массива объектов данных. В соответствии со стандартом ANSI прототип функции имеет следующий вид:

```
void qsort (void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
```

Первый аргумент представляет указатель начала сортируемого массива. В данной программе используется фактический аргумент `vals`. Это имя массива элементов типа `double`, а, следовательно, — указатель на первый элемент массива. ANSI-прототип предусматривает для аргумента `vals` приведение к типу указателя на `void`. Дело в том, что стандарт ANSI C допускает для любого указателя приведение к типу указателя на `void`. Поэтому первый фактический аргумент функции `qsort()` может ссылаться на массив любого вида.

Второй аргумент представляет количество сортируемых элементов. В листинге 16.15 это `N` — количество элементов массива. Прототип преобразует это значение к типу `size_t`.

Третий аргумент представляет размер каждого элемента, в данном случае — `sizeof(double)`.

Последний аргумент, `mycomp`, представляет адрес функции, применяемой для сравнения элементов.

Определение функции `mycomp()`

Как уже говорилось, прототип `qsort()` задает форму функции сравнения:

```
int (*compar)(const void *, const void *)
```

Здесь указано, что последний аргумент является указателем на функцию, которая возвращает значение типа `int` и принимает два аргумента. Каждый из них представляет собой указатель на тип `const void`. Программа содержит прототип функции `mycomp()`, который соответствует следующему прототипу:

```
int mycomp(const void * p1, const void * p2);
```

Вспомните, что имя функции, когда используется в качестве аргумента, является указателем на нее. Поэтому функция `mycomp` соответствует прототипу `compar`.

Функция `qsort()` передает функции сравнения адреса двух элементов, которые необходимо сравнить. В нашей программе адреса двух сравниваемых значений типа `double` присваиваются переменным `p1` и `p2`. Обратите внимание, что первый аргумент функции `qsort()` ссылается на массив в целом, а два аргумента функции сравне-

ния ссылаются на два элемента этого массива. Здесь возникает проблема. Чтобы выполнить сравнение указываемых значений, необходимо разыменовать указатель. Поскольку значения имеют тип double, следует выполнить разыменование указателя на тип double. Однако функция qsort() требует наличия указателей на тип void. Выход из положения заключается в объявлении указателей соответствующего типа внутри функции с их последующей инициализацией значениями, переданными в качестве аргументов:

```
/*сортировка по возрастанию */
int mycomp(const void * p1, const void * p2)
{
  /* для доступа к значениям нужны указатели на тип double */
  const double * a1 = (const double *) p1;
  const double * a2 = (const double *) p2;

  if (*a1 < *a2)
    return -1;
  else if (*a1 == *a2)
    return 0;
  else
    return 1;
}
```

Короче говоря, qsort() и функция сравнения для обобщенности используют указатели на void. Вследствие этого функции qsort() необходимо явно указать размер каждого элемента массива, а внутри определения функции сравнения преобразовать аргументы-указатели в указатели на тип данных, которые функция должна обрабатывать.

Тип данных void * в C и C++

Указатели на void в языках C и C++ обрабатываются по-разному. Оба языка позволяют присвоить указатель на любой тип данных переменной типа void *. Например, в листинге 16.15 при вызове функции qsort() значение типа double * присваивается указателю на double *. Однако язык C++ требует приведения типов, когда указатель void * присваивается указателю другого типа. При этом в C подобного требования нет. Например, в функции mycomp() из листинга 16.15 для указателя p1 типа void * такое приведение типов реализовано:

```
const double * a1 = (const double *) p1;
```

В языке C такое приведение типов необязательно, тогда как в языке C++ необходимо. Поскольку приведение типов применимо в обоих языках, имеет смысл выполнять его всегда. Это упростит адаптацию программы для языка C++.

Рассмотрим еще один пример функции сравнения. Предположим, имеются следующие объявления:

```
struct names {
    char first[40];
    char last[40];
};
struct names staff[100];
```

Как должен выглядеть вызов функции qsort()? Следуя модели листинга 16.15, можно применить следующий вызов:

```
qsort(staff, 100, sizeof(struct names), comp);
```

Здесь comp — имя функции сравнения. На что должна быть похожа эта функция?

Предположим, требуется выполнить сортировку по фамилии, а затем по имени. Можно написать следующую функцию:

```
#include <string.h>
int comp(const void * p1, const void * p2) /* обязательная форма */
{
  /* получение подходящего типа указателя */
  const struct names *ps1 = (const struct names *) p1;
  const struct names *ps2 = (const struct names *) p2;
  int res;

  res = strcmp(ps1->last, ps2->last); /* сравнение фамилий */
  if (res != 0)
    return res;
  else /* фамилии одинаковы, выполнить сравнение имен */
    return strcmp(ps1->first, ps2->first);
}
```

Данная функция вызывает функцию strcmp() для выполнения сравнения. Допускаемые для нее возвращаемые значения удовлетворяют требованиям функции сравнения. Обратите внимание, что для использования операции -> необходим указатель на структуру.

Библиотека assert

Библиотека assert, поддерживаемая заголовочным файлом assert.h, невелика и рассчитана на помощь при отладке программы. Она содержит макрос с именем assert(). Макрос принимает в качестве аргумента целочисленное выражение. Если выражение вычисляется как ложное (ненулевое значение), макрос assert() выводит в стандартный поток ошибок (stderr) сообщение об ошибке и вызывает функцию abort(), которая завершает программу. (Прототип функции abort() содержится в заголовочном файле stdlib.h.) Смысл состоит в том, чтобы идентифицировать критические места программы, где должны выполняться (быть истинными) определенные условия, и с помощью макроса assert() завершать программу, если одно из указанных условий не является истинным. Обычно аргументом служит выражение отношения или логическое выражение. Когда макрос assert() завершает программу, сначала выводится сообщение о проверке условия, которое привело к сбою, имени файла, содержащего проверку, а также указывается номер строки.

В листинге 16.16 показан простой пример. В нем ставится условие, что значение z должно быть больше или равно нулю, прежде чем будет предпринята попытка извлечь из него квадратный корень. Кроме того, ошибочно выполняется операция вычитания вместо сложения, в результате чего переменная z может принять недопустимое значение.

Листинг 16.16. Программа `assert.c`

```
/* assert.c -- использование макроса assert() */
#include <stdio.h>
#include <math.h>
#include <assert.h>
int main()
{
  double x, y, z;
  puts("Введите пару чисел (0 0 для выхода): ");
  while (scanf("%lf%lf", &x, &y) == 2
              && (x != 0 || y != 0))
  {
    z = x * x - y * y; /* должен быть + */
    assert(z >= 0);
    printf("ответ: %f\n", sqrt(z));
    puts("Следующая пара чисел: ");
  }
  puts("Программа завершена.");
  return 0;
}
```

Пример выполнения программы:

```
Введите пару чисел (0 0 для выхода):
4 3
ответ: 2.645751
Следующая пара чисел:
5 3
ответ: 4.000000
Следующая пара чисел:
3 5
Assertion failed: z >= 0, file C:\assert.c, line 14
```

Текст последней строки зависит от компилятора. Сообщение программы может сбить с толку, поскольку оно уведомляет о сбое `z >= 0` а не о том, что `claim z >= 0`.

Можно было бы воспользоваться следующим оператором `if`:

```
if (z < 0)
{
    puts("z меньше 0");
    abort();
}
```

Однако применение макроса `assert()` дает несколько преимуществ. Он автоматически идентифицирует файл и номер строки, где произошла ошибка. Наконец, существует механизм включения и отключения макроса `assert()` без изменения кода. Если разработчик считает, что устранил ошибки программы, он может поместить следующее определение макроса

```
#define NDEBUG
```

до позиции включения файла `assert.h`, а затем повторно скомпилировать программу. В результате компилятор отключит в файле операторы `assert()`.

Если ошибка возникнет снова, можно удалить директиву #define (или закомментировать ее), а затем выполнить повторную компиляцию. Все операторы assert() будут снова активизированы.

Функции memcpy() и memmove() из библиотеки string.h

Присвоить один массив другому невозможно. Поэтому в таких случаях использовались циклы для поэлементного копирования содержимого массива. Единственное исключение состоит в том, что для копирования символьных массивов в этой книге применялись функции strcpy() и strncpy(). Функции memcpy() и memmove() предлагают почти такие же услуги для других видов массивов. Рассмотрим прототипы этих функций:

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

Обе функции копируют n байтов из области, на которую указывает аргумент s2, в область, указываемую аргументом s1, а также возвращают значение s1. Различие между функциями, как указывает ключевое слово restrict, заключаются в следующем. Функция memcpy() предполагает, что области памяти не накладываются друг на друга.Функция memmove() не делает такого предположения, поэтому копирование происходит так, как будто все байты перед копированием в область назначения помещаются во временный буфер. Что произойдет при использовании функции memcpy() в случае наложения областей памяти? Поведение функции в этом случае не определено. Это означает, что она может работать как правильно, так и неправильно. Компилятор не запрещает использование функции memcpy(), когда этого делать не следует. Поэтому программисту придется принять меры, чтобы исключить наложение областей памяти.

Поскольку функции предназначены для работы с любым типом данных, у них два аргумента-указателя имеют типы указателя на void. Язык C позволяет присваивать указателю типа void * указатель любого типа.

Оборотная сторона такой гибкости связана с тем, что функции не способны распознавать, какого типа данные копируются. Поэтому в них присутствует третий аргумент, задающий количество копируемых байтов. Обратите внимание, что для массива количество байтов обычно не совпадает с количеством элементов. Поэтому если копируется массив из десяти значений типа double, в качестве третьего аргумента должно применяться выражение 10*sizeof(double), а не 10.

Примеры использования этих двух функций можно найти в листинге 16.17.

Листинг 16.17. Программа mems.c

```
// mems.c -- использование функций memcpy() и memmove()
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define SIZE 10
void show_array(const int ar[], int n);
int main()
{
```

```
  int values[SIZE] = {1,2,3,4,5,6,7,8,9,10};
  int target[SIZE];
  double curious[SIZE / 2] = {1.0, 2.0, 3.0, 4.0, 5.0};

  puts("Использование функции memcpy():");
  puts("значения (исходные данные): ");
  show_array(values, SIZE);
  memcpy(target, values, SIZE * sizeof(int));

  puts("целевой массив (копия значений):");
  show_array(target, SIZE);

  puts("\nИспользование memmove() для перекрывающихся областей:");
  memmove(values + 2, values, 5 * sizeof(int));

  puts("значения элементов 0-5, скопированных в 2-7:");
  show_array(values, SIZE);

  puts("\nИспользование memcpy() для копирования double в int:");
  memcpy(target, curious, (SIZE / 2) * sizeof(double));

  puts("целевой массив -- 5 значений double в 10 позиций int:");
  show_array(target, SIZE);

  return 0;
}

void show_array(const int ar[], int n)
{
  int i;
  for (i = 0; i < n; i++)
    printf("%d ", ar[i]);
  putchar('\n');
}
```

Выходные данные программы выглядят следующим образом:

```
Использование функции memcpy:
значения (исходные данные):
1 2 3 4 5 6 7 8 9 10
целевой массив (копия значений):
1 2 3 4 5 6 7 8 9 10

Использование memmove() для перекрывающихся областей:
значения элементов 0-5, скопированных в 2-7:
1 2 1 2 3 4 5 8 9 10

Использование memcpy() для копирования double в int:
целевой массив -- 5 значений double в 10 позиций int:
0 1072693248 0 1073741824 0 1074266112 0 1074790400 0 1075052544
```

В результате последнего вызова функции `memcpy()` данные из массива элементов типа `double` копируются в массив типа `int`. Это показывает, что типы данных для функции `memcpy()` значения не имеют. Она просто копирует байты из одной области в другую. (Например, можно копировать данные из структуры в массив символов.) Кроме того, здесь не применяется преобразование данных. Если организовать цикл, где выполняется поэлементное присваивание значений, значения типа `double` будут преобразованы к типу `int`. В данном же случае байты копируются в том виде, как они

есть, а программа интерпретирует комбинации разрядов, как если бы они принадлежали типу int.

Переменные аргументы: файл stdarg.h

В этой главе уже обсуждались варьируемые макросы. Так называют макросы, которые могут принимать различное количество аргументов. Заголовочный файл stdarg.h предоставляет подобную возможность функциям. Однако его использование несколько сложнее. Потребуется выполнить следующие действия:

1. Создать прототип функции, используя троеточия.
2. В определении функции создать переменную типа va_list.
3. Воспользоваться макросом для инициализации переменной списком аргументов.
4. Воспользоваться макросом для доступа к списку аргументов.
5. Воспользоваться макросом для удаления списка.

Рассмотрим эти действия подробнее. Прототип подобной функции должен содержать список параметров, где хотя бы после одного параметра следует троеточие:

```
void f1(int n, ...);                // допустимо
int f2(const char * s, int k, ...); // допустимо
char f3(char c1, ..., char c2);     // недопустимо, троеточие не в конце
double f3(...);                     // недопустимо, параметры отсутствуют
```

Крайний параметр справа (предшествующий троеточию) играет особую роль. Для его обозначения в стандарте используется имя parmN. В приведенных выше примерах роль parmN играют параметр n (в первом случае) и параметр k (во втором). Фактический аргумент, передаваемый этому параметру, задает количество аргументов, представленных троеточием. Например, в соответствии с прототипом функцию f1() можно использовать следующим образом:

```
f1(2, 200, 400);         // 2 дополнительных аргумента
f1(4, 13, 117, 18, 23);  // 4 дополнительных аргумента
```

Далее, тип va_list, объявленный в заголовочном файле stdargs.h, представляет объект данных, используемый для хранения параметров, соответствующих разделу троеточия в списке. Начало определения варьируемой функции выглядит примерно так:

```
double sum(int lim,...)
{
  va_list ap;    // объявление объекта для хранения аргументов
```

В этом примере lim является параметром parmN и указывает количество аргументов в списке переменных-аргументов.

Затем функция будет использовать макрос va_start(), также определенный в файле stdargs.h, для копирования списка аргументов в переменную va_list. Макрос содержит два аргумента: переменную va_list и параметр parmN. В предыдущем примере переменная типа va_list названа ap, а параметру parmN присвоено имя lim. Поэтому вызов функции имеет следующий вид:

```
va_start(ap, lim); // инициализация ap списком аргументов
```

На следующем этапе осуществляется доступ к содержимому списка аргументов. Для этой цели используется другой макрос — va_arg(). Он принимает два аргумента: переменную типа va_list, а также имя типа.

При первом вызове макрос возвращает первый элемент списка, при втором вызове — следующий элемент списка и так далее. Например, если первый аргумент списка имеет тип double, а второй — тип int, можно написать следующий код:

```
double tic;
int toc;
...
tic = va_arg(ap, double); // извлечение первого аргумента
toc = va_arg(ap, int);    // извлечение второго аргумента
```

Будьте внимательны. Тип аргумента должен соответствовать спецификации. Если первый аргумент имеет значение 10.0, код извлечения аргумента tic будет выполнен правильно. Если же аргумент имеет значение 10, код может работать неверно; в отличие от обычных операций присваивания, здесь не происходит автоматического преобразования типа double в тип int.

Наконец, необходимо удалить ненужные данные с помощью макроса va_end(). Например, можно высвободить память, которая динамически выделена для хранения аргументов. Этот макрос принимает в качестве аргумента переменную va_list:

```
va_end(ap);    // очистка памяти
```

После этого переменная ap может оказаться недоступной, если не выполнить ее повторную инициализацию с помощью макроса va_start.

Поскольку макрос va_arg() не обеспечивает резервного копирования предыдущих аргументов, иногда целесообразно сохранить копию переменной типа va_list. Для этой цели в стандарте C99 предусмотрен новый макрос с именем va_copy(). Он принимает два аргумента типа va_list и копирует второй аргумент в первый:

```
va_list ap;
va_list apcopy;
double
double tic;
int toc;
...
va_start(ap, lim);         // инициализация ap списком аргументов
va_copy(apcopy, ap);       // apcopy - это копия ap
tic = va_arg(ap, double); // извлечение первого аргумента
toc = va_arg(ap, int);     // извлечение второго аргумента
```

На данном этапе можно по-прежнему извлекать два первых элемента из переменной apcopy, даже если они удалены из переменной ap.

В листинге 16.18 представлен краткий пример применения этих средств для создания функции, которая суммирует переменное количество аргументов. Здесь первый аргумент функции sum() задает количество суммируемых элементов.

Листинг 16.18. Программа varargs.c

```
//varargs.c -- использование переменного количества аргументов
#include <stdio.h>
#include <stdarg.h>
```

```
double sum(int, ...);
int main(void)
{
  double s,t;

  s = sum(3, 1.1, 2.5, 13.3);
  t = sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1);
  printf("возвращаемое значение для "
       "sum(3, 1.1, 2.5, 13.3):            %g\n", s);
  printf("возвращаемое значение для "
       "sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1): %g\n", t);
  return 0;
}

double sum(int lim,...)
{
  va_list ap; // объявление объекта хранения аргументов
  double tot = 0;
  int i;

  va_start(ap, lim); // инициализация ap списком аргументов
  for (i = 0; i < lim; i++)
    tot += va_arg(ap, double); // доступ к каждому аргументу списка
  va_end(ap);  // удаление списка

  return tot;
}
```

Выходные данные программы имеют следующий вид:

```
возвращаемое значение для sum(3, 1.1, 2.5, 13.3):                16.9
возвращаемое значение для sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1): 31.6
```

Если проверить вычисления, легко убедиться, что при первом вызове функция `sum()` выполнила сложение трех аргументов, а при втором вызове — шести аргументов.

В конечном счете, варьируемые функции сложнее в использовании по сравнению с варьируемыми макросами. Однако функции обладают более широкой областью применения.

Ключевые понятия

Стандарт C описывает не только сам язык C. Он определяет пакет, включающий язык C, препроцессор C и стандартную библиотеку C. Препроцессор позволяет выполнить определенную подготовку перед компиляцией, выбирая строки кода, подлежащие компиляции, а также задавая другие аспекты поведения компилятора. Библиотека C расширяет возможности языка и предлагает готовые решения многих задач программирования.

Резюме

Препроцессор C и библиотека C представляют собой два важных дополнения языка C. Препроцессор C, следуя специальным директивам, корректирует исходный код до его компиляции. Библиотека C предоставляет множество функций, призванных

помочь в решении задач, таких как ввод, вывод, операции с файлами, управление памятью, сортировка и поиск, математические вычисления и обработка строк. И это далеко не полный перечень. Раздел V справочника (приложение Б) содержит полный список функций библиотеки ANSI C.

Вопросы для самоконтроля

1. Ниже приводятся группы из одного или нескольких макросов, сопровождаемые строками исходного кода, в которых они используются. Каким будет результат выполнения кода в каждом случае? Является ли код допустимым? (Предполагается, что переменные были объявлены.)

   а.
   ```
   #define FPM 5280     /* футов в миле */
   dist = FPM * miles;
   ```

   б.
   ```
   #define FEET 4
   #define POD FEET + FEET
   plort  = FEET * POD;
   ```

   в.
   ```
   #define SIX = 6;
   nex = SIX;
   ```

   г.
   ```
   #define NEW(X) X + 5
   y = NEW(y);
   berg = NEW(berg) * lob;
   est = NEW(berg) / NEW(y);
   nilp = lob * NEW(-berg);
   ```

2. Исправьте определение в части г) вопроса 1, чтобы сделать код более надежным.
3. Создайте определение макроса-функции, который возвращает наименьшее из двух значений.
4. Создайте определение макроса `EVEN_GT(X,Y)`, который возвращает значение 1, если `X` число четное и больше `Y`.
5. Создайте определение макроса-функции, которая выводит на печать представления и значения двух целочисленных выражений. Например, он может выводить данные

   `3 + 4 = 7` и `4 * 12 = 48`

   если аргументами служат выражения `3 + 4` и `4 * 12`.
6. Напишите операторы `#define` для решения следующих задач:
   а. Создание именованной константы со значением `25`.
   б. Создание идентификатора `SPACE`, представляющего символ пробела.
   в. Создание макроса `PS()`, который печатает символ пробела.
   г. Создание макроса `BIG(X)`, который представляет операцию сложения `3` и `X`.
   д. Создание макроса `SUMSQ(X,Y)`, представляющего сумму квадратов `X` и `Y`.
7. Создайте определение макроса, который печатает имя, значение и адрес переменной типа `int` в следующем формате:

   `имя: fop;  значение: 23;  адрес: ff46016`

8. Предположим, что во время тестирования программы требуется пропустить выполнение некоторого блока кода. Как это сделать без удаления кода из файла?
9. Создайте фрагмент кода, который печатает текущую дату работы препроцессора, если макрос PR_DATE определен.
10. Какую ошибку содержит следующая программа?

```
#include <stdio.h>
int main(int argc, char argv[])
{
   printf("Квадратный корень из %f равен %f\n", argv[1],
           sqrt(argv[1]) );
}
```

11. Предположим, scores представляет собой массив 1000 значений типа int, которые требуют сортировки в порядке убывания. Воспользуйтесь функцией сортировки qsort() и функцией сравнения comp().
    a. Как правильно вызвать функцию qsort()?
    б. Какое определение подойдет для функции comp()?
12. Предположим, data1 представляет собой массив из 100 значений типа double, а data2 — массив из 300 значений типа double.
    a. Напишите вызов функции memcpy(), чтобы скопировать 100 первых элементов массива data2 в массив data1.
    б. Напишите вызов функции memcpy(), чтобы скопировать 100 последних элементов data2 в массив data1.

Упражнения по программированию

1. Начните разработку заголовочного файла определений препроцессора, которые необходимы программе.
2. Гармоническое среднее двух чисел получают вычислением среднего от инверсий этих чисел с последующим инвертированием результата. Воспользуйтесь директивой #define для определения макроса-функции, который выполняет эту операцию. Напишите простую программу для тестирования этого макроса.
3. В полярной системе координат вектор описывается модулем и углом с осью x в направлении против часовой стрелки. В прямоугольной системе координат тот же вектор описывается составляющими x и y, как показано на рис. 16.3. Напишите программу, которая считывает значения модуля и угла (в градусах) вектора, а затем отображает составляющие x и y. Воспользуйтесь следующими уравнениями:

   x = r cos A y = r sin A

   Для выполнения преобразования примените функцию, которая принимает структуру, содержащую полярные координаты, и возвращает структуру, содержащую прямоугольные координаты (можно воспользоваться указателями на эти структуры).

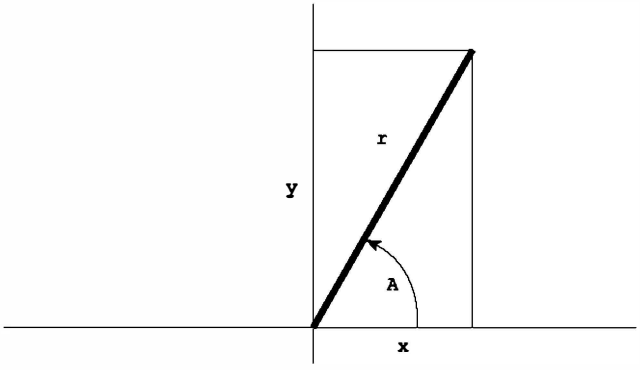


Рис. 16.3. *Прямоугольные и полярные координаты*

4. Библиотека ANSI содержит функцию `clock()` со следующим описанием:

   ```
   #include <time.h>
   clock_t clock (void);
   ```

   Здесь `clock_t` — тип данных, определенный в файле `time.h`. Функция возвращает процессорное время в единицах, которые зависят от реализации языка. (Если процессорное время недоступно или не может быть представлено, функция возвращает значение `-1`.) Однако в файле `time.h` также определена константа `CLOCKS_PER_SEC`, которая представляет количество единиц процессорного времени в секунде. Следовательно, в результате деления разницы между двумя возвращаемыми значениями функции `clock()` на константу `CLOCKS_PER_SEC` получается количество секунд, прошедшее между двумя вызовами функции. Приведение значений к типу `double` до операции деления позволит получать результат в долях секунды. Напишите функцию, которая принимает аргумент типа `double`, представляющий промежуток времени, а затем выполняет цикл до истечения указанного периода времени. Напишите простую программу для тестирования этой функции.

5. Напишите функцию, которая в качестве аргумента принимает имя массива элементов типа `int`, размер массива и значение, представляющее количество выборок. Функция должна случайным образом выбирать из массива указанное количество элементов и печатать их значения. Ни один элемент массива не должен выбираться более одного раза. (Это модель выбора чисел в лотерее или членов жюри.) Если в данной реализации доступна функция `time()` (которая обсуждалась в главе 12) или подобная ей функция, для вывода данных воспользуйтесь функцией `srand()`, чтобы инициализировать `rand()` — генератор случайных чисел. Напишите простую программу для тестирования этой функции.

6. Измените листинг 16.15 таким образом, чтобы программа использовала массив элементов `struct names` вместо массива элементов типа `double`. Задействуйте меньше элементов и явно инициализируйте массив подходящим набором имен.

7. Ниже приведен фрагмент программы, использующей варьируемую функцию:

   ```
   #include <stdio.h>
   #include <stdlib.h>
   ```

```
#include <stdarg.h>
void show_array(const double ar[], int n);
double * new_d_array(int n, ...);
int main()
{
  double * p1;
  double * p2;

  p1 = new_d_array(5, 1.2, 2.3, 3.4, 4.5, 5.6);
  p2 = new_d_array(4, 100.0, 20.00, 8.08, -1890.0);
  show_array(p1, 5);
  show_array(p2, 4);
  free(p1);
  free(p2);

  return 0;
}
```

Функция `new_d_array()` принимает аргумент типа `int` и переменное количество аргументов типа `double`. Она возвращает указатель на блок памяти, выделенный функцией `malloc()`. Аргумент типа `int` указывает количество элементов динамического массива, а значения типа `double` служат для инициализации элементов. При этом первое значение присваивается первому элементу и так далее. Завершите код программы, написав функции `show_array()` и `new_d_array()`.

ГЛАВА 17

Расширенное представление данных

В этой главе:

- Функции: дополнительные сведения о функции `malloc()`
- Использование C для представления различных типов данных
- Новые алгоритмы и увеличение возможностей концептуальной разработки программ
- Абстрактные типы данных

Изучение языка программирования подобно обучению музыке, плотницкому делу или инженерному искусству. Вначале вы знакомитесь с инструментами и средствами измерений, учитесь держать в руках молоток и избегать ударов по пальцам, а также решать бесчисленные проблемы, связанные с падением, соскальзыванием и утерей равновесия различных объектов. До сих пор в процессе прочтения этой книги читатели приобретали теоретические и практические навыки в создании переменных, структур, функций и тому подобного. Однако со временем происходит переход на более высокий уровень, на котором навыки использования инструментов превращаются во "вторую натуру", а реальной задачей становится разработка и создание проекта. Постепенно вырабатывается способность восприятия проекта как единого целого. Данная глава как раз и посвящена этому более высокому уровню работы. Изложенный в ней материал может показаться более сложным для восприятия, нежели материал, изложенный в предшествующих главах, однако его усвоение может оказаться и более плодотворным, поскольку позволяет ученику стать мастером.

Мы начнем с ознакомления с чрезвычайно важным аспектом проектирования программы: способа представления данных. Зачастую наиболее важным аспектом разработки программы является выбор подходящего представления данных, которыми будет манипулировать данная программа. Правильный выбор представления данных может превратить написание остальной программы в очень простую задачу. Вы уже знакомы с встроенными типами данных C: простыми переменными, массивами, указателями, структурами и объединениями.

Тем не менее, часто выбор правильного представления данных не ограничивается простым выбором типа. Необходимо также подумать о том, какие операции придется

выполнять. То есть потребуется выбрать способ хранения данных и определить, какие операции допустимы для того или иного типа данных. Например, как правило, в реализациях C оба типа: и `int`, и указатель хранятся в виде целочисленных значений, но для каждого из них определен свой набор допустимых операций. Например, одно целочисленное значение можно умножить на другое, но нельзя умножать указатель на указатель. Операцию `*` можно использовать для разыменования указателя, но она совершенно бессмысленна для целочисленного значения. Язык C определяет допустимые операции с основными типами. Однако при разработке схемы представления данных может требоваться определение допустимых операций вручную. В среде C это можно делать путем разработки функций, представляющих требуемые операции. Короче говоря, разработка типа данных состоит из определения способа хранения данных и разработки функций управления данными.

В процессе разработки программы потребуются также определенные *алгоритмы* — наборы команд для манипулирования данными. Каждый программист располагает определенным репертуаром таких наборов, которые он снова и снова применяет для решения схожих проблем.

В этой главе рассматривается процесс разработки типов данных — процесс сопоставления алгоритмов с представлениями данных. Здесь вы столкнетесь с рядом распространенных форм данных, таких как очередь, список и дерево бинарного поиска.

В главе будет также представлена концепция абстрактного типа данных (abstract data type — ADT). Тип ADT позволяет объединять методы и представления данных проблемно-ориентированным, а не языково-ориентированным образом. После того как ADT разработан, его можно многократно использовать в различных ситуациях. Понимание абстрактных типов данных концептуально подготовит вас к вступлению в мир объектно-ориентированного программирования и языка C++.

Исследование темы представления данных

Приступим к проектированию данных. Предположим, что требуется создать программу адресной книги. Какую форму данных необходимо использовать для хранения информации? Поскольку с каждой записью связана разнообразная информация, каждую запись имеет смысл представить структурой. А как представить несколько записей? С помощью стандартного массива структур? С помощью динамического массива? Посредством какой-либо иной формы? Должны ли записи быть упорядочены в алфавитном порядке? Требуется ли возможность выполнять поиск в записях по почтовому индексу? Или по междугородному телефонному коду? Действия, которые требуется выполнять, могут влиять на выбор способа хранения информации. Короче говоря, прежде чем приступать к созданию кода, придется принять массу проектных решений.

А как будет выполняться представление растровых графических изображений, которые должны храниться в памяти? В растровом изображении установка параметров каждого пикселя на экране выполняется индивидуально. Во времена черно-белых экранов для представления одного пикселя можно было использовать один бит (1 или 0) — отсюда и английское название растровых графических изображений *bitmapped* (побитовое представление). На цветных мониторах для описания одного пикселя требуется более одного бита. Например, выделение по 8 бит каждому пикселю позволяет получить 256 цветов. В настоящее время произошел переход к 65 536 цветам (16 бит на пиксель), 16 777 216 цветам (24 бита на пиксель), 2 147 483 648 (32 бита на пиксель) и

более. При наличии 16 миллионов цветов и разрешении экрана равном 1024×768 для представления единственного экрана растрового графического изображения потребуется 18,9 миллионов бит (2,25 Мбайт). Следует ли смириться с этим, или же разработать какой-либо метод сжатия информации? Должно ли это сжатие выполняться *без потерь* или *с потерями* (сравнительно неважных данных)? И снова, прежде чем приступить к созданию кода, придется принять множество проектных решений.

Рассмотрим конкретный случай представления данных. Предположим, нужно написать программу, которая позволяет вводить список всех фильмов (включая видеокассеты и диски DVD), просмотренных в течение года. Для каждого фильма требуется регистрация различной информации, такой как его название, год выпуска, имена и фамилии режиссера и ведущих актеров, продолжительность и жанр (комедия, научная фантастика, мелодрама, боевик и т.п.), рейтинг и так далее. Это предполагает применение для списка массива структур. Для простоты ограничим структуру двумя элементами: названием фильма и собственной оценкой его рейтинга по 10-бальной шкале. Упрощенная реализация с применением этого подхода приведена в листинге 17.1.

Листинг 17.1. Программа `films1.c`

```
/* films1.c -- использование массива структур */
#include <stdio.h>
#define TSIZE 45 /* размер массива для хранения названия     */
#define FMAX 5   /* максимальное количество названий фильмов */

struct film {
   char title[TSIZE];
   int rating;
};
int main(void)
{
   struct film movies[FMAX];
   int i = 0;
   int j;
   puts("Введите название первого фильма:");
   while (i < FMAX && gets(movies[i].title) != NULL &&
         movies[i].title[0] != '\0')
   {
      puts("Введите свое значение рейтинга <0-10>:");
      scanf("%d", &movies[i++].rating);
      while(getchar() != '\n')
         continue;
      puts("Введите название следующего фильма (или пустую строку для
прекращения ввода):");
   }
   if (i == 0)
      printf("Данные не были введены. ");
   else
      printf ("Список фильмов:\n");
   for (j = 0; j < i; j++)
      printf("Фильм: %s Рейтинг: %d\n", movies[j].title,
         movies[j].rating);
   printf("Программа завершена.\n");
   return 0;
}
```

Программа создает массив структур, а затем заполняет его данными, введенными пользователем. Ввод продолжается до заполнения массива (проверка условия FMAX), достижения конца файла (проверка условия NULL) или до нажатия пользователем клавиши <Enter> в начале строки (проверка условия '\0').

Применение этого алгоритма сопряжено с рядом проблем. Во-первых, скорее всего, программа будет напрасно тратить большой объем памяти, поскольку названия большинства фильмов содержат меньше 40 символов, но, в то же время, названия некоторых фильмов могут быть весьма длинными, такими как "Скромное обаяние буржуазии" или "Вон Тон Тон, пес, который спас Голливуд". Во-вторых, многим пользователям ограничение, равное пяти фильмам в год, покажется слишком строгим. Конечно, значение этого ограничения можно увеличить, но каким оно должно быть? Кое-кто просматривает до 500 фильмов в год, поэтому значение FMAX можно было бы увеличить до 500, но для некоторых пользователей и этого может оказаться слишком мало, в то время как для других пользователей это приводило бы к напрасной трате огромного объема памяти. Кроме того, некоторые компиляторы по умолчанию ограничивают объем памяти, доступной для автоматического сохранения переменных класса, подобных movies, и столь большой массив мог бы превысить это значение. Эту ситуацию можно решить с помощью статического или внешнего массива, либо вынудив компилятор использовать стек большего размера. Однако это не решает реальную проблему.

В данном случае реальная проблема состоит в том, что представление данных определено слишком жестко. Во время компиляции приходится принимать решения, которые целесообразнее принимать во время выполнения. Это предполагает переход к представлению данных, которое использует динамическое резервирование памяти. Можно попробовать воспользоваться кодом, подобным показанному ниже:

```
#define TSIZE 45 /* размер массива для хранения названия */
struct film {
   char title[TSIZE];
   int rating;
};
...
   int n, i;
   struct film * movies; /* указатель на структуру */
   ...
   printf("Введите максимальное количество фильмов:\n");
   scanf("%d", &n);
   movies = (struct film *) malloc(n * sizeof(struct film));
```

В этом примере, как и в главе 12, указатель movies можно использовать так, как если бы он был именем массива:

```
while (i < FMAX && gets(movies[i].title) != NULL &&
   movies[i].title[0] != '\0')
```

Применение функции malloc() позволяет отложить определение количества элементов до момента выполнения программы, и ей не придется резервировать память для 500 элементов, если их требуется только 20. Но такой подход возлагает обязанность ввода правильного значения для количества записей на пользователя.

От массива к связному списку

В идеале было бы желательно иметь возможность неограниченного добавления данных (в крайнем случае, до тех пор, пока программа не исчерпает лимит доступной памяти), не указывая заранее количество записей, которые будут созданы, и не вынуждая программу резервировать огромные области памяти без реальной на то необходимости. Этой цели можно достигнуть, вызывая функцию `malloc()` после ввода каждой записи и резервируя для нее именно такой объем памяти, который требуется. Если пользователь вводит информацию о трех фильмах, программа вызывает функцию `malloc()` три раза. Если пользователь вводит информацию о 300 фильмах, программа вызывает функцию `malloc()` триста раз.

Эта, прекрасная на первый взгляд, идея порождает новую проблему. Чтобы лучше понять, о чем идет речь, сравните однократный вызов функции `malloc()` для резервирования памяти для 300 структур `film` с 300-кратным вызовом этой функции для резервирования памяти каждый раз только для одной структуры `film`. В первом случае память резервируется в виде одного непрерывного блока, и для отслеживания содержимого требуется только единственная переменная указателя на структуру (`film`), указывающая на первую структуру в блоке. Как видно из приведенного фрагмента кода, простая форма записи массива обеспечивает указателю доступ к каждой из структур в блоке. Проблема применения второго подхода состоит в том, что нет никакой гарантии, что последовательные обращения к функции `malloc()` приведут к резервированию соседних блоков памяти. Следовательно, область памяти, используемая для хранения структур, может оказаться и не непрерывной (рис. 17.1). Поэтому вместо одного указателя блока 300 структур придется хранить 300 указателей — по одному для каждой независимо зарезервированной структуры!

Одно из возможных решений, которое мы не будем использовать, предполагает создание большого массива указателей и последовательное присвоение значений указателей по мере резервирования новых структур:

```
#define TSIZE 45 /* размер массива для хранения названий     */
#define FMAX 500 /* максимальное количество названий фильмов */
struct film {
   char title[TSIZE];
   int rating;
};
...
   struct film * movies[FMAX]; /* массив указателей на структуры */
   int i;
   ...
   movies[i] = (struct film *) malloc (sizeof (struct film));
```

Этот подход позволяет сэкономить очень большой объем памяти, поскольку массив из 500 указателей занимает значительно меньше памяти, чем массив из 500 структур. Однако при этом часть памяти все же напрасно тратится на хранение неиспользуемых указателей, а ограничение в 500 структур продолжает действовать.

Существует более эффективный способ решения задачи. При каждом вызове функции `malloc()` для резервирования памяти под новую структуру одновременно можно резервировать память и для нового указателя. “Но”, — возразите вы — “тогда потребуется еще один указатель для отслеживания только что зарезервированного

указателя, а для его отслеживания потребуется еще один указатель, и так до бесконечности". Способ предотвращения этой потенциальной проблемы состоит в переопределении структуры так, чтобы каждая структура содержала указатель на *следующую* структуру. Тогда при каждом создании новой структуры ее адрес можно будет сохранять в предыдущей структуре. Короче говоря, структуру `film` потребуется переопределить следующим образом:

```
#define TSIZE 45 /* размер массива для хранения названий */
struct film {
   char title[TSIZE];
   int rating;
   struct film * next;
};
```

Действительно, структура не может содержать структуру того же типа, но она может содержать указатель на структуру такого же типа. Такое определение служит основой определения *связного списка* — списка, в котором каждый элемент содержит информацию о местонахождении следующего элемента.

Прежде чем приступать к рассмотрению кода связного списка в C, подробнее рассмотрим концепцию такого списка. Предположим, что в качестве названия фильма пользователь вводит `Modern Times`, а качестве значения рейтинга — `10`. В этом случае программа выделила бы память для структуры `film`, скопировала бы строку `Modern Times` в элемент `title`, и установила бы значение элемент а `rating` равным `10`.

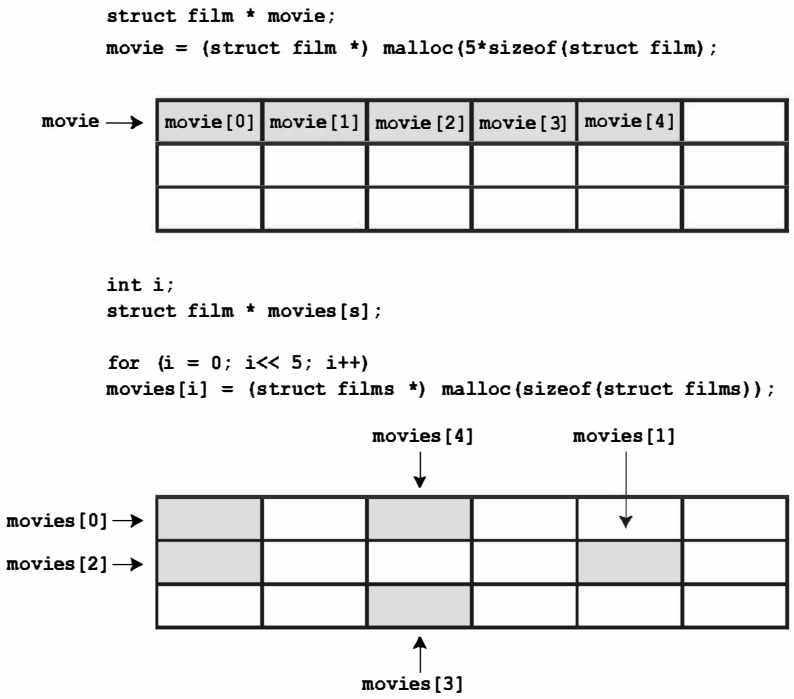


Рис. 17.1. *Сравнение резервирования структур в блоке с резервированием по отдельности*

Для указания того, что никакая структура не следует за данной, программа должна была бы установить значение элемента-указателя next равным NULL. (Вспомните, что NULL — символьная константа, определенная в файле stdio.h и представляющая нулевой указатель.). Конечно, при этом потребуется отслеживать место хранения первой структуры. Это можно делать, присваивая адрес отдельному указателю, который мы будем называть *указателем на заголовок*. Указатель на заголовок указывает на первый элемент в связном списке элементов. Вид этой структуры представлен на рис. 17.2. (Пустая область в элементе title сжата для уменьшения размера рисунка.)

```
#define TSIZE 45
struct film {
   char title[TSIZE]
   int rating;
   struct film * next;
};
struct film * head;
```

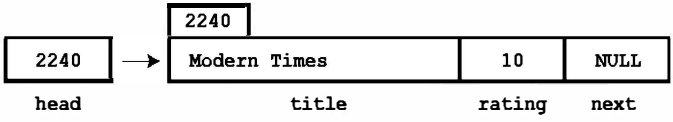


Рис. 17.2. *Первый элемент в связном списке*

Теперь предположим, что пользователь вводит название и рейтинг второго фильма — например, Titanic и 8. Программа выделяет память для второй структуры film, сохраняя адрес новой структуры в элементе next первой структуры (перезаписывая ранее сохраненное здесь значение NULL), чтобы этот указатель next указывал на следующую структуру в связном списке. Затем программа копирует значения Titanic и 8 в новую структуру и устанавливает значение ее элемента next равным NULL, указывая, что теперь эта структура является последней в списке. Этот список двух элементов показан на рис. 17.3.

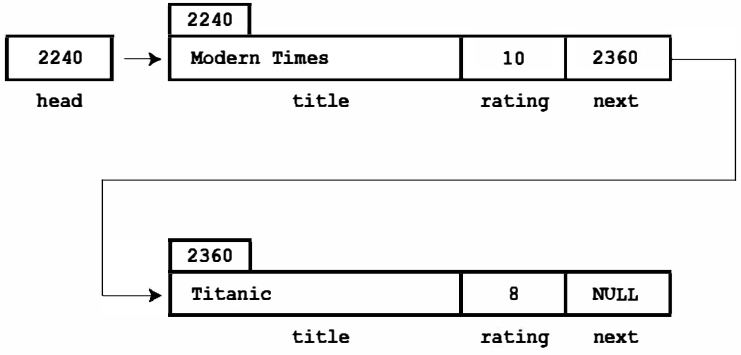


Рис. 17.3. *Связный список с двумя элементами*

Обработка каждой новой записи фильма будет выполняться аналогично. Ее адрес будет сохраняться в предыдущей структуре, новая информация будет помещаться в новую структуру, а значение элемента `next` новой структуры будет устанавливаться равным `NULL`, что приведет к созданию связного списка, подобного представленному на рис. 17.4.

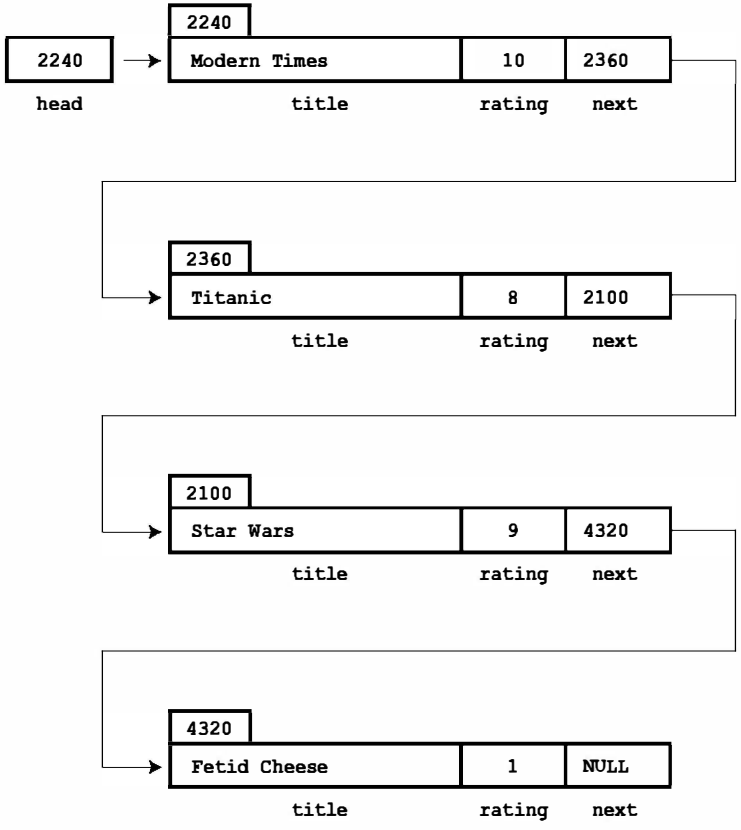


Рис. 17.4. *Связный список с несколькими элементами*

Предположим, что список необходимо отобразить. При каждом выводе элемента для обнаружения следующего отображаемого элемента можно использовать адрес, хранящийся в соответствующей структуре. Однако чтобы эта схема работала, указатель должен отслеживать первый элемент списка, поскольку ни одна структура в списке не хранит адрес первого элемента. К счастью, эта задача уже решена с помощью указателя на заголовок.

Использование связного списка

Теперь, когда вы получили представление о работе связного списка, реализуем его. В листинге 17.2 представлен измененный код листинга 17.1, в котором для хранения информации о фильмах вместо массива используется связный список.

Листинг 17.2. Программа `films2.c`

```
/* films2.c -- использование связного списка структур */
#include <stdio.h>
#include <stdlib.h> /* содержит прототип функции malloc()   */
#include <string.h> /* содержит прототип функции strcpy()   */
#define TSIZE 45    /* размер массива для хранения названия */

struct film {
   char title[TSIZE];
   int rating;
   struct film * next; /* указывает на следующую структуру в списке */
};

int main(void)
{
   struct film * head = NULL;
   struct film * prev, * current;
   char input[TSIZE];

/* Сбор и сохранение информации */
   puts("Введите название первого фильма:");
   while (gets(input) != NULL && input[0] != '\0')
   {
      current = (struct film *) malloc(sizeof(struct film));
      if (head == NULL) /* первая структура      */
         head = current;
      else              /* последующие структуры */
         prev->next = current;
      current->next = NULL;
      strcpy(current->title, input);
      puts("Введите свое значение рейтинга <0-10>:");
      scanf("%d", &current->rating);
      while(getchar() != '\n')
         continue;
      puts("Введите название следующего фильма (или пустую строку для
прекращения ввода информации):");
      prev = current;
   }
/* Отображение списка фильмов */
   if (head == NULL)
      printf("Данные не были введены. ");
   else
      printf ("Список фильмов:\n");
   current = head;
   while (current != NULL)
   {
      printf("Фильм: %s Рейтинг: %d\n",
         current->title, current->rating);
      current = current->next;
   }

/* Программа выполнена, поэтому можно освободить память */
   current = head;
```

```
   while (current != NULL)
   {
      free(current);
      current = current->next;
   }
   printf("Программа завершена.\n");
   return 0;
}
```

Программа использует связный список для решения двух задач. Во-первых, она конструирует список и заполняет его поступающими данными. Во-вторых, она отображает список. Отображение списка — более простая задача, поэтому вначале рассмотрим ее.

Отображение списка

Выполнение задачи начинается с определения указателя (назовем его `current`), указывающего на первую структуру. Поскольку указатель заголовка (назовем его `head`) уже указывает на эту структуру, достаточно воспользоваться следующим кодом:

```
current = head;
```

Теперь для обращения к элементам этой структуры можно применять форму записи через указатель:

```
printf("Фильм: %s Рейтинг: %d\n", current->title, current->rating);
```

Следующий шаг состоит в переопределении указателя `current`, чтобы он указывал на следующую структуру в списке. Эта информация хранится в элементе `next` структуры, поэтому для выполнения названной задачи необходим такой код:

```
current = current->next;
```

По завершении весь процесс нужно повторить. После отображения последнего элемента списка значение указателя `current` будет установлено равным `NULL`, поскольку именно это значение хранится в элементе `next` последней структуры. Этим обстоятельством можно воспользоваться для прекращения вывода. Фрагмент кода файла `films2.c`, используемый для отображения списка, выглядит следующим образом:

```
while (current != NULL)
{
   printf("Фильм: %s Рейтинг: %d\n", current->title, current->rating);
   current = current->next;
}
```

Почему бы для перемещения по списку вместо создания нового указателя (`current`) просто не использовать указатель `head`? В этом случае значение указателя `head` изменялось бы во время его применения, и программа лишалась бы средства отыскания начала списка.

Создание списка

Создание списка требует выполнения трех действий:

1. Использование функции `malloc()` для резервирования достаточного объема памяти для структуры.

2. Сохранение адреса структуры.
3. Копирование в структуру корректной информации.

Нет смысла создавать структуру, если она не требуется. Поэтому для получения информации о названии фильма, введенной пользователем, программа использует временную область хранения (массив input). Если пользователь эмулирует с помощью клавиатуры символ EOF или вводит пустую строку, цикл ввода завершается:

```
while (gets(input) != NULL && input[0] != '\0')
```

Адрес первой структуры должен быть сохранен в переменной указателя head. Адрес каждой последующей структуры должен сохраняться в элементе next предыдущей структуры. Поэтому программе требуется средство определения того, является ли текущая структура первой. Простейший способ решения этой задачи — инициализировать указатель head нулевым значением во время запуска программы. В этом случае программа может использовать значение указателя head для определения требуемых от нее действий:

```
if (head == NULL) /* первая структура      */
   head = current;
else              /* последующие структуры */
   prev->next = current;
```

В этом коде prev — указатель на структуру, выделенную в прошлый раз.

Теперь необходимо установить соответствующие значения элементов структуры. В частности, значение элемента next должно быть установлено равным NULL для указания того, что текущая структура является последней в списке. Необходимо скопировать название фильма из массива input в элемент title и получить значение элемента rating. Эти действия выполняет следующий код:

```
current->next = NULL;
strcpy(current->title, input);
puts("Введите свое значение рейтинга <0-10>:");
scanf("%d", &current->rating);
```

И, наконец, необходимо подготовить программу к следующему циклу ввода. В частности, значение указателя prev потребуется переопределить так, чтобы он указывал на текущую структуру, поскольку после ввода названия следующего фильма и распределения следующей структуры, текущая структура станет предыдущей. Программа устанавливает указатель на конец цикла:

```
prev = current;
```

Работает ли эта программа? Результат ее выполнения показан ниже.

```
Введите название первого фильма:
Без души
Введите свое значение рейтинга <0-10>:
8
Введите название следующего фильма (или пустую строку для прекращения ввода):
Дуэлянты
Введите свое значение рейтинга <0-10>:
7
Введите название следующего фильма (или пустую строку для прекращения ввода):
Морская пехота: Курган Гончей
```

```
Введите свое значение рейтинга <0-10>:
1
Введите название следующего фильма (или пустую строку для прекращения ввода):

Список фильмов:
Фильм: Без души  Рейтинг: 8
Фильм: Дуэлянты  Рейтинг: 7
Фильм: Морская пехота: Курган Гончей  Рейтинг: 1
Программа завершена.
```

Освобождение памяти, занимаемой списком

В случае завершения программа должна освободить память, выделенную функцией `malloc()`, при этом лучше, чтобы каждому вызову функции `malloc()` соответствовал вызов функции `free()`. В таком случае программа очищает память, применяя функцию `free()` к каждой выделенной структуре:

```
current = head;
while (current != NULL)
{
   free(current);
   current = current->next;
}
```

Дополнительные соображения

Программа `films2.c` несколько ограничена в своих возможностях. Например, она не проверяет, удалось ли функции `malloc()` найти запрошенную память, к тому же она лишена каких-либо средств удаления элементов из списка. Однако упомянутые недостатки легко устранить. Например, можно добавить код, который проверяет, является ли возвращаемое значение функции `malloc()` равным `NULL` (признак неудачи получения требуемой памяти). Если программа должна удалять записи, в нее можно поместить соответствующий код.

Такой специализированный подход к решению проблем и добавлению функциональных возможностей по мере необходимости — не всегда является наилучшим. С другой стороны, обычно невозможно предусмотреть все действия, которые должна выполнять программа. По мере увеличения масштабов проектов модель заблаговременного планирования всех необходимых функциональных возможностей становится все более нереальной. Как правило, наиболее успешными оказываются те большие программы, которые поэтапно расширялись на базе небольших удачных программ.

Учитывая, что планы могут изменяться, имеет смысл разрабатывать первоначальные проекты так, чтобы упростить возможные изменения. Пример, представленный в листинге 17.2, не соответствует этой концепции. В частности, в нем проявляется тенденция к смешиванию нюансов кодирования и концептуальной модели. Например, в этом коде концептуальная модель заключается в том, что элементы добавляются в список. Программа загромождает этот интерфейс, помещая на передний план такие детали, как функцию `malloc()` и указатель `current->next`. Было бы весьма желательно написать программу так, чтобы действия по добавлению каких-либо элементов в список были видны, а такие служебные действия, как вызов функций управления памятью и установка указателей, — наоборот, скрыты. Отделение интерфейса пользова-

теля от подробностей реализации упростит понимание и обновление программы. Упомянутых целей можно достичь, создав программу заново. Это можно выполнить следующим образом.

Абстрактные типы данных

В программировании принято использовать типы данных, соответствующие решению конкретной задачи. Например, для представления количества имеющихся пар обуви нужно было бы использовать тип `int`, а для представления средней цены одной пары — тип `float` или `double`. В приведенных примерах программ представления информации о фильмах данные образовывали список элементов, каждый из которых состоял из названия фильма (строки C) и значения рейтинга (типа `int`). Ни один из базовых типов C не соответствует этому описанию, поэтому для представления отдельных элементов мы определили структуру, а затем создали ряд методов для объединения последовательности структур в список. По сути, мы использовали предоставляемую языком C возможность разработки нового типа данных, соответствующего конкретным потребностям, но делали это бессистемно. Теперь мы применим более систематичный подход к определению типов.

Какие элементы образуют тип? *Тип* определяет два вида информации: набор свойств и набор операций. Например, свойство типа `int` состоит в том, что он представляет целочисленное значение и, следовательно, разделяет свойства целых значений. Для этого типа разрешены арифметические операции изменения знака, сложения, вычитания, и умножения двух значений типа `int`, деления одного значения типа `int` на другое, вычисления модуля одного значения типа `int` по другому значению. Объявление переменной типа `int` означает, что на нее могут воздействовать эти, и только эти, операции.

Свойства целочисленного типа

В основе типа `int` в C лежит более абстрактная концепция *целочисленного типа*. Математика позволяет определять свойства целочисленных значений формальным абстрактным образом. Например, если N и M – целочисленные значения, N+M=M+N, или для любых двух целых значений N и M существует целое значение S, для которого N+M=S. Если N+M=S и если N+Q=S, то M=Q. Можно считать, что математика предлагает абстрактную концепцию целочисленного значения, язык C предоставляет реализацию этой концепции. Например, язык C предоставляет средства хранения целочисленного значения и выполнения операций с целыми значениями, таких как сложение и умножение. Обратите внимание, что предоставление поддержки арифметических операций – важная часть представления целочисленных значений. Тип `int` был бы значительно менее полезным, если бы он позволял только хранить значения, но не использовать арифметические выражения. Следует отметить также, что реализация не обеспечивает идеального выполнения задачи представления целых значений. Например, множество целых значений бесконечно, но 2-байтовый тип `int` может представлять только 65536 из них. Не следует путать абстрактную идею с ее конкретной реализацией.

Предположим, что требуется определить новый тип данных. Во-первых, нужно обеспечить способ хранения данных — возможно, за счет проектирования структуры. Во-вторых, необходимо обеспечить способы манипулирования данными. В качестве примера рассмотрим программу `films2.c` (листинг 17.2).

Эта программа содержит связный набор структур для хранения информации, а также код добавления и отображения информации. Однако она выполняет эти задачи так, что создание при этом нового типа данных не очевидно. Как же следовало поступить?

Компьютерные науки предлагают очень эффективный способ определения новых типов данных. Он представляет собой трехэтапный процесс перехода от абстрактного к конкретному:

1. Приведите абстрактное определение свойств типа и операций, которые можно выполнять применительно к этому типу. Это описание не должно быть привязано ни к какой конкретной реализации. Более того, оно даже не должно быть привязано к конкретному языку программирования. Такое формально абстрактное описание называют *абстрактным типом данных* (ADT).
2. Разработайте программный интерфейс, реализующий этот тип ADT. То есть укажите, как следует хранить данные, и опишите набор функций, которые выполняют требуемые операции. Например, язык C позволяет приводить определение структуры вместе с прототипами функций манипулирования структурами. Эти функции играют для определенного пользователем типа ту же роль, которую встроенные операции исполняют для фундаментальных типов C. Любой, кто захочет воспользоваться новым типом, будет использовать этот интерфейс в своей программе.
3. Напишите код для реализации интерфейса. Конечно, этот шаг очень важен, но программисту, который использует новый тип, совершенно не обязательно знать подробности реализации.

Чтобы лучше разобраться в работе этого процесса, рассмотрим конкретный пример. Поскольку мы уже затратили определенные усилия на пример создания списка фильмов, переделаем его с применением нового подхода.

Получение абстракции

В основном, все, что требуется для проекта информации о фильмах — это список элементов. Каждый элемент содержит название фильма и значение рейтинга. Нам необходимо иметь возможность добавления новых элементов в конец списка и отображения его содержимого. Назовем абстрактный тип, который будет выполнять эти задачи, *списком*. Какими свойствами он должен обладать? Понятно, что список должен иметь возможность хранения последовательности элементов. То есть список может содержать несколько элементов, причем эти элементы каким-то образом упорядочены, что позволяет говорить о первом, втором или последнем элементе списка. Далее, тип списка должен поддерживать такие операции, как добавление элемента в список. Ниже перечислены некоторые полезные операции:

- Инициализация списка пустым значением.
- Добавление элемента в конец списка.
- Определение того, является ли список пустым.
- Определение того, является ли список полным.
- Определение количества элементов в списке.
- Посещение каждого элемента в списке для выполнения какого-либо действия, такого как его отображение.

Для этого проекта никакие дополнительные операции не требуются, однако более общий перечень операций со списками может включать следующие:

- Вставка элемента в любое место списка.
- Удаление элемента из списка.
- Извлечение элемента из списка (при этом список остается неизменным).
- Замена одного элемента в списке другим.
- Поиск элемента в списке.

Таким образом, неформальное, но абстрактное определение списка выглядит следующим образом: список — это объект данных, способный хранить последовательность элементов, к которому можно применять любые из перечисленных операций. Это определение не упоминает вид элементов, которые могут храниться в списке. Оно не указывает, нужно ли для хранения элементов использовать массив, связный набор структур или какую-либо иную форму данных. Оно не диктует и метод, который следует использовать, например, для выяснения количества элементов в списке. Все эти нюансы должны быть учтены реализацией.

Для простоты мы будем использовать упрощенный список абстрактного типа данных, содержащий только те функциональные возможности, которые требуются для проекта информации о фильмах.

Краткое описание типа приведено ниже:

Имя типа:	Простой список
Свойства типа:	Может содержать последовательность элементов.
Операции типа:	Инициализация списка пустым значением. Определение того, является ли список пустым. Определение того, является ли список полным. Определение количества элементов в списке. Добавление элемента в конец списка. Обход списка с обработкой каждого элемента. Опустошение списка.

Следующий шаг, который нужно выполнить — разработать интерфейс для ADT простого списка на языке C.

Построение интерфейса

Интерфейс для простого списка содержит две части. Первая часть описывает способ представления данных, а вторая — функции, реализующие операции ADT. Например, интерфейс будет содержать функции для добавления элемента в список и вывода количества элементов в списке. Структура интерфейса должна как можно больше соответствовать описанию ADT. Следовательно, она должна быть выражена в терминах некоего обобщенного типа `Item`, а не в терминах какого-то конкретного типа, подобного `int` или `struct film`.

Один из способов достижения этого предполагает применение средства `typedef` языка C для определения `Item` в качестве требуемого типа:

```
#define TSIZE 45 /* размер массива для хранения названия */
struct film
{
   char title[TSIZE];
   int rating;
};
typedef struct film Item;
```

Теперь тип `Item` можно использовать в остальных определениях. Если впоследствии потребуется список элементов какой-то другой формы данных, можно будет переопределить тип `Item` и оставить остальную часть определения интерфейса без изменений.

После того как тип `Item` определен, необходимо принять решение о способе хранения элементов этого типа. В действительности этот шаг относится к этапу реализации, но принятие решения на данном этапе облегчает понимание примера. Подход с использованием связанной структуры достаточно успешно работал в программе `films2.c`, поэтому применим его, как показано в следующем фрагменте кода:

```
typedef struct node
{
   Item item;
   struct node * next;
} Node;
typedef Node * List;
```

В реализации с применением связного списка каждая связь называется *узлом*. Каждый узел содержит информацию, формирующую содержимое списка, и указатель на следующий узел. Чтобы подчеркнуть используемую терминологию, мы назвали структуру узла пресловутым именем `node` (то есть "узел"), и использовали `typedef` для преобразования имени `Node` в имя типа структуры `struct node`. Наконец, чтобы связным списком можно было управлять, необходим указатель на его начало. Поэтому мы использовали `typedef` для преобразования имени `List` в имя указателя этого типа. Таким образом, объявление

```
List movies;
```

определяет `movies` как указатель, подходящий для ссылки на связный список.

Является ли этот способ определения типа `List` единственным? Нет. Например, для отслеживания количества записей можно было бы использовать переменную:

```
typedef struct list
{
   Node * head;     /* указатель на заголовок списка     */
   int size;        /* количество записей в списке       */
} List;             /* альтернативное определение списка */
```

Для отслеживания конца списка можно было бы добавить второй указатель. Позже читатели ознакомятся с соответствующим примером. А пока ограничимся первым определением типа `List`. Важно помнить, что объявление

```
List movies;
```

следует рассматривать как определение списка, а не указатель на узел или определение структуры. Конкретное представление данных списка `movies` — нюанс реализации, который должен быть невидим на уровне интерфейса.

Например, при запуске программа должна инициализировать указатель на заголовок значением `NULL`, но при этом не следует использовать код, подобный следующему:

```
movies = NULL;
```

Почему? Потому что впоследствии может оказаться, что реализация типа `List` в виде структуры подходит больше, а в этом случае потребуется следующая инициализация:

```
movies.next = NULL;
movies.size = 0;
```

Никто из тех, кто использует тип `List`, не должен беспокоиться о подобных нюансах. Вместо этого они должны быть в состоянии выполнять нечто наподобие следующего:

```
InitializeList(movies);
```

Программистам требуется знать только о том, что для инициализации списка им нужно использовать функцию `InitializeList()`. Им не требуется знание конкретной реализации данных переменной `List`. Все сказанное — пример *сокрытия данных* — искусства скрытия нюансов представления данных от более высоких уровней программирования.

Для облегчения задачи пользователя прототип функции можно сопровождать следующими строками:

```
/* операция:          инициализация списка                    */
/* начальные условия: plist указывает на список list          */
/* конечные условия:  список инициализирован пустым значением */
void InitializeList(List * plist);
```

Обратите внимание на три момента. Во-первых, комментарии описывают *начальные условия* — то есть условия, которые должны быть удовлетворены до вызова функции. Например, в данном случае необходимо указать список для инициализации. Во-вторых, комментарии описывают *конечные условия* — то есть условия, которые должны быть удовлетворены после выполнения функции. И, наконец, в качестве аргумента функции использован указатель на список, а не сам список. Поэтому вызов функции должен иметь следующий вид:

```
InitializeList(&movies);
```

Причина такого подхода состоит в том, что в языке C передача аргументов выполняется по значению. Поэтому единственный способ изменения значения в вызывающей программе функцией C — использование указателя на эту переменную. Как видите, в данном случае ограничения языка обуславливают некоторое отличие интерфейса от его абстрактного описания.

Метод связывания всей информации о типе и функциях в едином пакете, применяемый в языке C, состоит в помещении определений типа и прототипов функций (включая комментарии с описанием начальных и конечных условий) в файл заголовка. Этот файл должен предоставлять всю информацию, в которой нуждается программист для использования типа. Файл заголовка простого типа `list` показан в листинге 17.3. В нем конкретная структура определена как относящаяся к типу `Item`, а затем переменная `Node` определена в терминах этого типа, после чего термины типа

Node использованы для определения типа List. После этого типы Item и List могут применяться в качестве аргументов функций, представляющих операции списка. Если функции требуется изменить аргумент, она использует указатель на соответствующий тип, а не непосредственно сам тип. В файле имена функций начинаются с прописных букв для их обозначения как составной части пакета интерфейса. Кроме того, для защиты от множественных включений файла в нем применена методика #ifndef, рассмотренная в главе 16. Если используемый компилятор не поддерживает тип bool стандарта C99, в файле заголовка строку

```
#include <stdbool.h> /* функциональная возможность C99 */
```

можно заменить следующей строкой:

```
enum bool {false, true}; /* определение bool в качестве типа,
                            а false и true - в качестве значений */
```

Листинг 17.3. Файл заголовка интерфейса list.h

```
/* list.h -- файл заголовка для простого типа list                  */
#ifndef LIST_H_
#define LIST_H_
#include <stdbool.h> /* функциональная возможность C99              */
/* объявления, характерные для программы */
#define TSIZE 45 /* размер массива для хранения названия            */
struct film
{
   char title[TSIZE];
   int rating;
};
/* определения обобщенного типа                                     */
typedef struct film Item;
typedef struct node
{
   Item item;
   struct node * next;
} Node;
typedef Node * List;
/* прототипы функций                                                */
/* операция:          инициализация списка                          */
/* начальные условия: plist указывает на список */
/* конечные условия:  список инициализирован пустым значением       */
void InitializeList(List * plist);
/* операция:         определение того, является ли список пустым  */
/*                   plist указывает на инициализированный список */
/* конечные условия: функция возвращает значение True, если список */
/*                   пуст, и False - в противном случае            */
bool ListIsEmpty(const List *plist);
/* операция:         определение того, является ли список полным   */
/*                   plist указывает на инициализированный список  */
```

```
/* конечные условия: функция возвращает значение True, если список */
/*                   полон, и False - в противном случае           */
bool ListIsFull(const List *plist);

/* операция:         определение количества элементов в списке     */
/*                   plist указывает на инициализированный список  */
/* конечные условия: функция возвращает число элементов в списке   */
unsigned int ListItemCount(const List *plist);

/* операция:          добавление элемента в конец списка           */
/* начальные условия: item - элемент, добавляемый в список         */
/*                    plist указывает на инициализированный список */
/* конечные условия:  если возможно, функция добавляет элемент в   */
/*                    конец списка и возвращает значение True;     */
/*                    в противном случае она возвращает значение   */
/*                    False                                        */
bool AddItem(Item item, List * plist);

/* операция:          применение функции к каждому элементу списка */
/*                    plist указывает на инициализированный список */
/*                    pfun указывает на функцию, которая принимает */
/*                    аргумент Item и не имеет возвращаемого       */
/*                    значения                                     */
/* конечное условие:  функция, указанная pfun, выполняется один    */
/*                    раз для каждого элемента в списке            */
void Traverse (const List *plist, void (* pfun)(Item item) );

/* операция:          освобождение зарезервированной памяти, если  */
/*                    таковая существует                           */
/*                    plist указывает на инициализированный список */
/* конечные условия:  любая память, зарезервированная для списка,  */
/*                    освобождается, и список устанавливается в    */
/*                    пустое состояние                             */
void EmptyTheList(List * plist);

#endif
```

Только функции `InitializeList()`, `AddItem()` и `EmptyTheList` изменяют список, поэтому с технической точки зрения только они требуют использования аргумента-указателя. Однако если бы пользователю пришлось помнить о необходимости передачи аргумента `List` одним функциям и его адреса другим, это могло бы приводить к недоразумениям. Поэтому для упрощения задачи пользователя все функции используют аргументы-указатели.

Один из прототипов в файле заголовка несколько сложнее остальных:

```
/* операция:          применение функции к каждому элементу списка */
/*                    plist указывает на инициализированный список */
/*                    pfun указывает на функцию, которая принимает */
/*                    аргумент Item и не имеет возвращаемого       */
/*                    значения                                     */
/* конечное условие:  функция, указанная pfun, выполняется один    */
/*                    раз для каждого элемента в списке            */
void Traverse (const List *plist, void (* pfun)(Item item) );
```

Аргумент pfun представляет собой указатель на функцию. В данном конкретном случае он является указателем на функцию, которая принимает элемент в качестве аргумента и не имеет возвращаемого значения. Как вы, возможно, помните из главы 14, указатель на функцию можно передавать в качестве аргумента другой функции, и в этом случае вторая функция может использовать указанную функцию. Так, например, pfun может указывать на функцию, которая отображает элемент. Функция Traverse() будет применять эту функцию к каждому элементу списка, в результате отображая весь список.

Использование интерфейса

Мы берем на себя смелость утверждать, что этот интерфейс можно использовать для написания программы, не располагая никакими дополнительными сведениями — например, ничего не зная о том, как написаны функции. Поэтому мы сразу напишем новую версию программы вывода информации о фильмах, еще до написания поддерживающих функций. Поскольку интерфейс определен в терминах типов List и Item, программа должна быть создана с применением этих же типов. Представление одного из возможных планов в виде псевдокода приведено ниже:

```
Создать переменную List.
Создать переменную Item.
Инициализировать список пустым значением.
Пока список не заполнен, и пока существуют данные для ввода:
   Прочитать ввод в переменную Item.
   Добавить элемент в конец списка.
Посетить каждый элемент списка и отобразить его.
```

Программа, приведенная в листинге 17.4, соответствует этому общему плану, но в нее добавлена определенная проверка наличия ошибок. Обратите внимание на использование в ней интерфейса, описанного в файле list.h (листинг 17.3). Обратите также внимание, что листинг содержит код функции showmovies(), которая соответствует прототипу, требуемому функцией Traverse(). Поэтому программа может передавать указатель showmovies функции Traverse(), чтобы та могла применять функцию showmovies() к каждому элементу списка. (Вспомните, что имя функции является также указателем на эту функцию.)

Листинг 17.4. Программа `films3.c`

```
/* films3.c -- использование связного списка в стиле ADT */
/* компилировать вместе с list.c */
#include <stdio.h>
#include <stdlib.h> /* прототип функции exit()      */
#include "list.h"   /* определение типов List, Item */
void showmovies(Item item);

int main(void)
{
   List movies;
   Item temp;

/* инициализация */
   InitializeList(&movies);
   if (ListIsFull(&movies))
```

```
   {
      fprintf(stderr,"Доступная память отсутствует! Программа завершена.\n");
      exit(1);
   }
/* сбор и сохранение информации */
   puts("Введите название первого фильма:");
   while (gets(temp.title) != NULL && temp.title[0] != '\0')
   {
      puts("Введите свое значение рейтинга <0-10>:");
      scanf("%d", &temp.rating);
      while(getchar() != '\n')
         continue;
      if (AddItem(temp, &movies)==false)
      {
         fprintf(stderr,"Проблема с резервированием памяти\n");
         break;
      }
      if (ListIsFull(&movies))
      {
         puts("Список полон.");
         break;
      }
      puts("Введите название следующего фильма (или пустую строку для
прекращения ввода):");
   }
/* отображение */
   if (ListIsEmpty(&movies))
      printf("Данные не были введены. ");
   else
   {
      printf ("Список фильмов:\n");
      Traverse(&movies, showmovies);
   }
   printf("Вы ввели %d фильмов.\n", ListItemCount(&movies));
/* очистка */
   EmptyTheList(&movies);
   printf("Программа завершена.\n");
   return 0;
}
void showmovies(Item item)
{
   printf("Фильм: %s Рейтинг: %d\n", item.title,
      item.rating);
}
```

Реализация интерфейса

Конечно, интерфейс List все еще предстоит реализовать. В C принят подход, при котором определения функций собираются в одном файле — в данном случае в list.c. Тогда полная программа будет состоять из трех файлов: list.h, который определяет структуры данных и предоставляет прототипы для интерфейса пользователя, list.c, который предоставляет код функций реализации интерфейса, и films3.c, являющийся файлом исходного кода, который применяет интерфейс списка для решения конкретной программной задачи. Одна из возможных реализаций файла list.c показана в листинге 17.5. Чтобы выполнить программу, необходимо скомпилировать оба файла films3.c и list.c и скомпоновать их. (Компиляция многофайловых программ рассматривается в главе 9). Вместе файлы list.c, list.c и films3.c составляют полную программу (рис. 17.5).

list.h

```
/* list.h--header file for a simple list type */
/* program-specific declarations */
#define TSIZE 45 /* size of array to hold title */
struct film
{
  char title[TSIZE];
  int rating;
};
.
.
.
void Traverse (List l, void (* pfun)(Item item) );
```

list.c

```
/* list.c--functions supporting list operations */
#include<stdio.h>
#include<stdlib.h>
#include "list.h"

.
.
.
/* copies an item into node */
static void CopyToNode (Item item, Node * pnode)
{
pnode->item = item; /* structure copy */
}
```

films3.c

```
/* films3.c -- using and ADT-style linked list */
#include <stdio.h>
#include <stdlib.h> /* prototype for exit() */
#include "list.h"
void showmovies(Item item);

int main(void)
{
.
.
.
}
```

Рис. 17.5. *Три части программного пакета*

Листинг 17.5. Файл реализации `list.c`

```
/* list.c -- функции, поддерживающие операции со списком */
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

/* прототип локальной функции */
static void CopyToNode(Item item, Node * pnode);

/* функции интерфейса */
/* устанавливает список в пустое состояние */
void InitializeList(List * plist)
{
   * plist = NULL;
}

/* возвращает значение true, если список пуст */
bool ListIsEmpty(const List * plist)
{
   if (*plist == NULL)
      return true;
   else
      return false;
}

/* возвращает значение true, если список полон */
bool ListIsFull(const List * plist)
{
   Node * pt;
   bool full;

   pt = (Node *) malloc(sizeof(Node));
   if (pt == NULL)
      full = true;
   else
      full = false;
   free(pt);

   return full;
}

/* возвращает количество узлов */
unsigned int ListItemCount(const List * plist)
{
   unsigned int count = 0;
   Node * pnode = *plist; /* установка на начало списка */

   while (pnode != NULL)
   {
      ++count;
      pnode = pnode->next; /* установка на следующий узел */
   }
   return count;
}
```

```
/* создает узел для хранения элемента и добавляет его в конец */
/* списка, указанного переменной plist (медленная реализация) */
bool AddItem(Item item, List * plist)
{
   Node * pnew;
   Node * scan = *plist;

   pnew = (Node *) malloc(sizeof(Node));
   if (pnew == NULL)
   return false; /* выход из функции в случае ошибки */

   CopyToNode(item, pnew);
   pnew->next = NULL;
   if (scan == NULL) /* список пуст, поэтому помещает */
      *plist = pnew; /* pnew в начало списка */
   else
   {
      while (scan->next != NULL)
         scan = scan->next; /* ищет конец списка */
      scan->next = pnew; /* добавляет pnew в конец */
   }

   return true;
}

/* посещает каждый узел и выполняет функцию, указанную pfun */
void Traverse (const List * plist, void (* pfun)(Item item) )
{
   Node * pnode = *plist; /* устанавливает указатель на начало списка */

   while (pnode != NULL)
   {
      (*pfun)(pnode->item); /* применяет функцию к элементу */
      pnode = pnode->next; /* переход к следующему элементу */
   }
}

/* освобождает память, зарезервированную функцией malloc() */
/* устанавливает указатель списка в значение NULL */
void EmptyTheList(List * plist)
{
   Node * psave;

   while (*plist != NULL)
   {
      psave = (*plist)->next; /* сохранение адреса следующего узла */
      free(*plist);           /* освобождение текущего узла        */
      *plist = psave;         /* переход к следующему узлу         */
   }
}

/* определение локальной функции */
/* копирует элемент в узел */
static void CopyToNode(Item item, Node * pnode)
{
   pnode->item = item; /* копирование структуры */
}
```

Замечания по поводу программы

Файл list.c имеет много интересных особенностей. Во-первых, он иллюстрирует ситуацию, в которой можно использовать функции с внутренним связыванием. Как описано в главе 12, функции с внутренним связыванием известны только в том файле, котором они определены. При реализации интерфейса иногда удобно использовать вспомогательные функции, которые не являются частью официального интерфейса. Например, в приведенной программе функция CopyToNode() использована для копирования значения типа Item в переменную типа Item. Поскольку эта функция — часть реализации, но не интерфейса, с помощью спецификатора класса памяти static мы скрыли ее в файле list.c. А теперь проанализируем остальные функции.

Функция InitializeList() инициализирует список пустым значением. В нашей реализации это означает установку переменной типа List в значение NULL. Как уже было сказано ранее, это требует передачи функции указателя на переменную List.

Функция ListEmpty() достаточно проста, но она требует установки переменной списка в значение NULL в случае пустого списка. Поэтому важно инициализировать список до первого вызова функции ListEmpty().

Кроме того, если потребуется расширить интерфейс, включив в него удаление элементов, придется убедиться, что функция удаления переустанавливает список в пустое состояние после удаления последнего элемента. При использовании связного списка его размер ограничен объемом доступной памяти. Функция ListIsFull() пытается зарезервировать достаточный объем памяти для нового элемента. Если это ей не удается, это означает, что список полон. В случае успеха функция должна освободить только что зарезервированную ею память, чтобы она была доступна для реального элемента.

Функция ListItemCount() использует алгоритм обхода связного списка, подсчитывая при этом количество элементов:

```
unsigned int ListItemCount(const List * plist)
{
   unsigned int count = 0;
   Node * pnode = *plist; /* установка на начало списка */

   while (pnode != NULL)
   {
      ++count;
      pnode = pnode->next; /* установка на следующий узел */
   }
   return count;
}
```

Функция AddItem() наиболее сложная из всех:

```
bool AddItem(Item item, List * plist)
{
   Node * pnew;
   Node * scan = *plist;

   pnew = (Node *) malloc(sizeof(Node));
   if (pnew == NULL)
   return false; /* выход из функции в случае ошибки */

   CopyToNode(item, pnew);
```

```
    pnew->next = NULL;
    if (scan == NULL) /* список пуст, поэтому помещает */
        *plist = pnew; /* pnew в начало списка */
    else
    {
        while (scan->next != NULL)
            scan = scan->next; /* ищет конец списка */
        scan->next = pnew; /* добавляет pnew в конец */
    }
    return true;
}
```

Прежде всего, функция `AddItem()` резервирует память для нового узла. Если это ей удается, она использует функцию `CopyToNode()` для копирования элемента в узел. Затем она устанавливает элемент `next` узла в значение `NULL`. Это, как вы помните, служит сигналом того, что данный узел является последним в связном списке. И, наконец, после создания узла и присвоения соответствующих значений его элементам функция присоединяет узел к концу списка. Если элемент — первый добавленный элемент списка, программа устанавливает указатель заголовка на первый элемент. (Вспомните, что функция `AddItem()` вызывается с адресом указателя заголовка в качестве ее второго аргумента, поэтому `* plist` — значение указателя заголовка.) В противном случае код выполняет проход по связному списку до тех пор, пока не обнаружит элемент, элемент `next` которого установлен в значение `NULL`. На текущий момент времени этот узел является последним в списке, поэтому функция переустанавливает его элемент `next`, чтобы он указывал на новый узел.

Принятая практика программирования требует вызова функции `ListIsFull()` перед попыткой добавления элемента в список. Однако пользователь может упустить этот момент, поэтому функция `AddItem()` сама проверяет успешность выполнения функции `malloc()`. Кроме того, существует вероятность, что между вызовами функций `ListIsFull()` и `AddItem()` пользователь может выполнить еще какие-либо действия по резервированию памяти. Поэтому лучше на всякий случай проверить, была ли выполнена функция `malloc()`.

Функция `Traverse()` аналогична функции `ListItemCount()`, но в нее добавлено применение функции к каждому элементу списка:

```
void Traverse (const List * plist, void (* pfun)(Item item) )
{
    Node * pnode = *plist; /* устанавливает указатель на начало списка */
    while (pnode != NULL)
    {
        (*pfun)(pnode->item); /* применяет функцию к элементу */
        pnode = pnode->next; /* переход к следующему элементу */
    }
}
```

Вспомните, что `pnode->item` представляет данные, сохраненные в узле, а `pnode->next` определяет следующий узел в связном списке. Например, вызов функции

```
Traverse(movies, showmovies);
```

применяет функцию `showmovies()` к каждому элементу в списке.

И, наконец, функция EmptyTheList() освобождает память, которая ранее была зарезервирована с помощью malloc():

```
void EmptyTheList(List * plist)
{
   Node * psave;

   while (*plist != NULL)
   {
      psave = (*plist)->next; /* сохранение адреса следующего узла */
      free(*plist);           /* освобождение текущего узла */
      *plist = psave;         /* переход к следующему узлу */
   }
}
```

Эта реализация указывает пустой список путем установки переменной List в значение NULL. Поэтому, чтобы она могла изменять эту переменную, функции должен быть передан адрес переменной List. Поскольку переменная List уже является указателем, plist — это указатель на указатель. Таким образом, внутри кода выражение *plist является выражением типа указателя на Node. При выходе из списка значение *plist равно NULL — то есть теперь исходный актуальный аргумент установлен в значение NULL.

Код сохраняет адрес следующего узла, поскольку в принципе вызов функции free() может сделать содержимое текущего узла (указанного указателем *plist) недоступным.

Ограничения применения спецификатора

Несколько функций обработки списка содержат в качестве параметра выражение List * plist. Оно служит указанием того, что эти функции не изменяют список. В данном случае спецификатор const обеспечивает определенную защиту. Он предотвращает изменение указателя *plist (значения, на которое указывает plist). В этой программе plist указывает на переменную movies, поэтому спецификатор const предотвращает изменение этими функциями переменной movies, которая, в свою очередь, указывает на первую связь в списке. Таким образом, код, подобный показанному ниже, недопустим, например, в функции ListItemCount():

```
*plist = (*plist)->next; // не допускается, если *plist - константа
```

Это хорошо, поскольку изменение значения *plist, а, следовательно, и значения movies, привело бы невозможности отслеживания данных программой. Однако то, что переменные *plist и movies обрабатываются как значения типа const, не означает, что данные, указанные *plist или movies, также являются константами. Например, следующий код вполне допустим:

```
(*plist)->item.rating = 3; // допустимо, даже если *plist - константа
```

Это объясняется тем, что данный код не изменяет переменную *plist. Он изменяет данные, на которые указывает *plist. Вывод из всего сказанного состоит в том, что на спецификатор const нельзя полностью полагаться в деле выявления программных ошибок, которые ведут к случайному изменению данных.

Анализ проделанной работы

Теперь посвятим некоторое время оценке того, к чему привело применение подхода с использованием ADT. Прежде всего, сравним листинги 17.2 и 17.4. В обеих программах для решения задачи создания списка фильмов задействован один и тот же фундаментальный метод (динамическое резервирование связанных структур). Но программа, представленная в листинге 17.2, демонстрирует все программные ухищрения, помещая функцию `malloc()` и выражение `prev->next` в общедоступное представление. И напротив, код, показанный в листинге 17.4, скрывает эти нюансы и выражает программу на языке, который непосредственно связан с решаемой задачей. То есть в нем речь идет о создании списка и добавлении в него элементов, а не о вызове функций управления памятью или переустановке указателей. Короче говоря, листинг 17.4 представляет программу в терминах решаемой задачи, а не в терминах низкоуровневых инструментов, необходимых для ее решения. Версия с применением ADT ориентирована на выполнение задач конечного пользователя и ее значительно легче читать.

Во-вторых, файлы `list.h` и `list.c` вместе образуют ресурс многократного использования. Если пользователю потребуется другой простой список, ему достаточно воспользоваться этими файлами. Предположим, что необходимо хранить сведения о своих родственниках: имена, родственные отношения, адреса и номера телефонов. Прежде всего, следовало бы обратиться к файлу `list.h` и переопределить тип `Item`:

```
typedef struct itemtag
{
   char fname[14];
   char lname [24];
   char relationship[36];
   char address [60];
   char phonenum[20];
} Item;
```

Затем... что ж, в данном случае этого было бы достаточно, поскольку все функции простого списка определены в терминах типа `Item`. В некоторых случаях пришлось бы также переопределить функцию `CopyToNode()`. Например, если бы элемент был массивом, его нельзя было бы копировать с помощью операции присваивания.

Еще один важный момент связан с тем, что интерфейс пользователя определен в терминах операций абстрактного списка, а не в терминах какого-то конкретного набора представлений данных и алгоритмов. Это позволяет свободно манипулировать реализацией, не переделывая конечную программу. Например, созданная нами функция `AddItem()` несколько неэффективна, поскольку она всегда начинает работу с начала списка, а затем выполняет поиск его конца. Этот недостаток можно устранить, отслеживая конец списка. Например, тип `List` можно переопределить следующим образом:

```
typedef struct list
{
   Node * head; /* указывает на начало списка */
   Node * end;  /* указывает на конец списка */
} List;
```

Конечно, после этого пришлось бы переписать функции обработки списка, используя это новое определение, но при этом не нужно было бы ничего изменять в

листинге 17.4. Подобная изоляция реализации от конечного интерфейса особенно полезна в масштабных программных проектах. Этот подход называют *сокрытием данных*, поскольку подробное представление данных скрыто от конечного пользователя.

Обратите внимание, что данный конкретный тип ADT даже не требует реализации простого списка в виде связного списка. Ниже представлена еще одна возможность:

```
#define MAXSIZE 100
typedef struct list
{
   Item entries[MAXSIZE]; /* массив элементов */
   int items;             /* число элементов в списке */
} List;
```

И, наконец, подумайте о преимуществах, предоставляемых этим подходом процессу разработки программ. Если какая-либо часть программы работает не так, как следует, вполне возможно, что проблему удастся локализовать вплоть до единственной функции. Если удастся придумать более эффективный способ решения одной из задач, например, добавления элемента, достаточно переписать только соответствующую функцию. Если требуется новая функциональная возможность, задачу можно решить, добавляя новую функцию в пакет. Если окажется, что массив или двусвязный список больше подходят для решения задачи, можно изменить реализацию, не изменяя программы, которые используют эту реализацию.

Создание очереди с помощью ADT

Как было показано, подход к программированию на языке C с применением абстрактного типа данных подразумевает выполнение следующих трех шагов:

1. Описание типа, в том числе его операций в абстрактной обобщенной манере.
2. Определение функционального интерфейса для представления нового типа.
3. Написание подробного кода реализации интерфейса.

Этот подход был нами применен при создании простого списка. Теперь применим его для создания несколько более сложного объекта — очереди.

Определение абстрактного типа данных очереди

Очередь — это список, обладающий двумя особыми свойствами. Во-первых, новые элементы могут добавляться только в конец списка. В этом смысле очередь подобна простому списку. Во-вторых, элементы могут удаляться только из начала списка. Очередь можно сравнить с очередью людей, покупающих билеты в театр. Каждый новый человек становится в конец очереди и выходит из нее в начале после приобретения билетов. Очередь представляет собой форму данных типа "*первым прибыл, первым обслужен*" (first in, first out — FIFO), подобную очереди в кинотеатр (если только никто не вклинится в очередь).

Ниже дано неформальное абстрактное определение.

Имя типа: Очередь

Свойства типа: Может содержать упорядоченную последовательность элементов.

Операции типа: Инициализация очереди пустым значением.
Определение того, является ли очередь пустой.
Определение того, является ли очередь полной.
Определение количества элементов в очереди.
Добавления элемента в конец очереди.
Удаление и восстановление элемента в начале очереди.
Опустошение очереди.

Определение интерфейса

Определение интерфейса будет помещено в файл `queue.h`. Мы используем функциональную возможность `typedef` языка C для создания имен двух типов: `Item` и `Queue`. Конкретная реализация соответствующих структур должна быть частью файла `queue.h`, но с концептуальной точки зрения проектирование структур — часть этапа детальной реализации. А пока будем считать, что типы определены, и сосредоточим внимание на прототипах функций.

Прежде всего следует подумать над инициализацией. Она предполагает изменение типа `Queue`, поэтому функция должна принимать адрес переменной `Queue` в качестве аргумента.

```
void InitializeQueue (Queue * pq);
```

Во-вторых, определение того, является очередь пустой или полной, требует использования функции, которая должна возвращать значение `true` или `false`. В этом примере мы будем считать, что файл заголовка `stdbool.h` стандарта C99 доступен. Если это не так, можно использовать тип `int` или определить тип `bool` самостоятельно. Поскольку функция не изменяет очередь, она может принимать аргумент `Queue`. С другой стороны, в зависимости от реальной величины объекта типа `Queue`, передача только адреса переменной `Queue` может выполняться быстрее и требовать меньшего объема памяти. Еще одно преимущество такого подхода состоит в том, что в данном случае все функции будут принимать в качестве аргумента адрес. Для указания того, что эти функции не изменяют очередь, можно, да и должно, использовать спецификатор `const`:

```
bool QueueIsFull(const Queue * pq);
bool QueueIsEmpty (const Queue * pq);
```

Иначе говоря, указатель `pq` указывает на объект данных `Queue`, который не может изменяться через `pq`. Аналогичный прототип можно определить для функции, которая возвращает число элементов в очереди:

```
int QueueItemCount(const Queue * pq);
```

Добавление элемента в конец очереди требует идентификации элемента и очереди. На этот раз очередь изменяется, поэтому использование указателя обязательно. Функция могла бы иметь тип `void`, или же возвращаемое значение можно было бы использовать для указания того, была ли операция добавления элемента успешно выполнена. Мы используем второй подход:

```
bool EnQueue(Item item, Queue * pq);
```

И, наконец, удаление элемента может быть реализовано несколькими способами. Если элемент определен как структура или как один из фундаментальных типов, он может быть возвращен функцией. Аргументом функции могла бы быть переменная `Queue` или указатель на переменную. `Queue`. Следовательно, один из возможных прототипов выглядит так:

```
Item DeQueue(Queue q);
```

Однако следующий прототип является несколько более общим:

```
bool DeQueue(Item * pitem, Queue * pq);
```

Элемент, удаленный из очереди, помещается в место, которое указано указателем `pitem`, а возвращаемое значение указывает успешность выполнения операции.

Единственный аргумент, который потребуется функции опустошения очереди — адрес очереди, как показано в следующем прототипе:

```
void EmptyTheQueue(Queue * pq);
```

Реализация представления данных интерфейса

Прежде всего, необходимо выбрать форму данных C для использования в очереди. Одной из возможностей является массив. Преимущества применения массивов состоят в том, что их легко использовать, а добавление элемента в конец заполненной части массива не представляет сложности. Проблема возникает в связи с удалением элемента из начала очереди. Если снова воспользоваться аналогией очереди за билетами, удаление элемента из начала очереди состоит из копирования значения первого элемента массива (это просто) и последующего перемещения каждого элемента в массиве влево на одну позицию в сторону его начала. Хотя эти действия легко программировать, они требуют много процессорного времени (рис. 17.6).

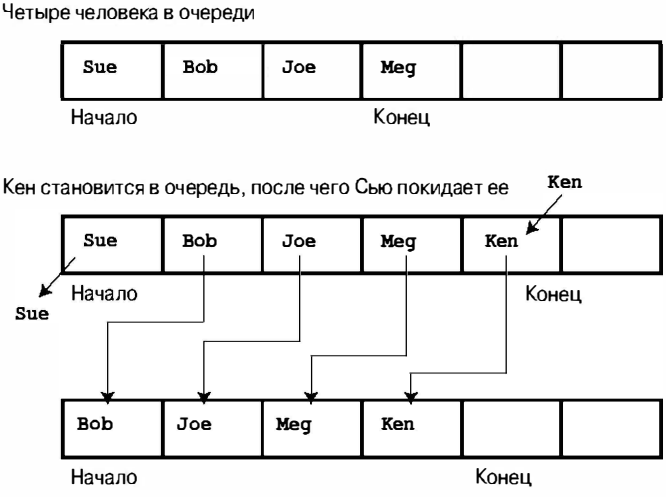


Рис. 17.6. *Использование массива в качестве очереди*

Второй способ решения задачи удаления элемента в реализации с использованием массива — оставление остающихся элементов в тех позициях, которых они находятся, и переопределение начального элемента (рис. 17.7). Проблема применения этого метода в том, что место, ранее занятое удаленными элементами, становится затрачиваемым впустую, что ведет к уменьшению доступного пространства в очереди.

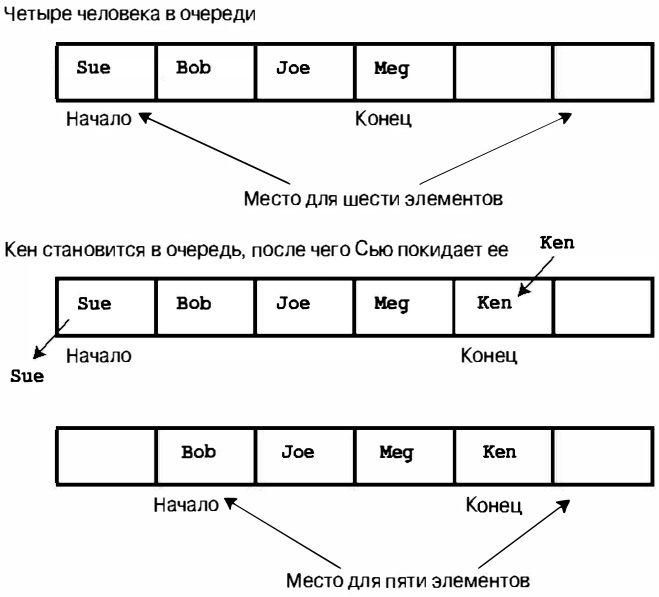


Рис. 17.7. *Переопределение начального элемента*

Более рациональное решение проблемы пустого места — создание *кольцевой* очереди. Это означает "заворачивание" конца массива в его начало. То есть потребуется принять, что первый элемент массива следует непосредственно за последним элементом. Иными словами, по достижении конца массива можно начинать добавлять элементы в начальные позиции, если они освобождены (рис. 17.8). Этот процесс можно сравнить с рисованием на бумажной ленте, склеенной в кольцо. Естественно, теперь придется выполнить дополнительную непростую задачу по обеспечению того, чтобы конец очереди не перекрывал ее начала.

Еще одно возможное решение предполагает применение связного списка. Преимущество этого подхода состоит в том, что удаление начального элемента не требует перемещения всех остальных. Достаточно просто переопределить указатель на начало, чтобы он указывал на новый первый элемент. Поскольку мы уже работали со связными списками, то пойдем этим путем. Чтобы проверить свои идеи, начнем с создания очереди целочисленных значений:

```
typedef int Item;
```

Связный список строится из узлов, поэтому определим тип данных узла node:

```
typedef struct node
{
   Item item;
   struct node * next;
} Node;
```

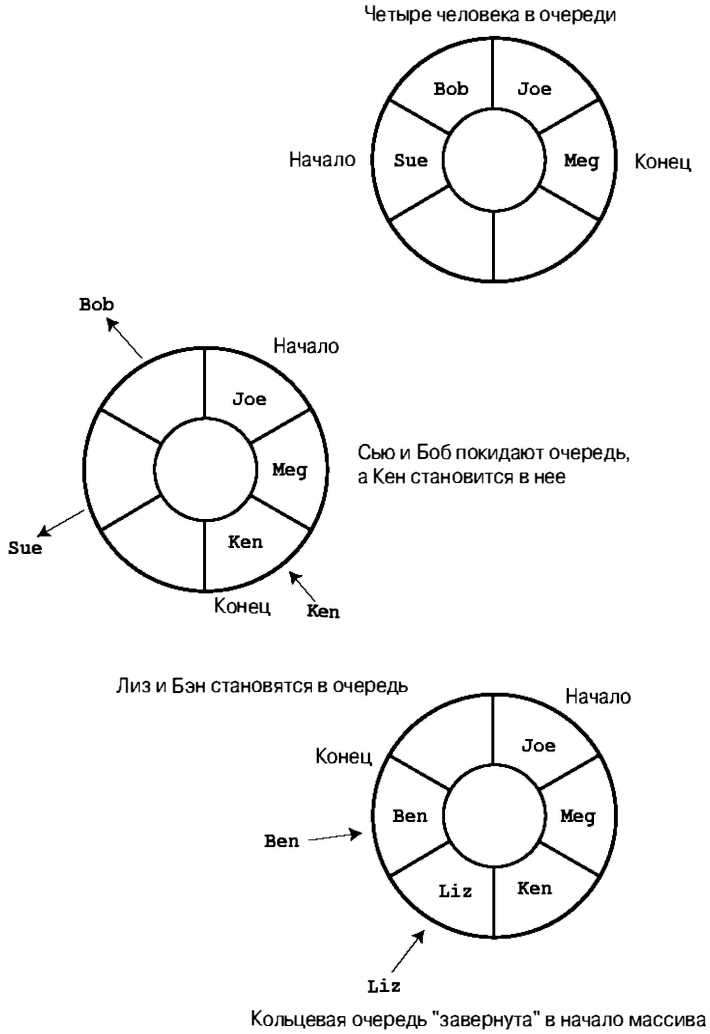


Рис. 17.8. *Кольцевая очередь*

Для реализации очереди необходимо отслеживать начальный и конечный элементы. Для этого можно использовать указатели. Кроме того, можно использовать счетчик для отслеживания количества элементов в очереди. Таким образом, структура будет содержать два элемента-указателя и один элемент типа `int`:

```
typedef struct queue
{
   Node * front;  /* указатель на начало очереди    */
   Node * rear;   /* указатель на конец очереди     */
   int items;     /* количество элементов в очереди */
} Queue;
```

Обратите внимание, что Queue — структура с тремя элементами, поэтому ранее принятое решение об использовании в качестве аргументов указателей на очереди, а не самих очередей, экономит время и объем используемой памяти.

Теперь пора подумать о размере очереди. При использовании связного списка размер очереди ограничен объемом доступной памяти, но часто имеет смысл применять очередь значительно меньшего размера. Например, очередь можно использовать для имитации самолетов, дожидающихся своей очереди на посадку в аэропорту. Если количество ожидающих самолетов становится слишком большим, прибывающие самолеты могут направляться в другие аэропорты. Установим максимальный размер очереди равным 10. Определения и прототипы интерфейса очереди представлены в листинге 17.6. В нем отсутствует конкретное определение типа Item. При использовании интерфейса в него придется поместить определение, соответствующее нуждам конкретной программы.

Листинг 17.6. Файл заголовка интерфейса queue.h

```
/* queue.h -- интерфейс очереди */
#ifndef _QUEUE_H_
#define _QUEUE_H_
#include <stdbool.h>

/* ЗДЕСЬ НЕОБХОДИМО ВСТАВИТЬ ОПРЕДЕЛЕНИЕ ТИПА ITEM */
/* НАПРИМЕР, */
typedef int Item;
/* ИЛИ typedef struct item {int gumption; int charisma;} Item; */

#define MAXQUEUE 10
typedef struct node
{
   Item item;
   struct node * next;
} Node;
typedef struct queue
{
   Node * front; /* указатель на начало очереди    */
   Node * rear;  /* указатель на конец очереди     */
   int items;    /* количество элементов в очереди */
} Queue;

/* операция:          инициализация очереди                          */
/* начальное условие: pq указывает на очередь                        */
/* конечное условие:  очередь инициализирована пустым значением      */
void InitializeQueue(Queue * pq);

/* операция:          проверка того, является ли очередь полной      */
/* начальное условие: pq указывает на ранее инициализированную очередь */
/* конечное условие:  возвращает значение True, если очередь полна;  */
/*                    иначе возвращает значение False */
bool QueueIsFull(const Queue * pq);

/* операция:          проверка того, является ли очередь пустой      */
/* начальное условие: pq указывает на ранее инициализированную очередь */
/* конечное условие:  возвращает значение True, если очередь пуста;  */
/*                    иначе возвращает значение False                */
bool QueueIsEmpty(const Queue *pq);
```

```
/* операция:          определение количества элементов в очереди   */
/* начальное условие: pq указывает на ранее инициализированную очередь */
/* конечное условие:  возвращает число элементов в очереди         */
int QueueItemCount(const Queue * pq);

/* операция:          добавление элемента в конец очереди          */
/* начальное условие: pq указывает на ранее инициализированную очередь */
/*                    элемент должен быть помещен в конец очереди  */
/* конечное условие:  если очередь не пуста, элемент помещается в  */
/*                    конец очереди и функция возвращает значение  */
/*                    True; в противном случае очередь остается неизменной, */
/*                    и функция возвращает значение False          */
bool EnQueue(Item item, Queue * pq);

/* операция:          удаление элемента из начала очереди          */
/* начальное условие: pq указывает на ранее инициализированную очередь */
/* конечное условие:  если очередь не пуста, элемент в начале      */
/*                    очереди копируется в *pitem и удаляется из   */
/*                    очереди, и функция возвращает значение True; */
/*                    если операция опустошает очередь, очередь    */
/*                    переустанавливается в пустое состояние.      */
/*                    Если очередь пуста с самого начала, очередь  */
/*                    остается неизменной, и функция возвращает    */
/*                    значение False                               */
bool DeQueue(Item *pitem, Queue * pq);

/* операция:          опустошение очереди                          */
/* начальное условие: pq указывает на ранее инициализированную очередь */
/* конечное условие:  очередь пуста                                */
void EmptyTheQueue(Queue * pq);

#endif
```

Реализация функций интерфейса

Теперь можно приступить к написанию кода интерфейса. Во-первых, инициализация очереди "пустым значением" означает установку указателей на начало и конец очереди в значения NULL, а счетчика (элемента item) — в значение 0.

```
void InitializeQueue(Queue * pq)
{
   pq->front = pq->rear = NULL;
   pq->items = 0;
}
```

Во-вторых, элемент item позволяет легко выполнять проверку на предмет того, является ли очередь полной или пустой, и возвращать количество элементов в очереди:

```
bool QueueIsFull(const Queue * pq)
{
   return pq->items == MAXQUEUE;
}
bool QueueIsEmpty(const Queue * pq)
{
   return pq->items == 0;
}
```

```
int QueueItemCount(const Queue * pq)
{
   return pq->items;
}
```

Добавление элемента в очередь требует выполнения следующих действий:

1. Создание нового узла.
2. Копирование элемента в узел.
3. Установка указателя `next` узла в значение `NULL`, что служит признаком того, что данный узел является последним в списке.
4. Установка указателя `next` текущего конечного узла так, чтобы он указывал на новый узел, что связывает новый узел с очередью.
5. Установка указателя `rear` для указания на новый узел, что облегчает поиск последнего узла.
6. Добавление 1 к значению счетчика элементов.

Функция должна также обрабатывать два особых случая. Во-первых, если очередь пуста, указатель `front` должен быть установлен для указания на новый узел. Это связано с тем, что при наличии только одного узла этот узел одновременно является и начальным и конечным узлом очереди. Во-вторых, если функции не удается распределить память для узла, она должна предпринять какие-то действия. Поскольку мы предполагаем использование небольших очередей, подобная ошибка должна возникать редко. Поэтому в случае переполнения памяти функция будет просто прерывать выполнение программы. Код функции `EnQueue()` выглядит следующим образом:

```
bool EnQueue(Item item, Queue * pq)
{
   Node * pnew;

   if (QueueIsFull(pq))
      return false;
   pnew = (Node *) malloc( sizeof(Node));
   if (pnew == NULL)
   {
      fprintf(stderr,"Не удается зарезервировать память!\n");
      exit(1);
   }
   CopyToNode(item, pnew);
   pnew->next = NULL;
   if (QueueIsEmpty(pq))
      pq->front = pnew;       /* элемент помещается в начало очереди  */
   else
      pq->rear->next = pnew; /* связь с концом очереди               */
   pq->rear = pnew;          /* запись конца очереди                 */
   pq->items++;              /* увеличение числа элементов на единицу*/

   return true;
}
```

Функция `CopyToNode()` — статическая функция, выполняющая копирование элемента в узел:

```
static void CopyToNode(Item item, Node * pn)
{
   pn->item = item;
}
```

Удаление элемента из начала очереди требует выполнения следующих действий:

1. Копирование элемента в ожидающую переменную.
2. Освобождение памяти, использованной удаляемым узлом.
3. Переустановка указателя на начало очереди на следующий элемент в очереди.
4. Переустановка указателей на начало и конец очереди в значение NULL в случае удаления последнего элемента.
5. Уменьшение значения счетчика элементов на единицу.

Все эти действия реализованы в следующем коде:

```
bool DeQueue(Item * pitem, Queue * pq)
{
   Node * pt;

   if (QueueIsEmpty(pq))
      return false;
   CopyToItem(pq->front, pitem);
   pt = pq->front;
   pq->front = pq->front->next;
   free(pt);
   pq->items--;
   if (pq->items == 0)
      pq->rear = NULL;

   return true;
}
```

Необходимо отметить несколько важных обстоятельств. Во-первых, код не выполняет явной установки указателя front в значение NULL при удалении последнего элемента. Это связано с тем, что он в любом случае устанавливает указатель front в значение указателя next удаляемого узла. Если этот узел — последний в очереди, значение его указателя next равно NULL, поэтому указатель front получает значение NULL.

Во-вторых, код использует временный указатель (pt) для отслеживания местоположения удаленного узла. Это связано с тем, что официальный указатель первого узла (pq->front) переустанавливается так, чтобы указывать на следующий узел. Поэтому без применения временного указателя программа утрачивала бы возможность отслеживания того, какой блок памяти нужно освобождать.

Для опустошения очереди можно использовать функцию DeQueue(). Для этого достаточно вызывать ее циклически до тех пор, пока очередь не станет пустой:

```
void EmptyTheQueue(Queue * pq)
{
   Item dummy;
   while (!QueueIsEmpty(pq))
      DeQueue(&dummy, pq);
}
```

Поддержание ADT в порядке

После того, как интерфейс ADT определен, вы должны использовать одну из его функций для обработки типа данных. Например, обратите внимание, что функция `Dequeue()` зависит от выполнения функцией `EnEqueue()` ее задачи по правильной установке указателей и установке указателя `next` узла `rear` в значение `NULL`. Если программе, в которой используется ADT, потребуется непосредственно манипулировать частями очереди, это может привести к нарушению координации между функциями в пакете интерфейса.

Все функции интерфейса, включая функцию `CopyToItem()`, используемую в функции `EnQueue()` представлены в листинге 17.7.

Листинг 17.7. Файл реализации `queue.c`

```
/* queue.c -- реализация типа Queue */
#include <stdio.h>
#include <stdlib.h>
#include "queue.h"

/* локальные функции */
static void CopyToNode(Item item, Node * pn);
static void CopyToItem(Node * pn, Item * pi);

void InitializeQueue(Queue * pq)
{
   pq->front = pq->rear = NULL;
   pq->items = 0;
}

bool QueueIsFull(const Queue * pq)
{
   return pq->items == MAXQUEUE;
}

bool QueueIsEmpty(const Queue * pq)
{
   return pq->items == 0;
}

int QueueItemCount(const Queue * pq)
{
   return pq->items;
}

bool EnQueue(Item item, Queue * pq)
{
   Node * pnew;
   if (QueueIsFull(pq))
      return false;
   pnew = (Node *) malloc( sizeof(Node));
   if (pnew == NULL)
   {
      fprintf(stderr,"Не удается зарезервировать память!\n");
      exit(1);
   }
```

```
   CopyToNode(item, pnew);
   pnew->next = NULL;
   if (QueueIsEmpty(pq))
      pq->front = pnew;      /* элемент помещается в начало очереди */
   else
      pq->rear->next = pnew; /* связь с концом очереди              */
   pq->rear = pnew;          /* запись местоположения конца очереди */
   pq->items++;    /* увеличение количества элементов в очереди на единицу */
   return true;
}
bool DeQueue(Item * pitem, Queue * pq)
{
   Node * pt;
   if (QueueIsEmpty(pq))
      return false;
   CopyToItem(pq->front, pitem);
   pt = pq->front;
   pq->front = pq->front->next;
   free(pt);
   pq->items--;
   if (pq->items == 0)
      pq->rear = NULL;
   return true;
}
/* Опустошение очереди */
void EmptyTheQueue(Queue * pq)
{
   Item dummy;
   while (!QueueIsEmpty(pq))
      DeQueue(&dummy, pq);
   }
/* Локальные функции */
static void CopyToNode(Item item, Node * pn)
{
   pn->item = item;
}
static void CopyToItem(Node * pn, Item * pi)
{
   *pi = pn->item;
}
```

Тестирование очереди

Прежде чем вставлять новый программный пакет, подобный пакету очереди, в важную программу, всегда следует его тестировать. Один из методов тестирования — создание короткой программы, иногда называемой *драйвером*, единственное назначение которой состоит в тестировании пакета. Например, в коде, приведенном в лис-

тинге 17.8, очередь используется для добавления и удаления целых чисел. Прежде чем выполнять программу, убедитесь в наличии следующей строки в файле queue.h:

```
typedef int item;
```

Следует помнить также о необходимости выполнения компоновки с файлами queue.c и use_q.c.

Листинг 17.8. Программа use_q.c

```
/* use_q.c -- тестирование интерфейса Queue с помощью драйвера */
/* компилируется вместе с queue.c */
#include <stdio.h>
#include "queue.h" /* определяет типы Queue, Item */

int main(void)
{
   Queue line;
   Item temp;
   char ch;

   InitializeQueue(&line);
   puts("Тестирование интерфейса Queue. Введите a, чтобы добавить значение,");
   puts("введите d, чтобы удалить значение, и введите q для выхода из программы.");
   while ((ch = getchar()) != 'q')
   {
      if (ch != 'a' && ch != 'd') /* игнорирование остального ввода */
         continue;
      if ( ch == 'a')
      {
         printf("Вводимое целое значение: ");
         scanf("%d", &temp);
         if (!QueueIsFull(&line))
         {
            printf("Помещение %d в очередь\n", temp);
            EnQueue(temp,&line);
         }
         else
            puts("Очередь полна!");
      }
      else
      {
         if (QueueIsEmpty(&line))
            puts("Значения для удаления отсутствуют!");
         else
         {
            DeQueue(&temp,&line);
            printf("Удаление %d из очереди\n", temp);
         }
      }
      printf("%d элементов в очереди\n", QueueItemCount(&line));
      puts("Введите a, чтобы добавить значение, d, чтобы удалить, q для
выхода из программы:");
   }
```

```
    EmptyTheQueue(&line);
    puts("Программа завершена.");

    return 0;
}
```

Результаты выполнения этой программы показаны ниже. Следует также проверить правильность работы реализации в ситуации, когда очередь полна.

```
Тестирование интерфейса Queue interface. Введите a, чтобы добавить значение,
Введите d, чтобы удалить значение, и введите q для выхода из программы.
a
Добавляемое целое значение: 40
Помещение 40 в очередь
1 элементов в очереди
Введите a, чтобы добавить значение, d, чтобы удалить, q для выхода из программы:
a
Добавляемое целое значение: 20
Помещение 20 в очередь
2 элементов в очереди
Введите a, чтобы добавить значение, d, чтобы удалить, q для выхода из программы:
a
Добавляемое целое значение: 55
Помещение 55 в очередь
3 элементов в очереди
Введите a, чтобы добавить значение, d, чтобы удалить, q для выхода из программы:
d
Удаление 40 из очереди
2 элементов в очереди
Введите a, чтобы добавить значение, d, чтобы удалить, q для выхода из программы:
d
Удаление 20 из очереди
1 элементов в очереди
Введите a, чтобы добавить значение, d, чтобы удалить, q для выхода из программы:
d
Удаление 55 из очереди
0 элементов в очереди
Введите a, чтобы добавить значение, d, чтобы удалить, q для выхода из программы:
d
Значения для удаления отсутствуют!
0 элементов в очереди
Введите a, чтобы добавить значение, d, чтобы удалить, q для выхода из программы:
q
Программа завершена.
```

Имитация реальной очереди

Что ж, очередь работает! Теперь пора с ее помощью решить какую-то более интересную задачу. Очереди встречаются во многих реальных ситуациях. Например, очереди клиентов в банках и универсамах, очереди самолетов в аэропортах и очереди задач в многозадачных компьютерных системах. Пакет очереди можно использовать для имитации ситуаций подобного рода.

Например, предположим, что некий Зигмунд Ландерс поставил консультационный киоск в пассаже. Клиенты могут заплатить за одну, две или три минуты получения консультации. Для обеспечения свободного прохода действующие в пассаже правила ограничивают количество клиентов в очереди числом 10 (это же значение вполне можно использовать для определения максимального размера очереди в программе). Предположим, что люди подходят к киоску случайным образом, и что время, которое они отводят на получение консультации, случайным образом распределяется между тремя возможными вариантами (одна, две или три минуты). Сколько в среднем клиентов придется обслужить Зигмунду в течение часа? Сколько в среднем каждому клиенту придется дожидаться своей очереди? Какой будет средняя длина очереди? Имитация может дать ответы на эти аналогичные вопросы.

Во-первых, решим, что следует помещать в очередь. Каждого клиента можно описывать значениями времени, когда он становится в очередь, и значениями времени (в минутах), которые он собирается затратить на консультацию. Это предполагает следующее определение элемента `Item`:

```
typedef struct item
{
   long arrive; /* время присоединения клиента к очереди */
   int processtime; /* желаемое количество минут консультации */
} Item;
```

Для преобразования пакета очереди, чтобы он обрабатывал эту структуру, а не тип `int`, который был использован в последнем примере, достаточно заменить первоначальное определение `typedef` типа `Item` приведенным выше. После этого можно будет не беспокоиться о нюансах реализации очереди. Вместо этого можно будет сосредоточить все внимание на реальной проблеме — имитации очереди к киоску Зигмунда.

Вот один из возможных подходов. Предположим, что отсчет времени выполняется с одноминутными интервалами. Тогда через каждую минуту необходимо проверять, не подошел ли новый клиент. Если клиент подошел, и очередь не переполнена, клиента необходимо добавить в очередь. Этот подход предполагает запись в структуру `Item` времени прибытия клиента и длительности консультации, которую клиент желает оплатить, с последующим добавлением элемента в очередь. Однако, если очередь полна, клиента нужно отослать. В целях учета мы будем отслеживать общее число клиентов и общее количество "отказов" (людей, которые не могут стать в очередь, поскольку она переполнена).

Далее потребуется выполнить обработку начала очереди. То есть, если очередь не пуста, и если Зигмунд не занят обслуживанием предыдущего клиента, необходимо удалить элемент из начала очереди. Вспомним, что элемент содержит значение времени присоединения клиента к очереди. Сравнивая это время с текущим, мы получим время пребывания клиента в очереди (в минутах). Элемент содержит также время, которое клиент намерен потратить на консультацию — это значение определяет интервал времени, в течение которого Зигмунд будет занят обслуживанием клиента. Для отслеживания этого времени ожидания мы используем переменную. Если Зигмунд занят, никто "не удаляется из очереди". Однако значение переменной отслеживания времени ожидания должно уменьшаться.

Основной код может выглядеть подобно приведенному ниже, причем каждый цикл соответствует одной минуте активности:

```
for (cycle = 0; cycle < cyclelimit; cycle++)
{
   if (newcustomer(min_per_cust))
   {
      if (QueueIsFull(&line))
          turnaways++;
      else
      {
          customers++;
          temp = customertime(cycle);
          EnQueue(temp, &line);
      }
   }
   if (wait_time <= 0 && !QueueIsEmpty(&line))
   {
      DeQueue (&temp, &line);
      wait_time = temp.processtime;
      line_wait += cycle - temp.arrive;
      served++;
   }
   if (wait_time > 0)
      wait_time--;
   sum_line += QueueItemCount(&line);
}
```

Ниже представлены краткие описания значений некоторых переменных и функций.

- `min_per_cust` — средний интервал (в минутах) между прибытиями клиентов.
- `newcustomer()` использует функцию `rand()` C для определения того, появляется ли клиент в течение данной конкретной минуты.
- `turnaways` — количество прибывших клиентов, которым было отказано в обслуживании.
- `customers` — количество прибывающих клиентов, помещаемых в очередь.
- `temp` — переменная типа `Item`, описывающая нового клиента.
- `customertime()` устанавливает значения элементов `arrive` и `processtime` структуры `temp`.
- `wait_time` — остающееся время (в минутах) до того момента, когда Зигмунд завершит обслуживание текущего клиента.
- `line_wait` — накапливаемое значение общего времени, проведенного в очереди всеми клиентами на текущий момент.
- `served` — количество действительно обслуженных клиентов.
- `sum_line` — накапливаемое значение длины очереди на текущий момент.

Подумайте, насколько более запутанным и непонятным выглядел бы код, если бы он был переполнен функциями `malloc()` и `free()` и указателями на узлы. Наличие пакета очереди позволяет сосредоточить внимание на проблеме имитации, а не на нюансах программирования.

Полный код имитации консультационного киоска представлен в листинге 17.9. Для генерации случайных значений методом, предложенным в главе 12, в нем применяются стандартные функции `rand()`, `srand()` и `time()`. Чтобы использовать программу, не забудьте дополнить определение типа `Item` в файле `queue.h` следующими строками:

```
typedef struct item
{
   long arrive; /* время присоединения клиента к очереди */
   int processtime; /* требуемая продолжительность консультации в минутах*/
} Item;
```

Не забудьте также выполнить компоновку файла `mall.c` с файлом `queue.c`.

Листинг 17.9. Программа `mall.c`

```
/* mall.c -- использует интерфейс Queue */
/* компилируется вместе с queue.c */
#include <stdio.h>
#include <stdlib.h> /* содержит определения функций rand() и srand() */
#include <time.h>  /* содержит определение функции time()     */
#include "queue.h" /* изменение определения typedef типа Item */
#define MIN_PER_HR 60.0

bool newcustomer(double x);   /* имеется новый клиент?        */
Item customertime(long when); /* установка параметров клиента */

int main(void)
{
   Queue line;
   Item temp; /* данные нового клиента */
   int hours; /* длительность периода имитации в часах*/
   int perhour; /* среднее число прибывающих клиентов в час */
   long cycle, cyclelimit; /* счетчик и граничное значение цикла */
   long turnaways = 0;  /* число отказов из-за переполненной очереди */
   long customers = 0;  /* число клиентов присоединившихся к очереди */
   long served = 0;     /* число клиентов обслуженных за время имитации */
   long sum_line = 0;   /* накопительное значение длины очереди */
   int wait_time = 0;   /* время до освобождения Зигмунда */
   double min_per_cust; /* среднее время между прибытиями клентов */
   long line_wait = 0;  /* накопительное значение ожидания в очереди */

   InitializeQueue(&line);
   srand(time(0)); /* случайная инициализация функции rand() */
   puts("Учебный пример: консультационный киоск Зигмунда Ландера");
   puts("Введите длительность периода имитации в часах:");
   scanf("%d", &hours);
   cyclelimit = MIN_PER_HR * hours;
   puts("Введите среднее количество клиентов, прибывающих за час:");
   scanf("%d", &perhour);
   min_per_cust = MIN_PER_HR / perhour;

   for (cycle = 0; cycle < cyclelimit; cycle++)
   {
      if (newcustomer(min_per_cust))
      {
```

```
         if (QueueIsFull(&line))
            turnaways++;
         else
         {
            customers++;
            temp = customertime(cycle);
            EnQueue(temp, &line);
         }
      }
      if (wait_time <= 0 && !QueueIsEmpty(&line))
      {
         DeQueue (&temp, &line);
         wait_time = temp.processtime;
         line_wait += cycle - temp.arrive;
         served++;
      }
      if (wait_time > 0)
         wait_time--;
      sum_line += QueueItemCount(&line);
   }
   if (customers > 0)
   {
      printf("   принятых клиентов: %ld\n", customers);
      printf("обслуженных клиентов: %ld\n", served);
      printf("             отказов: %ld\n", turnaways);
      printf(" средняя длина очереди: %.2f\n",
         (double) sum_line / cyclelimit);
      printf("среднее время ожидания: %.2f минут\n",
         (double) line_wait / served);
   }
   else
      puts("Клиенты отсутствуют!");
   EmptyTheQueue(&line);
   puts("Программа завершена.");
   return 0;
}
/* x = среднее время между прибытием клиентов, в минутах */
/* возвращаемое значение - true, если клиент появляется в течение данной минуты*/
bool newcustomer(double x)
{
   if (rand() * x / RAND_MAX < 1)
      return true;
   else
      return false;
}
/* when - время прибытия клиента */
/* функция возвращает структуру Item со значением времени прибытия */
/* установленным равным значению when, а значением времени обслуживания
/* установленным равным случайному значению в диапазоне 1 - 3 */
Item customertime(long when)
```

```
{
   Item cust;

   cust.processtime = rand() % 3 + 1;
   cust.arrive = when;

   return cust;
}
```

Программа позволяет задавать длительность периода имитации и среднее количество клиентов, обращающихся за консультацией в течение часа. Выбор длительного периода имитации позволяет получить достаточно хорошее усреднение, а выбор короткого периода позволяет видеть своего рода случайную вариацию, которая может иметь место от часа к часу. Следующие результаты выполнения программы иллюстрируют эти моменты. Обратите внимание, что средние значения длины очереди и времени ожидания для 80-часового и для 800-часового периода почти совпадают, но результаты двух одночасовых периодов существенно отличаются как друг от друга, так и от результатов усреднений более длительных периодов. Это обусловлено тем, что меньшие статистические выборки характеризуются бо́льшими относительными вариациями.

```
Учебный пример: консультационный киоск Зигмунда Ландера
Введите длительность периода имитации в часах:
80
Введите среднее количество клиентов, прибывающих за час:
20
     принятых клиентов: 1633
  обслуженных клиентов: 1633
               отказов: 0
 средняя длина очереди: 0.46
среднее время ожидания: 1.35 минут

Учебный пример: консультационный киоск Зигмунда Ландера
Введите длительность периода имитации в часах:
800
Введите среднее количество клиентов, прибывающих за час:
20
     принятых клиентов: 16020
  обслуженных клиентов: 16019
               отказов: 0
 средняя длина очереди: 0.44
среднее время ожидания: 1.32 минут

Учебный пример: консультационный киоск Зигмунда Ландера
Введите длительность периода имитации в часах:
1
Введите среднее количество клиентов, прибывающих за час:
20
    принятых клиентов: 20
  обслуженных клиентов: 20
               отказов: 0
 средняя длина очереди: 0.23
среднее время ожидания: 0.70 минут
```

```
Учебный пример: консультационный киоск Зигмунда Ландера
Введите длительность периода имитации в часах:
1
Введите среднее количество клиентов, прибывающих за час:
20
     принятых клиентов: 22
  обслуженных клиентов: 22
               отказов: 0
 средняя длина очереди: 0.75
среднее время ожидания: 2.05 минут
```

Еще один способ использования этой программы предусматривает сохранение длительности периода имитации неизменной, но варьирование средним количеством клиентов, прибывающих в течение часа. Результаты выполнения двух примеров исследования этой вариации выглядят следующим образом:

```
Учебный пример: консультационный киоск Зигмунда Ландера
Введите длительность периода имитации в часах:
80
Введите среднее количество клиентов, прибывающих за час:
25
     принятых клиентов: 1960
  обслуженных клиентов: 1959
               отказов: 3
 средняя длина очереди: 1.43
среднее время ожидания: 3.50 минут
```

```
Учебный пример: консультационный киоск Зигмунда Ландера
Введите длительность периода имитации в часах:
80
Введите среднее количество клиентов, прибывающих за час:
30
     принятых клиентов: 2376
  обслуженных клиентов: 2373
               отказов: 94
 средняя длина очереди: 5.85
среднее время ожидания: 11.83 минут
```

Обратите внимание на резкое возрастание среднего времени ожидания с увеличением частоты прибытия клиентов. Среднее время ожидания при 20 клиентах в час (80-часовая имитация) составило 1,35 минуты. Это значение возрастает до 3,5 минуты при 25 клиентах в час и до 11,83 минуты при 30 клиентах в час. Кроме того, количество отказов возрастает от 0 до 3 и до 94 соответственно. Зигмунд мог бы воспользоваться подобным анализом для принятия решения о необходимости открытия второго киоска.

Сравнение связного списка и массива

Многие задачи программирования, такие как создание списка или очереди, могут решаться с помощью связного списка, под которым мы понимаем связную последовательность динамически резервируемых структур, или с помощью массива. Каждая форма реализации обладает собственными преимуществами и недостатками, поэтому

конкретный выбор зависит от конкретных требований задачи. Основные характеристики связных списков и массивов перечислены в табл. 17.1.

Таблица 17.1. Сравнение массивов со связными списками

Форма данных	*Достоинства*	*Недостатки*
Массив	Непосредственно поддерживается в C.	Размер определяется во время компиляции.
	Предоставляет произвольный доступ к элементам данных.	Вставка и удаление элементов требует значительных затрат времени.
Связный список	Размер определяется во время выполнения.	Произвольный доступ к элементам невозможен.
	Вставка и удаление выполняются быстро.	Пользователь должен обеспечить программную поддержку.

Подробнее рассмотрим процесс вставки и удаления элементов. Для вставки элемента в массив необходимо переместить элементы, чтобы освободить место для нового элемента, как показано на рис. 17.9. Чем ближе к началу массива должен быть помещен новый элемент, тем больше элементов потребуется переместить. В то же время, для вставки узла в связный список достаточно присвоить значения двум указателям, как показано на рис. 17.10. Аналогично, удаление элемента из массива требует полного изменения расположения всех элементов, а для удаления узла из связного списка достаточно переустановки указателя и освобождения памяти, использованной удаленным узлом.

Теперь рассмотрим, как осуществляется доступ к элементам списка. При использовании массива для непосредственного обращения к любому элементу можно применять индекс массива. Такую форму доступа называют *произвольным доступом*. При использовании связного списка необходимо начинать с начала списка и переходить от узла к узлу до тех пор, пока не будет достигнут требуемый узел. Эту форму называют *последовательным доступом*.

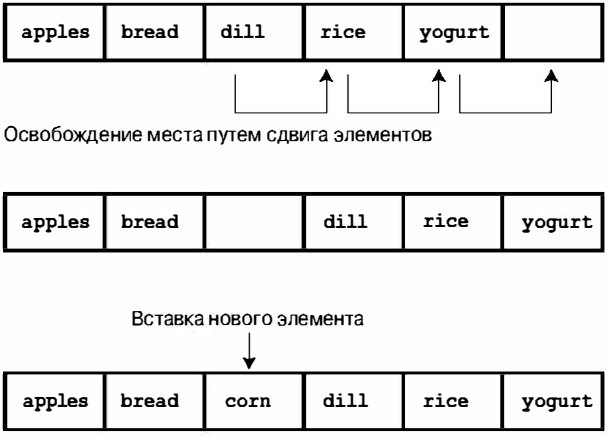


Рис. 17.9. *Вставка элемента в массив*

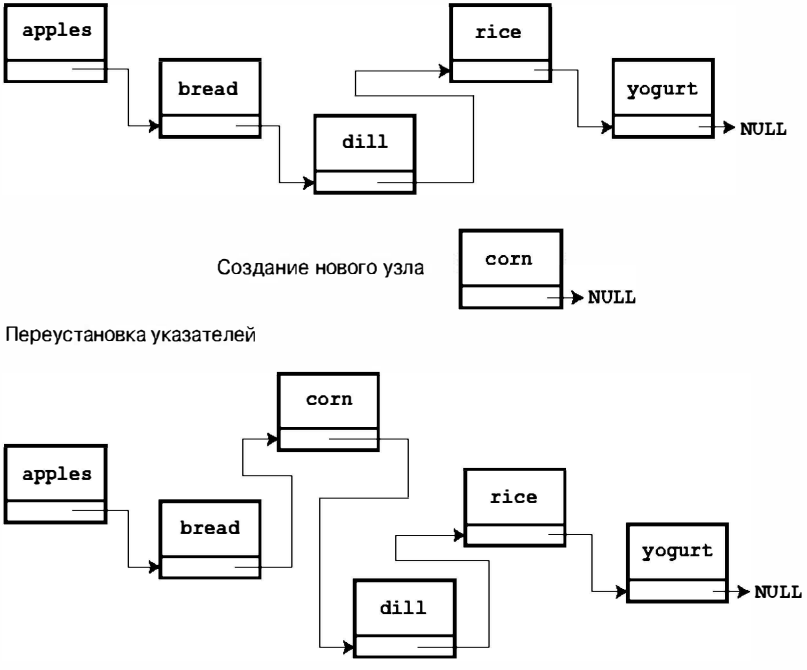


Рис. 17.10. *Вставка элемента в связный список*

При использовании связного списка необходимо начинать с начала списка и переходить от узла к узлу до тех пор, пока не будет достигнут требуемый узел. Эту форму называют *последовательным доступом*. Последовательный доступ может быть реализован и в массиве. Для последовательного обхода элементов массива достаточно увеличивать его индекс массива на единицу при каждом перемещении. В некоторых ситуациях последовательного доступа вполне достаточно. Например, если требуется отобразить каждый из элементов списка, последовательный доступ прекрасно подойдет. В других ситуациях, как будет показано ниже, наличие произвольного доступа — огромное преимущество.

Предположим, что в списке необходимо выполнить поиск конкретного элемента. Один из возможных алгоритмов может выглядеть так: начать поиск с начала списка и последовательного просматривать его элементы. Такой поиск называют *последовательным поиском*. Если только элементы не упорядочены каким-либо образом, последовательный поиск является практически единственным доступным методом решения задачи. Если искомый элемент отсутствует в списке, придется просмотреть все элементы, прежде чем можно будет сделать вывод о его отсутствии.

Последовательный поиск можно усовершенствовать, предварительно отсортировав список. Это позволяет прервать поиск, если искомый элемент не найден по достижении элемента, который должен был бы следовать за искомым. Например, предположим, что мы выполняем поиск элемента *Susan* в списке, упорядоченном по алфавиту, и через некоторое время наталкиваемся на элемент *Sylvia*, так и не найдя элемента *Susan*. В этом месте поиск можно прервать, поскольку элемент *Susan*, если бы он присутствовал в списке, предшествовал бы элементу *Sylvia*. В среднем этот метод будет сокращать время поиска элементов, отсутствующих в списке, вдвое.

В случае упорядоченного списка для поиска можно использовать значительно более эффективный метод *бинарного поиска*. Этот метод работает следующим образом. Прежде всего, назовем искомый элемент списка *целевым* и предположим, что список упорядочен по алфавиту. Затем выберем элемент, расположенный посередине списка, и сравним его с целевым элементом. Если эти два элемента равны, поиск завершен. Если элемент списка по алфавиту предшествует целевому элементу, целевой элемент, если он присутствует в списке, должен находиться во второй половине. Если элемент списка следует за целевым, целевой элемент должен располагаться в первой половине. В любом случае правила поиска уменьшают количество просматриваемых элементов вдвое. Затем применяем этот метод снова. То есть выберем элемент, расположенный посередине остающейся половины списка. Как и ранее, метод либо находит элемент, либо вдвое уменьшает размер просматриваемого списка. Эти действия следует продолжать до тех пор, пока элемент не будет найден или пока весь список не будет отброшен (рис. 17.11).

Описанный метод достаточно эффективен. Например, предположим, что список содержит 127 элементов. При использовании последовательного поиска для обнаружения элемента или установления его отсутствия в списке требовалось бы в среднем выполнение 64 операций сравнения. В то же время, бинарный поиск требовал бы выполнения не более 7 сравнений. Первая операция сравнения уменьшает количество возможных совпадений до 63, вторая — до 31 и так далее, пока шестое сравнение не уменьшит число возможных элементов до 1. После того седьмая операция сравнения определяет, является ли остающийся элемент целевым. В общем случае n сравнений позволяют обработать массив, содержащий $2^n - 1$ элементов, поэтому преимущество применения бинарного поиска по сравнению с последовательным поиском становится все более явным по мере увеличения длины списка.

В случае использования массива реализация бинарного поиска не отличается особой сложностью, поскольку для определения среднего элемента любого списка или любой его части можно использовать индекс массива. Для этого достаточно суммировать индексы первого и последнего элементов части списка и разделить результат на 2. Например, в списке, состоящем из 100 элементов, первый индекс равен 0, а последний 99. Тогда первый шаг поиска дал бы значение, равное (0 + 99)/2, или 49 (при целочисленном делении). Если бы элемент с индексом 49 располагался слишком далеко по алфавиту, искомый элемент должен был бы располагаться в диапазоне 0–48. Поэтому вторым предположением должен был бы стать индекс (0 + 48)/2, или 24. Если 24-й элемент расположен слишком близко, следующим предположением должен был бы стать индекс (25 + 48)/2, или 36. Именно в таких ситуациях вступает в действие свойство произвольного доступа к элементам массива. Оно позволяет выполнять переход от одного элемента к другому, не посещая все расположенные между ними элементы. Связные списки, которые поддерживают только последовательный доступ, не предоставляют средства для перехода к внутреннему элементу списка. Поэтому технологию бинарного поиска нельзя применять к связным спискам.

Как видите, выбор типа данных зависит от конкретной решаемой задачи. Если ситуация требует применения списка, размер которого постоянно изменяется за счет частых вставок и удалений элементов, но в котором поиск выполняется не очень часто, лучше использовать связный список. В тех же ситуациях, когда необходим стабильный список, в котором вставки и удаления выполняются лишь изредка, но в котором нужно часто выполнять поиск, лучше использовать массив.

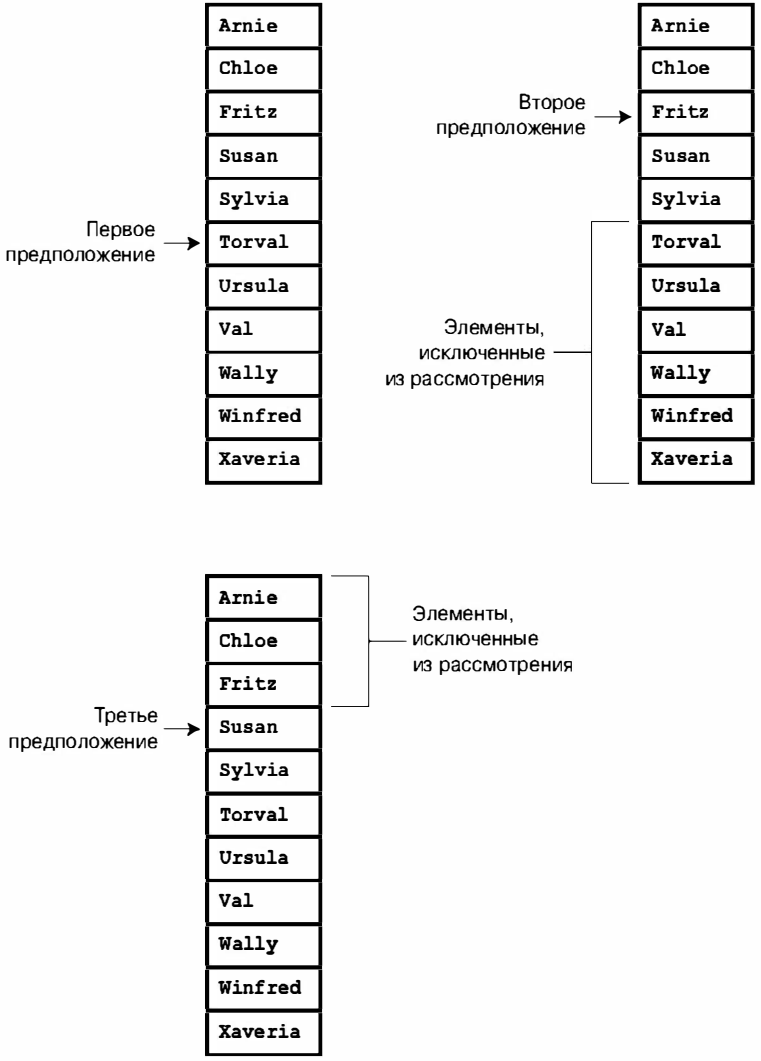


Рис. 17.11. *Бинарный поиск элемента Susan*

А как быть, если требуется форма данных, поддерживающая как частые вставки и удаления, так и частый поиск? Ни связный список, ни массив не являются идеальной формой данных для таких целей. Наиболее подходящей может оказаться другая форма данных – дерево бинарного поиска.

Деревья бинарного поиска

Дерево бинарного поиска – это связная структура, которая включает в себя поддержку стратегии бинарного поиска. Каждый узел дерева содержит элемент и два указателя на другие узлы, называемые *дочерними узлами*. Связь между узлами в дереве бинарного поиска показана на рис. 17.12.

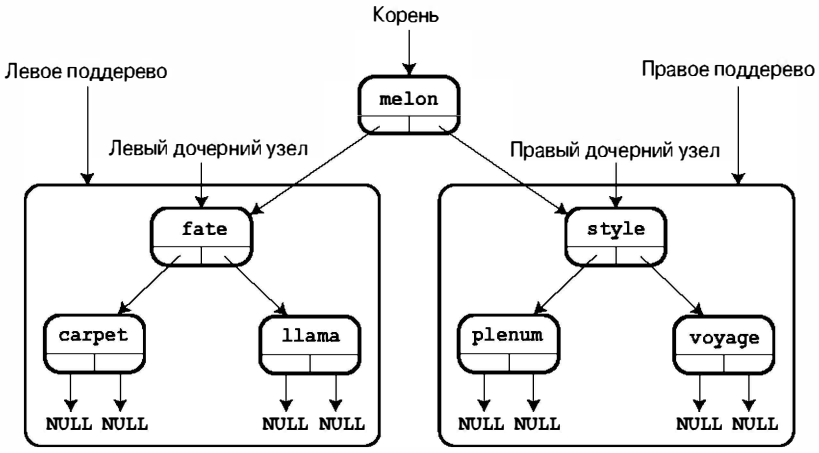


Рис. 17.12. *Хранение слов в дереве бинарного поиска*

Основная идея этой структуры состоит в том, что каждый узел имеет два дочерних узла — левый и правый. Порядок элементов определяется тем, что элемент в левом узле предшествует элементу в родительском узле, а элемент в правом узле следует за элементом родительского узла. Это отношение сохраняется для всех узлов внутри дочерних узлов. Более того, все элементы, родословная которых может быть прослежена до левого узла родительского узла, содержат элементы, которые предшествуют родительскому элементу, а все элементы, производные от правого узла, содержат элементы, следующие за родительским элементом. Слова в дереве на рис. 17.12 хранятся именно таким образом. Верхняя часть дерева, в отличие от ботаники, называется *корнем*. Дерево представляет собой *иерархическую* организацию данных, то есть данные организованы по рангам, или уровням, причем в общем случае каждому рангу соответствуют ранги, расположенные над и под ним. Если дерево бинарного поиска полностью заполнено, каждый уровень содержит вдвое больше узлов, чем уровень, расположенный над ним.

Каждый узел в дереве бинарного поиска и сам является корнем узлов, исходящих из него, что превращает узел и его "побеги" в *поддерево*. Например, на рис. 17.12 узлы, содержащие слова *fate*, *carpet* и *llama*, образуют левое поддерево всего дерева, а слово *voyage* — правое поддерево поддерева *style-plenum-voyage*.

Предположим, что в таком дереве необходимо найти элемент — назовем его *целью*. Если элемент предшествует корневому, поиск необходимо выполнять только в левой половине дерева, а если цель следует за корневым элементом, поиск нужно выполнять только в правом поддереве корневого узла. Таким образом, одна операция сравнения исключает из поиска половину дерева. Предположим, что мы выполняем поиск в левой половине. Это означает сравнение цели с элементом в левом дочернем узле. Если цель предшествует элементу левого дочернего узла, поиск необходимо выполнять только в левой половине дочерних узлов и так далее. Как и при бинарном поиске, каждая операция сравнения уменьшает количество возможных соответствий в два раза.

Применим этот метод для выяснения того, присутствует ли слово *puppy* в дереве, показанном на рис. 17.12. Сравнивая слово *puppy* со словом *melon* (элементом корневого узла) мы видим, что слово *puppy*, если оно присутствует, должно располагаться в

правой половине дерева. Поэтому мы переходим к правому дочернему узлу корневого узла и сравниваем слово *puppy* со словом *style*. В данном случае целевой элемент предшествует элементу узла, поэтому следует двигаться по связи левого узла. В нем расположено слово *plenum*, которое предшествует слову *puppy*. Теперь необходимо следовать правой ветвью, но она пуста. Таким образом, три операции сравнения показывают, что слово *puppy* в дереве отсутствует.

Таким образом, дерево бинарного поиска сочетает преимущества связанной структуры с эффективностью бинарного поиска. С точки зрения программирования формирование дерева — более трудоемкий процесс, нежели создание связного списка. Теперь создадим бинарное дерево и, наконец, проект ADT.

Тип ADT бинарного дерева

Как обычно, начнем с общего определения бинарного дерева. Данное конкретное определение предполагает, что дерево не содержит дублированных элементов. Многие его операции совпадают с операциями со списками. Различие состоит в иерархической организации данных. Неформальное определение этого типа ADT выглядит следующим образом:

Название типа:	Дерево бинарного поиска
Свойства типа:	Бинарное дерево является либо пустым набором узлов (пустым деревом), либо набором узлов, один из которых обозначает корень.
	Каждый узел имеет ровно два дерева, называемые *левым поддеревом* и *правым поддеревом*, исходящие из него.
	Каждое поддерево само является бинарным деревом, что включает возможность быть пустым деревом.
	Дерево бинарного поиска — упорядоченное бинарное дерево, где каждый узел содержит элемент, в котором все элементы в левом поддереве предшествуют корневому элементу, и в котором корневой элемент предшествует всем элементам правого поддерева.
Операции типа:	Инициализация дерева пустым значением.
	Определение того, является ли дерево пустым.
	Определение того, является ли дерево полным.
	Определение количества элементов в дереве
	Добавление элемента в дерево.
	Удаление элемента из дерева.
	Поиск элемента в дереве.
	Посещение каждого элемента в дереве.
	Опустошение дерева.

Интерфейс дерева бинарного поиска

В принципе, дерево бинарного поиска можно реализовать различными способами. Его можно даже реализовать в виде массива, манипулируя индексами массива. Но наиболее простой способ реализации дерева бинарного поиска предполагает использование динамически зарезервированных узлов, связанных между собой указателями.

Поэтому мы начнем с определений, подобных представленному ниже:

```
typedef SOMETHING Item;

typedef struct node
{
   Item item;
   struct node * left;
   struct node * right;
} Node;

typedef struct tree
{
   Node * root;
   int size;
} Tree;
```

Каждый узел содержит элемент, указатель на левый дочерний узел и указатель на правый дочерний узел. Структуру `Tree` можно было бы определить в качестве типа указателя на `Node`, поскольку для получения доступа ко всему дереву достаточно знать только местоположение корневого узла. Однако использование структуры с элементом размера упрощает отслеживание размера дерева.

Мы разработаем программу ведения реестра домашних животных клуба Nelville Pet Club, в котором каждый элемент хранит кличку домашнего животного и его вид. С учетом этого можно определить интерфейс, представленный в листинге 17.10. Мы ограничим размер дерева до 10. Небольшой размер облегчает тестирование программы при заполнении дерева. При необходимости значение `MAXITEMS` всегда можно увеличить.

Листинг 17.10. Файл заголовка интерфейса `tree.h`

```
/* tree.h -- дерево бинарного поиска */
/* в этом дереве никакие дублированные элементы не допускаются */
#ifndef _TREE_H_
#define _TREE_H_
#include <stdbool.h>

/* переопределение типа Item */
typedef struct item
{
   char petname[20];
   char petkind[20];
} Item;

#define MAXITEMS 10

typedef struct node
{
   Item item;
   struct node * left;  /* указатель на левую ветвь  */
   struct node * right; /* указатель на правую ветвь */
} Node;

typedef struct tree
{
```

```
   Node * root; /* указатель на корень дерева    */
   int size;    /* количество элементов в дереве */
} Tree;

/* прототипы функций */

/* операция:          инициализация дерева пустым значением       */
/* начальные условия: ptree указывает на дерево                   */
/* конечные условия:  дерево инициализировано пустым значением    */
void InitializeTree(Tree * ptree);

/* операция:          определение того, является ли дерево пустым  */
/* начальные условия: ptree указывает на дерево                   */
/* конечные условия:  функция возвращает значение true, если дерево*/
/*                    пусто, и false - в противном случае         */
bool TreeIsEmpty(const Tree * ptree);

/* операция:          определение того, является ли дерево полным  */
/* начальные условия: ptree указывает на дерево                   */
/* конечные условия:  функция возвращает значение true, если дерево*/
/*                    полно, и false - в противном случае         */
bool TreeIsFull(const Tree * ptree);

/* операция:          определение количества элементов в дереве    */
/* начальные условия: ptree указывает на дерево                   */
/* конечные условия:  функция возвращает число элементов в дереве  */
int TreeItemCount(const Tree * ptree);

/* операция:          добавление элемента в дерево                */
/* начальные условия: pi - адрес добавляемого элемента            */
/*                    ptree указывает на инициализированное дерево */
/* конечные условия:  если возможно, функция добавляет элемент в   */
/*                    дерево и возвращает значение true;          */
/*                    в противном случае она возвращает значение   */
/*                    false                                       */
bool AddItem(const Item * pi, Tree * ptree);

/* операция:          поиск элемента в дереве                     */
/* начальные условия: pi указывает на элемент                     */
/*                    ptree указывает на инициализированное дерево */
/* конечные условия:  функция возвращает значение true, если      */
/*                    элемент присутствует в дереве, и значение    */
/*                    false - в противном случае                  */
bool InTree(const Item * pi, const Tree * ptree);

/* операция:          удаление элемента из дерева                 */
/* начальные условия: pi - адрес удаляемого элемента              */
/*                    ptree указывает на инициализированное дерево */
/* конечные условия:  если возможно, функция удаляет элемент      */
/*                    из дерева и возвращает значение true;       */
/*                    в противном случае функция возвращает       */
/*                    значение false                              */
bool DeleteItem(const Item * pi, Tree * ptree);

/* операция:          применение функции к каждому элементу в дереве*/
/* начальные условия: ptree указывает на дерево                    */
```

```
/*                      pfun указывает на функцию, которая принимает */
/*                      аргумент Item и не имеет возвращаемого       */
/*                      значения                                     */
/* конечные условия:  функция, указанная pfun, выполняется один раз*/
/*                      для каждого элемента в дереве                */
void Traverse (const Tree * ptree, void (* pfun)(Item item));

/* операция:           удаление всех элементов из дерева            */
/* начальные условия: ptree указывает на инициализированное дерево */
/* конечные условия:  дерево пусто                                  */
void DeleteAll(Tree * ptree);

#endif
```

Реализация бинарного дерева

Теперь приступим к реализации множества функций, описанных в файле `tree.h`. Функции `InitializeTree()`, `EmptyTree()`, `FullTree()` и `TreeItems()` достаточно просты и работают подобно их аналогам в абстрактных типах данных списка и очереди. Поэтому уделим основное внимание остальным функциям.

Добавление элемента

При добавлении элемента в дерево вначале следует проверить наличие в дереве свободного места для нового узла. Затем, поскольку дерево бинарного поиска определено так, что не может содержать дублированных элементов, необходимо проверить, не присутствует ли данный элемент в дереве. Если новый элемент удовлетворяет этим двум начальным условиям, потребуется создать новый узел, скопировать элемент в него и установить левый и правый указатели узла в значения `NULL`. Это говорит об отсутствии дочерних узлов у дочернего узла. Затем следует обновить элемент `size` структуры `Tree` с целью отражения добавления нового элемента. Далее необходимо выяснить, в какую позицию дерева поместить новый узел. Если дерево пусто, корневой указатель нужно установить так, чтобы он указывал на новый узел. В противном случае потребуется просмотреть дерево, чтобы найти в нем место для добавления узла. Функция `AddItem()` выполняет эти действия, передавая часть работы функциям, которые еще не определены: `SeekItem()`, `MakeNode()` и `AddNode()`.

```
bool AddItem(const Item * pi, Tree * ptree)
{
   Node * new_node;

   if (TreeIsFull(ptree))
   {
      fprintf(stderr,"Дерево полно\n");
      return false;          /* преждевременный возврат        */
   }
   if (SeekItem(pi, ptree).child != NULL)
   {
      fprintf(stderr,"Попытка добавления дублированного элемента\n");
      return false;          /* преждевременный возврат        */
   }
```

```
    new_node = MakeNode(pi);   /* указывает на новый узел         */
    if (new_node == NULL)
    {
        fprintf(stderr,"Не удалось создать узел\n");
        return false;          /* преждевременный возврат         */
    }
    /* успешное создание нового узла */
    ptree->size++;

    if (ptree->root == NULL)   /* случай 1: дерево пусто          */
        ptree->root = new_node; /* новый узел - корень дерева     */
    else                       /* случай 2: не пусто              */
        AddNode(new_node,ptree->root); /* добавление узла к дереву */

    return true;               /* успешный возврат                */
}
```

Функции `SeekItem()`, `MakeNode()` и `AddNode()` не являются частью общедоступного интерфейса типа `Tree`. Напротив, они представляют собой статические функции, скрытые в файле `tree.c`, которые имеют дело с такими нюансами реализации, как узлы, указатели и структуры, не относящимися к общедоступному интерфейсу.

Функция `MakeNode()` достаточно проста. Она выполняет динамическое резервирование памяти и инициализацию узла. Аргумент функции — указатель на новый элемент, а ее возвращаемое значение — указатель на новый узел. Вспомните, что функция `malloc()` возвращает нулевой указатель, если она не может выполнить запрошенное резервирование памяти. Функция `MakeNode()` инициализирует новый узел только в случае успешного резервирования памяти. Код функции `MakeNode()` имеет следующий вид:

```
static Node * MakeNode(const Item * pi)
{
    Node * new_node;

    new_node = (Node *) malloc(sizeof(Node));
    if (new_node != NULL)
    {
        new_node->item = *pi;
        new_node->left = NULL;
        new_node->right = NULL;
    }

    return new_node;
}
```

Функция `AddNode()` — вторая по сложности в пакете дерева бинарного поиска. Она должна определить место, в которое должен быть помещен новый узел, а затем добавить его. В частности, ей необходимо сравнить новый элемент с корневым для выяснения того, в какое поддерево должен быть помещен новый элемент — в левое или правое. Если бы элемент был строкой, для выполнения сравнений можно было бы использовать функцию `strcmp()`. Но элемент является структурой, которая содержит две строки. Поэтому для выполнения сравнений придется предусмотреть собственные функции. Функция `ToLeft()`, которая будет определена позже, возвращает значение `True`, если новый элемент должен быть помещен в левое поддерево, а функция `ToRight()` возвращает значение `True`, если новый элемент должен быть помещен в

правое поддерево. Эти две функции — аналоги операций < и > соответственно. Предположим, что новый элемент должен быть помещен в левое поддерево. Оно может быть пустым. В этом случае функция просто определяет указатель левого дочернего узла так, чтобы он указывал на новый узел.

А что произойдет, если левое дерево не пусто? Тогда функция должна сравнить новый элемент с элементом в левом дочернем узле с целью определения того, должен ли новый узел быть помещен в левое поддерево дочернего узла или же в правое поддерево. Этот процесс должен продолжаться до тех пор, пока функция не достигнет пустого поддерева, в которое она и сможет добавить новый узел. Один из возможных способов реализации этого поиска — применение рекурсии, то есть вызов функции `AddNode()` для дочернего, а не корневого узла. Рекурсивная последовательность вызовов функции завершается, когда она приводит к пустому левому или правому поддереву — когда результат выражения `root->left` или `root->right` равен `NULL`. Следует помнить, что `root` — это указатель на верхушку текущего поддерева, поэтому он указывает на новое (расположенное на более низком уровне) поддерево каждого рекурсивного вызова. (Рекурсия подробно рассмотрена в 9 главе.)

```
static void AddNode (Node * new_node, Node * root)
{
   if (ToLeft(&new_node->item, &root->item))
   {
      if (root->left == NULL)      /* пустое поддерево */
           root->left = new_node; /* поэтому узел должен быть
                                     добавлен сюда */
      else
           AddNode(new_node, root->left);  /* в противном случае
                                     необходимо обработать поддерево*/
   }
   else if (ToRight(&new_node->item, &root->item))
   {
      if (root->right == NULL)
           root->right = new_node;
      else
           AddNode(new_node, root->right);
   }
   else /* дубликаты не допускаются */
   {
      fprintf(stderr,"Ошибка определения места вставки в функции
AddNode()\n");
      exit(1);
   }
}
```

Функции `ToLeft()` и `ToRight()` зависят от сущности типа `Item`. Члены клуба Nerfville Pet Club будут упорядочены в алфавитном порядке по кличкам. Если двое животных имеют одинаковые клички, они должны быть упорядочены по виду. Если их вид также совпадает, то два элемента являются дубликатами, что недопустимо в базовом дереве поиска. Вспомните, что функция `strcmp()` из стандартной библиотеки C возвращает отрицательное число, если строка, представленная ее первым аргументом, предшествует второй строке, ноль, если обе строки совпадают, и положительное

число, если первая строка следует за второй. Функция ToRight() содержит аналогичный код. Использование этих двух функций вместо выполнения сравнений непосредственно в функции AddNode() упрощает адаптацию кода к новым требованиям. Вместо того чтобы переписывать функцию AddNode(), когда требуется другая форма сравнения, достаточно изменить функции ToLeft() и ToRight().

```
static bool ToLeft(const Item * i1, const Item * i2)
{
   int comp1;

   if ((comp1 = strcmp(i1->petname, i2->petname)) < 0)
      return true;
   else if (comp1 == 0 &&
            strcmp(i1->petkind, i2->petkind) < 0 )
      return true;
   else
      return false;
}
```

Поиск элемента

Три функции интерфейса: AddItem(), InTree() и DeleteItem() требуют выполнения поиска конкретного элемента в дереве. В рассматриваемой реализации для этого использована функция SeekItem(). Функция DeleteItem() имеет дополнительное требование: она должна знать родительский узел удаленного элемента, чтобы дочерний указатель родительского узла можно было дополнить при удалении дочернего элемента. Поэтому функция SeekItem() была создана так, чтобы возвращать структуру, содержащую два указателя: один, указывающий на узел, который содержит искомый элемент (NULL, если элемент не найден), и другой, указывающий на родительский узел (NULL, если данный узел — корневой и не имеет родительского узла). Эта структура определена следующим образом:

```
typedef struct pair {
   Node * parent;
   Node * child;
} Pair;
```

Функцию SeekItem() можно реализовать рекурсивно. Однако чтобы ознакомить вас с различными технологиями программирования, для нисходящего обхода дерева мы используем цикл while. Подобно функции AddNode(), функция SeekItem() использует для обхода дерева функции ToLeft() и ToRight().

Вначале функция SeekItem() устанавливает указатель look.child так, чтобы он указывал на корень дерева, а затем, по мере трассировки пути к возможному местонахождению элемента, переустанавливает этот указатель на последующие поддеревья. Одновременно указатель look.parent устанавливается для указания на последующие родительские узлы. Если никакой подходящий элемент не найден, значение указателя look.child будет равно NULL. Если искомый элемент находится в корневом узле, значение указателя look.parent равно NULL, поскольку корневой узел не имеет родительского узла.

Код функции SeekItem() выглядит следующим образом:

```
static Pair SeekItem(const Item * pi, const Tree * ptree)
{
   Pair look;
   look.parent = NULL;
   look.child = ptree->root;
   if (look.child == NULL)
      return look;   /* преждевременный возврат */
   while (look.child != NULL)
   {
      if (ToLeft(pi, &(look.child->item)))
      {
          look.parent = look.child;
          look.child = look.child->left;
      }
      else if (ToRight(pi, &(look.child->item)))
      {
          look.parent = look.child;
          look.child = look.child->right;
      }
      else       /* если элемент не расположен ни слева, ни справа,
                    он должен быть таким же */
          break; /* look.child - адрес узла, содержащего элемент */
   }
   return look;   /* успешный возврат */
}
```

Обратите внимание, что поскольку функция `SeekItem()` возвращает структуру, ее можно использовать с оператором принадлежности к структуре. Например, функция `AddItem()` использует следующий код:

```
if (SeekItem(pi, ptree).child != NULL)
```

После того, как функция `SeekItem()` создана, написание кода функции `InTree()` общедоступного интерфейса не представляет особой сложности:

```
bool InTree(const Item * pi, const Tree * ptree)
{
   return (SeekItem(pi, ptree).child == NULL) ? false : true;
}
```

Соображения по поводу удаления элемента

Удаление элемента — наиболее трудная задача, поскольку необходимо заново соединить остающиеся поддеревья для формирования допустимого дерева. Прежде чем пытаться программировать решение этой задачи, имеет смысл представить действия, которые нужно выполнить, визуально. Простейший случай показан на рис. 17.13. В этом примере узел, который должен быть удален, не имеет дочерних узлов. Такой узел называют *листом*. В данном случае нужно лишь переустановить указатель в родительском узле в значение `NULL` и использовать функцию `free()` для освобождения памяти, использованной удаленным узлом.

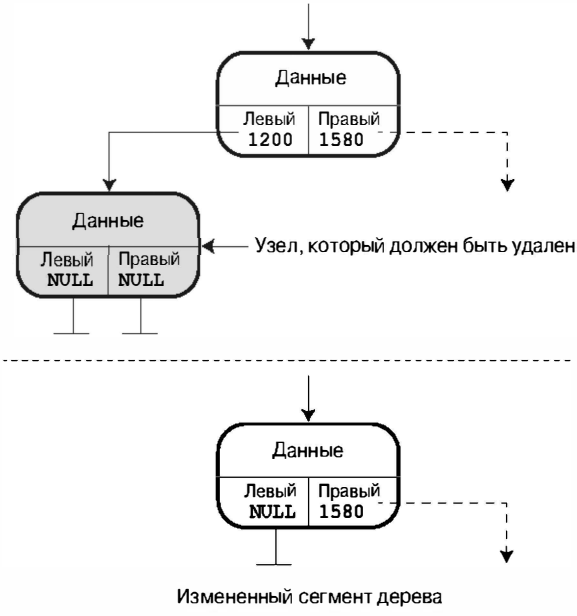


Рис. 17.13. *Удаление листа*

Следующая по сложности задача — удаление узла с одним дочерним узлом. Удаление узла ведет к отделению дочернего поддерева от остального дерева. Для исправления этой ситуации адрес дочернего поддерева должен быть сохранен в родительском узле в той позиции, которая ранее была занята адресом удаленного узла (рис. 17.14).

И последний случай — удаление узла с двумя поддеревьями. Одно дерево, например левое, может быть присоединено к тому узлу, к которому вначале был присоединен удаленный узел. Но куда следует поместить остающееся поддерево? Следует помнить об основном принципе формирования структуры дерева. Каждый элемент в левом поддереве предшествует элементу в родительском узле, а каждый элемент в правом поддереве следует за элементом в родительском узле. Это означает, что каждый элемент в правом поддереве расположен в структуре дальше любого элемента левого поддерева. Кроме того, поскольку правое поддерево ранее было частью поддерева, начинающегося с удаленного узла, каждый элемент в правом поддереве предшествует родительскому узлу удаленного узла. Представьте себе спуск по дереву в поисках позиции для помещения начала правого поддерева. Он предшествует родительскому узлу, поэтому далее необходимо следовать вниз по левому поддереву. Однако начало поддерева должно быть расположено после всех элементов в левом поддереве, поэтому необходимо следовать правой ветвью левого поддерева и выяснить, имеется ли в ней место для нового узла. Если нет, потребуется продолжать спуск по правой стороне левого поддерева до тех пор, пока свободное место не будет найдено. Этот подход проиллюстрирован на рис. 17.15.

Удаление узла

Теперь можно приступать к планированию необходимых функций, предварительно разделив задачу на две. Первая из них — связывание конкретного элемента с узлом, который должен быть удален, а вторая — действительное удаление узла.

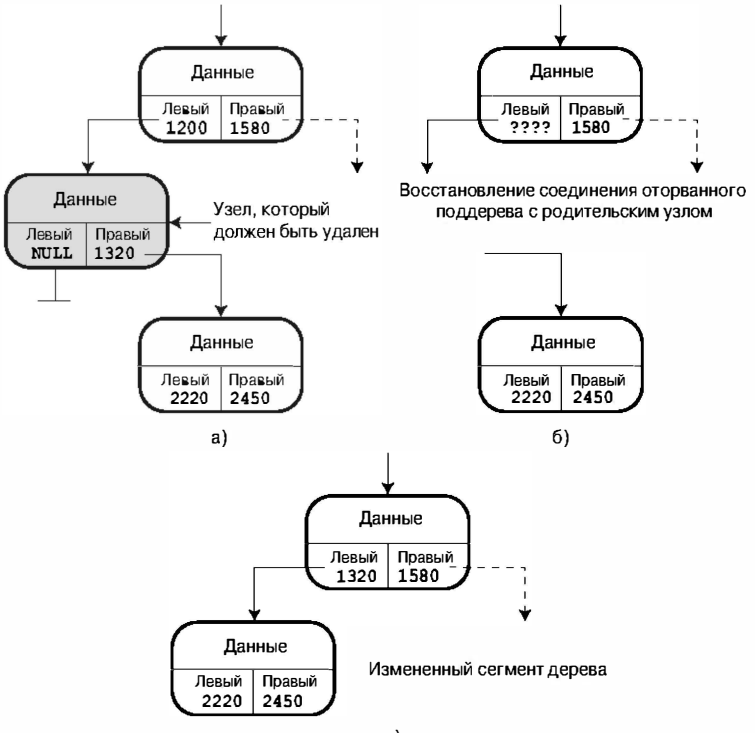


Рис. 17.14. *Удаление узла с одним дочерним узлом*

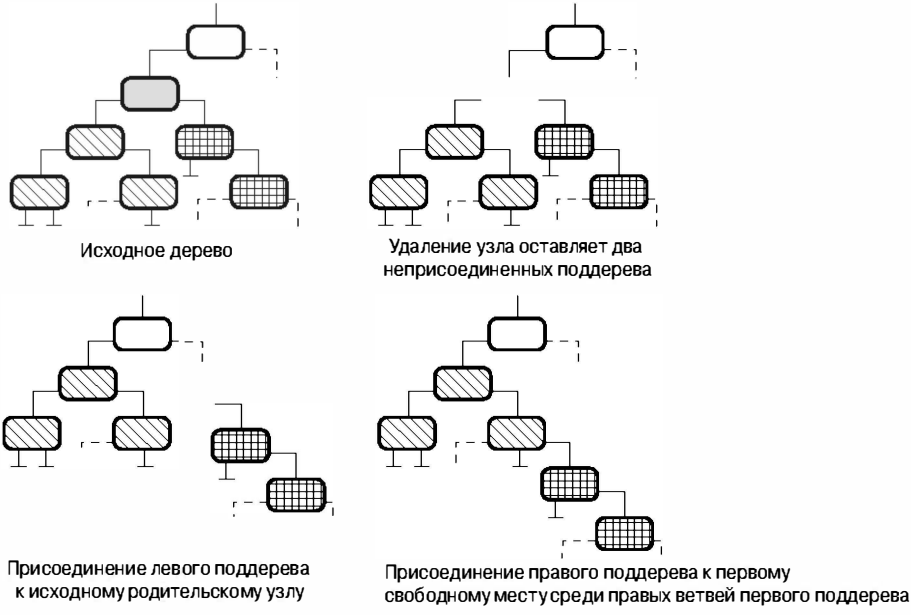


Рис. 17.15. *Удаление узла с двумя дочерними узлами*

Следует отметить, что во всех случаях требуется изменение указателя в родительском узле, а это приводит к двум важным последствиям:

- Программа должна определять родительский узел узла, который должен быть удален.
- Для изменения указателя код должен передавать *адрес* этого указателя на функцию удаления.

К первому моменту мы вернемся несколько позже. А пока проанализируем второй момент. Указатель, который нужно изменять, имеет тип `Node *`, то есть является указателем на `Node`. Поскольку аргумент функции — адрес этого указателя, типом аргумента будет `Node **` или указатель на указатель на `Node`. При условии наличия доступного соответствующего адреса функция удаления может быть реализована следующим образом:

```
static void DeleteNode(Node **ptr)
/* ptr -- адрес родительского элемента, указывающего на целевой узел */
{
   Node * temp;

   puts((*ptr)->item.petname);
   if ( (*ptr)->left == NULL)
   {
      temp = *ptr;
      *ptr = (*ptr)->right;
      free(temp);
   }
   else if ( (*ptr)->right == NULL)
   {
      temp = *ptr;
      *ptr = (*ptr)->left;
      free(temp);
   }
   else /* удаленный узел имеет два дочерних узла */
   {
      /* выяснение места присоединения правого поддерева */
      for (temp = (*ptr)->left; temp->right != NULL;
               temp = temp->right)
          continue;
      temp->right = (*ptr)->right;
      temp = *ptr;
      *ptr =(*ptr)->left;
      free(temp);
   }
}
```

Эта функция выполняет явную обработку трех случаев: узла без левого дочернего узла, узла без правого дочернего узла и узла с двумя дочерними узлами. Узел без дочерних узлов можно считать особым случаем узла без левого дочернего узла. Если узел не имеет левого дочернего узла, код присваивает адрес правого дочернего узла родительскому указателю. Но если узел не имеет и правого узла, значением этого указателя будет `NULL`, которое полностью соответствует случаю узла без дочерних узлов.

Обратите внимание, что для отслеживания адреса удаленного узла код использует временный указатель. После того как родительский указатель (*ptr) переустановлен, программа утрачивала бы информацию о местоположении удаленного узла, но эта информация требуется функции free(). Поэтому программа сохраняет исходное значение *ptr в переменной temp, а затем использует ее для освобождения памяти, использованной удаленным узлом.

Код обработки случая узла с двумя дочерними узлами вначале использует указатель temp в цикле for для поиска свободного места в правой части левого поддерева. После его нахождения программа присоединяет к нему правое поддерево. Затем она снова использует указатель temp для отслеживания местоположения удаленного узла. И, наконец, она присоединяет левое поддерево к родительскому узлу, после чего освобождает узел, указанный переменной temp.

Обратите внимание, что поскольку ptr имеет тип Node **, то *ptr — Node *, что делает его совпадающим с типом указателя temp.

Удаление элемента

Осталась одна нерешенная задача — связывание узла с конкретным элементом. Для ее решения можно использовать функцию SeekItem(). Вспомните, что она возвращает структуру, содержащую указатель на родительский узел и указатель на узел, содержащий элемент. Следовательно, указатель родительского узла можно использовать для получения соответствующего адреса, который должен быть передан функции DeleteNode(). Представленная ниже функция DeleteItem() соответствует этой концепции:

```
bool DeleteItem(const Item * pi, Tree * ptree)
{
   Pair look;

   look = SeekItem(pi, ptree);
   if (look.child == NULL)
      return false;

   if (look.parent == NULL) /* удаление корневого элемента */
      DeleteNode(&ptree->root);
   else if (look.parent->left == look.child)
      DeleteNode(&look.parent->left);
   else
      DeleteNode(&look.parent->right);
   ptree->size--;

   return true;
}
```

Возвращаемое значение функции SeekItem() присваивается переменной структуры look. Если значение look.child равно NULL, поиск элемента неудачен, и функция DeleteItem() завершает свое выполнение, возвращая значение false. Если элемент Item найден, функция обрабатывает три случая. Во-первых, значение NULL переменной look.parent означает, что элемент был найден в корневом узле. В этом случае никакого родительского узла, который нужно было бы обновить, не существует. Вместо этого программа должна обновить корневой указатель структуры Tree. Поэтому функция передает адрес этого указателя функции DeleteNode(). В противном случае

программа определяет, в каком дочернем узле родительского узла расположен удаляемый узел — левом или правом, а затем передает адрес соответствующего указателя.

Обратите внимание, что функция общедоступного интерфейса (`DeleteItem()`) манипулирует понятиями, близкими конечному пользователю (элементами и деревьями), а скрытая функция `DeleteNode()` выполняет "махинации" с указателями.

Обход дерева

Обход дерева — более сложная задача, чем обход связного списка, поскольку каждый узел имеет две ветви, которые нужно отследить. Эта природа ветвлений превращает рекурсию типа "разделяй и властвуй" (см. главу 9) в естественный метод решения задачи. В каждом узле функция должна выполнить следующие действия:

- Обработать элемент в данном узле.
- Обработать левое поддерево (рекурсивный вызов).
- Обработать правое поддерево (рекурсивный вызов).

Этот процесс можно разделить на две функции: `Traverdse()` и `InOrder()`. Обратите внимание, что функция `InOrder()` обрабатывает левое поддерево, затем элемент и, наконец — правое поддерево. Этот порядок обработки приводит к обходу дерева в алфавитном порядке. Если вы располагаете временем, можете самостоятельно проанализировать, к чему приводит другой порядок обработки, такой как "элемент — левое поддерево — правое поддерево" и "левое поддерево — правое поддерево — элемент".

```
void Traverse (const Tree * ptree, void (* pfun)(Item item))
{
   if (ptree != NULL)
      InOrder(ptree->root, pfun);
}

static void InOrder(const Node * root, void (* pfun)(Item item))
{
   if (root != NULL)
   {
      InOrder(root->left, pfun);
      (*pfun)(root->item);
      InOrder(root->right, pfun);
   }
}
```

Опустошение дерева

В основном, процесс опустошения дерева совпадает с процессом его обхода. То есть код должен посетить каждый узел и применить к нему функцию `free()`. Код должен также переустановить элементы структуры `Tree` для указания пустого дерева `Tree`. Функция выполняет обработку структуры `Tree`, а задачу освобождения памяти передает функции `DeleteAllNodes()`.

Последняя функция аналогична функции `InOrder()`. Она сохраняет значение указателя `root->right`, чтобы он оставался доступным после освобождения корня.

Код этих двух функций выглядит следующим образом:

```
void DeleteAll(Tree * ptree)
{
   if (ptree != NULL)
      DeleteAllNodes(ptree->root);
   ptree->root = NULL;
   ptree->size = 0;
}

static void DeleteAllNodes(Node * root)
{
   Node * pright;

   if (root != NULL)
   {
      pright = root->right;
      DeleteAllNodes(root->left);
      free(root);
      DeleteAllNodes(pright);
   }
}
```

Полный пакет

Полный код файла `tree.c` представлен в листинге 17.11. Файлы `tree.h` и `tree.c` образуют программный пакет дерева.

Листинг 17.11. Файл реализации `tree.c`

```
/* tree.c -- функции поддержки дерева */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

/* локальный тип данных */
typedef struct pair {
   Node * parent;
   Node * child;
} Pair;

/* прототипы локальных функций */
static Node * MakeNode(const Item * pi);
static bool ToLeft(const Item * i1, const Item * i2);
static bool ToRight(const Item * i1, const Item * i2);
static void AddNode (Node * new_node, Node * root);
static void InOrder(const Node * root, void (* pfun)(Item item));
static Pair SeekItem(const Item * pi, const Tree * ptree);
static void DeleteNode(Node **ptr);
static void DeleteAllNodes(Node * ptr);

/* определения функций */
void InitializeTree(Tree * ptree)
{
   ptree->root = NULL;
   ptree->size = 0;
}
```

```
bool TreeIsEmpty(const Tree * ptree)
{
   if (ptree->root == NULL)
      return true;
   else
      return false;
   }
bool TreeIsFull(const Tree * ptree)
{
   if (ptree->size == MAXITEMS)
      return true;
   else
   return false;
}
int TreeItemCount(const Tree * ptree)
{
   return ptree->size;
}
bool AddItem(const Item * pi, Tree * ptree)
{
   Node * new_node;
   if (TreeIsFull(ptree))
   {
      fprintf(stderr,"Дерево полно\n");
      return false; /* преждевременный возврат */
   }
   if (SeekItem(pi, ptree).child != NULL)
   {
      fprintf(stderr, "Попытка добавления дублированного элемента\n");
      return false; /* преждевременный возврат */
   }
   new_node = MakeNode(pi); /* указывает на новый узел */
   if (new_node == NULL)
   {
      fprintf(stderr, "Не удалось создать узел\n");
      return false; /* преждевременный возврат */
   }
   /* новый узел успешно создан */
   ptree->size++;
   if (ptree->root == NULL)  /* случай 1: дерево пустое */
      ptree->root = new_node; /* новый узел - корень дерева */
   else                      /* случай 2: не пустое */
      AddNode(new_node,ptree->root); /* добавление узла в дерево */
      return true; /* успешный возврат */
}
bool InTree(const Item * pi, const Tree * ptree)
{
   return (SeekItem(pi, ptree).child == NULL) ? false : true;
}
```

```
bool DeleteItem(const Item * pi, Tree * ptree)
{
   Pair look;
   look = SeekItem(pi, ptree);
   if (look.child == NULL)
      return false;
   if (look.parent == NULL) /* удаление корневого элемента */
      DeleteNode(&ptree->root);
   else if (look.parent->left == look.child)
      DeleteNode(&look.parent->left);
   else
      DeleteNode(&look.parent->right);
   ptree->size--;

   return true;
}

void Traverse (const Tree * ptree, void (* pfun)(Item item))
{
   if (ptree != NULL)
   InOrder(ptree->root, pfun);
}

void DeleteAll(Tree * ptree)
{
   if (ptree != NULL)
      DeleteAllNodes(ptree->root);
   ptree->root = NULL;
   ptree->size = 0;
}

/* локальные функции */
static void InOrder(const Node * root, void (* pfun)(Item item))
{
   if (root != NULL)
   {
      InOrder(root->left, pfun);
      (*pfun)(root->item);
      InOrder(root->right, pfun);
   }
}

static void DeleteAllNodes(Node * root)
{
   Node * pright;

   if (root != NULL)
   {
      pright = root->right;
      DeleteAllNodes(root->left);
      free(root);
      DeleteAllNodes(pright);
   }
}
```

```
static void AddNode (Node * new_node, Node * root)
{
   if (ToLeft(&new_node->item, &root->item))
   {
      if (root->left == NULL)  /* пустое поддерево */
        root->left = new_node; /* поэтому узел добавляется сюда */
      else
        AddNode(new_node, root->left);/*иначе выполняется обработка поддерева*/
   }
   else if (ToRight(&new_node->item, &root->item))
   {
      if (root->right == NULL)
         root->right = new_node;
      else
         AddNode(new_node, root->right);
   }
   else /* дубликаты не допускаются */
   {
      fprintf(stderr,"Ошибка определения местоположения в функции AddNode()\n");
      exit(1);
   }
}
static bool ToLeft(const Item * i1, const Item * i2)
{
   int comp1;
   if ((comp1 = strcmp(i1->petname, i2->petname)) < 0)
      return true;
   else if (comp1 == 0 &&
        strcmp(i1->petkind, i2->petkind) < 0 )
      return true;
   else
      return false;
}
static bool ToRight(const Item * i1, const Item * i2)
{
   int comp1;
   if ((comp1 = strcmp(i1->petname, i2->petname)) > 0)
      return true;
   else if (comp1 == 0 &&
        strcmp(i1->petkind, i2->petkind) > 0 )
      return true;
   else
      return false;
}
static Node * MakeNode(const Item * pi)
{
   Node * new_node;
   new_node = (Node *) malloc(sizeof(Node));
```

```
   if (new_node != NULL)
   {
      new_node->item = *pi;
      new_node->left = NULL;
      new_node->right = NULL;
   }
   return new_node;
}
static Pair SeekItem(const Item * pi, const Tree * ptree)
{
   Pair look;
   look.parent = NULL;
   look.child = ptree->root;
   if (look.child == NULL)
      return look; /* преждевременный возврат */
   while (look.child != NULL)
   {
      if (ToLeft(pi, &(look.child->item)))
      {
         look.parent = look.child;
         look.child = look.child->left;
      }
      else if (ToRight(pi, &(look.child->item)))
      {
         look.parent = look.child;
         look.child = look.child->right;
      }
      else      /* если узел не является ни левым, ни правым,
                   то должен совпадать с данным */
         break; /* look.child - адрес узла с элементом */
   }
   return look; /* успешный возврат */
}
static void DeleteNode(Node **ptr)
/* ptr - адрес родительского элемента, указывающего на целевой узел */
{
   Node * temp;
   puts((*ptr)->item.petname);
   if ( (*ptr)->left == NULL)
   {
      temp = *ptr;
      *ptr = (*ptr)->right;
      free(temp);
   }
   else if ( (*ptr)->right == NULL)
   {
      temp = *ptr;
      *ptr = (*ptr)->left;
```

```
      free(temp);
   }
   else /* удаленный узел имеет два дочерних узла */
   {
      /* выяснение места присоединения правого поддерева */
      for (temp = (*ptr)->left; temp->right != NULL;
            temp = temp->right)
         continue;
      temp->right = (*ptr)->right;
      temp = *ptr;
      *ptr =(*ptr)->left;
      free(temp);
   }
}
```

Тестирование дерева

Теперь, когда реализации интерфейса и функций созданы, применим их. Программа, код которой показан в листинге 17.12, использует меню с пунктами для добавления домашних животных в реестр членов клуба, вывода списка элементов клуба, вывода количества элементов, проверки членства и выхода из программы. Краткая функция `main()` выполняет основные действия по выводу результатов работы программы. Большая часть работы выполняется функциями поддержки.

Листинг 17.12. Программа `petclub.c`

```
/* petclub.c -- использование дерева бинарного поиска */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "tree.h"

char menu(void);
void addpet(Tree * pt);
void droppet(Tree * pt);
void showpets(const Tree * pt);
void findpet(const Tree * pt);
void printitem(Item item);
void uppercase(char * str);

int main(void)
{
   Tree pets;
   char choice;

   InitializeTree(&pets);
   while ((choice = menu()) != 'q')
   {
      switch (choice)
      {
         case 'a' : addpet(&pets);
            break;
```

```
        case 'l' : showpets(&pets);
           break;
        case 'f' : findpet(&pets);
           break;
        case 'n' : printf("%d животные в клубе\n",
              TreeItemCount(&pets));
           break;
        case 'd' : droppet(&pets);
           break;
        default : puts("Ошибка переключения");
      }
   }
   DeleteAll(&pets);
   puts("Программа завершена.");
   return 0;
}
char menu(void)
{
   int ch;
   puts("Программа членства в клубе Nerfville Pet Club ");
   puts("Введите букву, соответствующую вашему выбору:");
   puts("a) добавление животного   l) вывод списка животных");
   puts("n) количество животных    f) поиск животных");
   puts("d) удаление животного     q) выход");
   while ((ch = getchar()) != EOF)
   {
      while (getchar() != '\n') /* отбрасывание остальной части строки */
         continue;
      ch = tolower(ch);
      if (strchr("alrfndq",ch) == NULL)
         puts("Пожалуйста, введите букву a, l, f, n, d или q:");
      else
         break;
   }
   if (ch == EOF) /* ввод символа EOF приводит к выходу из программы */
      ch = 'q';
   return ch;
}
void addpet(Tree * pt)
{
   Item temp;
   if (TreeIsFull(pt))
      puts("Вакансии в клубе отсутствуют!");
   else
   {
      puts("Пожалуйста, введите кличку животного:");
      gets(temp.petname);
      puts("Теперь введите вид животного:");
```

```
      gets(temp.petkind);
      uppercase(temp.petname);
      uppercase(temp.petkind);
      AddItem(&temp, pt);
   }
}

void showpets(const Tree * pt)
{
   if (TreeIsEmpty(pt))
      puts("Записи отсутствуют!");
   else
      Traverse(pt, printitem);
}

void printitem(Item item)
{
   printf("Животное: %-19s Вид: %-19s\n", item.petname,
      item.petkind);
}

void findpet(const Tree * pt)
{
   Item temp;
   if (TreeIsEmpty(pt))
   {
      puts("Записи отсутствуют!");
      return; /* если дерево пусто, выход из функции */
   }

   puts("Пожалуйста, введите кличку животного, которое хотите найти:");
   gets(temp.petname);
   puts("Теперь введите вид животного:");
   gets(temp.petkind);
   uppercase(temp.petname);
   uppercase(temp.petkind);
   printf("%s животное %s ", temp.petname, temp.petkind);
   if (InTree(&temp, pt))
      printf("является элементом клуба.\n");
   else
      printf("не является элементом клуба.\n");
}

void droppet(Tree * pt)
{
   Item temp;

   if (TreeIsEmpty(pt))
   {
      puts("Записи отсутствуют!");
      return; /* если дерево пусто, выход из функции */
   }

   puts("Пожалуйста, введите кличку животного, которое нужно исключить из клуба:");
   gets(temp.petname);
```

```
   puts("Теперь введите вид животного:");
   gets(temp.petkind);
   uppercase(temp.petname);
   uppercase(temp.petkind);
   printf("%s животное %s ", temp.petname, temp.petkind);
   if (DeleteItem(&temp, pt))
      printf("исключено из клуба.\n");
   else
      printf("не является членом клуба.\n");
}
void uppercase(char * str)
{
   while (*str)
   {
      *str = toupper(*str);
      str++;
   }
}
```

Программа преобразует все буквы в прописные, поэтому *SNUFFY, Snuffy и snuffy* – считаются совпадающими кличками. Результаты примера выполнения программы приведены ниже:

```
Программа членства в клубе Nerfville Pet Club
Введите букву, соответствующую вашему выбору:
a) добавление животного   l) вывод списка животных
n) количество животных    f) поиск животных
d) удаление животного     q) выход
a
Пожалуйста, введите кличку животного:
Quincy
Теперь введите вид животного:
поросенок
Программа членства в клубе Nerfville Pet Club
Введите букву, соответствующую вашему выбору:
a) добавление животного   l) вывод списка животных
n) количество животных    f) поиск животных
d) удаление животного     q) выход
a
Пожалуйста, введите кличку животного:
Betty
Теперь введите вид животного:
Удав
Программа членства в клубе Nerfville Pet Club
Введите букву, соответствующую вашему выбору:
a) добавление животного   l) вывод списка животных
n) количество животных    f) поиск животных
d) удаление животного     q) выход
a
Пожалуйста, введите кличку животного:
Hiram Jinx
```

```
Теперь введите вид животного:
домашняя кошка
Программа членства в клубе Nerfville Pet Club
Введите букву, соответствующую вашему выбору:
a) добавление животного   l) вывод списка животных
n) количество животных    f) поиск животных
d) удаление животного     q) выход
n
3 животных в клубе
Программа членства в клубе Nerfville Pet Club
Введите букву, соответствующую вашему выбору:
a) добавление животного   l) вывод списка животных
n) количество животных    f) поиск животных
d) удаление животного     q) выход
l
Животное: BETTY Вид: УДАВ
Животное: HIRAM JINX Вид: ДОМАШНЯЯ КОШКА
Животное: QUINCY Kind: ПОРОСЕНОК
Программа членства в клубе Nerfville Pet Club
Введите букву, соответствующую вашему выбору:
a) добавление животного   l) вывод списка животных
n) количество животных    f) поиск животных
d) удаление животного     q) выход
q
Программа завершена.
```

Соображения по поводу дерева

Дерево бинарного поиска обладает рядом недостатков. Например, дерево бинарного поиска эффективно, только если оно полностью заполнено, или *сбалансировано*. Предположим, что вы сохраняете слова, вводимые в произвольном порядке. Существует вероятность, что дерево будет иметь весьма беспорядочный вид, как можно видеть на рис. 17.12. Теперь предположим, что данные вводятся в алфавитном порядке. В этом случае каждый новый узел будет добавляться справа от существующего, и дерево может выглядеть, как показано на рис. 17.16. Дерево на рис. 17.12 называют *сбалансированным*, а на рис. 17.16 — *несбалансированным*. Поиск в этом дереве ненамного эффективнее последовательного поиска в связном списке.

Один из способов предотвращения создания вытянутых деревьев — более тщательный подход к их построению. Если дерево или поддерево начинает становится слишком "перекошенным" в одну из сторон, необходимо реорганизовать узлы для восстановления баланса дерева. Аналогично реорганизация дерева может требоваться после удаления узлов. Русские математики Адельсон-Вельский и Ландис разработали алгоритм выполнения этой задачи. Деревья, построенные их методом, называют AVL-деревьями (от "Adel'son-Vel'skii, Landis"). Построение сбалансированного дерева занимает больше времени, что связано с необходимостью выполнения дополнительных действий по реструктуризации. Однако при этом можно получить максимальную, или близкую к максимальной, эффективность поиска.

Иногда может требоваться дерево бинарного поиска, которое допускает наличие дублированных элементов.

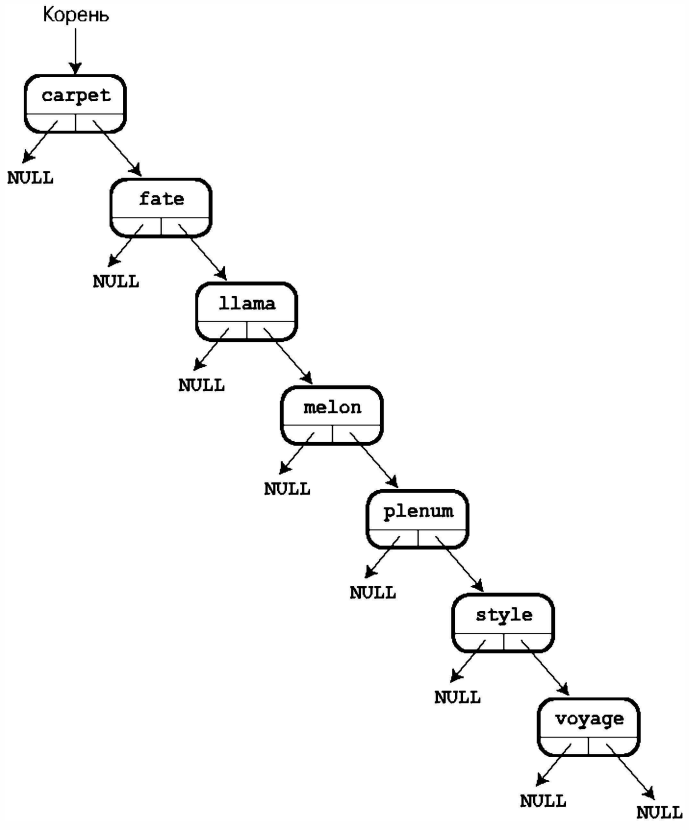


Рис. 17.16. *Плохо сбалансированное дерево бинарного поиска*

Например, предположим, что нужно проанализировать какой-то текст, отслеживая количество появлений в нем каждого слова. Один из возможных подходов к решению этой задачи — определение `Item` в качестве структуры, которая содержит одно слово и число. При первом появлении слова оно добавляется в дерево, а число устанавливается равным 1. При следующем появлении этого же слова программа находит содержащий его узел и увеличивает число на 1. Преобразование базового дерева бинарного поиска, чтобы оно работало описанным образом, не требует внесения больших изменений.

Для ознакомления с еще одним возможным вариантом дерева, снова обратимся к примеру клуба Nelville Pet Club. В этом примере сортировка дерева выполнялась как по кличкам, так и по видам животных. Поэтому оно могло бы содержать Сэма-кота в одном узле, Сэма-пса — во втором, и Сэма-козла — в третьем. Однако в нем не могло бы быть двух котов с кличкой Сэм. Еще один возможный подход — упорядочение дерева только по кличкам животных. Само по себе это изменение допустило бы существование только одного Сэма, независимо от вида, но затем можно было бы определить `Item` как список структур, а не как одиночную структуру. Тогда при первом появлении животного с кличкой Салли программа создала бы новый узел, затем новый список, а затем добавила бы Салли и ее вид в этот список. Следующая появившаяся Салли была бы направлена в этот же узел и добавлена в список.

Дополнительные библиотеки

Вероятно, вы уже поняли, что реализация абстрактных типов данных, таких как связный список или дерево – трудоемкая задача, сопряженная с множеством потенциальных ошибок. Дополнительные библиотеки предоставляют альтернативную возможность: пусть кто-то другой выполнит всю работу по созданию и тестированию типа. Ознакомившись с двумя сравнительно простыми примерами, приведенными в этой главе, вы сможете лучше понять и оценить значение таких библиотек.

Другие направления

В этой книге были освещены основные функциональные возможности языка C, но мы лишь вскользь затронули библиотеку доступных функций. Библиотека ANSI C содержит множество полезных функций. Большинство реализаций также предоставляют обширные библиотеки, характерные для конкретных систем. Компиляторы для DOS предлагают функции поддержки управления оборудованием, клавиатурного ввода и генерирование графических изображений для компьютеров IBM PC и их клонов. Компиляторы для Windows поддерживают графический интерфейс операционной системы Windows. Компиляторы C для Macintosh предоставляют функции для доступа к наборам инструментов Macintosh, облегчающим создание программ со стандартным интерфейсом Macintosh. Поэтому найдите время для исследования возможностей, предлагаемых используемой системой. Если нужные функциональные возможности в системе отсутствуют, создайте собственные функции. Возможность их создания — неотъемлемая часть C. Если полагаете, что можете создать более совершенную функцию ввода — создайте ее! И по мере совершенствования и оттачивания своей техники программирования вы сможете перейти от простого программирования на C к блестящему программированию на C.

Если тема списков, очередей и деревьев показалась вам интересной и полезной, можете почитать соответствующие книги или прослушать курс по более сложным технологиям программирования. Ученые-компьютерщики тратят массу энергии и таланта на разработку и анализ алгоритмов и способов представления данных. Возможно, что кто-либо уже разработал именно то средство, в котором вы нуждаетесь.

После усвоения языка C можете заняться изучением языка C++, Objective C или Java. Эти *объектно-ориентированные* языки были созданы на основе C. Язык C уже содержит объекты данных, варьирующиеся по сложности от простой переменной типа `char` до больших и сложных структур. Объектно-ориентированные языки развивают идею объектов еще больше. Например, свойства объекта определяют не только то, какие виды информации он может хранить, но и то, какие виды операций могут над ним выполняться. Описанные в этой главе абстрактные типы данных соответствуют этому подходу. Кроме того, объекты могут наследовать свойства других объектов. Объектно-ориентированное программирование переносит концепцию модульности на более высокий уровень абстракции, чем это имеет место в языке C, и применяется при разработке больших программ.

Перечень дополнительных книг, которые могут заинтересовать читателей, приведен в разделе I справочника (приложение Б).

Ключевые понятия

Тип данных характеризуется способами структурирования и хранения данных, а также возможными операциями над ними. Абстрактный тип данных (ADT) абстрактным образом определяет свойства и операции, характеризующие тип. Концептуально тип ADT можно преобразовать в код конкретного языка программирования в два этапа. Первый этап связан с определением программного интерфейса. На языке C это можно сделать за счет использования файла заголовка для определения имен типов и объявления прототипов функций, соответствующих допустимым операциям. Второй этап состоит в реализации интерфейса. На языке C это можно сделать в файле исходного кода, который предоставляет определения функций, соответствующих прототипам.

Резюме

Список, очередь и бинарное дерево — это примеры абстрактных типов данных, обычно используемых в программировании. Их часто реализуют посредством динамического резервирования памяти и связанных структур, но иногда их лучше реализовать с помощью массивов.

Если программа использует конкретный тип (например, очередь или дерево), программу следует писать с применением понятий интерфейса типа. Это позволит изменять и совершенствовать реализацию, не изменяя программы, которые используют этот интерфейс.

Вопросы для самоконтроля

1. Что требуется для определения типа данных?
2. Почему обход списка, приведенного в листинге 17.2, может выполняться только в одном направлении? Как можно было бы изменить определение `struct film`, чтобы обход списка можно было выполнять в обоих направлениях?
3. Что такое ADT?
4. Функция `QueueIsEpty()` принимает в качестве аргумента указатель на структуру `queue`, но ее можно было бы написать так, чтобы она принимала в качестве аргумента саму структуру `queue`, а не ее указатель. Каковы преимущества и недостатки каждого из этих подходов?
5. *Стек* — еще одна форма данных из семейства списков. В стеке добавления и удаления могут выполняться только с одного конца списка. Элементы должны "заталкиваться" в стек и "выталкиваться" из него. Следовательно, стек представляет собой структуру LIFO (*last in, first out* — "последним прибыл, первым обслужен").
   а. Определите тип ADT стека.
   б. Определите программный интерфейс стека.
6. Каково максимальное количество сравнений, требуемых при последовательном поиске и бинарном поиске для определения того, что конкретный элемент отсутствует в упорядоченном списке из 3 элементов? 1023 элементов? 65535 элементов?

7. Предположим, что программа создает дерево бинарного поиска слов, используя алгоритм, описанный в этой главе. Нарисуйте дерево, исходя из предположения, что слова были введены в следующем порядке:
   а. nice food roam dodge gate office wave
   б. wave roam office nice gate food dodge
   в. food dodge roam wave office gate nice
   г. nice roam office food wave gate dodge
8. Рассмотрите бинарные деревья, созданные при ответе на контрольный вопрос 7. Как они будут выглядеть после удаления из них слова *food* с помощью алгоритма, описанного в этой главе.

Упражнения по программированию

1. Измените листинг 17.2 так, чтобы он отображал список фильмов как в исходном, так и в обратном порядке. Один из возможных подходов — изменение определения связного списка для обеспечения обхода списка в обоих направлениях. Другой подход — применение рекурсии.
2. Предположим, что в файле `list.h` (листинг 17.3) использовано следующее определение списка:

```
typedef struct list
{
   Node * head; /* указывает на начало списка */
   Node * end;  /* указывает на конец списка */
} List;
```

   Перепишите функции в файле `list.c` (листинг 17.5), чтобы они соответствовали этому определению и протестируйте результирующий код с помощью программы `films3.c` (листинг 17.4).
3. Предположим, что в файле `list.h` (листинг 17.3) использовано следующее определение списка:

```
#define MAXSIZE 100
typedef struct list
{
   Item entries[MAXSIZE]; /* массив элементов */
   int items;             /* количество элементов в списке */
} List;
```

   Перепишите функции в файле `list.c` (листинг 17.5), чтобы они соответствовали этому определению и протестируйте результирующий код с помощью программы `films3.c` (листинг 17.4).
4. Перепишите программу `mall.c` (листинг 17.7), чтобы она имитировала киоск с двумя окошками и двумя очередями.
5. Напишите программу, которая вводит строку. Затем программа по одному заталкивает символы строки в стек (см. вопрос 5 выше), а затем выталкивает символы из стека и отображает их. Это приведет к отображению символов в обратном порядке.

6. Напишите функцию, которая принимает три аргумента: имя массива упорядоченных целых чисел, количество элементов в массиве и целое число, которое нужно найти. Функция возвращает значение 1, если целое число присутствует в массиве, и 0 — если оно отсутствует. Воспользуйтесь бинарным поиском.
7. Напишите программу, которая открывает и считывает текстовый файл, фиксируя количество появлений в нем каждого слова. Для хранения слова и числа его повторений используйте модифицированное дерево бинарного поиска. После того как программа выполнит считывание файла, она должна отобразить меню, состоящее из трех пунктов. Первый — вывод списка всех слов с указанием их повторений. Второй должен обеспечить возможность ввода слова, причем программа должна сообщить количество повторений этого слова в файле. Третий пункт меню должен выполнять выход из программы.
8. Измените программу для клуба любителей животных, чтобы все животные с одинаковыми кличками хранились в одном узле списка. Когда пользователь выберет поиск животного, программа должна запросить кличку животного, после чего вывести список всех животных (вместе с их видами), имеющих данную кличку.

ПРИЛОЖЕНИЕ А

Ответы на вопросы для самоконтроля

Ответы на вопросы для самоконтроля из главы 1

1. Полностью переносимая программа – это программа, код которой без каких-либо изменений может быть скомпилирован в успешно выполняемую программу во множестве различных компьютерных систем.
2. Файл исходного кода содержит код в том виде, каком он был написан на используемом программистом языке программирования. Файл объектного кода содержит код на машинном языке; ему не обязательно быть полным кодом всей программы. Исполняемый файл содержит полный код на машинном языке, формирующий исполняемую программу.
3. а. Определение целей программы.
   б. Проектирование программы.
   в. Написание кода программы.
   г. Компиляция программы.
   д. Выполнение программы.
   е. Тестирование и отладка программы.
   ж. Поддержка и модификация программы.
4. Компилятор преобразует исходный код (например, код, написанный на языке C) в эквивалентный код на машинном языке, называемый также *объектным кодом*.
5. Компоновщик объединяет исходный код с кодом библиотек и кодом запуска для создания исполняемой программы.

Ответы на вопросы для самоконтроля из главы 2

1. Они называются функциями.
2. Синтаксическая ошибка – это нарушение правил, управляющих составлением предложений или программ. Примером ошибки синтаксиса русского языка может служить предложение: “Моя хорошо говорить по-русски”. Ниже приведен пример синтаксической ошибки в программе на языке C:

```
printf"Куда подевались скобки?";
```

3. Семантическая ошибка — это ошибка, связанная с неправильным применением понятий. Например: “Это предложение — прекрасный образец английского языка”. Или в программе на языке C:

   ```
   тройное_n=3 + n;
   ```

4. Строка 1: начните строку с символа #; правильно введите имя файла `stdio.h`.

   Строка 2: используйте круглые скобки `()`, а не фигурные `{}`; завершите комментарий символами `*/`, а не `/*`.

   Строка 3: используйте скобку `{`, а не `(`.

   Строка 4: дополните оператор символом точки с запятой.

   Строка 5: только эта строка (пустая) в программе является правильной!

   Строка 6: в качестве символа операции присваивания необходимо использовать символ =, а не :=. (Судя по всему, Индиана немного знаком с языком Pascal.). В качестве количества недель в году используйте значение `52`, а не `56`.

   Строка 7: эта строка должна выглядеть следующим образом:

   ```
   printf("В году %d недель.\n", s);
   ```

   Строка 9: строка 9 в программе отсутствует, но она должна присутствовать и должна состоять из закрывающей фигурной скобки `}`.

   После внесения всех исправлений код должен выглядеть следующим образом:

   ```
   #include <stdio.h>
   int main(void) /* программа печатает количество недель в году */
   {
      int s;
      s = 52;
      printf("В году %d недель.\n", s);
      return 0;
   }
   ```

5.

   а.

   ```
   Бе-е, Бе-е, Черная Овечка.У тебя найдется шерсть для меня?
   ```

   (Обратите внимание на отсутствие пробела перед точкой. Пробел можно было бы вставить, используя " У вместо "У.)

   б.

   ```
   Убирайся!\
   Нет, толстое создание!
   ```

   (Обратите внимание на то, что курсор остается в конце второй строки.)

   в.

   ```
   Что?
   Нет/nC ума сошел?
   ```

   (Обратите внимание, что косая черта (/) оказывает не то влияние, которое оказывает обратная косая черта (\); программа просто печатает символ косой черты.)

г.

```
2 + 2 = 4
```

(Обратите внимание на замену каждой последовательности символов `%d` значением соответствующей переменной из списка. Обратите также внимание, что символ + означает операцию сложения, и что это вычисление может выполняться внутри оператора `printf()`.

6. `int` и `char` (`main` — имя функции, "function" (функция) — технический термин для описания работы программ C, а = — символ операции.)
7. `printf("Было %d слов и %d строк.\n", words, lines);`
8. После выполнения строки 7 значение `a` равно 5, а значение `b` — 2. После выполнения строки 8 значения обеих переменных `a` и `b` равны 5. После выполнения строки 9 значения обеих переменных `a` и `b` по-прежнему равны 5. (Обратите внимание, что значение `a` не может быть равным 2, поскольку на момент выполнения оператора `a = b;` значение переменной `b` уже изменено на 5.)

Ответы на вопросы для самоконтроля из главы 3

1. а. `int`, возможно `short` или `unsigned` либо `unsigned short`; население города представляет собой целое число.

   б. `float`; маловероятно, чтобы стоимость была целым числом. (Можно было бы использовать тип `double`, но в данном случае повышенная точность не требуется.)

   в. `char`.

   г. `int`, возможно `unsigned`.
2. Во-первых, тип `long` может вмещать бо́льшие числа, чем тип `int`; во-вторых, если требуется обработка больших значений, использование типа, длина которого в любой системе не меньше 32 битов, повышает переносимость программ.
3. Для получения в точности 32 битов можно было бы использовать тип `int32_t`, если он определен в данной системе. Для получения более короткого типа, который мог бы хранить числа длиной не менее 32 разрядов, следует использовать тип `int_least32_t`. И для получения типа, который бы обеспечивал наиболее быстрые вычисления с 32-разрядными значениями, следует использовать тип `int_fast32_t`.
4. а. Константа типа `char`.

   б. Константа типа `double`.

   в. Константа типа `unsigned int`, представленная в шестнадцатеричном формате.

   г. Константа типа `double`.
5. Строка 1: должна иметь вид `#include <stdio.h>`.

   Строка 2: должна иметь вид `int main(void)`.

   Строка 3: используйте `{`, а не `(`.

   Строка 4: между идентификаторами `g` и `h` необходимо использовать запятую, а не точку с запятой.

Строка 6 (пустая): все нормально.

Строка 7: перед символом e должна присутствовать хотя бы одна цифра. В данном случае вполне подошло бы значения 1e21 или 0e21, хотя они и достаточно длинные.

Строка 8: все нормально, по крайней мере, с точки зрения синтаксиса.

Строка 9: необходимо использовать символ }, а не).

Отсутствующие строки: во-первых, переменная rate никогда не получает присвоенного значения. Во-вторых, переменная h никогда не используется. Кроме того, программа не сообщает пользователю о результатах вычислений. Ни одна из этих ошибок не помешает выполнению программы (хотя может быть выведено сообщение о неиспользуемой переменной), но они скажутся на и без того незначительной полезности программы. Кроме того, в конце программы должен присутствовать оператор return.

Ниже представлена одна из возможных правильных версий этой программы:

```
#include <stdio.h>
int main(void)
{
   float g, h;
   float tax, rate;

   rate = 0.08;
   g = 1.0e5;
   tax = rate*g;
   h = g + tax;
   printf("Вам причитается $%f плюс $%f комиссионных, что вместе
составляет $%f.\n", g, tax, h);
   return 0;
}
```

6.

	Константа	*Тип*	*Спецификатор*
а.	12	int	%d
б.	0X3	unsigned int	%#X
в.	'C'	char (в действительности int)	%c
г.	2.34E07	double	%e
д.	'\040'	char (в действительности int)	%c
е.	7.0	double	%f
ж.	6L	long	%ld
з.	6.0f	float	%f

7.

	Константа	*Тип*	*Спецификатор*
а.	`12`	`unsigned int`	`%#o`
б.	`2.9e05L`	`long double`	`%Le`
в.	`'s'`	`char` (в действительности `int`)	`%c`
г.	`100000`	`long`	`%ld`
д.	`'\n'`	`char` (в действительности `int`)	`%c`
е.	`20.0f`	`float`	`%f`
ж.	`0x44`	`unsigned int`	`%x`

8.

```
printf("Ставка на %d равна %ld к 1.\n", imate, shot);
printf("Рейтинг %f не соответствует %c позиции.\n", log, grade);
```

9.

```
ch = '\r';
ch = 13;
ch = '\015'
ch = '\xd'
```

10. Строка 0: должна содержать `#include <stdio.h>`.

Строка 1: используйте символы `/*` и `*/` либо `//`.

Строка 3: `int cows, legs;`

Строка 4: `count?\n");`

Строка 5: `%d`, а не `%c`; замените `legs` на `&legs`.

Строка 7: `%d`, а не `%f`.

Добавьте также оператор `return`.

Одна из возможных правильных версий может иметь следующий вид:

```
#include <stdio.h>
int main(void) /* эта программа работает правильно */
{
   int cows, legs;
   printf("Сколько коровьих ног вы насчитали?\n");
   scanf("%d", &legs);
   cows = legs / 4;
   printf("Отсюда следует, что имеется %d коров.\n", cows);
   return 0;
}
```

11.

а. Символ новой строки.

б. Символ обратной косой черты.

в. Двойная кавычка.

г. Символ табуляции.

Ответы на вопросы для самоконтроля из главы 4

1. Программа работает неправильно. Первый оператор `scanf()` считывает только имя, оставляя фамилию неизменной, но все же сохраняет ее в "буфере" ввода. (Этот буфер представляет собой всего лишь временную область хранения вводимых данных.) Когда следующий оператор `scanf()` переходит к считыванию веса, он продолжает считывание с того места, где была завершена предыдущая попытка считывания данных, и пытается считать фамилию в качестве веса. Это ведет к ошибке работы `scanf()`. С другой стороны, если в ответ на запрос имени ввести что-либо наподобие `Александр 144`, программа использует значение `144` в качестве значения веса, несмотря на то, значение веса было введено ранее в ответ на запрос веса.

2.

   а.

   ```
   Он продал эту картину за $234.50
   ```

   б.

   ```
   Hi!
   ```

   (Примечание: первый символ — символьная константа, второй — десятичное целое значение, преобразованное в символ, а третий — восьмеричное ASCII-представление символьной константы.)

   в.

   ```
   Его Гамлет был хорош и без намека на вульгарность.
   Содержит 42 символа.
   ```

   г.

   ```
   Является ли 1.20e+003 тем же, что и 1201.00?
   ```

3. Необходимо использовать символы `\"`, как показано в следующем примере:

   ```
   printf("\"%s\"\nсодержит %d символов.\n", Q, strlen(Q));
   ```

4. Правильная версия программы выглядит следующим образом:

   ```
   #include <stdio.h> /* не забудьте включить эту строку */
   #define B "booboo" /* добавьте символы # и кавычек    */
   #define X 10       /* добавьте символ #               */
   int main(void)     /* вместо main(int)                */
   {
      int  age;
      int xp;         /* объявите все переменные         */
      char name[40];  /* создайте массив                 */

      printf{"Введите свое имя.\n"); /* символ \n вставлен для  */
                                     /* повышения читабельности */
      scanf("%s", name);
      printf("Хорошо, %s, а сколько вам лет?\n", name); /*%s - указание строки*/
      scanf("%d", &age);   /* %d, а не %f, и &age, а не age  */
      xp = age + X;
   ```

```
    printf ("Неужели, %s! Вам должно быть, по меньшей мере, %d лет.\n", B, xp);
    rerun 0;              /* возвращаемое значение отсутствует */
}
```

5. Вспомните о конструкции %%, предназначенной для вывода символа %.

```
printf("Этот экземпляр книги \"%s\" стоит $%0.2f.\n", BOOK, cost);
printf("Это %0.0f%% от цены прайс-листа.\n", percent);
```

6. а. %d
   б. %4X
   в. %10.3f
   г. %12.2e
   д. %-30s
7. а. %15lu
   б. %#4x
   в. %-12.2E
   г. %+10.3f
   д. %8.8s
8. а. %6.4d
   б. %*o
   в. %2c
   г. %+0.2f
   д. %-7.5s
9. а.

```
int dalmations;
   scanf("%d", &dalmations);
```

   б.

```
float kgs, share;
   scanf("%f%f", &kgs, &share);
```

   (Примечание: при вводе спецификаторы формата e, f и g можно использовать взаимозаменяемо. Кроме того, для всех спецификаторов, кроме %c, пробелы между спецификаторами преобразования не имеют значения.)

   в.

```
char pasta[20];
   scanf("%s", pasta);
```

   г.

```
char action[20];
   int value;
   scanf("%s %d", action, &value);
```

   д.

```
int value;
scanf("%*s %d", &value);
```

10. Пробельные символы — это символы пробела, табуляции и новой строки. В C пробельные символы служат для разделения выражений; в операторе `scanf()` пробелы используются для разделения последовательных элементов ввода.

11. Замещение было бы выполнено. К сожалению, препроцессор не в состоянии различить, какие фигурные скобки должны быть заменены круглыми, а какие нет. Поэтому программа

```
#define ( {
#define ) }
int main(void)
(
   printf("Привет, Богатырь!\n");
)
```

превратилась бы в

```
int main{void}
{
   printf{"Привет, Богатырь!\n"};
}
```

Ответы на вопросы для самоконтроля из главы 5

1. а. `30`.
   б. `27` (а не `3`). Выражение `(12 + 6)/(2*3)` дало бы результат равный `3`.
   в. `x = 1`, `y = 1` (целочисленное деление).
   г. `x = 3` (целочисленное деление) и `y = 9`.

2. а. `6` (сводится к `3 + 3.3`).
   б. `52`.
   в. `0` (сводится к `0 * 22.0`).
   г. `13` (сводится к `66.0 / 5` или `13.2`, а затем присваивается переменной типа `int`).

3. Строка 0: необходимо включить `<stdio.h>`.

   Строка 3: должна заканчиваться точкой с запятой, а не запятой.

   Строка 6: оператор `while` образует бесконечный цикл, поскольку значение `i` остается равным `1` и всегда меньшим `30`. Вероятно, имелось в виду `while(i++ < 30)`.

   Строки 6–8: судя по отступам, строки 7 и 8 должны были образовывать блок, однако отсутствие фигурных скобок означает, что цикл `while` включает в себя только строку 7. Необходимо добавить фигурные скобки.

   Строка 7: поскольку и `1`, и `i` — целые, результат деления будет равен `1` при `i = 1` и `0` при всех больших значениях. Использование выражения `n = 1.0/i;` привело бы к преобразованию `i` в значение с плавающей запятой перед выполнением операции деления, и общий результат был бы ненулевым.

   Строка 8: в управляющем операторе опущен символ новой строки (`\n`). Это ведет к выводу чисел в одной строке, когда это возможно.

   Строка 10: должна содержать оператор `return 0;`

Правильная версия программы выглядит следующим образом:

```
#include <stdio.h>
int main(void)
{
   int i = 1;
   float n;
   printf("Будьте осторожны! Далее идет последовательность дробей!\n");
   while (i++ < 30)
   {
      n = 1.0/i;
      printf(" %f\n", n);
   }
   printf("Вот и все!\n");
   return 0;
}
```

4. Основная проблема заключается во взаимодействии оператора проверки условия (является ли значение `sec` большим 0?) и оператором `scanf()`, который загружает значение переменной `sec`. В частности, при первом выполнении проверки условия программа не имеет ни малейшей возможности получить значение переменной `sec`, и сравнение будет выполняться с совершенно случайным значением, оказавшимся в используемой ячейке памяти. Одно из возможных решений, хотя и не очень изящное, — инициализация переменной `sec` значением равным, например, 1, чтобы проверку условия можно было выполнить в первый раз. Это позволяет выявить вторую проблему. Когда, в конце концов, пользователь вводит значение 0, чтобы остановить программу, значение `sec` не проверяется вплоть до завершения цикла, и программа выводит результаты, соответствующие 0 секунд. В действительности в программе требуется оператор `scanf()`, который бы выполнялся перед проверкой условия оператора `while`. Этого можно добиться, изменяя центральную часть программы следующим образом:

```
scanf("%d", &sec);
while ( sec > 0 ) {
   min = sec/S_TO_M;
   left = sec % S_TO_M;
   printf("%d секунд равно %d минут, %d секунд. \n", sec, min, left);
   printf("Следующий ввод данных?\n");
   scanf("%d", &sec);
}
```

На первом проходе программа будет выполнять оператор `scanf()`, находящийся снаружи цикла. На других проходах она будет выполнять оператор `scanf()`, указанный в конце цикла (и, следовательно, до начала новой итерации цикла). Описанный подход является общепринятым методом решения проблем подобного рода. Именно поэтому мы использовали программу, представленную в листинге 5.9.

5. Вывод программы имеет следующий вид:

```
%s! C - это круто!
! C - это круто!
11
11
12
11
```

Позвольте объяснить. Первый оператор printf() эквивалентен следующему оператору:

```
printf("%s! C - это круто!\n","%s! C - это круто!\n");
```

Второй оператор вывода вначале увеличивает значение num до 11, а затем выводит значение. Третий оператор вывода выводит значение num, которое равно 11, а затем увеличивает его до 12. Четвертый оператор выводит текущее значение n, которое по-прежнему равно 12, а затем уменьшает n до 11. Заключительный оператор вывода выводит текущее значение переменной num, которое равно 11.

6. Вывод имеет следующий вид:

```
SOS:4 4.00
```

Значение выражения c1 - c2 совпадает со значением выражения 'S' - '0', которое в ASCII-коде выглядит как 83 - 79.

7. Она печатает одну строку цифр от 1 до 10 в полях шириной по пять символов, а затем начинает новую строку:

```
    1    2    3    4    5    6    7    8    9   10
```

8. Возможный вариант программы, в которой предполагается, что буквы кодируются последовательно, как это имеет место в кодировке ASCII, выглядит следующим образом:

```
#include <stdio.h>

int main(void)
{
   char c = 'a';
   while (c <= 'g')
      printf("%5c", c++);
   printf("\n");
   return 0;
}
```

9. Эти фрагменты выводили бы следующие результаты:

   а.
   ```
    1   2
   ```

   Обратите внимание, что программа вначале выполняет увеличение значения x, а затем сравнение. Курсор остается в той же строке.

   б.
   ```
   101
   102
   103
   104
   ```

   Обратите внимание, что на этот раз сначала выполняется сравнение значения x, а затем его увеличение. И в этом примере, и в примере а), значение x увеличивается перед выполнением вывода. Обратите также внимание, что запись второго оператора printf() с отступом не делает его частью цикла while. Следовательно, этот оператор вызывается только один раз, после завершения цикла while.

   в.
   ```
   stuvw
   ```

   В этом примере увеличение значения выполняется только по завершении выполнения первого оператора printf().

10. Эта программа написана неудачно. Поскольку оператор `while` не содержит фигурных скобок, только оператор `printf()` является частью цикла. Поэтому программа бесконечно повторяет вывод сообщения `COMPUTER BYTES DOG` до тех пор, пока пользователю не удастся ее остановить.
11.
    а. `x = x + 10;`
    б. `x++;` или `++x;` или `x = x + 1;`
    в. `c = 2 * (a + b);`
    г. `c = a + 2* b;`
12.
    а. `x--;` или `--x;` или `x = x - 1;`
    б. `m = n % k;`
    в. `p = q / (b - a);`
    г. `x = (a + b) / (c * d);`

Ответы на вопросы для самоконтроля из главы 6

1. `2, 7, 70, 64, 8, 2`
2. Он должен вывести следующее:

    ```
    36 18 9 4 2 1
    ```

    Если бы переменная `value` имела тип `double`, результат проверки условия оставался бы равным `true` даже при значениях `value`, меньших 1. Выполнение цикла продолжалось бы до тех пор, пока потеря значимости при вычислениях с плавающей запятой не привела бы к значению 0. Кроме того, в этом случае выбор спецификатора `%3d` был бы неправильным.
3.
    а. `x > 5`
    б. `scanf("%lf",&x) != 1`
    в. `x == 5`
4.
    а. `scanf("%d", &x) == 1`
    б. `x != 5`
    в. `x >= 20`
5. Строка 4: должна содержать `list[10]`.

    Строка 9: переменная `i` должна изменяться в диапазоне от 0 до 9, а не от 1 до 10.

    Строка 9: запятые необходимо заменить точками с запятой.

    Строка 9: символ операции >= необходимо заменить символом операции <=. В противном случае цикл будет выполняться достаточно медленно.

Строка 11: Между строками 11 и 12 должна присутствовать еще одна закрывающая фигурная скобка. Одна скобка закрывает блочный оператор, а вторая — программу. Между этими скобками необходимо вставить строку `return 0;`

Правильная версия программы имеет следующий вид:

```
#include <stdio.h>
int main(void)
{                                   /* строка 3  */
   int i, j, list[10];              /* строка 4  */

   for (i = 1; i <= 10; i++)        /* строка 6  */
   {                                /* строка 7  */
      list[i] = 2*i + 3;            /* строка 8  */
      for (j = 1; j <= i; j++)      /* строка 9  */
         printf(" %d", list[j]);    /* строка 10 */
   printf("\n");                    /* строка 11 */
   }
   return 0;
}
```

6. Вот один из возможных вариантов:

```
#include <stdio.h>
int main(void)
{
   int col, row;

   for (row = 1; row <= 4; row++)
   {
      for (col = 1; col <= 8; col++)
         printf("$");
      printf("\n");
   }
   return 0;
}
```

7.

   а. Программа выведет на печать следующую строку:

   ```
   Здравствуйте! Здравствуйте! Здравствуйте! До свидания! До свидания!
   До свидания! До свидания! До свидания!
   ```

   б. Программа выведет на печать следующую строку:

   ```
   ACGM
   ```

8. Эти программы выведут на печать следующее:

   а. `Go west, youn`

   б. `Hp!xftu-!zpvo`

   в. `Go west, young`

   г. `Go west, youn`

9. Вы должны получить следующий вывод:

```
31|32|33|30|31|32|33|
***
```

```
1
5
9
13
***
2 6
4 8
8 10
***
======
=====
====
===
==
```

10.

а. `mint`.

б. 10 элементов.

в. Значения типа `double`.

г. правильной является строка `ii`; `mint[2]` — значение типа `double`, а `&mint[2]` — его местоположение.

11. Поскольку индекс первого элемента равен `1`, переменная цикла должна изменяться в диапазоне от `1` до `SIZE - 1`, а не от `1` до `SIZE`. Однако внесение этого изменения приводит к присвоению первому элементу значения `0` вместо `2`. Поэтому цикл необходимо переписать следующим образом:

```
for (index = 0; index < SIZE; index++)
   by_twos[index] = 2 * (index + 1);
```

Аналогично, изменений требуют и пределы второго цикла. Кроме того, имя массива должно сопровождаться индексом массива:

```
for( index = 0; index < SIZE; index++)
   printf("%d ", by_twos[index]);
```

Один из опасных аспектов неправильного указания диапазона цикла состоит в том, что при этом программа может работать. Однако, поскольку она будет помещать данные в те ячейки памяти, в которые не должна, она может перестать работать в будущем, представляя собой своего рода "мину замедленного действия".

12. Оно должно объявлять тип возвращаемого значения как `long` и должно содержать оператор `return`, который возвращает значение типа `long`.

13. Преобразование типа `num` в `long` гарантирует выполнение вычислений с типом `long`, а не `int`. В системе, использующей 16-разрядный тип `int`, умножение двух значений типа `int` создает результат, который перед возвратом значения усекается до типа `int`, что может вести к потере данных.

```
long square(int num)
{
   return ((long) num) * num;
}
```

14. Ее вывод имеет следующий вид:

```
1: Здравствуйте!
k = 1
k равно 1 на итерации
Теперь k равно 3
k = 3
k равно 3 на итерации
Теперь k равно 5
k = 5
k равно 5 на итерации
Теперь k равно 7
k = 7
```

Ответы на вопросы для самоконтроля из главы 7

1. Истинно: б.
2.
   а. `number >= 90 && number < 100`
   б. `ch != 'q' && ch != 'k'`
   в. `(number >= 1 && number <= 9) && number != 5`
   г. Один из возможных вариантов `!(number >= 1 && number <= 9)`, но выражение `number < 1 || number > 9` проще для понимания
3. Строка 5: эта строка должна иметь вид `scanf("%d %d", &weight, &height);`. Не забудьте использовать символы `&` в операторе `scanf()`. Кроме того, этой строке должна предшествовать строка с приглашением к вводу данных.

   Строка 9: в данном случае подразумевается выражение `(height < 72 && height > 64)`. Однако первая часть выражения излишня, поскольку, чтобы программа достигла строки `else if`, рост `height` должен быть меньше 72. Поэтому вполне достаточно использовать выражение `(height > 64)`. Но строка 6 уже гарантирует выполнение этого условия, поэтому никакая дополнительная проверка вообще не требуется и выражение `if else` следует заменить выражением `else`.

   Строка 11: это условие избыточно. Второе подвыражение (`weight` не меньше или равно 300) означает то же, что и первое. В данном случае требуется использовать простое выражение `(weight > 300)`. Однако в этой строке присутствует значительно более серьезная ошибка. Строка 11 связана не с тем оператором `if`! Понятно, что это выражение `else` предполагалось использовать с оператором `if` строки 6. Однако в соответствии с правилом связи с ближайшим предшествующим оператором `if` она будет связана с оператором `if` строки 9. Поэтому строка 11 выполняется тогда, когда значение `weight` меньше 100, а значение `height` меньше или равно 64. В результате в момент выполнения этого оператора значение `weight` никак не может превышать 300.

   Строки 7–9: эти строки должны быть заключены в фигурные скобки. В этом случае строка 11 станет альтернативной строкой строки 6, а не 9. Или же выражение `if else`

в строке 9 можно заменить выражением `else`, и тогда никакие фигурные скобки не нужны.

Строка 13: ее следует упростить до `if (height > 48)`. В действительности эту строку можно полностью опустить, поскольку строка 12 уже выполняет данную проверку.

Строка 15: это выражение `else` связано с последним оператором `if`, указанным в строке 13. Чтобы связать это выражение с оператором `if` строки 11, строки 13 и 14 потребуется заключить в фигурные скобки. Или, как было предложено ранее, строку 13 можно просто опустить.

Правильная версия программы имеет следующий вид:

```
#include <stdio.h>
int main(void)
{
   int weight, height; /* вес в фунтах, рост в дюймах */

   printf("Введите свой вес в фунтах и ");
   printf("свой ростр в дюймах.\n");
   scanf("%d %d", &weight, &height);
   if (weight < 100 && height > 64)
      if (height >= 72)
         printf("Ваш вес слишком мал для вашего роста.\n");
      else
         printf("Ваш вес мал для вашего роста.\n");
   else if (weight > 300 && height < 48)
         printf(" Ваш рост мал для вашего веса.\n");
   else
         printf("У вас идеальный вес.\n");

   return 0;
}
```

4.

а. `1`. Утверждение истинно, что численно равно значению `1`.

б. `0`. 3 не меньше 2.

в. `1`. Если первое выражение ложно, то второе истинно, и наоборот. Чтобы все выражение было истинным, достаточно истинности только одного из его подвыражений.

г. `6`, поскольку значение выражения `6 > 2` равно `1`.

д. `10`, поскольку проверяемое условие истинно.

е. `0`. Если выражение `x > y` истинно, значением выражения будет `y > x`, которое в этом случае ложно, или равно `0`. Если выражение `x > y` ложно, значением выражения будет `x > y`, которое в данном случае ложно.

5. Программа выводит на печать следующую строку:

```
*#%*#%$#%*#%*#%$#%*#%*#%$#%*#%*#%
```

Несмотря на присутствующие в коде программы отступы, она печатает символ # на каждой итерации цикла, поскольку этот оператор вывода не является частью блочного оператора.

6. Программа выводит на печать следующие строки:

```
fat hat cat Oh no!
hat cat Oh no!
cat Oh no!
```

7. Комментарии в строках с 5 по 7 должны завершаться символами */, либо же символы /* можно заменить символами //. Выражение 'a' <= ch >= 'z' потребуется заменить следующим выражением:

```
ch >= 'a' && ch <= 'z'
```

Или же можно воспользоваться более простым и переносимым вариантом, включив файл ctype.h и вызвав функцию islower(). По случайному стечению обстоятельств выражение 'a' <= ch >= 'z' является допустимой конструкцией С. Просто оно затрудняет получение правильного представления о его смысле. Поскольку объединение членов в операциях отношения выполняется слева направо, это выражение интерпретируется как ('a' <= ch) >= 'z'. Выражение в скобках принимает значение 1 или 0 (истинно или ложно), а затем программа проверяет, является ли это значение большим или равным числовому коду 'z'. Ни 0, ни 1 не удовлетворяют этому условию, поэтому значение всего выражения всегда равно 0 (ложно). Во втором условном выражении символы || необходимо заменить символами &&. Кроме того, хотя выражение !(ch < 'A') вполне допустимо и правильно, выражение !(ch < 'A') проще. За выражением 'Z' должны следовать две закрывающие скобки, а не одна. И снова, проще использовать функцию isupper(). Оператору oc++; должно предшествовать выражение else. В противном случае программа будет увеличивать значение каждого символа. Управляющее выражение в вызове функции printf() должно быть заключено в двойные кавычки.

Правильная версия программы выглядит следующим образом:

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   char ch;
   int lc = 0; /* подсчет строчных символов */
   int uc = 0; /* подсчет прописных символов */
   int oc = 0; /* подсчет всех остальных символов */

   while ((ch = getchar()) != '#')
   {
      if (islower(ch))
         lc++;
      else if (isupper(ch))
         uc++;
      else
         oc++;
   }
   printf("%d строчных, %d прописных, %d остальных", lc, uc, oc);
   return 0;
}
```

8. К сожалению, она бесконечно повторяет вывод одной и той же строки:

```
Вам 65 лет. Получите ваши золотые часы.
```

Проблема в том, что строка

```
if (age = 65)
```

устанавливает значение переменной age равным 65, которое на каждой итерации цикла интерпретируется как истинное.

9. Результат выполнения программы с введенными значениями выглядит следующим образом:

```
q
Шаг 1
Шаг 2
Шаг 3
c
Шаг 1
g
Шаг 1
Шаг 3
b
Шаг 1
Готово
```

Обратите внимание, что ввод обоих символов — и b, и # — ведет к прерыванию цикла, но ввод символа b ведет к выводу строки Шаг 1, а ввод символа # — нет.

10. Одно из возможных решений выглядит следующим образом:

```
#include <stdio.h>
int main(void)
{
   char ch;

   while ((ch = getchar()) != '#')
   {
      if (ch != '\n')
      {
         printf("Шаг 1\n");
         if (ch == 'b')
            break;
         else if (ch != 'c')
         {
            if (ch != 'g')
               printf("Шаг 2\n");
            printf("Шаг 3\n");
         }
      }
   }
   printf("Готово\n");
   return 0;
}
```

Ответы на вопросы для самоконтроля из главы 8

1. Выражение `putchar(getchar())` вынуждает программу выполнить считывание следующего вводимого символа и вывести его на печать. В данном случае возвращаемое значение функции `getchar()` служит аргументом функции `putcar()`. И напротив, выражение `getchar(putchar())` недопустимо, поскольку функция `getchar()` не использует аргумент, а функция `putcar()` нуждается в нем.
2. а. Отображает символ `H`.
   б. Если система использует ASCII-кодировку, этот оператор вызывает подачу звукового сигнала.
   в. Перемещает курсор к началу следующей строки.
   г. Вызывает затирание последнего напечатанного символа и возврат курсора на один символ назад.
3. `count <essay >essayct` или `count >essayct <essay`
4. Только вариант в) является допустимым.
5. Это сигнал (специальное значение), возвращаемое функциями `getchar()` и `scanf()` для указания того, что они обнаружили конец файла.
6. а. Вывод имеет следующий вид:

   ```
   If you qu
   ```

   Обратите внимание, что символ `I` отличается от символа `i`. Обратите также внимание, что символ `i` не выводится, поскольку выход из цикла выполняется до обнаружения этого символа.

   б. Если система использует ASCII-символы, вывод будет следующим:

   ```
   HJacrthjacrt
   ```

   При первом обращении значением переменной `ch` является символ `H`. Операция `ch++` приводит к использованию (печати) этого значения, а затем к его инкременту (до символа `I`). После этого операция `++ch` инкрементирует значение (до `J`), а затем его использует (выводит на печать). После этого выполняется считывание следующего символа (`a`) и процесс повторяется. Важно отметить, что операции инкремента значения воздействуют на переменную `ch` после присвоения ей значения. Они не приводят к перемещению программы по очереди ввода.
7. Стандартная библиотека ввода-вывода C преобразует различные формы файлов в унифицированные потоки, обработку которых можно выполнять одинаково.
8. При вводе чисел пробелы и символы новой строки пропускаются, а при вводе символов — нет. Предположим, что имеется код, подобный следующему:

```
int score;
char grade;
printf("Введите число баллов.\n");
scanf("%s", &score);
printf("Введите буквенное обозначение рейтинга.\n");
grade = getchar();
```

Если в качестве значения баллов (score) ввести число 98, а затем нажать клавишу <Enter> для передачи этого значения программе, ей будет передан и символ новой строки, который станет следующим вводимым символом и будет считан в переменную grade в качестве значения рейтинга. Если ввод числа предшествует вводу символа, в программу потребуется добавить код, который будет отбрасывать символ новой строки перед вводом символа.

Ответы на вопросы для самоконтроля из главы 9

1. Формальный параметр — это переменная, которая определена в вызываемой функции. Фактический аргумент — это значение, появляющееся в вызове функции. Это значение присваивается формальному аргументу. Фактический аргумент можно считать значением, которым инициализируется формальный параметр при вызове функции.
2.
   а. `void donut(int n)`
   б. `int gear(int t1, int t2)`
   в. `void stuff_it(double d, double *pd)`
3.
   а. `char n_to_char(int n)`
   б. `int digits(double x, int n)`
   в. `int random(void)`
4.
```
int sum(int a, int b)
{
   return a + b;
}
```
5. Все идентификаторы типа int необходимо заменить идентификаторами типа double:
```
double sum(double a, double b)
{
   return a + b;
}
```
6. В этой функции необходимо использовать указатели:
```
void alter(int * pa, int * pb)
{
   int temp;
   temp = *pa + *pb;
   *pb = *pa - *pb;
   *pa = temp;
}
```
   или

```
void alter(int * pa, int * pb)
{
   *pa += *pb;
   *pb = *pa - 2 * *pb;
}
```

7. Да, ошибки присутствуют. В списке аргументов аргумент num должен быть объявлен в списке аргументов функции salami(), а не после фигурной скобки. Кроме того, необходимо использовать операцию count++, а не num++.

8. Одно из возможных решений выглядит следующим образом:

```
int largest(int a, int b, int c)
{
   int max = a;
   if (b > max)
      max = b;
   if (c > max)
      max = c;
   return max;
}
```

9. Минимальная по объему кода программа приведена ниже. Функции showmenu() и getchoice() можно использовать в решениях заданий пунктов а) и б).

```
#include <stdio.h>
void showmenu(void); /* объявление используемых функций */
int getchoice(int, int);
main()
{
   int res;

   showmenu();
   while ((res = getchoice(1,4)) != 4)
      printf("Меня устраивает вариант %d.\n", res);
   printf("До свидания!\n");
   return 0;
}

void showmenu(void)
{
   printf("Пожалуйста, выберите один из следующих вариантов:\n");
   printf("1) копирование файлов   2) перемещение файлов\n");
   printf("3) удаление файлов      4) выход\n");
   printf("Введите номер выбранного варианта:\n");
}

int getchoice(int low, int high)
{
   int ans;
   scanf("%d", &ans);
   while (ans < low || ans > high)
   {
      printf("%d недопустимый вариант; повторите попытку\n", ans);
      showmenu();
      scanf("%d", &ans);
   }
   return ans;
}
```

Ответы на вопросы для самоконтроля из главы 10

1. Вывод выглядит следующим образом:
   ```
   8 8
   4 4
   0 0
   2 2
   ```
2. Массив `ref` содержит четыре элемента, поскольку таково количество значений в списке инициализации.
3. Имя массива `ref` указывает на первый элемент массива — целое число `8`. Выражение `ref + 1` указывает на второй элемент, целое число `4`. Конструкция `++ref` не является допустимым выражением C. `ref` — константа, а не переменная.
4. `ptr` указывает на первый элемент, а `ptr + 2` указывает на третий элемент, который будет первым элементом второй строки.
   а. `12` и `16`.
   б. `12` и `14` (в соответствии со скобками только число 12 помещается в первую строку).
5. `ptr` указывает на первую строку, а `ptr+1` — на вторую строку; `*ptr` указывает на первый элемент в первой строке, а `*(ptr + 1)` — на первый элемент второй строки.
   а. `12` и `16`.
   б. `12` и `14` (в соответствии со скобками только число 12 помещается в первую строку).
6. а. `&grid[22][56]`
   б. `&grid[22][0]` или `grid[22]`

   (Последний идентификатор — имя одномерного массива, состоящего из 100 элементов, то есть адрес его первого элемента, которым является элемент `grid[22][0]`.)
   в. `&grid[0][0]` или `grid[0]` или `(int *) grid`.

   (Здесь `grid[0]` — адрес элемента `grid[0][0]` типа `int`, а `grid` — адрес 100-элементного массива `grid[0]`. Эти два адреса имеют одно и то же значение, но различные типы. Преобразование типа делает типы одинаковыми.
7. а. `int digits[10];`
   б. `float rates[6];`
   в. `int mat[3][5];`
   г. `char * psa[20];`

   Обратите внимание, что приоритет квадратных скобок `[]` выше приоритета операции умножения, поэтому при отсутствии скобок вначале применяется описатель массива, а затем описатель указателя. Таким образом, это объявление эквивалентно объявлению `char * (psa[20]);`
   д. `char (*pstr)[20];`

На заметку!

В пункте д) нельзя использовать объявление `char *pstr[20];`. Это привело бы к созданию массива указателей, а не указателя на массив. В частности, `pstr` указывал бы на единственную переменную `char` – первый элемент массива; `pstr + 1` указывал бы на следующий байт. При использовании правильного объявления `pstr` представляет переменную, а не имя массива, а `pstr + 1` указывает на позицию, которая на 20 байт отстоит от начального байта.

8. а. `int sextet[6] = {1, 2, 4, 8, 16, 32};`
   б. `sextet[2]`
   в. `int lots[100] = { [99] = -1};`
9. От 0 до 9.
10. а. `rootbeer[2] = value;`
    Допустим.
    б. `scanf("%f", &rootbeer );`
    Недопустим; `&rootbeer` не является значением типа `float`.
    в. `rootbeer = value;`
    Недопустим; `rootbeer` не является значением типа `float`.
    г. `printf("%f", rootbeer);`
    Недопустим; `rootbeer` не является значением типа `float`.
    д. `things[4][4] = rootbeer[3];`
    Допустим.
    е. `things[5] = rootbeer;`
    Недопустим; нельзя присваивать массивы.
    ж. `pf = value;`
    Недопустим; `value` не является адресом.
    з. `pf = rootbeer;`
    Допустим.
11. `int screen[800][600];`
12.
    а.
    ```
    void process(double ar[], int n);
    void processvla(int n, double ar[n]);
    process(trots, 20);
    processvla(20, trots);
    ```
    б.
    ```
    void process2(short ar2[30], int n);
    void process2vla(int n, int m, short ar2[n][m]);
    process2(clops, 10);
    process2vla(10, 30, clops);
    ```
    в.
    ```
    void process3(long ar3[10][15], int n);
    void process3vla(int n, int m,int k, long ar3[n][m][k]);
    process3(shots, 5);
    process3vla(5, 10, 15, shots);
    ```
13.
    а. `show( (int [4]) {8,3,9,2}, 4);`
    б. `show2( (int [][3]){{8,3,9}, {5,4,1}}, 2);`

Ответы на вопросы для самоконтроля из главы 11

1. Чтобы результат был строкой, выражение инициализации должно содержать нулевой символ '\0'. Разумеется, применение альтернативной синтаксической формы ведет к автоматическому добавлению нулевого символа:

   ```
   char name[] = "Fess";
   ```

2.
   ```
   Увидимся в кафе.
   видимся в кафе.
   Увидимся
   идимся
   ```

3.
   ```
   y
   my
   mmy
   ummy
   Yummy
   ```

4. `Я читал часть этого всю дорогу.`

5. а. `Xo Xo Xo!!oX oX oX`

   б. Указатель на `char` (то есть `char *`)

   в. Адрес начальной буквы `X`.

   г. Выражение `*--pc` означает уменьшение указателя на 1 и использование значения, найденного по этому адресу. `--*pc` означает извлечение значения, указанного указателем `pc` и уменьшение этого значения на 1 (например, символ `X` становится символом `Ф`).

   д. `Xo Xo Xo!!oX oX o`

На заметку!

Нулевой символ присутствует между символами `!` и `!`, но он не оказывает никакого влияния на вывод.

   е. `while(*pc)` проверяет, не указывает ли указатель `pc` на нулевой символ (то есть на конец строки). Выражение использует значение, расположенное по указанному адресу.

   `while(pc - str)` проверяет, не указывает ли указатель `pc` на тот же адрес, что и `str` (начало строки). Выражение использует значения самих указателей.

   ж. После выполнения первой итерации цикла `while` указатель `pc` указывает на нулевой символ. При входе во вторую итерацию цикла он указывает на ячейку памяти, предшествующую нулевому символу (то есть расположенную непосредственно перед той, на которую указывает `str`). Этот байт интерпретируется как символ и выводится на печать. Затем указатель возвращается к предыдущему байту. Условие выхода из цикла `(pc == str)` никогда не выполняется, и процесс продолжается до тех пор, пока не будет прерван пользователем или системой.

   з. `pr()` должен быть объявлен в вызывающей программе:

   ```
   char * pr(char *);
   ```

6. Символьная переменная занимает один байт, поэтому и sign занимает один байт. Но символьная константа сохраняется с типом данных int — то есть, как правило '$' будет использовать 2 или 4 байта. Однако для действительного хранения кода '$' будет использован только один байт типа данных int. Строка "$" использует два байта: один для хранения кода символа '$', второй — для хранения кода символа '\0'.

7. Эта программа выводит на печать следующее:

```
How are ya, sweetie? How are ya, sweetie?
Beat the clock.
eat the clock.
Beat the clock. Win a toy.
Beat
chat
hat
at
t
t
at
How are ya, sweetie?
```

8. Ее вывод имеет следующий вид:

```
faavrhee
*le*on*sm
```

9. Вот одно из возможных решений:

```
int strlen(const char * s)
{
   int ct = 0;

   while (*s++) // или while (*s++ != '\0')
      ct++;

   return(ct);
}
```

10. Одно из возможных решений выглядит следующим образом:

```
#include <stdio.h>       /* для определения NULL */
char * strblk(char * string)
{
   while (*string != ' ' && *string != '\0')
      string++;          /* остановка при обнаружении первого
                            символа пробела или нулевого символа */
   if (*string == '\0')
      return NULL;       /* NULL - нулевой указатель */
   else
      return string;
}
```

Второе решение предотвращает изменение строки функцией, но позволяет использовать возвращаемое значение для изменения строки. Выражение (char *) string называют “избавлением от const”.

```
#include <stdio.h>       /* для определения NULL */
char * strblk(const char * string)
{
   while (*string != ' ' && *string != '\0')
      string++;          /* остановка при обнаружении первого
                            символа пробела или нулевого символа */
   if (*string == '\0')
      return NULL;       /* NULL - нулевой указатель */
   else
      return (char *) string;
}
```

11. Одно из решений имеет вид:

```
/* compare.c -- эта программа будет работать */
#include <stdio.h>
#include <string.h> /* объявляет функцию strcmp() */
#include <ctype.h>
#define ANSWER "ГРАНТ"
#define MAX 40
void ToUpper(char * str);
int main(void)
{
   char try[MAX];

   puts("Кто похоронен в могиле Гранта?");
   gets(try);
   ToUpper(try);
   while (strcmp(try,ANSWER) != 0)
   {
      puts("Нет, неправильно. Попытайтесь еще раз.");
      gets(try);
      ToUpper(try);
   }
   puts("Теперь правильно!");
   return 0;
}

void ToUpper(char * str)
{
   while (*str != '\0')
   {
      *str = toupper(*str);
      str++;
   }
}
```

Ответы на вопросы для самоконтроля из главы 12

1. Автоматический класс памяти, регистровый класс памяти и статический без связывания класс памяти.

2. Статический без связывания класс памяти, статический с внутренним связыванием класс памяти и статический с внешним связыванием класс памяти.
3. Статический с внешним связыванием класс памяти. Статический с внутренним связыванием класс памяти.
4. Класс памяти без связывания.
5. Ключевое слово `extern` используется в объявлениях для указания переменной или функции, которая объявлена в каком-либо другом модуле.
6. Оба оператора резервируют память для массива 100 значений типа `int`. Кроме того, оператор, в котором использована функция `calloc()`, устанавливает значение каждого элемента равным 0.
7. Переменная `daisy` известна функции `main()` по умолчанию, а функциям `petal()`, `stem()` и `root()` — благодаря объявлению `extern`. Объявление `extern int daisy;` во втором файле делает переменную `daisy` известной всем функциям в этом файле. Первая переменная `lily` локальна для функции `main()`. Ссылка на переменную `lily` в функции `petal()` — ошибка, поскольку ни один из файлов не содержит объявления внешней переменной `lily`. Существует внешняя статическая переменная `lily`, но она известно только функциям второго файла. Первая внешняя переменная `rose` известна функции `root()`, но функция `stem()` заменяет ее собственной локальной переменной `rose`.
8. Эта программа выводит на печать следующее:

```
color в main() равно B
color в first() равно R
color в main() равно B
color в second() равно G
color в main() равно G
```

9. а. Они указывают, что программа будет использовать переменную `plink`, которая локальна для файла, содержащего функцию. Первый аргумент функции `value_ct()` — указатель на целочисленное значение, по всей вероятности являющееся первым элементом массива, состоящего из `n` членов. В данном случае важно отметить, что программа не сможет использовать указатель `arr` для изменения значений в исходном массиве.

   б. Нет. `value` и `n` уже являются копиями исходных данных, поэтому функция никак не может изменять соответствующие значения в вызывающей программе. Эти объявления предотвращают изменение значений `value` и `n` внутри самой функции. Например, функция не могла бы использовать выражение `n++`, если бы переменная `n` была определена как `const`.

Ответы на вопросы для самоконтроля из главы 13

1. Программа должна содержать строку `<stdio.h>`, чтобы можно было использовать определения этого файла. Переменная `fp` должна быть объявлена как указатель на файл: `FILE *fp;` Функция `fopen()` требует указания режима: `fopen("gelatin", "w")`,

или, возможно режима "a". Порядок указания аргументов функции fputs() должен быть обратным. Для повышения удобочитаемости строка вывода должна содержать символ новой строки, поскольку функция fputs() не выполняет автоматического добавления этого символа. Функция fclose() требует использования в качестве аргумента указателя файла, а не имени файла: fclose(fp); Правильная версия программы выглядит следующим образом:

```
#include <stdio.h>
int main(void)
{
   FILE * fp;
   int k;

   fp = fopen("gelatin", "w");
   for (k = 0; k < 30; k++)
      fputs("Кто-то не особо умный поглощает желатин.\n", fp);
   fclose(fp);
   return 0;
}
```

2. При возможности она будет открывать файл, имя которого задано в первом аргументе командной строки, и выводить на экран каждый присутствующий в файле цифровой символ.

3. а. `ch = getc(fp1);`

   б. `fprintf(fp2,"%c"\n",ch);`

   в. `putc(ch,fp2);`

   г. `fclose(fp1); /* закрыть файл terky */`

На заметку!

Указатель fp1 используется для операций ввода, поскольку он идентифицирует файл, открытый в режиме чтения. Аналогично файл fp2 был открыт в режиме записи, поэтому он используется с функциями вывода.

4. Один из возможных вариантов программы имеет вид:

```
#include <stdio.h>
#include <stdlib.h>
/* #include <console.h> */ /* для Macintosh */

int main(int argc,char * argv[])
{
   FILE * fp;
   double n;
   double sum = 0.0;
   int ct = 0;

/* argc = ccommand(&argv); */ /* для Macintosh */
   if (argc == 1)
      fp = stdin;
   else if (argc == 2)
   {
      if ((fp = fopen(argv[1], "r")) == NULL)
      {
```

```
            fprintf(stderr, "Невозможно открыть %s\n", argv[1]);
            exit(EXIT_FAILURE);
        }
    }
    else
    {
        fprintf(stderr, "Применение: %s [имя_файла]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    while (fscanf(fp, "%lf", &n) == 1)
    {
        sum += n;
            ++ct;
    }
    if (ct > 0)
        printf("Среднее арифметическое %d значений = %f\n", ct, sum / ct);
    else
        printf("Допустимые данные отсутствуют.\n");
    return 0;
}
```

Пользователи Macintosh C должны не забыть использовать заголовочный файл `console.h` и функцию `ccommand()`.

5. Одно из возможных решений приведено ниже. (Пользователи Macintosh C должны не забыть использовать заголовочный файл `console.h` и функцию `ccommand()`.)

```
#include <stdio.h>
#include <stdlib.h>
/* #include <console.h> */ /* для Macintosh */
#define BUF 256
int has_ch(char ch, const char * line);
int main(int argc,char * argv[])
{
    FILE * fp;
    char ch;
    char line [BUF];

/* argc = ccommand(&argv); */ /* для Macintosh */
    if (argc != 3)
    {
        printf("Применение: %s символ имя_файла\n", argv[0]);
        exit(1);
    }
    ch = argv[1][0];
    if ((fp = fopen(argv[2], "r")) == NULL)
    {
        printf("Невозможно открыть %s\n", argv[2]);
        exit(1);
    }
    while (fgets(line,BUF,fp) != NULL)
    {
        if (has_ch(ch,line))
```

```
            fputs(line,stdout);
        }
        fclose(fp);
        return 0;
    }
    int has_ch(char ch, const char * line)
    {
        while (*line)
            if (ch == *line++)
                return(1);
        return 0;
    }
```

Функции `fgets()` и `fputs()` работают вместе, поскольку `fgets()` оставляет в строке символ новой строки `\n`, созданный в результате нажатия клавиши <Enter>, а функция `fputs()` не добавляет его, как это делает функция `puts()`.

6. Различие между двоичным и текстовым файлом определяется системно-зависимыми особенностями этих файловых форматов. Различие между двоичным и текстовым потоком заключается в преобразованиях, выполняемых программой во время считывания или записи потоков. (Двоичный поток не выполняет никаких преобразований; текстовый поток может выполнять преобразования символов новой строки и других символов.)
7. а. При сохранении числа `8238201` с помощью функции `fprintf()` оно сохраняется в виде семи символов, занимающих 7 байтов. При использовании функции `fwrite()` число сохраняется в виде двоичного представления 4-байтового целого числового значения.

   б. Ни в чем. В обоих случаях символ сохраняется в виде 1-байтового двоичного кода.
8. Первый оператор представляет собой всего лишь сокращенную форму записи второго. Третий оператор выполняет запись в стандартное устройство сообщений об ошибках. Обычно стандартные сообщения об ошибках направляются туда же, куда и стандартный вывод, но переадресация стандартного вывода не оказывает влияния на стандартные сообщения об ошибках.
9. Режим `"r+"` позволяет выполнять чтение и запись в любом месте файла, поэтому он наиболее подходит в данном случае. Режим `"a+"` позволяет только дописывать данные в конец файла, а режим `"w+"` начинает работу с “чистого листа”, удаляя предыдущее содержимое файла.

Ответы на вопросы для самоконтроля из главы 14

1. Правильное ключевое слово — `struct`, а не `structure`. Шаблон требует наличия либо дескриптора перед открывающей скобкой, либо имени переменной после закрывающей скобки. Кроме того, символ точки с запятой должен присутствовать после выражения `* togs` и в конце шаблона.

2. Вывод имеет следующий вид:

```
6 1
22 Spiffo Road
S p
```

3.

```
struct month {
   char name[10];
   char abbrev[4];
   int days;
   int monumb;
};
```

4.

```
struct month months[12] =
{
   {"Январь", "янв", 31, 1},
   {"Февраль", "фев", 28, 2},
   {"Март", "мар", 31, 3},
   {"Апрель", "апр", 30, 4},
   {"Май", "май", 31, 5},
   {"Июнь", "июн", 30, 6},
   {"Июль", "июл", 31, 7},
   {"Август", "авг", 31, 8},
   {"Сентябрь", "сен", 30, 9},
   {"Октябрь", "окт", 31, 10},
   {"Ноябрь", "ноя", 30, 11},
   {"Декабрь", "дек", 31, 12}
};
```

5.

```
extern struct month months[];
int days(int month)
{
   int index, total;

   if (month < 1 || month > 12)
      return(-1); /* сигнал ошибки */
   else
   {
      for (index = 0, total = 0; index < month; index ++)
         total += months[index].days;
      return( total);
   }
}
```

Обратите внимание, что значение `index` на единицу меньше номера месяца. Это связано с тем, что нумерация индексов в массивах начинается с 0. Поэтому необходимо использовать операцию сравнения `index < month`, а не `index <= month`.

6.

a. Чтобы можно было использовать функцию `strcpy()`, включите заголовочный файл `string.h`:

```
typedef struct lens {    /* дескриптор структуры lens         */
   float foclen;         /* фокусное расстояние в миллиметрах */
   float fstop;          /* диафрагма                         */
   char brand[30];       /* марка производителя               */
} LENS;

LENS bigEye[10];
bigEye[2].foclen = 500;
bigEye[2] fstop = 2.0;
strcpy(bigEye[2].brand, "Remarkatar");
```

б.

```
LENS bigEye[10] = { [2] = {500, 2, "Remarkatar"} };
```

7.

а.

```
6
Arcturan
cturan
```

б. Для этого можно использовать имя структуры и указатель:

```
deb.title.last
pb->title.last
```

в. Одна из возможных версий выглядит следующим образом:

```
#include <stdio.h>
#include "starfolk.h" /* обеспечение доступности определений структуры */
void prbem (const struct bem * pbem )
{
   printf("%s %s - это %d-ветвленный %s.\n", pbem->title.first,
      pbem->title.last, pbem->limbs, pbem->type);
}
```

8.

а. `willie.born`

б. `pt->born`

в. `scanf("%d", &willie.born);`

г. `scanf("%d", &pt->born);`

д. `scanf("%s", willie.name.lname);`

е. `scanf("%s", pt->name.lname);`

ж. `willie.name.fname[2]`

з. `strlen(willie.name.fname) + strlen(willie.name.lname)`

9. Один из возможных вариантов имеет вид:

```
struct car {
   char name[20];
   float hp;
   float epampg;
   float wbase;
   int year;
};
```

10. Функции могли бы иметь следующий вид:

```
struct gas {
   float distance;
   float gals;
   float mpg;
};

struct gas mpgs(struct gas trip)
{
   if (trip.gals > 0)
      trip.mpg = trip.distance / trip.gals ;
   else
      trip.mpg = -1.0;
   return trip;
}

void set_mpgs(struct gas &ptrip)
{
   if (ptrip->gals > 0)
      ptrip->mpg = ptrip->distance / ptrip->gals ;
   else
      ptrip->mpg = -1.0;
}
```

Обратите внимание, что первая функция не может непосредственно менять значения в вызывающей программе, поэтому для передачи информации необходимо использовать возвращаемое значение:

```
struct gas idaho = {430.0, 14.8}; // установка значений двух первых
членов
idaho = mpgs(idaho);              // переустановка структуры
```

Однако вторая функция обращается к исходной структуре непосредственно:

```
struct gas ohio = {583, 17.6}; // установка значений двух первых членов
set_mpgs(ohio);                // установка значения третьего члена
```

11. `enum choices {no, yes, maybe};`

12. `char * (*pfun)(char *, char);`

13.
```
double sum(double, double);
double diff(double, double);
double times(double, double);
double divide(double, double);
double (*pf1[4])(double, double) = {sum, diff, times, divide};
```

Или для упрощения кода последнюю строку можно заменить следующими двумя строками:

```
typedef double (*ptype)(double, double);
ptype pf[4] = {sum, diff, times, divide};
```

Ответы на вопросы для самоконтроля из главы 15

1. а. `00000011`
   б. `00001101`
   в. `00111011`
   г. `01110111`
2. а. `21, 025, 0x15`
   б. `85, 0125, 0x55`
   в. `76, 0114, 0x4C`
   г. `157, 0235, 0x9D`
3. а. `252`
   б. `2`
   в. `7`
   г. `7`
   д. `5`
   е. `3`
   ж. `28`
4. а. `255`
   б. `1` (“не ложно” равно “истинно”)
   в. `0`
   г. `1` (“истинно” И “истинно” равно “истинно”)
   д. `6`
   е. `1` (“истинно” ИЛИ “истинно” равно “истинно”)
   ж. `40`
5. В двоичной форме маска имеет вид `1111111`, в десятичной — `127`, в восьмеричной `0177`, а в шестнадцатеричной — `0x7F`.
6. Оба выражения `bitval *= 2` и `bitval << 1` удваивают текущее значение переменной `bitval`, поэтому они эквивалентны. Однако выражения `mask += bitval` и `mask |= bitval` оказывают одинаковое влияние, только если переменные `bitval` и `mask` не имеют ни одного общего установленного бита. Например, `2 | 4` равно `6`, но этому же значению равен результат выражения `3 | 6`.
7. а.
```
struct tb_drives {
    nsigned int diskdrives   : 2;
    unsigned int             : 1;
    unsigned int cdromdrives : 2;
    unsigned int             : 1;
    unsigned int harddrives  : 2;
};
```

б.
```
struct kb_drives {
   unsigned int harddrives  : 2;
   unsigned int             : 1;
   unsigned int cdromdrives : 2;
   unsigned int             : 1;
   unsigned int diskdrives  : 2;
};
```

Ответы на вопросы для самоконтроля из главы 16

1.

а. Результатом выполнения кода будет допустимое выражение `dist = 5280 * miles;`

б. Результатом будет допустимое выражение `plort = 4 * 4 + 4;`, но если в действительности требуется выполнить вычисление выражения `4 * (4 + 4)`, следует воспользоваться выражением `#define POD (FEET + FEET)`.

в. Результатом будет недопустимое выражение `nex = = 6;;`. Пользователь явно забыл, что пишет макрос для препроцессора, а не программу на языке С.

г. Результирующее выражение `y = y + 5` допустимо. Выражение `berg = berg + 5 * lob;` допустимо, но, вероятно, представляет не тот результат, который хотел получить пользователь. Выражение `est = berg + 5/y + 5;` допустимо, но, скорее всего, представляет не тот результат, который хотел получить пользователь. Выражение `nilp = lob *-berg + 5;` допустимо, но, вероятно, представляет не тот результат, к которому стремился пользователь.

2. `#define NEW(X) ((X) + 5)`

3. `#define MIN(X,Y) ( (X) < (Y) ? (X) : (Y) )`

4. `#define EVEN_GT(X,Y) ( (X) > (Y) && (X) % 2 == 0 ? 1 : 0 )`

5. `#define PR(X,Y) printf(#X " is %d and " #Y " is %d\n", X,Y)`

   Поскольку в этом макросе переменные `X` и `Y` не являются объектами каких-то других операций (подобных умножению), выражения можно не заключать в скобки.

6.

а. `#define QUARTERCENTURY 25`

б. `#define SPACE ' '`

в. `#define PS() putchar(' ')`

   или

   `#define PS() putchar(SPACE)`

г. `#define BIG(X) ((X) + 3)`

д. `#define SUMSQ(X,Y) ((X)*(X) + (Y)*(Y))`

7. Попытайтесь воспользоваться следующим определением:

   `#define P(X) printf("имя: "#X"; значение: %d; адрес: %p\n", X, &X)`

   Или, если используемая реализация не распознает спецификацию адреса `%p`, попытайтесь использовать спецификацию `%u` или `%lu`.

8. Используйте директивы условной компиляции. Один из возможных способов предусматривает применение директивы #ifdef:

```
#define _SKIP_ /* удалите эту строку, если пропуск кода не требуется */
#ifndef _SKIP_
   /* код, который нужно пропустить */
#endif
```

9.

```
#ifdef PR_DATE
   printf("Date = %s\n", __DATE__);
#endif
```

10. Аргумент argv должен быть объявлен в качестве типа char *argv[]. Аргументы командной строки хранятся в виде строк, поэтому вначале программа должна преобразовать строку, хранящуюся в элементе массива argv[1] в значение типа double — например, с помощью функции atof() из библиотеки stdlib.h. Чтобы можно было использовать функцию sqrt(),в программу потребуется включить заголовочный файл math.h. Прежде чем извлекать квадратный корень, программа должна выполнить проверку на предмет передачи отрицательных значений.

11.

а. Вызов функции должен выглядеть подобно следующему:

```
qsort( (void *)scores, (size_t) 1000, sizeof (double), comp);
```

б. Определение функции сравнения может выглядеть следующим образом:

```
int comp(const void * p1, const void * p2)
{
   /* Для получения доступа к значениям необходимо */
   /* использовать указатели на константу int */
   /* Приведения типов не обязательны в C, но */
   /* обязательны в C++ */
   const int * a1 = (const int *) p1;
   const int * a2 = (const int *) p2;

   if (*a1 > *a2)
      return -1;
   else if (*a1 == *a2)
      return 0;
   else
      return 1;
}
```

12.

а. Вызов функции должен выглядеть подобно следующему:

```
memcpy(data1, data2, 100 * sizeof(double));
```

б. Вызов функции должен выглядеть подобно следующему:

```
memcpy(data1, data2 + 200 , 100 * sizeof(double));
```

Ответы на вопросы для самоконтроля из главы 17

1. Определение типа данных заключается в определении способа хранения данных и набора функций манипулирования данными.
2. Обход списка может выполняться только в одном направлении, поскольку каждая структура содержит адрес следующей, но не предыдущей структуры. Определение структуры можно было бы изменить, чтобы каждая структура содержала два указателя — один на предыдущую структуру и один на следующую. Конечно, программа должна была бы присваивать соответствующие адреса этим указателям при каждом добавлении новой структуры.
3. ADT — аббревиатура от *abstract data type* (абстрактный тип данных). ADT представляет собой формальное определение свойств типа и операций, которые можно выполнять с этим типом. ADT должен быть выражен в обобщенных терминах, а не терминах какого-то конкретного языка программирования или нюансов реализации.
4. **Преимущества непосредственной передачи переменной**. Данная функция проверяет очередь, но не должна ее изменять. Непосредственная передача переменной очереди означает, что функция работает с копией исходных данных, что гарантирует невозможность их изменения функцией. При непосредственной передаче переменной не нужно помнить о необходимости использования операции адресации или указателя.

   Недостатки непосредственной передачи переменной. Программа должна зарезервировать достаточный объем памяти для хранения переменной, а затем скопировать информацию из оригинала в копию. Если переменная представляет крупную структуру, ее использование будет сопряжено с большими затратами времени и памяти.

   Преимущества передачи адреса переменной. Если переменная представляет крупную структуру, передача адреса и доступ к исходным данным выполняются быстрее и требуют меньшего объема памяти, чем при передаче переменной.

   Недостатки передачи адреса переменной. Необходимо помнить о применении операции адресации или указателя. В K&R C функция могла бы неумышленно изменить исходные данные, но этой опасности можно избежать, используя спецификатор `const` стандарта ANSI C.
5. а.

Имя типа:	Стек.
Свойства типа:	Может содержать упорядоченную последовательность элементов.
Операции типа:	Инициализация стека пустым значением. Определение того, является ли стек пустым. Определение того, является ли стек полным. Добавление элемента в верхушку стека (заталкивание элемента) Удаление и восстановление элемента из верхушки стека (выталкивание элемента).

   б. Ниже представлен код реализации стека в виде массива, но это влияет только на определение структуры и нюансы определений функций. Реализация не влияет на интерфейс, описанный прототипами функций.

```
/* stack.h -- интерфейс стека */
#include <stdbool.h>
/* ЗДЕСЬ ВСТАВЬТЕ ТИП ЭЛЕМЕНТА */
/* НАПРИМЕР, typedef int Item; */
#define MAXSTACK 100
typedef struct stack
{
Item items[MAXSTACK]; /* содержит сведения о стеке          */
int top;              /* индекс первой пустой ячейки        */
} Stack;
/* операция:          инициализация стека                    */
/* начальное условие: ps указывает на стек                   */
/* конечное условие:  стек инициализирован пустым значением  */
void InitializeStack(Stack * ps);
/* операция:          проверяет, является ли стек полным          */
/* начальное условие: ps указывает на ранее инициализированный стек*/
/* конечное условие:  возвращает значение true, если стек полон,  */
/*                    иначе возвращает значение false             */
bool FullStack(const Stack * ps);
/* операция:          проверяет, является ли стек пустым          */
/* начальное условие: ps указывает на ранее инициализированный стек*/
/* конечное условие: возвращает значение true, если стек пуст,    */
/*                    иначе возвращает значение false             */
bool EmptyStack(const Stack *ps);
/* операция:          заталкивает элемент в стек                  */
/* начальное условие: ps указывает на ранее инициализированный стек*/
/*                    элемент должен помещаться                   */
/*                    в верхушку стека                            */
/* конечное условие:  если стек не полон, элемент помещается      */
/*                    в верхушку стека и функция возвращает       */
/*                    значение true; иначе стек остается          */
/*                    неизменным, и функция возвращает            */
/*                    значение false                              */
bool Push(Item item, Stack * ps);
/* операция:          удаляет элемент из верхушки стека           */
/* начальное условие: ps указывает на ранее инициализированный стек*/
/* конечное условие:  если стек не пуст, элемент в верхушке       */
/*                    стека копируется в *pitem и удаляется       */
/*                    из стека, а функция возвращает              */
/*                    значение true; если операция                */
/*                    опустошает стек, стек                       */
/*                    переустанавливается в пустое состояние.     */
/*                    Если стек пуст с самого начала, он          */
/*                    остается неизменным, а функция              */
/*                    возвращает значение false                   */
bool Pop(Item *pitem, Stack * ps);
```

6. Максимальное количество требуемых сравнений:

Количество элементов	*Последовательный поиск*	*Бинарный поиск*
3	3	2
1023	1023	10
65535	65535	16

7. См. рис. А.1.
8. См. рис. А.2.

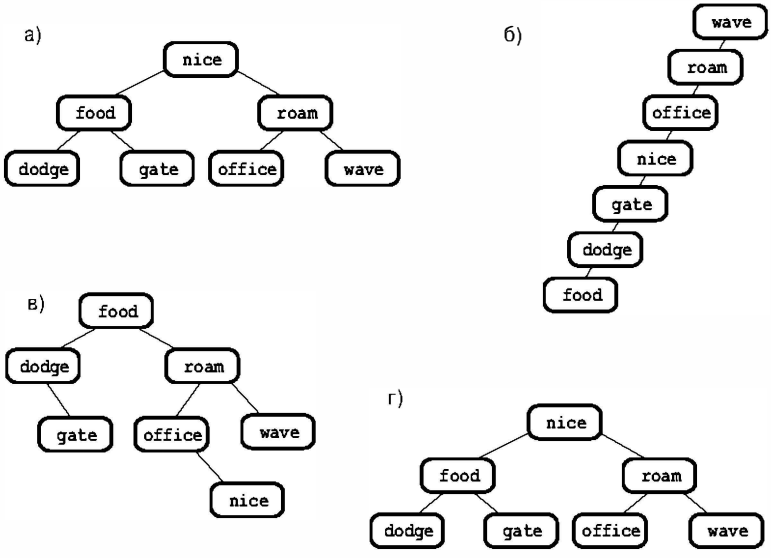


Рис. А.1. *Дерево бинарного поиска слов*

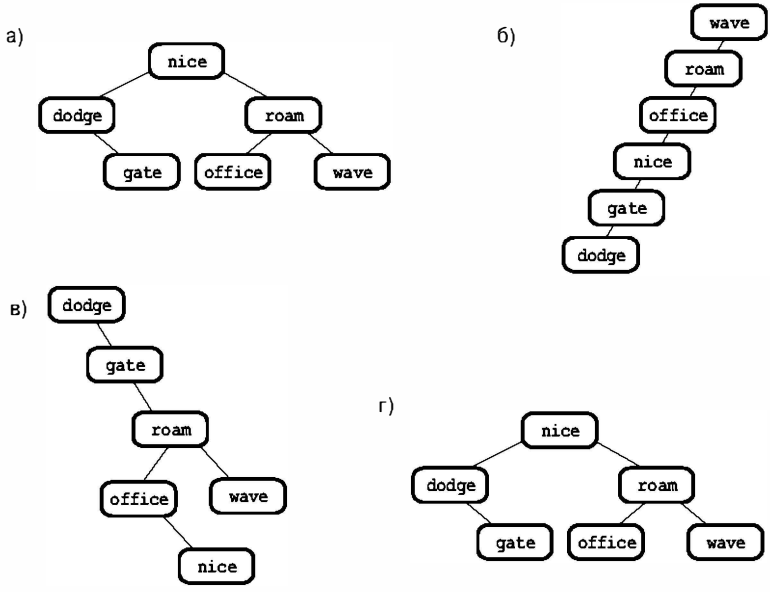


Рис. А.2 *Дерево бинарного поиска слов после удаления*

ПРИЛОЖЕНИЕ Б

Справочный раздел

В этой части книги представлен итоговый перечень базовых средств C наряду с более детализированным рассмотрением определенных тем. Ниже приведен список разделов данного приложения.

- Раздел I. Дополнительные источники информации
- Раздел II. Операции C
- Раздел III. Базовые типы и классы памяти
- Раздел IV. Выражения, операторы и поток управления программы
- Раздел V. Стандартная библиотека ANSI C с дополнениями C99
- Раздел VI. Расширенные целочисленные типы
- Раздел VII. Расширенная поддержка символов
- Раздел VIII. Расширенные средства вычислений C99
- Раздел IX. Различия между C и C++

Раздел I. Дополнительные источники информации

Если вы хотите узнать больше о языке C и программировании, вам будут полезны следующие ссылки.

Журнал

C/C++ Users Journal

Этот ежемесячный журнал (с подзаголовком *Advanced Solutions for C/C++ Programmers* (Профессиональные решения для программистов на C/C++)) является очень полезным ресурсом для всех программистов C и C++.

Сетевые ресурсы

Программисты на C помогали создавать Internet, а Internet может помочь вам в изучении C. Internet всегда растет и изменяется; перечисленные ниже ресурсы — лишь пример того, что вы можете найти.

В качестве возможного места старта, если у вас есть специфические вопросы о C, или же вы хотите расширить свои знания, обратитесь на сайт C FAQ (Frequently Asked Questions — часто задаваемые вопросы):

```
http://www.eskimo.com/~scs/C-faq/top.html
```

Если у вас есть вопросы по библиотеке C, то соответствующую информацию можно найти на следующем сайте:

```
http://www.dinkumware.com/htm_cl/index.html
```

Приведенный ниже сайт предлагает всестороннюю дискуссию об указателях:

```
http://pweb.netcom.com/~tjensen/ptr/pointers.htm
```

Также вы можете использовать средства поиска — такие, как Google и Yahoo! Попробуйте поискать интересующие вас сайты и темы здесь:

```
http://www.google.com
http://search.yahoo.com
```

Вы можете использовать расширенные средства поиска, представленные на этих сайтах для более тонкой настройки поиска.

Также доступно множество онлайновых руководств. Вот лишь несколько из них:

```
http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/teaching_C.html
http://www.strath.ac.uk/CC/Courses/NewCcourse/ccourse.html
```

Группы новостей позволяют задавать вопросы через Internet. Обычно группы новостей доступны с помощью программ чтения новостей, работающих через учетную запись, предоставляемую поставщиком Internet-услуг. Другой способ доступа — через Web-браузер, по следующему адресу:

```
http://groups.google.com
```

Вам следует сначала потратить некоторое время на чтение групп новостей, чтобы составить представление о том, как они организованы по темам. Например, если у вас есть вопросы о том, как сделать что-либо на C, попробуйте такую группу новостей:

```
comp.lang.c
```

Здесь вы всегда найдете людей, готовых и желающих помочь. Вопросы должны касаться стандартного языка C. Не спрашивайте здесь о том, как организовать небуферизованный ввод в Unix — на то имеются специализированные группы новостей, посвященные специфичным для платформ вопросам. Не спрашивайте их, как вам справиться с домашними проблемами!

Если у вас есть вопросы об интерпретации стандарта C, попробуйте следующую группу:

```
comp.std.c
```

Но не задавайте здесь вопросов о том, как объявлять указатель на трехмерный массив; это вопрос такого рода, что больше подходит для группы `comp.lang.c`.

И, наконец, если вы интересуетесь историей C, то Деннис Ритчи (Dennis Ritchie), создатель C, описал происхождение и разработку C в статье на следующем сайте:

```
http://cm.bell-labs.com/cm/cs/who/dmr/chist.html
```

Книги по языку C

- Feuer, Alan R. *The C Puzzle Book, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1989

  Эта книга содержит множество программ, чей вывод вы должны предсказать. Предсказание вывода программы на основе ее исходного кода дают хорошую

возможность проверить и расширить свое понимание C. Эта книга включает также ответы и объяснения.

- Керниган, Брайан У., Ритчи, Денис М.. *Язык программирования C, 2-е издание.* Издательский дом "Вильямс", 2006

  Это второе издание первой книги о языке C. (Отметьте, что среди ее авторов Деннис Ритчи — создатель языка C). Первое издание представило определение "K&R" C — неофициальный стандарт, существующий на протяжении многих лет. Новое издание включило в себя изменения ANSI, основанные на черновом проекте ANSI, который был стандартом на момент написания книги. Книга включает множество интересных примеров. Однако она предполагает знакомство читателя с системным программированием.

- Koenig, Andrew. *C Traps and Pitfalls.* Reading, MA: Addison-Wesley, 1988

  Название ("Капканы и ловушки C") говорит само за себя.

- Summit, Steve. *C Programming FAQs.* Reading, MA: Addison-Wesley, 1995

  Это расширенная версия Internet FAQ.

Книги по программированию

- Kernighan, Brian W. and P.J. Plauger. *The Elements of Programming Style, Second Edition.* New York: McGraw-Hill, 1978

  Эта книга содержит тонкие, ранее не издававшиеся классические эскизы примеров, собранные из других текстов для иллюстрации того, что нужно, и чего не нужно делать при написании ясных и эффективных программ.

- Кнут, Дональд Э. *Искусство программирования*, том 1. Основные алгоритмы, 3-е издание. Издательский дом "Вильямс", 2000

  В этом обширном классическом руководстве во всех подробностях рассматриваются представления данных и анализ алгоритмов. Весьма глубокое и по природе своей — математическое. Том 2 (Получисленные методы; Издательский дом "Вильямс", 2000) включает расширенное обсуждение проблемы псевдослучайных чисел. Том 3 (Сортировка и поиск; Издательский дом "Вильямс", 2000), как следует из названия, посвящен вопросам сортировки и поиска. Примеры в книгах представлены в псевдокоде и на языке ассемблера.

- Sedgewick, Robert. *Algorithms in C: Fundamentals, Data Structures, Sorting, Searching.* Reading, MA: Addison-Wesley, 1995

  Как и следовало ожидать, книга посвящена структурам данных, сортировке и поиску.

Справочные руководства

- Harbison, Samuel P. and Steele, Guy L. *C: A Reference Manual, Fifth Edition.* Englewood Cliffs, NJ: Prentice Hall, 2002

  Это справочное руководство представляет правила языка C и описывает большую часть стандартных библиотечных функций. Включает обсуждение C99 и множество примеров.

- Plauger, P.J. *The Standard C Library.* Englewood Cliffs, NJ: Prentice Hall, 1992
  Огромное справочное руководство, описывает стандартные библиотечные функции, но с более подробными объяснениями, чем можно найти в типичном руководстве по компилятору.
- *The International C Standard. ISO/IEC 9899:1999*
  На момент написания книги этот стандарт доступен для загрузки за $18 на сайте `www.ansi.org`. Не рассчитывайте изучить C по этому документу, потому что он не является учебным пособием. Вот лишь одна довольно-таки красноречивая цитата: "Если более чем одно объявление определенного идентификатора видимо в любой точке единицы трансляции, используется синтаксический контекст для обращения к различным сущностям."

Книги по C++

- Prata, Stephen. *C++ Primer Plus, Fifth Edition.* Indianapolis, IN: Sams Publishing, 2005
  Эта книга представляет собой введение в язык C++ и философию объектно-ориентированного программирования.
- Stroustrup, Bjarne. *The C++ Programming Language, Third Edition.* Reading, MA: Addison-Wesley, 1997
  Книга, написанная создателем C++, представляет сам язык C++ и включает в себя справочник по C++.

Раздел II. Операции C

Язык C богат операциями. В табл. RS.II.1 перечислены операции C в порядке убывания приоритетов с указанием направления ассоциации. Если не указано иначе, все операции являются бинарными (с двумя операндами). Обратите внимание, что некоторые бинарные и унарные операции, такие как `*` (умножение) и `*` (разыменование), обозначаются одним и тем же символом, но имеют разный приоритет.

Таблица RS.II.1. Операции в C

Операции (в порядке снижения приоритетов)	*Ассоциативность*
`++` (постфиксный) `--` (постфиксный) `( )` (вызов функции) `[ ]` `{}` (составной литерал) `.` `->`	Слева направо
`++` (префиксный) `--` (префиксный) `-` `+` `~` `!` `sizeof` `*` (разыменование) `&` (адрес) `(type)` (все унарные)	Справа налево
(имя типа)	Справа налево
`*` `/` `%`	Слева направо
`+` `-` (оба бинарные)	Слева направо
`<<` `>>`	Слева направо
`<` `>` `<=` `>=`	Слева направо

Окончание табл. RS.II.1

Операции (в порядке снижения приоритетов)	*Ассоциативность*
`== !=`	Слева направо
`&`	Слева направо
`^`	Слева направо
`|`	Слева направо
`&&`	Слева направо
`||`	Слева направо
`? :` (условное выражение)	Справа налево
`= *= /= %= += -= <<= >>= &= |= ^=`	Справа налево
`,` (операция запятой)	Слева направо

Арифметические операции

`+` прибавляет значение справа к значению слева.

`+` (унарная операция) — дает значение, равное модулю операнда справа (с тем же знаком).

`-` вычитает значение справа из значения слева.

`-` (унарная операция) — дает значение, равное модулю операнда справа (с противоположным знаком).

`*` умножает значение справа на значение слева.

`/` делит значение слева на значение справа. Если оба операнда целочисленные, результат усекается до целого.

`%` дает остаток от целочисленного деления значения слева на значение справа (только для целых чисел).

`++` добавляет 1 к значению переменной справа (в префиксном режиме), либо прибавляет 1 к значению переменной слева (в постфиксном режиме).

`--` подобен `++`, но вычитает 1.

Операции отношений

Каждая из следующих операций сравнивает значение слева от нее со значением справа:

`<` меньше чем

`<=` меньше или равно

`==` равно

`>=` больше или равно

`>` больше чем

`!=` не равно

Выражения сравнения

Простейшее выражение сравнения состоит из операции отношения с двумя операндами. Если сравнение истинно, выражение сравнения имеет значение 1, а если ложно — сравнивающее выражение имеет значение 0. Ниже показаны два примера:

`5 > 2`	истинно, имеет значение 1.
`(2 + a) == a`	ложно, имеет значение 0.

Операции присваивания

Язык C имеет одну базовую и несколько комбинированных операций присваивания. Базовая форма записывается, как одиночный знак равенства:

`=` присваивает значение справа l-значению слева.

Каждый из следующих операторов присваивания обновляет l-значение, стоящее слева, значением, указанным справа, используя указанную операцию (с помощью R-H обозначается правый операнд, а посредством L-H — левый):

`+=` добавляет число R-H к значению переменной L-H и помещает результат в переменную L-H.

`-=` вычитает число R-H из значения переменной L-H и помещает результат в переменную L-H.

`*=` умножает число R-H на значение переменной L-H и помещает результат в переменную L-H.

`/=` делит значение переменной L-H на число R-H и помещает результат в переменную L-H.

`%=` получает остаток от деления переменной L-H на число R-H и помещает результат в переменную L-H.

`&=` выполняет операцию “И” над L-H и R-H и помещает результат в переменную L-H.

`|=` выполняет операцию “ИЛИ” над L-H и R-H и помещает результат в переменную L-H.

`^=` выполняет операцию исключающего “ИЛИ” над L-H и R-H и помещает результат в переменную L-H.

`>>=` выполняет поразрядный сдвиг вправо значения L-H на величину R-H и помещает результат в переменную L-H.

`<<=` выполняет поразрядный сдвиг влево значения L-H на величину R-H и помещает результат в переменную L-H.

Пример

`rabbits *= 1.6;` дает тот же эффект, что и `rabbits = rabbits * 1.6;`

Логические операции

Логические операции обычно принимают в качестве операндов выражения сравнения.

Операция ! принимает один операнд, остальные — два: слева и справа.

`&&`	И
`||`	ИЛИ
`!`	НЕ

Логические выражения

`выражение1 && выражение2`	истинно тогда и только тогда, когда оба выражения истинны.
`выражение1 || выражение2`	истинно тогда, когда любое из выражений, либо оба сразу, истинны.
`!выражение`	истинно, когда выражение ложно и наоборот.

Порядок оценки логических выражений

Логические выражения оцениваются слева направо. Оценка прекращается, как только становится ясно, что выражение ложно.

Примеры

`6 > 2 && 3 == 3`	истинно.
`! ( 6 > 2 && 3 == 3 )`	ложно.
`x != 0 && 20/x < 5`	Второе выражение оценивается только тогда, когда `x` не равно 0.

Условная операция

Операция `? :` принимает три операнда, каждый из которых является выражением. Они располагаются следующим образом:

```
выражение1 ? выражение2 : выражение3
```

Значение полного выражения равно значению `выражение2`, если `выражение1` истинно, и значению `выражение3` в противном случае.

Примеры

`( 5 > 3 ) ? 1 : 2`	имеет значение 1.
`( 3 > 5 ) ? 1 : 2`	имеет значение 2.
`( a > b ) ? a : b`	имеет большее значение среди `a` и `b`.

Операции, связанные с указателями

`&`	операция взятия адреса. Когда за ней следует имя переменной, `&` возвращает ее адрес.
`*`	операция разыменования. Когда за ней следует указатель, `*` возвращает значение, хранимое по указанному адресу.

Примеры

Здесь &nurce — это адрес переменной nurce:

```
nurse = 22;
ptr = &nurse; /* указатель на nurse */
val = *ptr;
```

Общий эффект состоит в присваивании переменной val значения 22.

Операции знаков

- знак минуса (унарный), меняет знак операнда на противоположный.
+ знак плюса (унарный) оставляет знак операнда без изменений.

Операции структур и объединений

Структуры и объединения используют операции для идентификации их отдельных элементов. Операция принадлежности используется со структурами и объединениями, а косвенная операция принадлежности — с указателями на структуры и объединения.

Операция принадлежности

Операция принадлежности (.) используется с именем структуры или объединения для указания элемента структуры или объединения. Если name — имя структуры или объединения, а member — элемент, определенный в шаблоне структуры, то name.member идентифицирует этот элемент структуры. Типом name.member является тип, заданный для member. Операция принадлежности может использоваться в той же манере с объединениями.

Пример

```
struct {
   int code;
   float cost;
} item;
item.code = 1265;
```

Этот оператор присваивает значение элементу code структуры item.

Косвенная операция принадлежности (или операция указателя на структуру)

Косвенная операция принадлежности (->) используется с указателем на структуру или объединение для идентификации элемента структуры или объединения. Предположим, что ptrstr — указатель на структуру, а member — элемент, определенный в шаблоне структуры. Тогда ptrstr->member идентифицирует этот элемент структуры, на которую указывает указатель. Точно таким же образом косвенная операция принадлежности может применяться в отношении объединений.

Пример

```
struct {
   int code;
   float cost;
} item, * ptrst;
ptrst = &item;
ptrst->code = 3451;
```

Этот фрагмент программы присваивает значение элементу code структуры item. Следующие три выражения эквивалентны:

```
ptrst->code  item.code  (*ptrst).code
```

Поразрядные операции

Все представленные ниже поразрядные операции за исключением ~ являются бинарными.

- `~` унарная операция "НЕ", дает в результате значение операнда, в котором каждый бит инвертирован.
- `&` операция "И", дает в результате значение, в котором каждый бит установлен в 1, если соответствующие биты в обоих операндах равны 1.
- `|` операция "ИЛИ", дает в результате значение, в котором каждый бит установлен в 1, когда любой из соответствующих битов операндов, либо оба сразу, равны 1.
- `^` операция исключающего "ИЛИ", дает в результате значение, в котором каждый бит установлен в 1, когда любой из соответствующих битов операндов (но не оба сразу), равны 1.
- `<<` операция сдвига влево, дает значение, полученное в результате сдвига разрядов левого операнда влево на количество позиций, указанное правым операндом. Освобождаемые места заполняются нулями.
- `>>` операция сдвига вправо, дает значение, полученное в результате сдвига разрядов левого операнда вправо на количество позиций, указанное правым операндом. Для беззнаковых целых освобождаемые места заполняются нулями. Поведение для целых со знаком зависит от реализации.

Примеры

Предположим, имеются следующие переменные:

```
int x = 2;
int y = 3;
```

Тогда `x & y` дает в результате значение 2, потому что только один бит "включен" как в x, так и в y. Также `x<<y` дает в результате значение 12, поскольку это значение получается, когда битовый шаблон 3 сдвигается на 2 разряда влево.

Прочие операции

- Операция sizeof возвращает размер операнда, находящегося справа, измеренный в единицах, представляющих размер значения char. Обычно размер значения char равен 1 байту. Операнд может быть спецификатором типа в скобках, как в случае sizeof(float), или же именем определенной переменной, массива, и тому подобного, как в случае sizeof foo. Типом выражения sizeof является size_t.
- (type) — это операция приведения, она преобразует следующее за ней значение к типу, указанному с помощью ключевого слова в скобках. Например, (float)9 преобразует целое число 9 в число с плавающей запятой 9.0.
- , — это операция запятой, она связывает два выражения в одно и гарантирует, что левое выражение выполняется первым. Значением всего результирующего выражения является значение правого выражения. Операция запятой обычно используется для включения большего количества информации в управляющее выражение цикла for.

Пример

```
for (step = 2, fargo = 0; fargo < 1000; step *= 2)
fargo += step;
```

Раздел III. Базовые типы и классы памяти

Обзор: базовые типы данных

Базовые типы данных C подразделяются на две категории: целые числа и числа с плавающей запятой. Различные вариации характеризуются разными диапазонами значений и точностью.

Ключевые слова

Базовые типы данных описываются следующими восемью ключевыми словами: int, long, short, unsigned, char, float, double и signed (ANSI C).

Целые со знаком

Целые со знаком могут иметь положительные и отрицательные значения:

- int — базовый целочисленный тип для данной системы.
- long или long int — может содержать целое, как минимум, равное самому большому значению int, а может, и больше; long занимает не менее 32 разрядов.
- Самое большое значение short или short int не больше самого большого int, но может быть меньше. short занимает минимум 16 разрядов. Обычно long больше, чем short, а int — такой же, как один из них. Например, компиляторы C для DOS на IBM PC поддерживают 16-разрядные int и short и 32-разрядный long.
- Тип long long, предусмотренный стандартом C99, имеет размер не менее чем long, и занимает минимум 64 разряда.

Целые без знака

Целые без знака могут иметь только нулевое или положительные значения, что расширяет диапазон допустимых положительных чисел. Используйте ключевое слово `unsigned` перед именем типа: `unsigned int`, `unsigned long`, `unsigned short` или `unsigned long long`. Отдельно `unsigned` — то же самое, что `unsigned int`.

Символы

Символами называются типографские символы, такие как `A`, `&` и `+`. По определению для переменной типа `char` используется один байт памяти. В прошлом наиболее типичным был размер `char`, равный 8 разрядам. Однако возможность языка C обрабатывать расширенные наборы символов может привести к использованию 16-разрядных и даже 32-разрядных символов.

`char` — ключевое слово для обозначения этого типа. Некоторые реализации используют `char` со знаком, другие –`char` без знака. ANSI C позволяет использовать ключевые слова `signed` и `unsigned` для указания требуемой формы `char`. Технически `char`, `unsigned char` и `signed char` — три разных типа, причем тип `char` имеет такое же представление, как один из двух других.

Булевский тип (C99)

Булевский тип C99 — `_Bool`. Это целочисленный тип без знака, который может принимать одно из двух значений: `0` — обозначающий ложь (`false`), и `1` — истину (`true`). Включение заголовочного файла `stdbool.h` позволяет использовать `bool` вместо `_Bool`, `true` — вместо `1` и `false` — вместо `0`, что обеспечивает совместимость кода с C++.

Действительные и комплексные типы с плавающей запятой

Стандарт C99 выделяет два домена типов с плавающей запятой: действительные и комплексные. Вместе они образуют тип с плавающей запятой.

Действительные числа с плавающей запятой могут иметь положительные и отрицательные значения. Язык C распознает три действительных типа с плавающей запятой:

- `float` — базовый тип с плавающей запятой. Может представлять как минимум шесть значащих десятичных разрядов. Обычно занимает 32 разряда.
- `double` — (возможно) более крупная единица хранения чисел с плавающей запятой. Может допускать больше значащих цифр и, возможно, большие экспоненты, чем тип `float`. Может представлять, по меньшей мере, 10 десятичных разрядов. Обычно занимает 64 разряда.
- `long double` — (возможно) еще более крупная единица хранения чисел с плавающей запятой. Может допускать больше значащих цифр и, возможно, большие экспоненты, чем тип `double`.

Комплексные числа имеют два компонента: действительную часть и мнимую часть. Стандарт C99 внутренне организует хранение комплексных чисел в виде массива из двух элементов, где первый представляет действительную часть, а второй — мнимую.

Существуют три типа комплексных чисел:

- `float _Complex` — представляет действительную и мнимую части значениями типа `float`.
- `double _Complex` — представляет действительную и мнимую части значениями типа `double`.
- `long double _Complex` — представляет действительную и мнимую части значениями типа `long double`.

В каждом случае префиксный тип называется *соответствующим действительным типом*. Например, `double` — соответствующий действительный тип для `double_Complex`.

Комплексные типы необязательны для автономных сред, где программы С могут выполняться без операционной системы.

Существует три мнимых типа; они необязательны как для автономных сред, так и сред, где программы на С выполняются под управлением операционной системы. Мнимые числа состоят только из мнимой части. Вот их список:

- `float _Imaginary` — представляет мнимую часть значениями типа `float`.
- `double _Imaginary` — представляет мнимую часть значениями типа `double`.
- `long double _Imaginary` — представляет мнимую часть значениями типа `long double`.

Комплексные числа могут быть инициализированы действительными числами и значением `I`, определенным в `complex.h`, и представляющем `i`, квадратный корень из –1:

```
#include <complex.h>                  // для I
double _Complex z = 3.0;              // действительная часть = 3.0,
                                      // мнимая часть = 0
double _Complex w = 4.0 * I;          // действительная часть = 0.0,
                                      // мнимая часть = 4.0
double Complex u = 6.0 - 8.0 * I;     // действительная часть = 6.0,
                                      // мнимая часть = -8.0
```

Обзор: объявление простой переменной

1. Выберите необходимый тип.
2. Выберите имя переменной.
3. Используйте следующий формат оператора объявления:

   спецификатор-типа имя-переменной;

   спецификатор-типа формируется из одного или более ключевых слов. Вот некоторые примеры:

   ```
   int erest;
   unsigned short cash;
   ```

4. Чтобы объявить более одной переменной одного и того же типа, разделите имена переменных запятыми:

   ```
   char ch, init, ans;
   ```

   В операторе объявления переменную можно инициализировать:

   ```
   float mass = 6.0E24;
   ```

Обзор: классы памяти

Ключевые слова:

`auto`, `extern`, `static`, `register`

Общие комментарии:

Класс памяти переменной определяет ее область видимости, связывание и продолжительность хранения. Класс памяти определяется и местом ее определения, и ассоциированными ключевыми словами. Переменные, определенные вне функций, являются внешними, имеют область видимости в пределах файла, внешнее связывание и статическую продолжительность хранения. Переменные, определенные внутри функции, являются автоматическими, если только не использовано другое ключевое слово. Они имеют область видимости в пределах блока, никакого связывания и автоматическую продолжительность хранения. Переменные, определенные с ключевым словом `static` внутри функции, имеют область видимости в пределах блока, никакого связывания и статическую продолжительность хранения. Переменные, определенные с ключевым словом `static` вне функции, имеют область видимости в пределах файла, внутреннее связывание и статическую продолжительность хранения.

Свойства:

Ниже представлены свойства классов памяти:

Класс памяти	Продолжительность хранения	Область видимости	Связывание	Как объявлена
Автоматический	Автоматическая	Блок	Нет	В блоке
Регистровый	Автоматическая	Блок	Нет	В блоке с ключевым словом `register`
Статический с внешним связыванием	Статическая	Файл	Внешнее	Вне функций
Статический с внутренним связыванием	Статическая	Файл	Внутреннее	Вне функций с ключевым словом `static`
Статический без связывания	Статическая	Блок	Нет	В блоке ключевым словом `static`

Обратите внимание, что ключевое слово `extern` используется только для повторного объявления переменной, которая уже определена как внешняя где-то еще. Объявление переменной вне функции делает ее внешней.

В дополнение к этим классам памяти C представляет распределяемую память. Эта память выделяется вызовом одной из функций семейства `malloc()`, возвращающей указатель, который может быть использован для доступа к памяти. Память остается распределенной до тех пор, пока не будет вызвана функция `free()` либо завершена программа. Доступ к выделенной таким образом памяти возможен из любой функции, которой доступен данный указатель. Например, функция может вернуть значение указателя другой функции, которая затем может обратиться к памяти.

Обзор: квалификаторы

Ключевые слова

Для квалификации переменных используются следующие ключевые слова:

```
const, volatile, restrict
```

Общие комментарии

Квалификатор ограничивает использование переменной определенным образом. Переменная `const` после инициализации не может быть изменена. Компилятор не может предполагать, что переменная `volatile` не изменится некоторым внешним агентом, таким как аппаратное обновление. Указатель, квалифицированный с помощью `restrict`, понимается как обеспечивающий единственный доступ (в определенной области видимости) к блоку памяти.

Свойства

Объявление

```
const int joy = 101;
```

устанавливает, что значение `joy` фиксировано значением `101`.

Объявление

```
volatile unsigned int incoming;
```

устанавливает, что значение `incoming` может измениться между несколькими ее упоминаниями в программе.

Объявление

```
const int * ptr = &joy;
```

устанавливает, что указатель `ptr` не может быть использован для изменения переменной `joy`. Однако указатель может быть переустановлен на другой адрес.

Объявление

```
int * const ptr = &joy;
```

устанавливает, что указатель `ptr` не может изменяться; то есть он может указывать только на `joy`. Однако он может быть использован для изменения значения `joy`.

Прототип

```
void simple(const char * s);
```

устанавливает, что после инициализации формального аргумента `s` любым переданным в функцию `simple()` значением то, на что он указывает, не может быть изменено в теле `simple()`.

Прототип

```
void supple(int * const pi);
```

и эквивалентный ему прототип

```
void supple(int pi[const]);
```

устанавливают, что `supple()` не может изменять значение параметра `pi`.

Прототип

```
void interleave(int * restrict p1, int * restrict p2, int n);
```

указывает, что p1 и p2 имеют исключительный доступ к соответствующим блокам памяти, на которые они указывают; это предполагает, что данные блоки не перекрываются.

Раздел IV. Выражения, операторы и поток управления программы

Обзор: выражения и операторы

В языке C выражения представляют значения, а операторы — исполняемые инструкции компьютеру.

Выражения

Выражение — это комбинация операций и операндов. Простейшее выражение — это просто константа или переменная без операций, такая как `22` или `beebop`. Более сложные примеры выглядят так: `55+22` и `vap=2*(vip+(vup=4))`.

Операторы

Оператор — это команда компьютеру. Любое выражение, за которым следует точка с запятой, формирует оператор, хотя и не обязательно осмысленный. Операторы могут быть простыми или составными. *Простые операторы* завершаются точкой с запятой, как показано в следующих примерах.

Оператор объявления	`int toes;`
Оператор присваивания	`toes = 12;`
Оператор вызова функции	`printf ("%d\n", toes);`
Управляющий оператор	`while (toes < 20) toes = toes + 2;`
Пустой оператор	`; /* ничего не делает */`

(Технически стандарт относит объявления к отдельной категории, а не объединяет их с операторами.)

Составные операторы, или *блоки*, состоят из одного или более операторов (каждый из которых сам может быть составным), заключенных в фигурные скобки. Примером тому может служить следующий оператор `while`:

```
while (years < 100)
{
   wisdom = wisdom + 1;
   printf("%d %d\n", years, wisdom);
   years = years + 1;
}
```

Обзор: оператор `while`

Ключевое слово

Ключевым словом оператора `while` является `while`.

Общие комментарии

Оператор `while` создает цикл, который повторяется, пока управляющее *выражение* не станет равно `false` или нулю. Оператор `while` представляет собой цикл с проверкой условия на входе; решение о выполнении очередной итерации принимается *перед* выполнением тела цикла. Таким образом, существует возможность, что тело цикла не будет выполнено ни разу. Часть `оператор` этой формы может быть как простым оператором, так и составным.

Форма

```
while(выражение)
   оператор
```

Часть *оператор* повторяется до тех пор, пока *выражение* не станет равно `false` или нулю.

Примеры

```
while (n++ < 100)
   printf(" %d %d\n",n, 2*n+1);

while (fargo < 1000)
{
   fargo = fargo + step;
   step = 2 * step;
}
```

Обзор: оператор `for`

Ключевое слово

Ключевым словом оператора `for` является `for`.

Общие комментарии

Оператор `for` использует три управляющих выражения, разделенных точкой с запятой, для управления циклическим процессом. Выражение `инициализация` выполняется однажды, перед любыми другими операторами цикла. Если выражение `проверка` равно `true` (или не ноль), выполняется одна итерация цикла. Затем оценивается выражение `обновление`, после чего вновь оценивается выражение `проверка`. Оператор `while` представляет собой цикл с проверкой условия на входе; решение о выполнении очередной итерации принимается *перед* выполнением тела цикла. То есть, возможно, что цикл не выполнится ни разу. Часть `оператор` также может быть как простым оператором, так и составным.

Форма

```
for (инициализация ; проверка ; обновление)
   оператор
```

Цикл повторяется до тех пор, пока выражение *проверка* не вернет `false` или ноль.

C99 позволяет в часть *инициализация* включать объявление. В этом случае область видимости и продолжительность существования переменной ограничено циклом `for`.

Примеры

```
for (n = 0; n < 10 ; ++n)
   printf("%d %d\n", n, 2 * n+1);

for (int k = 0; k < 10 ; ++k)    // C99
   printf("%d %d\n", k, 2 * k+1);
```

Обзор: оператор `do while`

Ключевые слова

Ключевыми словами операторы `do while` являются `do` и `while`.

Общие комментарии

Оператор `do while` создает цикл, повторяющийся до тех пор, пока проверочное выражение не вернет `false` или ноль. Оператор `do while` является циклом с проверкой на выходе; решение о выполнении очередной итерации принимается *после* выполнения тела цикла. Таким образом, цикл должен выполниться как минимум, один раз. Часть `оператор` этой формы также может быть как одиночным оператором, так и составным.

Форма

```
do
   оператор
while (выражение);
```

Часть *оператор* повторяется до тех пор, пока *выражение* не станет равно `false` или нулю.

Пример

```
do
   scanf("%d", &number)
while(number != 20);
```

Обзор: использование операторов `if` для реализации выбора

Ключевые слова

Ключевыми словами оператора `if` являются `if` и `else`.

Общие комментарии

В каждой из следующих форм оператор может быть как одиночным оператором, так и составным. "Истинное" выражение в общем случае означает такое, которое возвращает отличное от нуля значение.

Форма 1

```
if(выражение)
   оператор
```

Если *выражение* истинно, выполняется *оператор*.

Форма 2

```
if(выражение)
   оператор1
else
   оператор2
```

Если *выражение* истинно, выполняется *оператор1*. В противном случае исполняется *оператор2*.

Форма 3

```
if(выражение1)
   оператор1
else if(выражение2)
   оператор2
else
   оператор3
```

Если *выражение1* истинно, выполняется *оператор1*. Если же *выражение1* ложно, но *выражение2* истинно, выполняется *оператор2*. В противном случае, если оба выражения ложны, выполняется *оператор3*.

Пример

```
if (legs == 4)
   printf("Это, должно быть, лошадь.\n");
else if (legs > 4)
   printf("Это не лошадь.\n");
else /* ног < 4 */
{
   legs++;
   printf("Теперь на одну ногу стало больше.\n");
}
```

Обзор: множественный выбор с помощью `switch`

Ключевые слова

Ключевым словом оператора switch является switch.

Общие комментарии

Управление передается оператору, имеющему выражение в качестве метки. Поток программы затем проходит остальные операторы внутри блока switch, если только не будет перенаправлен. Как выражение, так и метки case должны иметь целочисленные значения (включая тип char), а метки должны быть константами или выражениями, состоящими из констант. Если ни одна метка не соответствует значению выражения, управление переходит к оператору, помеченному меткой default, если она есть. Иначе управление переходит к оператору, следующему за оператором switch. После того, как управление передается по определенной метке, выполняются все последующие операторы внутри switch — до конца switch, или до оператора break — что встретится раньше.

Форма

```
switch (выражение)
{
   case метка1 : оператор1
   case метка2 : оператор2
   default     : оператор3
}
```

Операторов, снабженных метками, может быть более двух, а конструкция default является необязательной.

Примеры

```
switch (value)
   case 1 : find_sum(ar, n);
            break;
   case 2 : show_array(ar, n);
            break;
   case 3 : puts("Всего хорошего!");
            break;
   default: puts("Неправильный выбор, попробуйте еще раз.");
            break;
}
switch (letter)
{
   case 'a' :
   case 'e' : printf("%d - гласная\n", letter);
   case 'c' :
   case 'n' : printf("%d - согласная\n", letter);
   default  : printf("Всего хорошего.\n");
}
```

Если letter имеет значение 'a' или 'e', то печатаются все три сообщения; 'c' и 'n' приводит к выводу двух последних. Все прочие значения вызывают печать лишь последнего сообщения.

Обзор: переходы в программе

Ключевые слова

Ключевыми словами для переходов являются `break`, `continue` и `goto`.

Общие комментарии

Эти три инструкции — `break`, `continue` и `goto` — заставляют поток управления программы перейти из одного места кода в другое.

Команда `break`

Команда `break` может быть использована с любой из трех форм циклов, а также оператором `switch`. Она принуждает поток управления программы пропустить остаток цикла или операторы `switch`, и продолжить выполнение со следующей инструкции после цикла или `switch`.

Пример

```
while ((ch = getchar()) != EOF)
{
   putchar(ch);
   if (ch == ' ')
      break; // прервать цикл
   chcount++;
}
```

Команда `continue`

Команда `continue` может использоваться с любой из трех форм циклов, но не с оператором `switch`. Она заставляет поток управления программы пропустить остаток операторов цикла. В циклах `for` и `while` при этом запускается следующая итерация цикла. В цикле `do while` проверяется условие выхода, а затем, если необходимо, запускается новая итерация.

Пример

```
while ((ch = getchar()) != EOF)
{
   if (ch == ' ')
      continue; // перейти к проверочному условию
   putchar(ch);
   chcount++;
}
```

Этот фрагмент отображает и подсчитывает непробельные символы.

Команда `goto`

Оператор `goto` передает управление программой оператору, снабженному указанной меткой. Метка отделяется от оператора двоеточием. Имена меток подчиняются правилам, определенным для имен переменных. Помеченный оператор может располагаться как до, так и после `goto`.

Форма

```
goto метка;
   метка : оператор
```

Пример

```
top : ch = getchar();
      if (ch != 'y')
        goto top;
```

Раздел V. Стандартная библиотека ANSI C с дополнениями C99

Библиотека ANSI C разбивает функции на несколько групп, с каждой из которых ассоциирован свой заголовочный файл. В этом разделе представлен обзор библиотеки, список заголовочных файлов и краткое описание ассоциированных с ними функций. Некоторые из этих функций (например, некоторые функции ввода-вывода) обсуждаются в тексте более подробно. Но вообще за полным описанием следует обращаться к документации, сопровождающей вашу реализацию, к справочному руководству, или же заглянуть в онлайновые руководства вроде следующего:

`http://www.dinkumware.com/htm_cl/index.html`

Диагностика: `assert.h`

Этот заголовочный файл определяет `assert()` как макрос. Определение макроидентификатора `NDEBUG` перед включением заголовочного файла `assert.h` отключает макрос `assert()`. Выражение, используемое в качестве аргумента, является обычно выражением сравнения или логическим, которое должно быть истинным в данной точке программы, если программа функционирует правильно. В табл. RS.V.1 описан макрос `assert()`.

Таблица RS.V.1. Макрос диагностики

Прототип	*Описание*
`void assert(int exprs);`	Если выражение `exprs` оценивается как ненулевое (или `true`), макрос не делает ничего. Если же `exprs` равно 0 (`false`), `assert()` отображает выражение, номер строки, в которой находится оператор `assert()`, а также имя файла, содержащего этот оператор. Затем вызывается `abort()`.

Комплексные числа: `complex.h` (C99)

Стандарт C99 добавил расширенную поддержку вычислений с комплексными числами. Реализации могут представлять тип `_Imaginary` в дополнение к типу `_Complex`. Заголовочный файл определяет макросы, перечисленные в табл. RS.V.2.

Таблица RS.V.2. Макросы complex.h

Прототип	*Описание*
`complex`	Расширяется до ключевого слова типа `_Complex`.
`_Complex_I`	Расширяется до выражения типа `const float _Complex`, значение которого, возведенное в квадрат, равно –1.
`imaginary`	Если поддерживаются мнимые типы, расширяется до ключевого слова `_Imaginary_I`.
`_Imaginary_I`	Если поддерживаются мнимые типы, расширяется до выражения типа `const float _Imaginary_I`, значение которого, возведенное в квадрат, равно –1.
`I`	Расширяется либо до `_Complex_I`, либо до `_Imaginary_I`.

Реализация комплексных чисел на языке C, поддерживаемая заголовочным файлом `complex.h`, существенно отличается от реализации C++, поддерживаемой заголовочным файлом `complex`. Язык C++ использует классы для определения типов комплексных чисел.

Указание компилятору (или прагма) `STDC CX_LIMITED_RANGE` может использовать для уведомления того, могут ли применяться обычные математические формулы (в случае установки в `on`), или же следует уделить особое внимание обработке предельных значений (при установке в `off`):

```
#include <complex.h>
#pragma STDC CX_LIMITED_RANGE on
```

Библиотечные функции поставляются в трех версиях: `double`, `float` и `long double`. В табл. RS.V.3 перечислены функции для версии `double`. Версии `float` и `long double` добавляют к именам функций, соответственно, `f` и `l`. То есть `csinf()` — это `float`-версия `csin()`, а `csinl()` — версия `long double`.

Углы измеряются в радианах.

Таблица RS.V.3. Функции комплексных чисел

Прототип	*Описание*
`double complex cacos(double complex z)`	Возвращает комплексный арккосинус `z`.
`double complex casin(double complex z);`	Возвращает комплексный арксинус `z`.
`double complex catan(double complex z);`	Возвращает комплексный арктангенс `z`.
`double complex ccos(double complex z);`	Возвращает комплексный косинус `z`.
`double complex csin(double complex z);`	Возвращает комплексный синус `z`.
`double complex ctan(double complex z);`	Возвращает комплексный тангенс `z`.

Окончание табл. RS.V.3

Прототип	*Описание*
`double complex cacosh(double complex z);`	Возвращает комплексный гиперболический арккосинус `z`.
`double complex casinh(double complex z);`	Возвращает комплексный гиперболический арксинус `z`.
`double complex catanh(double complex z);`	Возвращает комплексный гиперболический арктангенс `z`.
`double complex ccosh(double complex z);`	Возвращает комплексный гиперболический косинус `z`.
`double complex csinh(double complex z);`	Возвращает комплексный гиперболический синус `z`.
`double complex ctanh(double complex z);`	Возвращает комплексный гиперболический тангенс `z`.
`double complex cexp(double complex z);`	Возвращает комплексное значение e в степени `z`.
`double complex clog(double complex z);`	Возвращает комплексный натуральный (по основанию *e*) логарифм `z`.
`double complex cabs(double complex z);`	Возвращает абсолютное значение `z`.
`double complex cpows(double complex z, double complex y);`	Возвращает значение `z` в степени `y`.
`double complex csqrt(double complex z);`	Возвращает комплексный квадратный корень из `z`.
`double complex carg(double complex z);`	Возвращает фазовый угол (или аргумент) `z` в радианах.
`double complex cimag(double complex z);`	Возвращает мнимую часть `z` как действительное число.
`double complex conj(double complex z);`	Возвращает комплексное сопряжение `z`.
`double complex cproj(double complex z);`	Возвращает проекцию `z` на сферу Римана.
`double complex creal(double complex z);`	Возвращает действительную часть `z` как действительное число.

Обработка символов: `ctype.h`

Эти функции принимают аргументы `int`, которые могут быть представлены либо как `unsigned char`, либо как `EOF`; в случае передачи других значений поведение не определено. В табл. RS.V.4 `true` используется как синоним "ненулевого значения". Интерпретация некоторых определений зависит от текущих локальных установок, кото-

рые управляются функциями из заголовочного файла `locale.h`; таблица демонстрирует интерпретацию для локальной установки "C".

Таблица RS.V.4. Функции обработки символов

Прототип	*Описание*
`int isalnum(int c);`	Возвращает `true`, если `c` — число или буква.
`int isalpha(int c);`	Возвращает `true`, если `c` — буква.
`int isblank(int c);`	Возвращает `true`, если `c` — пробел или горизонтальная табуляция (C99).
`int iscntrl(int c);`	Возвращает `true`, если `c` — управляющий символ, такой как <Ctrl+B>.
`int isdigit(int c);`	Возвращает `true`, если `c` — десятичная цифра.
`int isgraph(int c);`	Возвращает `true`, если `c` — печатаемый символ, отличный от пробела.
`int islower(int c);`	Возвращает `true`, если `c` — символ нижнего регистра.
`int isprint(int c);`	Возвращает `true`, если `c` — печатаемый символ.
`int ispunct(int c);`	Возвращает `true`, если `c` — знак препинания (любой печатаемый символ, отличный от пробела, букв и цифр).
`int isspace(int c);`	Возвращает `true`, если `c` — пробельный символ: пробел, символ новой строки, перевода страницы, возврата каретки, вертикальной табуляции, горизонтальной табуляции или, возможно, другой определяемый реализацией символ.
`int isupper(int c);`	Возвращает `true`, если `c` — символ верхнего регистра.
`int isxdigit(int c);`	Возвращает `true`, если `c` — шестнадцатеричная цифра.
`int tolower(int c);`	Если аргумент — символ верхнего регистра, возвращает его версию в нижнем регистре; иначе просто возвращает исходный аргумент.
`int toupper(int c);`	Если аргумент — символ нижнего регистра, возвращает его версию в верхнем регистре; иначе просто возвращает исходный аргумент.

Сообщения об ошибках: `errno.h`

Заголовочный файл `errno.h` поддерживает старый механизм сообщений об ошибках. Механизм обеспечивает внешнее статическое место в памяти, которое доступно через идентификатор (или, возможно, макрос) `ERRNO`. Некоторые библиотечные функции помещают значение в это место для обработки ошибки. Программа, включающая этот заголовочный файл, затем может проверить значение `ERRNO`, чтобы обнаружить сообщение об ошибке.

Механизм, использующий `ERRNO`, считается устаревшим, и от основных функций более не требуется устанавливать значения `ERRNO`. Стандарт предусматривает три макросы-значения, представляющие определенные ошибки, но конкретные реализации могут дополнять их список. В табл. RS.V.5 перечислены стандартные макросы.

Таблица RS.V.5. Макросы `errno.h`

Прототип	*Описание*
EDOM	Ошибка домена при вызове функции (аргумент вышел за допустимые пределы).
ERANGE	Ошибка диапазона возвращаемого значения функции (возвращаемое значение вышло за пределы допустимого диапазона).
EILSEQ	Ошибка трансляции расширенных символов.

Среда плавающей запятой: `fenv.h` (C99)

Стандарт C99 обеспечивает доступ и управление средой плавающей запятой через заголовочный файл `fenv.h`. Это средство поддерживает более агрессивный подход к вычислениям, однако может потребоваться немало времени, прежде чем он будет полностью реализован.

Среда плавающей запятой состоит из набора флагов и режимов управления. Исключительные ситуации, возникающие во время вычислений с плавающей запятой, подобные делению на ноль, могут "возбудить исключение". Это значит, что событие устанавливает один из флагов среды плавающей запятой. Значение режима управления может управлять, например, направлением округления. Заголовочный файл `fenv.h` определяет набор макросов, представляющих несколько исключений и режимов управления, а также прототипы функций, взаимодействующих со средой. Заголовок также представляет указание компилятору по поводу включения и отключения доступа к среде плавающей запятой.

Директива

```
#pragma STDC FENV_ACCESS on
```

включает доступ к среде, а директива

```
#pragma STDC FENV_ACCESS off
```

отключает его. Если указание компилятору является внешним, оно должно находиться перед любым внешним объявлением или же в начале составного блока. Оно остается в силе до тех пор, пока не будет переключено другим появлением этого же указания, либо до конца файла (в случае внешней директивы) или же до конца составного оператора (в случае блочной директивы).

В заголовочном файле определены два типа, представленные в табл. RS.V.6.

Таблица RS.V.6. Типы `fenv.h`

Тип	*Описание*
fenv_t	Вся среда плавающей запятой.
fexcept_t	Коллекция флагов состояния среды плавающей запятой.

Заголовочный файл определяет макросы, представляющие несколько возможных исключений плавающей запятой и управляющих состояний. Реализации могут определять дополнительные макросы, присваивая им имена, состоящие из заглавных букв и начинающихся с `FE_`. В табл. RS.V.7 показаны стандартные макросы исключений.

Таблица RS.V.7. Макросы `fenv.h`

Макрос	*Описание*
`FE_DIVBYZERO`	Исключение деления на ноль.
`FE_INEXACT`	Исключение неточного значения.
`FE_INVALID`	Исключение неверного значения.
`FE_OVERFLOW`	Исключение переполнения.
`FE_UNDERFLOW`	Исключение потери значимости.
`FE_ALL_EXCEPT`	Объединение с помощью операции поразрядного "ИЛИ" всех исключений плавающей запятой, поддерживаемых реализацией.
`FE_DOWNWARD`	Округление в меньшую сторону.
`FE_TONEAREST`	Округление до ближайшего значения.
`FE_TOWARDZERO`	Округление до нуля.
`FE_UPWARD`	Округление в большую сторону.
`FE_DFL_ENV`	Представляет среду по умолчанию и имеет тип `const fenv_t *`.

Таблица RS.V.8 иллюстрирует прототипы стандартных функций из заголовочного файла `fenv.h`. Обратите внимание, что очень часто значения аргументов и возвращаемые значения соответствуют макросам из табл. RS.V.7. Например, `FE_UPWARD` является подходящим аргументом для `fesetround()`.

Таблица RS.V.8. Прототипы `fenv.h`

Прототип	*Описание*
`void feclearexcept( int excepts);`	Очищает исключения, представленные `excepts`.
`void fegetexceptflag( fexcept_t *flagp, int excepts);`	Сохраняет состояния флагов среды плавающей запятой, переданные `excepts`, в объекте, на который указывает `flagp`.
`void feraiseexcept( int excepts);`	Возбуждает исключения, специфицированные в `excepts`.
`void fesetexceptflag( const fexcept_t *flagp, int excepts);`	Устанавливает флаги состояния среды плавающей запятой, указанные `excepts`, в значения, представленные `flagp`; `flagp` должен быть ранее установлен с помощью вызова `fegetexceptflag()`.
`int fetestexcept( int excepts);`	`excepts` специфицирует флаги состояния, которые следует опросить; функция возвращает объединенные поразрядным "ИЛИ" значения указанных флагов состояния.
`int fegetround(void);`	Возвращает текущую установку направления округления.
`int fesetround( int round);`	Устанавливает направление округления равным значению `round`; возвращает `0` только в случае успеха.

Окончание табл. RS.V.8

Прототип	*Описание*
`void fegetenv( fenv_t *envp);`	Сохраняет текущую среду в месте, на которое указывает `envp`.
`int feholdexcept( fenv_t *envp);`	Сохраняет текущую среду плавающей запятой в месте, указанном `envp`, очищает флаги состояния, а затем, если возможно, устанавливает режим "nonstop", в котором выполнение продолжается, несмотря на исключения; возвращает `0` в случае успеха.
`void fesetenv( const fenv_t *envp);`	Инсталлирует среду плавающей запятой, представленную `envp`; `envp` должен указывать на объект данных, установленный предыдущим вызовом `fegetenv()` или `feholdexcept()`, либо же макросом среды плавающей запятой.
`void feupdateenv( const fenv_t *envp);`	Функция сохраняет текущие возбужденные исключения плавающей запятой в автоматическом хранилище, инсталлирует среду плавающей запятой, представленную объектом, на который указывает `envp`, а затем возбуждает сохраненные исключения плавающей запятой; `envp` должен указывать на объект данных, установленный предыдущим вызовом `fegetenv()` или `feholdexcept()`, либо же макросом среды плавающей запятой.

Преобразование формата целочисленных типов: `inttypes.h` (C99)

Этот заголовочный файл определяет несколько макросов, которые могут использоваться в качестве спецификаторов формата для расширенных целочисленных типов. Более подробно это обсуждается в разделе VI справочника. В данном заголовочном файле также объявлен следующий тип:

`imaxdiv_t`

Этот тип представляет собой структуру, возвращаемую функцией `idivmax()`.

Файл включает в себя `stdint.h` и объявляет несколько функций, использующих самый широкий целочисленный тип, определенный в `stdint.h` как `intmax`. В табл. RS.V.9 перечислены эти функции.

Таблица RS.V.9. Целочисленные функции максимальной ширины

Прототип	*Описание*
`intmax_t imaxabs(intmax_t j);`	Возвращает абсолютное значение `j`.
`imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);`	Вычисляет частное и остаток от деления `numer/denom` за одну операцию, сохраняя эти два значения в возвращаемой структуре.
`intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);`	Эквивалент функции `strtol()`, отличающийся тем, что преобразует строку в тип `intmax_t` и возвращает это значение.

Окончание табл. RS.V.9

Прототип	*Описание*
`uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base);`	Эквивалент функции `strtoul()`, отличающийся тем, что преобразует строку в тип `uintmax_t` и возвращает это значение.
`intmax_t wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);`	Версия `wchar_t` функции `strtoimax()`.
`uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);`	Версия `wchar_t` функции `strtoumax()`.

Локализация: `locale.h`

Локальная установка — группа настроек, управляющая такими вещами, как символ, используемый для представления десятичной точки. Локальные установки сохраняются в структуре типа `struct lconv`, которая определена в заголовочном файле `locale.h`. Локальная установка может быть задана строкой, которая указывает определенный набор значений элементов структуры. Локальная установка по умолчанию обозначается строкой "C". В табл. RS.V.10 перечислены функции локализации с кратким описанием каждой.

Таблица RS.V.10. Функции локализации

Прототип	*Описание*
`char * setlocale( int category, const char * locale);`	Функция устанавливает определенные значения локальной установки равными значениям, заданным в `locale`. `category` управляет тем, какие именно значения локальной установки должны быть изменены (см. табл. RS.V.11). Функция возвращает нулевой указатель, если не может удовлетворить запрос. В противном случае возвращает указатель, связанный с указанной категорией в новой локальной установке.
`struct lconv * localeconv(void);`	Возвращает указатель на структуру `struct lconv`, заполненную значениями текущей локальной установки.

Возможными значениями параметра `locale` при вызове `setlocale()` могут быть `"C"`, что является принятым по умолчанию, и `""`, что представляет родную среду, определенную реализацией. Реализация может определять дополнительные локальные установки. Возможные значения параметра `category` при вызове `setlocale()` представлены макросами, перечисленными в табл. RS.V.11.

Таблица RS.V.11. Макросы категорий

Макрос	*Описание*
NULL	Оставляет локальную установку неизменной и возвращает указатель на текущую локальную установку.
LC_ALL	Изменяет все локальные значения.
LC_COLLATE	Изменяет локальные значения последовательности сортировки, используемой `strcoll()` и `strxfrm()`.
LC_CTYPE	Изменяет локальные значения, используемые символьными функциями и многобайтными функциями.
LC_MONETARY	Изменяет локальные значения, имеющие отношение к форматированию денежных величин.
LC_NUMERIC	Изменяет локальные значения символа десятичной точки и установки форматирования, не связанного с денежными значениями, используемыми при форматированном вводе-выводе и в функциях преобразования строк.
LC_TIME	Изменяет локальные значения для форматирования значений времени, используемых `strftime()`.

В табл. RS.V.12 перечислены обязательные элементы структуры `struct lconv`.

Таблица RS.V.12. Обязательные элементы структуры `struct lconv`

Элемент `struct lconv`	*Описание*
`char *decimal_point`	Символ десятичной точки для не денежных значений.
`char *thousands_sep`	Символ, используемый для разделения групп десятичных разрядов до точки для значений, отличных от денежных.
`char *grouping`	Строка, чьи элементы указывают размер каждой группы десятичных разрядов для значений, отличных от денежных.
`char *int_curr_symbol`	Интернациональный символ валюты.
`char *currency_symbol`	Локальный символ валюты.
`char *mon_decimal_point`	Символ десятичной точки для денежных значений.
`char *mon_thousands_sep`	Символ, используемый для разделения групп десятичных разрядов до точки для денежных значений.
`char *mon_grouping`	Строка, элементы которой указывают размер каждой группы десятичных разрядов для денежных значений.
`char *positive_sign`	Строка, используемая для обозначения неотрицательных денежных значений.
`char *negative_sign`	Строка, используемая для обозначения отрицательных денежных значений.
`char int_frac_digits`	Количество десятичных разрядов после точки для интернационально форматированных денежных величин.

Окончание табл. RS.V.12

Элемент `struct lconv`	*Описание*
`char frac_digits`	Количество десятичных разрядов после точки для локально форматированных денежных величин.
`char p_cs_precedes`	Устанавливается в `1` или `0` в зависимости от того, предшествует ли `currency_symbol` или следует за неотрицательными форматированными денежными величинами.
`char p_sep_by_space`	Устанавливается в `1` или `0` в зависимости от того, отделяется ли `currency_symbol` пробелом от неотрицательных форматированных денежных величин.
`char n_cs_precedes`	Устанавливается в `1` или `0` в зависимости от того, предшествует ли `currency_symbol` или следует за отрицательными форматированными денежными величинами.
`char n_sep_by_space`	Устанавливается в `1` или `0` в зависимости от того, отделяется ли `currency_symbol` пробелом от отрицательных форматированных денежных величин.
`char p_sign_posn`	Устанавливает значение, указывающее позицию строки `positive_sign`; `0` — означает, что скобки окружают число и символ валюты; `1` — означает, что строка предшествует числу и символу валюты; `2` — означает, что строка следует за числом и символом валюты; `3` — означает, что строка предшествует непосредственно символу валюты; `4` — означает, что строка следует немедленно за символом валюты.
`char n_sign_posn`	Устанавливается в значение, указывающее положение строки `negative_sign`; значения — такие же, как в `p_sign_posn`.
`char int_p_cs_precedes`	Устанавливается в `1` или `0` в зависимости от того, предшествует ли `int_currency_symbol` значению неотрицательных форматированных денежных величин, или следует за ним.
`char int_p_sep_by_space`	Устанавливается в `1` или `0` в зависимости от того, отделяется ли пробелом `int_currency_symbol` от значения неотрицательных форматированных денежных величин.
`char int_n_cs_precedes`	Устанавливается в `1` или `0`, в зависимости от того, предшествует ли `int_currency_symbol` значению отрицательных форматированных денежных величин, или следует за ним.
`char int_n_sep_by_space`	Устанавливается в `1` или `0` в зависимости от того, отделяется ли пробелом `int_currency_symbol` от значения отрицательных форматированных денежных величин.
`char int_p_sign_posn`	Устанавливает значение, указывающее позицию `positive_sign` для неотрицательных интернационально форматированных денежных величин.
`char int_n_sign_posn`	Устанавливает значение, указывающее позицию `negative_sign` для неотрицательных интернационально форматированных денежных величин.

Математическая библиотека: math.h

В C99 заголовочный файл math.h определяет два типа:

```
float_t
double_t
```

Эти типы в разрядах по ширине равны, по меньшей мере, float и double, соответственно, а double_t, по меньшей мере, — float_t. Их предназначение состоит в том, чтобы быть типами, обеспечивающими наиболее эффективные вычисления, соответственно, с float и double.

Этот заголовочный файл также определяет несколько макросов, как описано в табл. RS.V.13; все, кроме HUGE_VAL, добавлены C99. Некоторые из них более подробно описаны в разделе VIII.

Таблица RS.V.13. Макросы math.h

Макрос	*Описание*
HUGE_VAL	Положительная константа double, не обязательно выражаемая как float; в прошлом использовалась в качестве возвращаемого значения функций, когда абсолютная величина результата превышала максимальное представимое значение.
HYGE_VALF	Дополнение HUGE_VAL типа float.
HUGE_VALL	Дополнение HUGE_VAL типа long double.
INFINITY	Расширяется до константного выражения float, представляющего положительную или беззнаковую бесконечность, если возможно; в противном случае расширяется до положительной константы float, которая приводит к переполнению во время компиляции.
NAN	Определен тогда и только тогда, когда реализация молча поддерживает NaN (значения, трактуемые, как нечисловые) для типа float.
FP_INFINITE	Классификационное число, символизирующее бесконечное значение с плавающей запятой.
FP_NAN	Классификационное число, символизирующее значение с плавающей запятой, не являющееся числом.
FP_NORMAL	Классификационное число, символизирующее нормальное значение с плавающей запятой.
FP_SUBNORMAL	Классификационное число, символизирующее неполноценное значение с плавающей запятой (с пониженной точностью).
FP_ZERO	Классификационное число, символизирующее значение с плавающей запятой, представляющее 0.
FP_FAST_FMA	(Необязательный). Если определен, этот макрос говорит о том, что функция fma() работает почти так же быстро, или быстрее, чем умножение и сложение операндов типа double.
FP_FAST_FMAF	(Необязательный). Если определен, этот макрос говорит о том, что функция fmaf() работает почти так же быстро, или быстрее, чем умножение и сложение операндов типа float.

Окончание табл. RS.V.13

Макрос	*Описание*
FP_FAST_FMAL	(Необязательный). Если определен, этот макрос говорит о том, что функция fmal() работает почти так же быстро, или быстрее, чем умножение и сложение операндов типа long double.
FP_ILOGB0	Целое константное выражение, представляющее значение, возвращаемое ilogb(0).
FP_ILOGBNAN	Целое константное выражение, представляющее значение, возвращаемое ilogb(NaN).
MATH_ERRNO	Расширяется до целой константы 1.
MATH_ERREXCEPT	Расширяется до целой константы 2.
math_errhandling	Имеет значение MATH_ERRNO или MATH_ERREXCEPT, либо объединение этих двух значений с помощью поразрядного "ИЛИ".

Математические функции обычно работают со значениями типа double. C99 добавляет версии float и long double этих функций, что указывается добавлением к их именам суффиксов f и l соответственно. Например, в языке теперь представлены следующие прототипы:

```
double sin(double);
float sinf(float);
long double sinl(long double);
```

Для краткости в табл. RS.V.14 перечислены только double-версии функций математической библиотеки. Таблица ссылается на константу FLT_RADIX. Эта константа, определенная в float.h, в основном используется для возведения в степень во внутреннем представлении плавающей запятой. Наиболее часто имеет значение 2.

Таблица RS.V.14. Стандартные математические функции ANSI C

Прототип	*Описание*
int classify(real-floating x);	Макрос C99, возвращает классификационное значение плавающей запятой, соответствующее x.
int isfinite(real-floating x);	Макрос C99, возвращает ненулевое значение тогда и только тогда, когда x конечно.
int isfin(real-floating x);	Макрос C99, возвращает ненулевое значение тогда и только тогда, когда x бесконечно.
int isnan(real-floating x);	Макрос C99, возвращает ненулевое значение тогда и только тогда, когда x – NaN.
int isnormal(real-floating x);	Макрос C99, возвращает ненулевое значение тогда и только тогда, когда x нормально.
int signbit(real-floating x);	Макрос C99, возвращает ненулевое значение тогда и только тогда, когда x отрицательно.

Продолжение табл. RS.V.14

Прототип	*Описание*
`double acos(double x);`	Возвращает угол (от 0 до π радиан), косинус которого равен `x`.
`double asin(double x);`	Возвращает угол (от $-\pi/2$ до $\pi/2$ радиан), синус которого равен `x`.
`double atan(double x);`	Возвращает угол (от $-\pi/2$ до $\pi/2$ радиан), тангенс которого равен `x`.
`double atan2(double y, double x);`	Возвращает угол (от $-\pi$ до π радиан), тангенс которого равен `x/y`.
`double cos(double x);`	Возвращает косинус `x` (`x` — в радианах).
`double sin(double x);`	Возвращает синус `x` (`x` — в радианах).
`double tan(double x);`	Возвращает тангенс `x` (`x` — в радианах).
`double cosh(double x);`	Возвращает гиперболический косинус `x`.
`double sinh(double x);`	Возвращает гиперболический синус `x`.
`double tanh(double x);`	Возвращает гиперболический тангенс `x`.
`double exp(double x);`	Возвращает экспоненциальную функцию `x` (e^x).
`double exp2(double x);`	Возвращает 2 в степени `x` (C99).
`double expm1(double x);`	Возвращает $e^x - 1$ (C99).
`double frexp(double v, int *pt_e);`	Разбивает значение `v` на нормализованную дробную часть, которая возвращается, и степень 2, которая помещается в место, указанное `pt_e`.
`int ilogb(double x);`	Возвращает экспоненту `x`, как целое со знаком (C99).
`double ldexp(double x, int p);`	Возвращает 2 в степени `p`, умноженное на `x`.
`double log(double x);`	Возвращает натуральный логарифм `x`.
`double log10(double x);`	Возвращает логарифм `x` по основанию 10.
`double log1p(double x);`	Возвращает `log(1 + x)` (C99).
`double log2(double x);`	Возвращает логарифм `x` по основанию 2.
`double logb(double x);`	Возвращает экспоненту аргумента со знаком для лежащей в основе базы, используемой в системе для представления значений с плавающей запятой (`FLT_RADIX`) (C99).
`double modf(double x, double *p);`	Разбивает значение `x` на целую и дробную части, обе с одинаковым знаком, возвращает дробную часть, а целую помещает в место, указанное `p`.
`double scalbn(double x, int n);`	Возвращает `x` × `FLT_RADIXn` (C99).
`double scalbln(double x, long n);`	Возвращает `x` × `FLT_RADIXn` (C99).

Продолжение табл. RS.V.14

Прототип	*Описание*
`double cbrt(double x);`	Возвращает корень кубический из x (C99).
`double hypot(double x, double y);`	Возвращает корень квадратный суммы квадратов x и y (C99).
`double pow(double x, double y);`	Возвращает x в степени y.
`double sqrt(double x);`	Возвращает квадратный корень из x.
`double erf(double x);`	Возвращает функцию ошибки x.
`double erfc(double x);`	Возвращает дополнительную функцию ошибки x.
`double lgamma(double x);`	Возвращает натуральный логарифм абсолютного значения гамма-функции x (C99).
`double tgamma(double x);`	Возвращает гамма-функцию x (C99).
`double ceil(double x);`	Возвращает минимальное целое значение, не меньшее чем x.
`double fabs(double x);`	Возвращает абсолютное значение x.
`double floor(double x);`	Возвращает максимальное целое значение, не большее чем x.
`double nearbyint(double x);`	Округляет x до ближайшего целого в формате с плавающей запятой; использует направление округления, установленное средой плавающей запятой, если она доступна. Исключение "неточности" не возбуждается (C99).
`double rint(double x);`	Подобно `nearbyint()`, но может быть возбуждено исключение "неточности" (C99).
`long int lrint(double x);`	Округляет x до ближайшего целого в формате `long int`; использует направление округления, установленное средой плавающей запятой, если она доступна (C99).
`long long int llrint(double x);`	Округляет x до ближайшего целого в формате `long long int`; использует направление округления, установленное средой плавающей запятой, если она доступна (C99).
`double round(double x);`	Округляет x до ближайшего целого в формате с плавающей запятой; всегда округляет частичные значения в сторону от нуля (C99).
`long int lround(double x);`	Подобно `round()`, но возвращается ответ типа `long int` (C99).
`long long int llround(double x);`	Подобно `round()`, но возвращается ответ типа `long long int` (C99).

Продолжение табл. RS.V.14

Прототип	*Описание*
`double trunc(double x);`	Округляет x до ближайшего целого в формате с плавающей запятой, которое не больше абсолютного значения x (C99).
`int fmod(double x, double y);`	Возвращает дробную часть x/y; если y — не ноль, то результат получает тот же знак, что x, и меньше по абсолютному значению, чем y.
`double remainder(double x, double y);`	Возвращает x REM y, что стандарт IEC 60559 определяет как x - n*y, где n — ближайшее к x/y целое; n — четное, если абсолютное значение (n - x/y) равно 1/2 (C99).
`double remquo(double x, double y, int *quo);`	Возвращает то же значение, что и reminder(), и помещает в место, указываемое quo, значение, имеющее тот же знак, что и x/y, и имеющее абсолютную целую величину x/y по модулю 2^k, где k — зависящее от реализации целое, значение которого не меньше 3 (C99).
`double copysign(double x, double y);`	Возвращает значение абсолютной величины x со знаком y (C99).
`double nan(const char *tagp);`	Возвращает double-представление NaN; nan("n-char-seq") эквивалентно strtod("NAN(n-char-seq) ", (char **)NULL); nan("") — эквивалент strtod("NAN()", (char **)NULL); для других строк-аргументов вызов эквивалентен strtod("NAN",(char **)NULL). Возвращает 0, если расширенные NaN не поддерживаются (C99).
`double nextafter(double x, double y);`	Возвращает следующее представимое значение типа double после x в направлении y; возвращает x, если x равно y (C99).
`double nexttoward(double x, long double y);`	То же самое, что и nextafter(), за исключением того, что второй аргумент имеет тип long double, и если x равно y, функция возвращает y, преобразованное в double (C99).
`double fdim(double x, double y);`	Возвращает положительную разность аргументов (C99).
`double fmax(double x, double y);`	Возвращает максимальное числовое значение из двух аргументов; если один из аргументов NaN, а второй — число, возвращается второй (C99).
`double fmin(double x, double y);`	Возвращает минимальное числовое значение из двух аргументов; если один из аргументов NaN, а второй — число, возвращается второй (C99).

Окончание табл. RS.V.14

Прототип	*Описание*
`double fma(double x, double y, double z);`	Возвращает величину `(x*y)+z` как тернарную операцию, округленную один раз в конце (C99).
`int isgreater(real-floating x, real-floating y);`	Макрос C99, возвращающий значение `(x) > (y)` без возбуждения исключения плавающей запятой типа "неверное число", если один или оба аргумента – NaN.
`int isgreaterequal(real-floating x, real-floating y);`	Макрос C99, возвращающий значение `(x) >= (y)` без возбуждения исключения плавающей запятой типа "неверное число", если один или оба аргумента – NaN.
`int isless(real-floating x, real-floating y);`	Макрос C99, возвращающий значение `(x) < (y)` без возбуждения исключения плавающей запятой типа "неверное число", если один или оба аргумента – NaN.
`int islessequal(real-floating x, real-floating y);`	Макрос C99, возвращающий значение `(x) <= (y)` без возбуждения исключения плавающей запятой типа "неверное число", если один или оба аргумента – NaN.
`int islessgreater(real-floating x, real-floating y);`	Макрос C99, возвращающий значение `(x) < (y) || (x) > (y)` без возбуждения исключения плавающей запятой типа "неверное число", если один или оба аргумента – NaN.
`int isunordered(real-floating x, real-floating y);`	Возвращает единицу, если аргументы неупорядочены (то есть хотя бы один является NaN), в противном случае возвращает 0.

Нелокальные переходы: `setjmp.h`

Заголовочный файл `setjmp.h` позволяет пропускать обычную последовательность вызовов и возвратов функций. Функция `setjmp()` сохраняет информацию о текущей среде выполнения (например, указатель на текущую инструкцию) в переменной типа `jmp_buf` (тип массива, определенный в этом заголовочном файле), а функция `longjmp()` передает выполнение в эту среду. Функции предназначены для обработки ошибочных ситуаций, но не в качестве части нормального потока управления программы. В табл. RS.V.15 описаны эти функции.

Таблица RS.V.15. Функции `setjmp.h`

Прототип	*Описание*
`int setjmp(jmp_buf env);`	Сохраняет текущую среду вызовов в массиве `env` и возвращает `0`, если вызвана непосредственно, и `1` — если выполнена в результате вызова `longlmp()`.
`void longjmp(jmp_buf env, int val);`	Восстанавливает среду, сохраненную последним вызовом `setjmp()`, который записал массив `env`; после этого программа выполняется с того места, где была вызвана `setjmp()`, как если бы эта функция была вызвана, но при этом возвратила `1` вместо `0`.

Обработка сигналов: `signal.h`

Сигнал — это условие, которое может быть сообщено программе во время ее выполнения. Сигнал представлен положительным целым числом. Функция `raise()` посылает, или *возбуждает* сигнал, а функция `signal()` устанавливает ответ на определенный сигнал.

Стандарт предусматривает макросы, перечисленные в таблице RS.V.16, для представления возможных сигналов; реализация может добавить собственные значения. Эти макросы могут использоваться в качестве аргументов `raise()` и `signal()`.

Таблица RS.V.16. Макросы сигналов

Макрос	*Описание*
`SIGABRT`	Ненормальное завершение, такое как инициированное вызовом `abort()`.
`SIGFPE`	Ошибочная арифметическая операция.
`SIGILL`	Обнаружен неверный образ функции (такой как недопустимая инструкция).
`SIGINT`	Принят интерактивный сигнал внимания (такой как прерывание DOS).
`SIGSERV`	Неверное обращение к хранилищу.
`SIGTERM`	Программа получила запрос на прекращение работы.

В качестве второго аргумента функция `signal()` принимает указатель на функцию `void`, принимающую аргумент `int`. Она также возвращает указатель того же типа. Функция, вызываемая в ответ на сигнал, называется *обработчиком сигнала*. Стандарт определяет три макроса, подходящие к этому прототипу:

```
void (*funct)(int);
```

В табл. RS.V.17 перечислены эти макросы.

Таблица RS.V.17. Макросы типа `void (*f)(int)`

Макрос	*Описание*
`SIG_DFL`	Когда используется в качестве аргумента `signal()` наряду со значением сигнала, этот макрос указывает, что произойдет обработка данного сигнала по умолчанию.
`SIG_ERR`	Используется в качестве возвращаемого значения `signal()`, если она не может вернуть свой второй аргумент.
`SIG_IGN`	Когда используется в качестве аргумента `signal()` наряду со значением сигнала, этот макрос указывает, что сигнал будет проигнорирован.

Если сигнал `sig` возбуждается, а `func` указывает на функцию (см. прототип `signal()` в табл. RS.V.18), то вначале в большинстве случаев вызывается `signal(sig, SIG_DFL)` для сброса обработчика сигнала в значение по умолчанию, а затем вызывается `(*func)(sig)`. Функция-обработчик сигнала, на которую указывает `func`, может быть завершена выполнением оператора `return` либо вызовом `abort()`, `exit()` или `longjmp()`. В табл. RS.V.18 перечислены функции сигналов.

Таблица RS.V.18. Функции сигналов

Прототип	*Описание*
`void (*signal(int sig, void(*func)(int)))(int);`	Заставляет функцию, на которую указывает `func`, автоматически вызываться при получении программой сигнала `sig`. Если возможно, возвращает `func`, иначе — `SIG_ERR`.
`int raise(int sig);`	Посылает сигнал `sig` выполняющейся программе; в случае успеха возвращает ноль, в противном случае — не ноль.

Переменное количество аргументов: `stdarg.h`

Заголовочный файл `stdarg.h` предоставляет средства для определения функций, принимающих переменное количество аргументов. Прототип для такой функции должен содержать список параметров, в котором указан как минимум один параметр, за которым следует многоточие:

```
void f1(int n, ...);                  /* правильно   */
int f2(int n, float x, int k, ...);   /* правильно   */
double f3(...);                       /* неправильно */
```

В следующей таблице термин `parmN` — это идентификатор, используемый для обозначения последнего параметра, предшествующего многоточию. В предыдущих примерах таким параметром был `n` в первом случае и `k` — во втором.

В заголовочном файле объявлен тип `va_list` для представления объекта данных, который используется для хранения параметров, соответствующих многоточию в списке параметров. В табл. RS.V.19 перечислены три макроса, которые должны использоваться в функциях с переменным количеством параметров. Перед использованием этих макросов должен быть объявлен объект типа `va_list`.

Таблица RS.V.19. Макросы переменных списков аргументов

Прототип	*Описание*
`void va_start(va_list ap, parmN);`	Макрос инициализирует `ap` перед использованием `va_arg()` и `va_end()`; `parmN` – идентификатор последнего именованного параметра в списке аргументов.
`void va_copy(va_list dest, va_list src);`	Макрос инициализирует `dest` как копию текущего состояния `src` (C99).
`type va_arg(va_list ap, type);`	Макрос расширяется до выражения, имеющего то же значение и тип, что и следующий элемент в списке аргументов, представленном `ap`; `type` – тип этого элемента. Каждый вызов переходит к следующему элементу `ap`.
`void va_end(va_list ap);`	Этот макрос прекращает процесс и может сделать `ap` недоступным без повторного вызова `va_start()`.

Поддержка булевских значений: `stdbool.h` (C99)

Этот заголовочный файл определяет четыре макроса, перечисленных в табл. RS.V.20.

Таблица RS.V.20. Макросы `stdbool.h`

Макрос	*Описание*
`Bool`	Расширяется до `_Bool`.
`False`	Расширяется до целочисленной константы 0.
`True`	Расширяется до целочисленной константы 1.
`__bool_true_false_are_defined`	Расширяется до целочисленной константы 1.

Общие определения: `stddef.h`

Этот заголовочный файл определяет некоторые типы и макросы, показанные в таблицах RS.V.21 и RS.V.22.

Таблица RS.V.21. Типы `stddef.h`

Тип	*Описание*
`ptrdiff_t`	Целочисленный тип со знаком, представляющий результат вычитания одного указателя из другого.
`size_t`	Целочисленный тип без знака, представляющий результат, возвращаемый операцией `sizeof`.

Таблица RS.V.22. Макросы `stddef.h`

Макрос	*Описание*
`wchar_t`	Целочисленный тип, который может представлять наибольший расширенный набор символов, определенный поддерживаемыми локальными установками.
`NULL`	Зависящая от реализации константа, представляющая нулевой указатель.
`offsetof(type, member-esignator)`	Расширяется до `size_t`, представляющего смещение в байтах указанного элемента от начала структуры типа `type`; поведение не определено, если элемент является разрядным полем.

Пример

```
#include <stddef.h>
struct car
{
   char brand[30];
   char model[30];
   double hp;
   double price;
};
int main(void)
{
   size_t into = offsetof(struct car, hp); /* смещение элемента hp */
   ...
```

Целочисленные типы: `stdint.h`

Заголовочный файл использует средство `typedef` для определения имен целочисленных типов, описывающих свойства целых чисел. Этот файл включен в заголовок `inttypes.h`, который представляет макросы для использования в вызовах функций ввода-вывода.

Типы заданной ширины

Один из наборов `typedef` определяет типы с заданной шириной. В табл. RS.V.23 перечислены имена и размеры. Однако следует отметить, что не все системы могут поддерживать все эти типы.

Таблица RS.V.23. Типы заданной ширины

Тип	*Описание*
`int8_t`	8-разрядный со знаком
`int16_t`	16-разрядный со знаком
`int32_t`	32-разрядный со знаком
`int64_t`	64-разрядный со знаком
`uint8_t`	8-разрядный без знака
`uint16_t`	16-разрядный без знака
`uint32_t`	32-разрядный без знака
`uint64_t`	64-разрядный без знака

Типы минимальной ширины

Типы минимальной ширины гарантируют, что каждый данный тип будет иметь размер, равный, как минимум, определенному количеству байтов. В табл. RS.V.24 перечислены типы минимальной ширины. Эти типы существуют всегда.

Таблица RS.V.24. Типы минимальной ширины

Тип	*Описание*
`int_least8_t`	Минимум 8-разрядный со знаком
`int_least16_t`	Минимум 16-разрядный со знаком
`int_least32_t`	Минимум 32-разрядный со знаком
`int_least64_t`	Минимум 64-разрядный со знаком
`uint_least8_t`	Минимум 8-разрядный без знака
`uint_least16_t`	Минимум 16-разрядный без знака
`uint_least32_t`	Минимум 32-разрядный без знака
`uint_least64_t`	Минимум 64-разрядный без знака

Скоростные типы минимальной ширины

В определенной системе некоторые представления целых чисел могут быть быстрее, чем другие. Поэтому `stdint.h` также определяет наиболее быстрые типы для представления, как минимум, определенного количества разрядов. В табл. RS.V.25 перечислены все скоростные типы минимальной ширины. Эти типы также существуют всегда. Иногда выбор наиболее быстрого типа не однозначен; в этом случае система просто определяет один из возможных.

Таблица RS.V.25. Скоростные типы минимальной ширины

Тип	*Описание*
`int_fast8_t`	Минимум 8-разрядный со знаком
`int_fast16_t`	Минимум 16-разрядный со знаком
`int_fast32_t`	Минимум 32-разрядный со знаком
`int_fast64_t`	Минимум 64-разрядный со знаком
`uint_fast8_t`	Минимум 8-разрядный без знака
`uint_fast16_t`	Минимум 16-разрядный без знака
`uint_fast32_t`	Минимум 32-разрядный без знака
`uint_fast64_t`	Минимум 64-разрядный без знака

Типы минимальной ширины

Заголовочный файл `stdint.h` также определяет типы максимальной ширины. Переменные такого типа могут вместить любое целое значение, которое может быть представлено в системе. В табл. RS.V.26 перечислены эти типы.

Таблица RS.V.26. Типы максимальной ширины

Тип	*Описание*
`intmax_t`	Самый широкий целочисленный тип со знаком.
`uintmax_t`	Самый широкий целочисленный тип без знака.

Целые, которые могут хранить указатели

Заголовочный файл также определяет два целочисленных типа, перечисленных в табл. RS.V.27, которые могут корректно сохранять значения указателей. То есть, если переменной одного из этих типов присвоить значение типа `void *`, а затем обратно присвоить значение этого целого типа указателю, то никакой потери информации не происходит. В конкретных реализациях может отсутствовать любой из этих типов или же оба сразу.

Таблица RS.V.27. Целочисленные типы для хранения значений указателей

Тип	*Описание*
`intptr_t`	Целый тип со знаком, способный хранить указатели
`uintptr_t`	Целый тип без знака, способный хранить указатели

Определенные константы

Заголовочный файл `stdint.h` также определяет ряд констант, представляющих предельные значения типов, определенных в нем. Константы названы по именам типов. Чтобы получить имя константы, представляющей минимальное и максимальное значения данного типа, возьмите имя типа, замените `_t` на `_MAX` или `_MIN` и переведите все символы в верхний регистр. Например, минимальным значением `int32_t` является INT32_MIN, а максимальным значением типа `uint_fast16_t` — `UINT_FAST16_MAX`. В табл. RS.V.28 перечислены все эти константы, причем в именах типов `N` означает количество разрядов, которое, наряду с константами, относится к типам `intptr_t`, `uintptr_t`, `intmax_t` и `intmax_t`. Абсолютные значения этих констант равны или превышают (если только не помечено “в точности”) указанные величины.

Таблица RS.V.28.Целочисленные константы

Идентификатор константы	*Минимальное (абсолютное) значение*
`INTN_MIN`	В точности $-(2^{N-1}-1)$
`INTN_MAX`	В точности $2^{N-1}-1$
`UINTN_MAX`	В точности 2^{N-1}
`INT_LEASTN_MIN`	$-(2^{N-1}-1)$
`INT_LEASTN_MAX`	$2^{N-1}-1$
`UINT_LEASTN_MAX`	$2^{N-1}-1$
`INT_FASTN_MAX`	$-(2^{N-1}-1)$
`INT_FASTN_MIN`	$2^{N-1}-1$

Окончание табл. RS.V.28

Идентификатор константы	*Минимальное (абсолютное) значение*
UINT_FASTN_MAX	$2^{N-1}-1$
INTPTR_MIN	$-(2^{15}-1)$
INTPTR_MAX	$2^{15}-1$
UINTPTR_MAX	$2^{16}-1$
INTMAX_MIN	$-(2^{15}-1)$
INTMAX_MAX	$2^{63}-1$
UINTMAX_MAX	264-1

В данном заголовочном файле также определены некоторые константы для типов, определенных в других местах. Они перечислены в табл. RS.V.29.

Таблица RS.V.29. Дополнительные целые константы

Идентификатор константы	*Значение*
PTRDIFF_MIN	Минимальное значение типа ptrdiff_t.
PTRDIFF_MAX	Максимальное значение типа ptrdiff_t.
SIG_ATOMIC_MIN	Минимальное значение типа sig_atomic_t.
SIG_ATOMIC_MAX	Максимальное значение типа sig_atomic_t.
WCHAR_MIN	Минимальное значение типа wchar_t.
WCHAR_MAX	Максимальное значение типа wchar_t.
WINT_MIN	Минимальное значение типа wint_t.
WINT_MAX	Максимальное значение типа wint_t.
SIZE_MAX	Максимальное значение типа size_t.

Расширенные целые константы

Заголовочный файл stdint.h определяет макросы, специфицирующие константы различных расширенных целочисленных типов. По сути дела, такой макрос — это приведение одного типа к некоторому другому лежащему в основе типу, то есть к фундаментальному типу, представляющему расширенный в определенной реализации.

Имя макроса формируется из имени типа с заменой _t на _C и переводом к верхнему регистру. Например, чтобы объявить 1000 константой типа uint_least64_t, используйте выражение UINT_LEAST64_C(1000).

Стандартная библиотека ввода-вывода: stdio.h

Стандартная библиотека ANSI C предлагает множество стандартных функций ввода-вывода, ассоциированных с потоками и файлом stdio.h. В табл. RS.V.30 представлены прототипы ANSI этих функций вместе с кратким объяснением их работы. (Многие функции были описаны более полно в главе 13.) Заголовочный файл также определяет тип FILE, значения EOF и NULL, а также стандартные потоки ввода-вывода stdin, stdout и stderr, а также константы, используемые в функциях этой библиотеки.

Таблица RS.V.30. Функции ввода-вывода стандарта ANSI C

Прототип	*Описание*
`void clearerr(FILE *);`	Очищает индикаторы признака конца файла и ошибки.
`int fclose(FILE *);`	Закрывает указанный файл.
`int feof(FILE *);`	Проверяет на предмет достижения конца файла.
`int ferror(FILE *);`	Проверяет индикатор ошибки.
`int fflush(FILE *);`	Сбрасывает буфер указанного файла.
`int fgetc(FILE *);`	Получает следующий символ из указанного входного потока.
`int fgetpos(FILE * restrict, fpos_t * restrict);`	Сохраняет текущее значение индикатора позиции файла.
`char * fgets(char * restrict, FILE * restrict);`	Получает следующую строку (или значение `int` количества символов) из указанного потока.
`FILE * fopen(const char * restrict, const char *restrict);`	Открывает указанный файл.
`int fprintf(FILE * restrict, const char * restrict, ...);`	Пишет форматированный вывод в указанный поток.
`int fputc(int, FILE *);`	Пишет один символ в указанный поток.
`int fputs(const char * restrict, FILE * restrict);`	Пишет в поток строку, на которую указывает первый аргумент.
`size_t fread(void * restrict, size_t, size_t, FILE * restrict);`	Читает бинарные данные из указанного потока.
`FILE * freopen( const char * restrict, const char * restrict, FILE * restrict);`	Открывает указанный файл и ассоциирует его с потоком.
`int fscanf(FILE * restrict, const char * restrict, ...);`	Читает форматированный ввод из указанного потока.
`int fsetpos(FILE *, const fpos_t *);`	Устанавливает маркер позиции файла в заданное положение.
`int fseek(FILE *, long, int);`	Устанавливает маркер позиции файла в заданное положение.
`long ftell(FILE *);`	Получает текущую позицию маркера в файле.
`size_t fwrite( const void * restrict, size_t, size_t, FILE * restrict);`	Пишет в указанный поток бинарные данные.
`int getc(FILE *);`	Читает следующий символ из указанного ввода.
`int getchar();`	Читает следующий символ из стандартного ввода.
`char * gets(char *);`	Получает следующую строку из стандартного ввода.
`void perror(const char *);`	Пишет системные сообщения об ошибках в стандартный поток ошибок.

Окончание табл. RS.V.30

Прототип	*Описание*
`int printf(const char * restrict,...);`	Пишет форматированный вывод в стандартный вывод.
`int putc(int, FILE *);`	Пишет заданный символ в указанный вывод.
`int putchar(int);`	Пишет заданный символ в стандартный вывод.
`int puts(const char *);`	Пишет строку в стандартный вывод.
`int remove(const char *);`	Удаляет названный файл.
`int rename(const char *,const char *);`	Переименовывает названный файл.
`void rewind(FILE *);`	Устанавливает маркер позиции в начало файла.
`int scanf(const char * restrict,...);`	Читает форматированный ввод из стандартного ввода.
`void setbuf(FILE * restrict, char * restrict);`	Устанавливает размер и местоположение буфера.
`int setvbuf(FILE * restrict, char *restrict, int, size_t);`	Устанавливает размер, местоположение и режим буфера.
`int snprintf(char * restrict, size_t n,const char * restrict, ...);`	Пишет до `n` символов форматированного вывода в указанную строку.
`int sprintf(char * restrict, const char * restrict, ...);`	Пишет форматированный вывод в указанную строку.
`int sscanf(const char *restrict, const char * restrict, ...);`	Читает форматированный ввод из указанной строки.
`FILE * tmpfile(void);`	Создает временный файл.
`char * tmpnam(char *);`	Генерирует уникальное имя для временного файла.
`int ungetc(int, FILE *);`	Заталкивает указанный символ обратно во входной поток.
`int vfprintf(FILE * restrict, const char * restrict, va_list);`	Подобна `fprintf()`, но использует единственный списочный аргумент типа `va_list`, инициализированный `va_start`, вместо списка аргументов переменной длины.
`int vprintf(const char * restrict, va_list);`	Подобна `printf()`, но использует единственный списочный аргумент типа `va_list`, инициализированный `va_start`, вместо списка аргументов переменной длины.
`int vsprintf(char * restrict, size_t n, const char * restrict, va_list);`	Подобна `snprintf()`, но использует единственный списочный аргумент типа `va_list`, инициализированный `va_start` вместо списка аргументов переменной длины.
`int vsprintf(char * restrict, const char * restrict, va_list);`	Подобна `sprintf()`, но использует единственный списочный аргумент типа `va_list`, инициализированный `va_start`, вместо списка аргументов переменной длины.

Общие утилиты: stdlib.h

Стандартная библиотека ANSI C включает множество служебных функций, определенных в stdlib.h. Функции из этого заголовочного файла перечислены в табл. RS.V.31.

Таблица RS.V.31. Типы, определенные в stdlib.h

Тип	*Описание*
size_t	Целочисленный тип, возвращаемый операцией sizeof.
wchar_t	Целочисленный тип, используемый для представления широких символов.
div_t	Структурный тип, возвращаемый div(); имеет элементы quot и rem, оба типа int.
ldiv_t	Структурный тип, возвращаемый ldiv(); имеет элементы quot и rem, оба типа long.
lldiv_t	Структурный тип, возвращаемый lldiv(); имеет элементы quot и rem, оба типа long long (C99).

Заголовочный файл определяет константы, перечисленные в табл. RS.V.32.

Таблица RS.V.32. Константы, определенные в stdlib.h

Тип	*Описание*
NULL	Нулевой указатель (эквивалент 0).
EXIT_FAILURE	Может быть использована в качестве аргумента exit() для обозначения неудачного завершения программы.
EXIT_SUCCESS	Может быть использована в качестве аргумента exit() для обозначения успешного завершения программы.
RAND_MAX	Максимальное целое значение, возвращаемое rand().
MB_CUR_MAX	Максимальное количество байт в многобайтном символе из расширенного набора символов, соответствующего текущей локальной установке.

В табл. RS.V.33 перечислены функции, определенные в stdlib.h.

Таблица RS.V.33. Утилиты общего назначения

Прототип	*Описание*
`double atof(const char * nptr);`	Возвращает начальную часть строки `nptr`, преобразованную в значение типа `double`; преобразование завершается при обнаружении первого символа, не являющегося частью числа; начальные пробелы пропускаются; если число в строке не найдено, возвращается ноль.
`int atoi(const char * nptr);`	Возвращает начальную часть строки `nptr`, преобразованную в значение типа `int`; преобразование завершается при обнаружении первого символа, не являющегося частью числа; начальные пробелы пропускаются; если число в строке не найдено, возвращается ноль.
`int atol(const char * nptr);`	Возвращает начальную часть строки `nptr`, преобразованную в значение типа `long`; преобразование завершается при обнаружении первого символа, не являющегося частью числа; начальные пробелы пропускаются; если число в строке не найдено, возвращается ноль.

`double strtod(`
`char * restrict npt,`
`char ** restrict ept);`
	Возвращает начальную часть строки `npt`, преобразованную в значение типа `double`; преобразование завершается при обнаружении первого символа, не являющегося частью числа; начальные пробелы пропускаются; если число в строке не найдено, возвращается ноль. Если преобразование прошло успешно, адрес первого символа после числа сохраняется по адресу, указанному `ept`; если неудачно – `npt` записывается по адресу, указанному `ept`.

`float strtof(const char * restrict npt, char ** restrict ept);`	То же, что и `strtod()`, но преобразует строку, на которую указывает `npt`, в значение типа `float`.
`long double strtols(const char * restrict npt,char ** restrict ept);`	То же, что и `strtod()`, но преобразует строку, на которую указывает `npt`, в значение типа `long double` (C99).

`long strtol(`
`const char * restrict npt,`
`char ** restrict ept,`
`int base);`
	Возвращает начальную часть строки `npt`, преобразованную в значение типа `long`; преобразование завершается при обнаружении первого символа, не являющегося частью числа; начальные пробелы пропускаются; если число в строке не найдено, возвращается ноль. Если преобразование прошло успешно, адрес первого символа после числа сохраняется по адресу, указанному `ept`; если неудачно – `npt` записывается по адресу, указанному `ept`. Предполагается, что число в строке записано с основанием, заданным в `base`.

`long long strtoll(`
`const char * restrict npt,`
`char ** restrict ept, int base);`
	То же, что и `strtol()`, но преобразует строку, указанную `npt`, в значение типа `long long` (C99).

Продолжение табл. RS.V.33

Прототип	*Описание*
`unsigned long strtol( const char * restrict npt, char ** restrict ept, int base);`	Возвращает начальную часть строки `npt`, преобразованную в значение типа `unsigned long`; преобразование завершается при обнаружении первого символа, не являющегося частью числа; начальные пробелы пропускаются; если число в строке не найдено, возвращается ноль. Если преобразование прошло успешно, адрес первого символа после числа сохраняется по адресу, указанному `ept`; если неудачно — `npt` записывается по адресу, указанному `ept`. Предполагается, что число в строке записано с основанием, заданным в `base`.
`unsigned long long strtoll( const char * restrict npt, char ** restrict ept, int base);`	То же, что и `strtol()`, но строка, указанная `npt`, преобразуется в тип `unsigned long long` (C99).
`int rand(void);`	Возвращает псевдослучайное число в диапазоне от `0` до `RAND_MAX`.
`void srand(unsigned int seed);`	Устанавливает начальное значение для генератора случайных чисел равным `seed`; если `rand()` вызывается до вызова `srand()`, `seed` равно 1.
`void *calloc(size_t nmem, size_t size);`	Выделяет память для массива из `nmem` элементов, каждый элемент которого имеет размер `size` байтов; все биты в выделенной области инициализируются нулями. Функция возвращает адрес массива в случае удачи, в противном случае возвращает `NULL`.
`void free(void *ptr);`	Освобождает память, на которую указывает `ptr`; `ptr` предварительно должен быть возвращен вызовом `calloc()`, `malloc()` или `realloc()`, или же `ptr` должен быть нулевым указателем — в этом случае никаких действий не предпринимается. Для других значений указателя поведение не определено.
`void *malloc(size_t size);`	Выделяет неинициализированный блок памяти размером в `size` байт; в случае удачи функция возвращает адрес блока памяти, в противном случае — `NULL`.
`void *realloc(void *ptr, size_t size);`	Изменяет размер блока памяти, на который указывает `ptr`, до `size` байт; содержимое блока в части, меньшей из двух размеров — старого и нового, остается неизменным; функция возвращает местоположение блока, который может быть перемещен; если же память не может быть перераспределена, функция возвращает `NULL` и оставляет исходный блок неизменным. Если `ptr` равно `NULL`, поведение аналогично вызову `malloc()` с аргументом `size`; если `size` равно нулю, а `ptr` не равен `NULL`, то поведение аналогично вызову `free()` с `ptr` в качестве аргумента.

Продолжение табл. RS.V.33

Прототип	*Описание*
`void abort(void);`	Приводит к аварийному прерыванию программы, если только не вызван сигнал `SIGABRT` и не возвращен соответствующий обработчик сигнала; закрытие потоков ввода-вывода и временных файлов зависит от реализации; функция выполняет `raise(SIGABRT)`.
`int atexit(void (*func)(void));`	Регистрирует функцию, на которую указывает `func`, для вызова при нормальном завершении программы; реализация должна поддерживать регистрацию как минимум 32 функций, которые должны быть вызваны в порядке, противоположном порядку их регистрации; функция возвращает ноль, если регистрация удалась и ненулевое значение – в противном случае.
`void exit(int status);`	Вызывает нормальное завершение программы, сначала вызывая функции, зарегистрированные `atexit()`, затем сбрасывая все открытые выходные потоки, после этого закрывая все потоки ввода-вывода, затем закрывая все файлы, созданные `tmpfile()`, после чего передает управление среде хоста (операционной системе). Если `status` равен `0` или `EXIT_SUCCESS`, то в среду хоста передается зависящий от реализации код возврата, означающий нормальное завершение. Если `status` равен `EXT_FAILURE`, в среду хоста передается зависящий от реализации код, означающий неудачное завершение. Эффект от других значений `status` также зависит от реализации.
`void _Exit(int status);`	Подобна `exit()`, но с тем отличием, что зарегистрированные `atexit()` и `signal()` функции не вызываются, а обработка открытых потоков зависит от реализации (C99).
`char *getenv(const char * name);`	Возвращает указатель на строку, представляющую значение переменной среды, имя которой указано в `name`. Если переменной с таким именем нет, возвращает `NULL`.
`int system(const char *str);`	Передает строку, указанную `str`, среде хоста для выполнения командным процессором, таким как `DOS` или `UNIX`. Если `str` равно `NULL`, функция возвращает ненулевое значение, если командный процессор доступен, и нулевое – если нет. Если `str` – не `NULL`, то возвращаемое значение зависит от конкретной реализации.

Продолжение табл. RS.V.33

Прототип	*Описание*
`void *bsearch(const void *key, const void *base, size_t nmem, size_t size, int (*comp)(const void *, const void *));`	Выполняет поиск объекта `key` в массиве, на который указывает `base`, размером `nmem` элементов, каждый из которых имеет размер `size`. Элементы сравниваются с помощью функции `comp`. Функция сравнения должна возвращать значение меньше нуля, если ключевой объект меньше элемента массива, ноль – если они эквивалентны, и значение больше нуля, если ключевой объект больше. Функция возвращает указатель на найденный элемент или `NULL`, если элемент не найден. Если найдены два или более искомых элемента, то какой из них будет выбран – не регламентируется.
`void qsort(void *base, size_t nmem, size_t size, int (*comp) (const void *, const void *));`	Сортирует массив, указанный `base`, в порядке, заданном функцией `comp`; массив состоит из `nmem` элементов, каждый размером в `size` байт. Функция сравнения должна возвращать значение меньше нуля, если объект, указанный в ее первом аргументе, меньше, чем объект, указанный вторым аргументом, ноль, если объекты эквивалентны и значение больше нуля, если первый объект больше.
`int abs(int n);`	Возвращает абсолютное значение `n`; возвращаемое значение может быть неопределенным, если `n` – отрицательная величина, не имеющая положительного аналога, что может случиться, когда `n` равно `INT_MIN` в двух дополняющих представлениях.
`div_t div(int numer, int denom);`	Вычисляет частное и остаток от деления `numer` на `denum`, помещая частное в элемент `quot` структуры `div_t`, а остаток от деления – в элемент `rem` этой структуры. При неточном делении частное равно целому значению наименьшей ближайшей величины к алгебраическому частному (то есть округленному в сторону нуля).
`long labs(int n);`	Возвращает абсолютное значение `n`; возвращаемое значение может быть неопределенным, если `n` – отрицательная величина, не имеющая положительного аналога, что может случиться, когда `n` равно `LONG_MIN` в двух дополняющих представлениях.
`ldiv_t ldiv(long numer, long denom);`	Вычисляет частное и остаток от деления `numer` на `denum`, помещая частное в элемент `quot` структуры `div_t`, а остаток от деления – в элемент `rem` этой структуры. При неточном делении частное равно целому значению наименьшей ближайшей величины к алгебраическому частному (то есть округленному в сторону нуля).

Продолжение табл. RS.V.33

Прототип	*Описание*
`long long llabs(int n);`	Возвращает абсолютное значение `n`; возвращаемое значение может быть неопределенным, если `n` — отрицательная величина, не имеющая положительного аналога, что может случиться, когда `n` равно `LONG_LONG_MIN` в двух дополняющих представлениях (C99).
`lldiv_t lldiv(long numer, long denom);`	Вычисляет частное и остаток от деления `numer` на `denum`, помещая частное в элемент `quot` структуры `div_t`, а остаток от деления — в элемент `rem` этой структуры. При неточном делении частное равно целому значению наименьшей ближайшей величины к алгебраическому частному (то есть округленному в сторону нуля) (C99).
`int mblen(const char *s, size_t n);`	Возвращает количество байт (до `n`), составляющих многобайтный символ, указанный `s`. Возвращает `0`, если `s` указывает на нулевой символ, и `-1`, если `s` не указывает на многобайтный символ. Если `s` равно `NULL`, возвращает ненулевое значение, когда многобайтные символы имеют зависящую от состояния кодировку, или ноль — в противном случае.
`int mbtowc(wchar_t *pw, const char *s, size_t n);`	Если `s` — не `NULL`, определяет количество символов (до `n`), составляющих многобайтный символ, на который указывает `s`, и определяет код символа типа `wchar_t`. Если `pw` не равно `NULL`, записывает код в место, указанное `pw`; возвращает то же значение, что и `mblen(s, n)`.
`int wctomb(char *s, wchar_t wc);`	Преобразует код символа `wc` в соответствующее представление многобайтного символа, и сохраняет его в массиве, на который указывает `s`, если только `s` не равно `NULL`. Если `s` не равно `NULL`, возвращает `-1`, когда `wc` не соответствует корректному многобайтному символу. Если же `wc` корректно, возвращает количество байт, составляющих многобайтный символ. Если `s` равно `NULL`, возвращает ненулевое значение, когда многобайтный символ имеет зависящую от состояния кодировку, и ноль — в противном случае.
`size_t mbstowcs( wchar_t * restrict pwcs, const char *s restrict, size_t n);`	Преобразует массив многобайтных символов, указанный `s`, в массив кодов расширенных символов, сохраняя его в место, указанное `pwcs`. Преобразование выполняется для, максимум, `n` элементов массива `pwcs` либо до нулевого байта в массиве `s`, в зависимости от того, что произойдет раньше. Если встречается некорректный многобайтный символ, возвращает `(size_t) (-1)`; иначе возвращает количество загрузленных элементов массива (за исключением нулевого символа, если он есть).

Окончание табл. RS.V.33

Прототип	*Описание*

`size_t wcstombs(`
`char * restrict s,`
`const wchart_t * restrict pwcs,`
`size_t n);`
	Преобразует последовательность кодов расширенных символов в массиве, указанном `pwcs`, в последовательность многобайтных символов, копируя ее в место, указанное `s`. Обработка прекращается либо до сохранения `n` байт, либо до нулевого символа — в зависимости от того, что произойдет раньше. Если встречается некорректный многобайтный символ, возвращает `(size_t) (-1)`; иначе возвращает количество заполненных элементов массива (за исключением нулевого символа, если он есть).

Обработка строк: `string.h`

Библиотека `string.h` определяет тип `size_t` и макрос `NULL` для нулевого указателя. Она также представляет ряд функций для анализа и манипулирования символьными строками, а также несколько функций, работающих памятью в более общем виде. Эти функции перечислены в табл. RS.V.34.

Таблица RS.V.34. Строковые функции

Прототип	*Описание*
`void *memchr(const void *s,` `int c, size_t n);`	Ищет первое вхождение `c` (преобразованное в `unsigned char`) в первых `n` символах объекта, на который указывает `s`; возвращает указатель на первое вхождение, или `NULL`, если вхождение не найдено.
`int memcmp(const void *s1,` `const void *s2, size_t n);`	Сравнивает первые `n` символов объекта, на который указывает `s1`, с первыми `n` символами объекта, на который указывает `s2`, интерпретируя каждый символ как `unsigned char`. Два объекта идентичны, если первые `n` пар совпадают; в противном случае объекты сравниваются по первой несовпадающей паре. Возвращает ноль, если объекты идентичны, значение меньше нуля — если первый объект в числовом виде меньше второго, и значение больше нуля — если первый объект больше второго.

`void *memcpy(void * restrict s1,`
`const void * restrict s2,`
`size_t n);`
	Копирует `n` байт из места, указанного `s2`, в место, указанное `s1`; поведение не определено, если две области перекрываются; возвращает значение `s1`.

`void *memmove(void *s1,` `const void *s2, size_t n);`	Копирует `n` байт из места, указанного `s2`, в место, указанное `s1`; поведение аналогично `memmove()`. Сначала использует временное место, чтобы обеспечить перекрывающееся копирование; возвращает значение `s1`.

Продолжение табл. RS.V.34

Прототип	*Описание*
`void *memset(void *s, int v, size_t n);`	Копирует значение `v` (преобразованное в тип `unsigned char`) в первые `n` байт, находящиеся по адресу, указанному `s`; возвращает `s`.
`char *strcat(char * restrict s1, const char * restrict s2);`	Добавляет копию строки, на которую указывает `s2` (включая нулевой символ), в место, указанное `s1`; при этом первый символ `s2` перекрывает нулевой символ `s1`; возвращает `s1`.
`char *strncat(char * restrict s1, const char * restrict s2, size_t n);`	Добавляет копию до `n` символов, или до нулевого символа строки, указанной `s2`, в место, указанное `s1`; при этом первый символ `s2` перекрывает нулевой символ `s1`; всегда добавляется нулевой символ; функция возвращает `s1`.
`char *strcpy(char * restrict s1, const char * restrict s2);`	Копирует строку, указанную `s2` (включая нулевой символ) в место, указанное `s1`; возвращает `s1`.
`char *strncpy(char * restrict s1, const char * restrict s2, size_t n);`	Копирует до `n` символов, или до нулевого символа из строки, указанной `s2`, в место, указанное `s1`. Если в `s2` встретится нулевой символ до того, как будут скопированы `n` символов, то будут дописаны нулевые символы, недостающие до общего количества `n`. Если же `n` символов будут скопированы до того, как встретится нулевой символ, то никакого нулевого символа не добавляется. Строка возвращает `s1`.
`int strcmp(const char *s1, const char *s2);`	Сравнивает строки, указанные `s1` и `s2`. Две строки идентичны, если совпадают все пары соответствующих символов; иначе строки сравниваются по первой несовпадающей паре. Символы сравниваются по значениям их кодов; функция возвращает ноль, если строки идентичны, отрицательное значение, если первая строка меньше второй, и положительное – если первая строка больше второй.
`int strcoll(const char *s1, const char *s2);`	Работает подобно `strcmp()`, но использует последовательность сопоставления, описанную категорией `LC_COLLATE` текущей локальной установки, установленной функцией `setlocale()`.
`int strncmp(const char *s1, const char *s2, size_t n);`	Сравнивает до `n` первых символов или до нулевого символа двух массивов, указанных `s1` и `s2`; два массива идентичны, если все проверенные пары символов совпадают; иначе массивы сравниваются по первой несовпадающей паре. Символы сравниваются по значениям их кодов; функция возвращает ноль, если массивы идентичны, значение меньше нуля, если первый массив меньше второго, и значение больше нуля, если первый массив больше второго.

Продолжение табл. RS.V.34

Прототип	*Описание*
`size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);`	Трансформирует строку `s2` и копирует до `n` символов, включая завершающий нулевой символ, в массив, указанный `s1`. Критерием трансформации является то, что трансформированные строки должны упорядочиваться функцией `strcmp()` в том же порядке, в каком `strcol()` размещала бы их нетрансформированные версии. Функция возвращает длину трансформированной строки (исключая нулевой символ).
`char *strchr(const char *s, int c);`	Ищет первое вхождение `c` (преобразованного в `char`) в строке, на которую указывает `s`; нулевой символ рассматривается как часть строки; возвращает указатель на первое вхождение или `NULL`, если символ не найден.
`size_t strcspn(const char *s1, const char *s2);`	Возвращает длину максимального начального сегмента `s1`, который не содержит ни одного символа из `s2`.
`char *strpbrk(const char *s1, const char *s2);`	Возвращает указатель на первый символ `s1`, совпадающий с любым из символов `s2`; возвращает `NULL`, если соответствие не обнаружено.
`char *strrchr(const char *s, int c);`	Ищет последнее вхождение `c` (преобразованного в `char`) в строке, на которую указывает `s`; нулевой символ рассматривается как часть строки; возвращает указатель на первое вхождение, или `NULL`, если символ не найден.
`size_t strspn(const char *s1, const char *s2);`	Возвращает длину максимального начального сегмента `s1`, который полностью состоит из символов, входящих в `s2`.
`char *strstr(const char *s1, const char *s2);`	Возвращает указатель на место первого вхождения в `s1` последовательности символов `s2` (исключая нулевой символ); возвращает `NULL`, если вхождение не обнаружено.
`char *strtok(char * restrict s1, const char * restrict s2);`	Эта функция разбивает строку `s1` на отдельные лексемы. Строка `s2` содержит символы, служащие разделителями лексем. Функция вызывается последовательно. При начальном вызове `s1` должно указывать на строку, которую требуется разбить на лексемы. Функция находит первый разделитель, который следует за символом, не являющимся разделителем, и заменяет его нулевым символом. Она возвращает указатель на строку, содержащую первую лексему. Если никаких лексем не найдено, возвращается `NULL`. Чтобы найти последующие лексемы строки, `strtok()` нужно вызвать снова, но в качестве первого аргумента указать `NULL`. Каждый последующий вызов возвращает указатель на следующую лексему или `NULL`, если больше лексем не найдено. (См. пример, следующий за настоящей таблицей.)

Окончание табл. RS.V.34

Прототип	*Описание*
`char * strerror(int errnum);`	Возвращает указатель на зависящую от реализации строку сообщения об ошибке, которая соответствует номеру ошибки, переданному в `errnum`.
`int strlen(const char * s);`	Возвращает количество символов (исключая нулевой) в строке `s`.

Функция `strtok()` применяется несколько необычно, поэтому рассмотрим небольшой пример:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char data[] = "  C is\t too#much\nfun!";
   const char tokseps[] = " \t\n#";  /* разделители         */
   char * pt;

   puts(data);
   pt = strtok(data,tokseps);        /* первый вызов        */
   while (pt)                        /* выход в случае NULL */
   {
      puts (pt);                     /* показать лексему    */
      pt = strtok(NULL, tokseps);    /* следующая лексема   */
   }
   return 0;
}
```

Вывод этого примера представлен ниже:

```
 C is      too#much
fun!
C
is
too
much
fun!
```

Математические функции для общих типов: `tgmath.h` (C99)

Библиотеки `math.h` и `complex.h` представляют множество экземпляров функций, которые отличаются лишь типами параметров и возврата. Например, все следующие функции вычисляют синус:

```
double sin(double);
float sinf(float);
long double sinl(long double);
double complex csin(double complex);
```

```
float csinf(float complex);
long double csinl(long double complex);
```

В заголовочном файле tgmath.h определены макросы, которые расширяют обобщенный вызов в соответствующую функцию на основе типа аргумента. Следующий код иллюстрирует использование макроса sin(), расширяемого в различные формы функции синуса:

```
#include <tgmath.h>
...
double dx, dy;
float fx, fy;
long double complex clx, cly;
dy = sin(dx);       // расширяется в dy = sin(dx) (функция)
fy = sin(fx);       // расширяется в fy = sinf(fx)
cly = sin(clx);     // расширяется в cly = csinl(clyx)
```

Этот заголовочный файл определяет обобщенные макросы для трех классов функций. Первый класс состоит из функций math.h и complex.h, определенных в шести вариациях, используя суффиксы f и l и префикс c, как в предыдущем примере с sin(). В данном случае обобщенный макрос носит то же имя, что и double-версия функции.

Второй класс состоит из функций math.h, определенных в трех вариациях с применением суффиксов f и l, и не имеющих комплексных аналогов, таких как erf(). В этом случае имя макроса такое же, как и имя функции без суффикса, в данном примере — erf(). Эффект от применения этого макроса с комплексным аргументом не определен.

Третий класс состоит из функций complex.h, определенных в трех вариациях с применением суффиксов f и l, и не имеющих действительных аналогов, таких как cimag(). В данном случае имя макроса — такое же, как и у функции без суффикса, в данном примере — cimag().Эффект от применения этого макроса с действительным аргументом не определен.

Дата и время: time.h

Заголовочный файл time.h определяет два макроса. Первый, также определенный во многих других заголовочных файлах, — это NULL, представляющий нулевой указатель. Второй макрос — CLOCKS_PER_SEC. Разделение этим макросом значения, возвращенного функцией clock(), дает время в секундах.

Определенные в этом заголовочном файле типы перечислены в табл. RS.V.35.

Таблица RS.V.35. Типы, определенные в time.h

Тип	*Описание*
size_t	Целочисленный тип, возвращаемый операцией sizeof.
clock_t	Арифметический тип, подходящий для представления времени.
time_t	Арифметический тип, подходящий для представления времени.
struct tm	Структурный тип, содержащий компоненты календарного времени.

Компоненты календарного типа называют *разбитым временем*. В табл. RS.V.36 перечислены обязательные элементы структуры struct tm.

Таблица RS.V.36. Элементы структуры `struct tm`

Элемент	*Описание*
`int tm_sec`	Секунды после минут (0–61).
`int tm_min`	Минуты после часов (0–59).
`int tm_hour`	Часы после полуночи (0–23).
`int tm_mday`	Дни месяца (1–31).
`int tm_mon`	Месяцы, начиная с января (0–11).
`int tm_year`	Годы, начиная с 1900.
`int tm_wday`	Дни, начиная с воскресенья (0–6).
`int tm_yday`	Дни, начиная с 1 января (0–365).
`int tm_isdst`	Флаг перехода на летнее время (больше нуля – значит, переход на летнее время включен; ноль – нет; отрицательное значение говорит о том, что информация недоступна).

Термин *календарное время* означает текущую дату и время; например, это может быть количество секунд, прошедших после первой секунды 1900 года. Термин *локальное время* – это календарное время, выраженное для локального часового пояса. В табл. RS.V.37 перечислены функции времени.

Таблица RS.V.37. Функции времени

Прототип	*Описание*
`clock_t clock(void);`	Возвращает наилучшее для данной реализации приближение процессорного времени, прошедшего с момента запуска программы; для получения времени в секундах значение необходимо разделить на `CLOCKS_PER_SEC`. Если время не доступно или не может быть представлено, возвращает `(clock_t)(-1)`.
`double difftime(time_t t1, time_t t0);`	Вычисляет разницу `(t1 - t0)` между двумя моментами календарного времени; выражает результат в секундах и возвращает результат.
`time_t mktime(struct tm *tmptr);`	Преобразует разбитое время, представленное в структуре, на которую указывает `tmptr`, в календарное время; используя ту же кодировку, что функция `time()`, структура изменяется тем, что значения, выходящие за диапазон допустимых, исправляются (например, 2 минуты и 100 секунд становятся 3 минутами и 40 секундами), а `tm_wday` и `tm_yday` устанавливаются в величины, вытекающие из значений остальных элементов структуры. Если календарное время не может быть правильно представлено, возвращает `(time_t)(-1)`, иначе возвращает календарное время в формате `time_t`.

Окончание табл. RS.V.37

Прототип	*Описание*
`time_t time(time_t *ptm)`	Возвращает текущее календарное время и помещает его в место, на которое указывает `ptm`, предполагая, что `ptm` не равен `NULL`. Если календарное время не доступно, возвращает `(time_t)(-1)`.
`char *asctime(const struct tm *tmpt);`	Преобразует разбитое время из структуры, указанной `tmpt`, в строку вида `Sun Feb 26 13:14:33 2006\n\0` и возвращает указатель на эту строку.
`char *ctime(const time_t *ptm);`	Преобразует календарное время из структуры, указанной `ptm`, в строку вида `Mon Feb 27 10:48:24 2006\n\0` и возвращает указатель на эту строку.
`struct tm *gmtime(const time_t *ptm);`	Преобразует календарное время из структуры, указанной `ptm`, в разбитое время, выраженное в виде универсального глобального времени (UTC), ранее известного как время по Гринвичу (GMT), и возвращает указатель на структуру, содержащую эту информацию. Если UTC недоступно, возвращает `NULL`.
`struct tm *localtime(const time_t *ptm);`	Преобразует календарное время, указанное `ptm`, в разбитое время, выраженное как местное. Сохраняет его в структуре `tm` и возвращает указатель на нее.
`size_t strftime(char * restrict s, size_t max const char * restrict fmt, const struct tm * restrict tmpt);`	Копирует строку `fmt` в строку `s`, заменяя спецификаторы формата (см. табл. RS.V.38) в `fmt` соответствующими данными, взятыми из содержимого структуры разбитого времени, на которое указывает `tmpt`; в строку `s` помещается не более `max` символов. Функция возвращает количество символов (исключая нулевой) в результирующей строке. Если результирующая строка (включая нулевой символ) содержит больше, чем `max` символов, функция возвращает `0`, а содержимое `s` не определено.

В табл. RS.V.38 представлены спецификаторы формата, используемые функцией `strftime()`. Многие заменяемые значения, такие как названия месяцев, зависят от текущей локальной установки.

Таблица RS.V.38. Спецификаторы формата, используемые функцией `strftime()`

Спецификатор формата	*Заменяется на*
`%a`	Локальное сокращенное название дня недели.
`%A`	Локальное полное название дня недели.
`%b`	Локальное сокращенное название месяца.
`%B`	Локальное полное название месяца.
`%c`	Локальный разделитель даты и времени.

Окончание табл. RS.V.38

Спецификатор формата	*Заменяется на*
`%d`	День месяца в виде десятичного числа (01–31).
`%D`	Эквивалент `"%m/%d%y"`.
`%e`	День месяца в виде десятичного числа — однозначные числа дополнены пробелом.
`%F`	Эквивалент `"%Y-%m-%d"`.
`%g`	Последние два разряда года (00–99).
`%G`	Год в виде десятичного числа.
`%h`	Эквивалент “%b”.
`%H`	Часы (по 24-часовой шкале) в виде десятичного числа (00–23).
`%I`	Часы (по 12-часовой шкале) в виде десятичного числа (01–12).
`%j`	День года в виде десятичного числа (001–366).
`%m`	Месяц в виде десятичного числа (01–12).
`%n`	Символ новой строки.
`%M`	Минуты в виде десятичного числа (00–59).
`%p`	Локальный эквивалент a.m./p.m. для 12-часовой временной шкалы.
`%r`	Локальное 12-часовое время.
`%R`	Эквивалент `"%H:%M"`.
`%S`	Секунды в виде десятичного числа (00–61).
`%t`	Символ горизонтальной табуляции.
`%T`	Эквивалент `"%H:%M:%S"`.
`%u`	Номер дня недели по ISO 8601 (1–7), где 1 соответствует понедельнику.
`%U`	Номер недели в году, считая воскресенье первым днем недели (00–53).
`%V`	Номер недели в году в соответствие с ISO 8601, считая воскресенье первым днем недели (00–53).
`%w`	Номер дня недели в виде десятичного числа, начиная с воскресенья (0–6).
`%W`	Номер недели в году, считая понедельник первым днем недели (00–53).
`%x`	Локальное представление даты.
`%X`	Локальное представление времени.
`%y`	Год без века в виде десятичного числа (00–99).
`%Y`	Год с веком в виде десятичного числа.
`%z`	Смещение от UTC в формате ISO 8601 (“–800” означает восемь часов по Гринвичу, то есть на восемь часов западнее); никакие символы не подставляются, если информация недоступна.
`%Z`	Наименование часового пояса; никакие символы не подставляются, если информация недоступна.
`%%`	`%` (то есть знак процента).

Утилиты для работы с многобайтными и расширенными символами: wchar.h (C99)

Каждая реализация имеет свой базовый набор символов, и тип char языка C должен быть достаточно широким, чтобы поддерживать этот набор. Реализация может также поддерживать расширенные наборы символов, а эти символы могут требовать для своего представления более одного байта на символ. Многобайтные символы могут сохраняться наряду с однобайтными в обычном массиве char, где определенные значения байта служат признаками присутствия многобайтного символа и его размера. Интерпретация многобайтных символов может зависеть от *состояния сдвига*. В начальном состоянии сдвига однобайтные символы интерпретируются обычным образом. Специфические многобайтные символы затем могут изменять состояние сдвига. Определенное состояние сдвига остается в силе до тех пор, пока не будет изменено явно.

Тип wchar_t представляет второй способ представления расширенных символов, когда ширина типа выбирается достаточной для представления кодировки любого элемента расширенного набора символов. Такое представление расширенных символов позволяет помещать отдельные символы в переменные типа wchar_t, а строки таких символов представлять в виде массивов wchar_t. Представление расширенных символов не обязательно должно совпадать с многобайтным представлением, потому что последние могут использовать состояния сдвига, в то время как первые — нет.

Заголовочный файл wchar_t предоставляет средства для обработки обоих представлений расширенных символов. Он определяет типы, перечисленные в табл. RS.V.39 (некоторые из этих типов также представлены в других заголовочных файлах).

Таблица RS.V.39. Типы, определенные в wchar.h

Тип	*Описание*
wchar_t	Целочисленный тип, который может представить самый большой набор символов, описанный поддерживаемыми локальными установками.
wint_t	Целочисленный тип, который может вместить любое значение из расширенного набора символов плюс, по крайней мере, одно значение, не входящее в расширенный набор символов.
size_t	Целочисленный тип, возвращаемый операцией sizeof.
mbstate_t	Немассивный тип, который может содержать информацию преобразования, необходимую для преобразования между последовательностями многобайтных и широких символов.
struct tm	Структурный тип, содержащий компоненты календарного времени.

Заголовочный файл также содержит определения некоторых макросов, перечисленных в табл. RS.V.40.

Таблица RS.V.40. Макросы, определенные в wchar.h

Макрос	*Описание*
NULL	Нулевой указатель.
WCHAR_MAX	Максимальное значение wchar_t.
WCHAR_MIN	Минимальное значение wchar_t.
WEOF	Константное выражение типа wchar_t, которое не соответствует ни одному элементу расширенного набора символов; эквивалент EOF в расширенных символах, используемый для индикации состояния конца файла при вводе расширенных символов.

Библиотека содержит функции ввода-вывода, аналогичные стандартным функциям ввода-вывода, определенным в stdio.h. В тех случаях, когда стандартная функция ввода-вывода возвращает EOF, соответствующая функция для расширенных символов возвращает WEOF. Эти функции перечислены в табл. RS.V.41.

Таблица RS.V.41. Функции ввода-вывода, работающие с расширенными символами

Прототипы функций
`int fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...);`
`int fwscanf(FILE * restrict stream,const wchar_t * restrict format, ...);`
`int swprintf(wchar_t * restrict s, size_t n,const wchar_t * restrict format, ...);`
`int swscanf(const wchar_t * restrict s,const wchar_t * restrict format, ...);`
`int vfwprintf(FILE * restrict stream, const wchar_t * restrict format, va_list arg);`
`int vfwscanf(FILE * restrict stream, const wchar_t * restrict format, va_list arg);`
`int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, va_list arg);`
`int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format, va_list arg);`
`int vwprintf(const wchar_t * restrict format, va_list arg);`
`int vwscanf(const wchar_t * restrict format, va_list arg);`
`int wprintf(const wchar_t * restrict format, ...);`
`int wscanf(const wchar_t * restrict format, ...);`
`wint_t fgetwc(FILE *stream);`
`wchar_t *fgetws(wchar_t * restrict s, int n, FILE * restrict stream);`
`wint_t fputwc(wchar_t c, FILE *stream);`
`int fputws(const wchar_t * restrict s, FILE * restrict stream);`
`int fwide(FILE *stream, int mode);`
`wint_t getwc(FILE *stream);`
`wint_t getwchar(void);`
`wint_t putwc(wchar_t c, FILE *stream);`
`wint_t putwchar(wchar_t c);`
`wint_t ungetwc(wint_t c, FILE *stream);`

Существует одна функция ввода-вывода с расширенными символами, не имеющая аналогов в стандартной библиотеке ввода-вывода:

```
int fwide(FILE *stream, int mode);
```

Если аргумент mode положительный, то сначала она пытается трактовать поток, представленный параметром stream, как ориентированный на расширенные символы, а если mode отрицательный — то как ориентированный на байты. Если же mode равно нулю, функция не пытается изменить ориентацию потока. Попытка изменить ориентацию предпринимается, если поначалу она вообще не была указана. Во всех случаях функция возвращает положительное значение, если поток ориентирован на расширенные символы, отрицательное — если поток байт-ориентированный и ноль — если ориентация потока не установлена.

Заголовочный файл представляет несколько функций манипуляции и преобразования строк, которые моделируют определенные в string.h. Вообще, фрагмент "str" в идентификаторах из string.h заменяется на "wcs", поэтому wcstod() — это версия strtod() для расширенных символов. В табл. RS.V.42 эти функции перечислены.

Таблица RS.V.42. Строковые утилиты для расширенных символов

Прототипы функций
double wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
long long int wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
unsigned long int wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
unsigned long long int wcstoull(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);
wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
wchar_t *wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);
wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
int wcscmp(const wchar_t *s1, const wchar_t *s2);
int wcscoll(const wchar_t *s1, const wchar_t *s2);
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
size_t wcsxfrm(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
wchar_t *wcschr(const wchar_t *s, wchar_t c);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
size_t wcslen(const wchar_t *s);
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);

Окончание табл. RS.V.42

Прототипы функций
`wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);`
`wchar_t *wcstok(wchar_t * restrict s1, const wchar_t * restrict s2, wchar_t ** restrict ptr);`
`wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);`
`int wmemcmp(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);`
`wchar_t *wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);`
`wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);`
`wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);`

Этот заголовочный файл также объявляет функцию, моделирующую `strftime()` из `time.h`:

```
size_t wcsftime(wchar_t * restrict s, size_t maxsize,
const wchar_t * restrict format,
const struct tm * restrict timeptr);
```

И, наконец, здесь также объявлено несколько функций для преобразования строк с расширенными символами в многобайтные строки и наоборот, как показано в табл. RS.V.43.

Таблица RS.V.43. Функции преобразования расширенных и многобайтных символов

Прототип	*Описание*
`wint_t btowc(int c);`	Если `(unsigned char) c` – допустимый однобайтный символ в начальном состоянии сдвига, то функция возвращает его представление в расширенных символах, иначе возвращает `WEOF`.
`int wctob(wint_t c);`	Если `c` – элемент расширенного набора символов, чье многобайтное символьное представление в начальном состоянии сдвига представлено одним байтом, то функция возвращает однобайтное представление `unsigned char`, преобразованное в `int`; иначе функция возвращает `EOF`.
`int mbsinit(const mbstate_t *ps);`	Функция возвращает ненулевое значение, если `ps` — нулевой указатель или указывает на объект данных, описывающий начальное состояние преобразования; в противном случае функция возвращает ноль.
`size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);`	Функция `mbrlen()` эквивалентна вызову `mbrtowc(NULL, s, n, ps != NULL ? ps : &internal)`, где `internal` — объект `mbstate_t` для функции `mbrlen()`, за исключением того, что выражение, назначенное `ps`, оценивается только однажды.

Продолжение табл. RS.V.43

Прототип	*Описание*

`size_t mbrtowc(`
`wchar_t * restrict pwc,`
`const char * restrict s,`
`size_t n,`
`mbstate_t * restrict ps);`
	Если `s` — нулевой указатель, этот вызов эквивалентен установке `pwc` в нулевой указатель, а `n` — в `1`. Если же `s` не равно нулю, функция инспектирует максимум `n` байт с целью определения количества байт, необходимых для завершения следующего многобайтного символа (включая любые последовательности сдвига). Если функция определяет, что следующий многобайтный символ завершен и корректен, она определяет значение соответствующего расширенного символа и затем, если `pwc` — ненулевой указатель, сохраняет это значение в объекте, на который указывает `pwc`. Если соответствующий расширенный символ является нулевым, результирующее состояние описывается как начальное состояние преобразования. Функция возвращает `0`, если обнаружен нулевой расширенный символ. Если же обнаружен другой допустимый расширенный символ, она возвращает количество байтов, необходимых для завершения символа. Если `n` байт недостаточно для того, чтобы описать допустимый расширенный символ, и потенциально возможно его частичное представление, то функция возвращает `-2`. Если обнаружена ошибка кодирования, функция возвращает `-1`, записывает `EILSEQ` в `errno` и ничего не сохраняет.

`size_t wcrtomb(`
`char * restrict s,`
`wchar_t wc,`
`mbstate_t * restrict ps);`
	Если `s` — нулевой указатель, этот вызов эквивалентен установке `wc` в расширенный нулевой символ и использованию внутреннего буфера для первого аргумента. Если `s` — ненулевой указатель, функция `wcrtomb()` определяет количество байт, необходимое для представления многобайтного символа, соответствующего расширенному символу, заданному `wc` (включая любые последовательности сдвига), и сохраняет многобайтное представление в массиве, на первый элемент которого указывает `s`. Сохраняется максимум `MB_CUR_MAX` символов. Если `wc` — расширенный нулевой символ, сохраняется нулевой байт, которому предшествует необходимая для восстановления начального состояния сдвига последовательность сдвига; результирующее состояние описывает начальное состояние преобразования. Если `wc` — допустимый расширенный символ, функция возвращает количество байтов, необходимых для сохранения многобайтной версии, включая байты, описывающие состояние сдвига (если они есть). Если `wc` не является допустимым, функция записывает `EILSEQ` в `errno`, и возвращает `-1`.

Окончание табл. RS.V.45

Прототип	*Описание*
`size_t mbsrtowcs( wchar_t * restrict dst, const char ** restrict src, size_t len, mbstate_t * restrict ps);`	Функция `mbstrtowcs()` преобразует последовательность многобайтных символов, начинающуюся в состоянии преобразования, описанном объектом, на который указывает `ps`, из массива, косвенно указанного `src`, в последовательность соответствующих расширенных символов. Если `dst` — ненулевой указатель, преобразованные символы сохраняются в массиве, на который указывает `dst`. Преобразование продолжается вплоть до завершающего нулевого символа, включая его. Преобразование прекращается раньше в двух случаях: когда встречается последовательность байтов, которая не может формировать допустимый многобайтный символ, и (если `dst` — ненулевой символ) когда `len` расширенных символов уже сохранено в массив, указанный в `dst`. Каждое преобразование выполняется, как будто оно инициировано вызовом функции `mbrtowc()`. Если `dst` — ненулевой указатель, то объекту указателя, на который указывает `src`, присваивается нулевое значение (если преобразование остановилось по достижении нулевого символа) или адрес, следующий сразу за последним преобразованным многобайтным символом (если он есть). Если же преобразование прекратилось по достижении нулевого символа и `dst` — ненулевой указатель, то результирующее состояние описывается как начальное состояние конверсии. В случае успеха функция возвращает количество преобразованных многобайтных символов (исключая нулевой, если он есть); в противном случае функция возвращает `-1`.
`size_t wcsrtombs( char * restrict dst, const wchar_t ** restrict src, size_t len, mbstate_t * restrict ps);`	Функция `wcsrtombs()` преобразует последовательность расширенных символов из массива, косвенно указанного `src`, в последовательность соответствующих многобайтных символов, начинающихся в состоянии преобразования, описанном объектом, на который указывает `ps`. Если `dst` — ненулевой указатель, то преобразованные символы затем сохраняются в массиве, указанном `dst`. Преобразование продолжается вплоть до нулевого символа, включая его, а прекращается раньше в двух случаях: когда встречается расширенный символ, который не соответствует допустимому многобайтному символу, и (если `dst` — ненулевой символ) когда следующий многобайтный символ должен превысить лимит общего количества `len` байтов, которые должны быть сохранены в массиве, указанном `dst`. Каждое преобразование выполняется, как если бы была вызвана функция `wcrtomb()`. Если `dst` — ненулевой символ, то объекту указателя, на который указывает `src`, присваивается либо нулевое значение (если преобразование остановлено по достижении расширенного нулевого символа), либо адрес символа, следующего сразу за последним преобразованным расширенным символом (если он был). Если же преобразование остановлено по достижении нулевого расширенного символа, то результирующее состояние описывается как начальное состояние преобразования. В случае успеха функция возвращает количество многобайтных символов в результирующей многобайтной последовательности (исключая нулевой символ, если он есть); в противном случае возвращается `-1`.

Утилиты классификации и отображения расширенных символов: wctype.h (C99)

Библиотека wctype.h предлагает аналоги символьных функций из ctype.h, а также несколько дополнительных функций. Кроме того, она определяет три типа и макрос, перечисленные в табл. RS.V.44.

Таблица RS.V.44. Типы и макросы, определенные в wctype.h

Тип/макрос	*Описание*
wint_t	Целочисленный тип, который может содержать любое значение расширенного символьного набора, плюс как минимум, одно значение, не входящее в этот набор.
wctrans_t	Скалярный тип, который может представлять локальные специфичные символьные отображения.
wctype_t	Скалярный тип, который может представлять локальные специфичные символьные классификации.
WEOF	Константное выражение типа wint_t, которое не соответствует ни одному элементу из расширенного символьного набора; эквивалент EOF в расширенных символах, применяется для обозначения конца файла при вводе с использованием расширенных символов.

Символьные классификации в этой библиотеке возвращают true (ненулевое значение), если аргумент расширенного символа удовлетворяет условиям, описанным функцией. В общем случае, функция расширенных символов возвращает true, если соответствующая функция ctype.h возвращает true для однобайтного символа, соответствующего расширенному. В табл. RS.V.45 эти функции перечислены.

Таблица RS.V.45. Функции классификации расширенных символов

Прототип	*Описание*
int iswalnum(wint_t wc);	Возвращает true, если wc представляет цифру или букву.
int iswalpha(wint_t wc);	Возвращает true, если wc представляет букву.
int iswblank(wint_t wc);	Возвращает true, если wc представляет пробел.
int iswcntrl(wint_t wc);	Возвращает true, если wc представляет управляющий символ.
int iswdigit(wint_t wc);	Возвращает true, если wc представляет десятичную цифру.
int iswgraph(wint_t wc);	Возвращает true, если iswprint(wc) равно true, а iswspace(wc) — false.
int iswlower(wint_t wc);	Возвращает true, если wc представляет символ нижнего регистра.
int iswprint(wint_t wc);	Возвращает true, если wc представляет печатаемый символ.
int iswpunct(wint_t wc);	Возвращает true, если wc представляет знак пунктуации.
int iswspace(wint_t wc);	Возвращает true, если wc представляет символ табуляции, пробел или символ новой строки.
int iswupper(wint_t wc);	Возвращает true, если wc представляет символ верхнего регистра.
int iswxdigit(wint_t wc);	Возвращает true, если wc представляет шестнадцатеричную цифру.

Библиотека также содержит две классифицирующие функции, которые называются *расширяемыми*, поскольку для классификации символов они используют значение LC_CTYPE текущей локальной установки. Эти функции перечислены в табл. RS.V.46.

Таблица RS.V.46. Расширяемые функции классификации широких символов

Прототип	*Описание*
`int iswctype(wint_t wc, wctype_t desc);`	Возвращает `true`, если `wc` обладает свойством, описанным в `desc`.
`wctype_t wctype( const char *property);`	Функция `wctype` конструирует значение типа `wctype_t`, описывающее класс расширенных символов, идентифицированный строковым аргументом `property`. Если `property` идентифицирует допустимый класс расширенных символов в соответствии с категорией LC_CTYPE текущей локальной установки, то функция `wctype()` возвращает ненулевое значение, которое подходит в качестве второго аргумента функции `iswctype()`; в противном случае функция возвращает ноль.

Допустимые аргументы для `wctype()` состоят из имен функций классификации расширенных символов, из которых исключен префикс "isw", и выраженных в виде строк. Например, `wctype("alpha")` характеризует класс символов, проверяемых функцией `iswalpha()`. Таким образом, вызов

```
iswctype(wc, wctype("alpha"))
```

эквивалентен вызову

```
iswalpha(wc)
```

за исключением того, что символы классифицируются с применением категорий LC_CTYPE.

Библиотека представляет четыре функции преобразования. Две из них — `toupper()` и `tolower()` из библиотеки `ctype.h`. Третья — расширенная версия, использующая локальные настройки LC_CTYPE для определения символов верхнего и нижнего регистра. Четвертая представляет подходящие классификационные аргументы для третьей. Все эти функции перечислены в табл. RS.V.47.

Таблица RS.V.47. Функции трансформации расширенных символов

Прототип	*Описание*
`wint_t towlower(wint_t wc);`	Возвращает версию `wc` нижнего регистра, если `wc` представлено в верхнем регистре; в противном случае возвращает `wc`.
`wint_t towupper(wint_t wc);`	Возвращает версию `wc` верхнего регистра, если `wc` представлено в нижнем регистре; в противном случае возвращает `wc`.

Окончание табл. RS.V.47

Прототип	*Описание*
`wint_t towctrans(wint_t wc, wctrans_t desc);`	Возвращает версию `wc` нижнего регистра (как определено настройками `LC_CTYPE`), если `desc` эквивалентно возвращаемому значению `wctrans("lower")`; возвращает версию `wc` верхнего регистра (как определено настройками `LC_CTYPE`), если `desc` эквивалентно возвращаемому значению `wctrans("upper")`.
`wctrans_t wctrans( const char *property);`	Если аргументом является `"upper"` или `"lower"`, функция возвращает значение `wctrans_t`, используемое в качестве аргумента `towctrans()` и отображающее установки `LC_CTYPE`; в противном случае возвращается `0`.

Раздел VI. Расширенные целочисленные типы

Как было описано в главе 3, заголовочный файл C99 `inttypes.h` представляет систематизированный набор альтернативных имен различных целочисленных типов. Эти имена описывают свойства типа более ясно, чем стандартные имена. Например, тип `int` может быть 16-, 32- или 64-разрядным, однако тип `int32_t` — всегда 32-разрядный.

Выражаясь более точно, заголовочный файл `inttypes.h` определяет макросы, которые могут использоваться в функциях `scanf()` и `printf()` для чтения и записи целых значений этих типов. Этот заголовочный файл включает в себя заголовочный файл `stdlib.h`, который обеспечивает действительные определения типов. Форматирующие макросы — это строки, которые могут быть сцеплены с другими строками для формирования допустимых директив формата.

Типы определены через `typedef`. Например, система с 32-разрядным `int` может использовать такое определение:

```
typedef int int32_t;
```

Спецификаторы формата определены с помощью директив `#define`. Например, система, использующая приведенное выше определение `int32_t`, может иметь следующие определения:

```
#define PRId32 "d" // спецификатор вывода
#define SCNd32 "d" // спецификатор ввода
```

Используя эти определения, вы можете объявлять расширенные целочисленные переменные, вводить их значения и отображать следующим образом:

```
int32_t cd_sales; // 32-разрядное целое
scanf("%" SCNd32, &cd_sales);
printf("Продажи компакт-дисков составили %10" PRId32 " единиц\n", cd_sales);
```

При необходимости применяется конкатенация строк, чтобы получить финальную управляющую строку. Таким образом, предыдущий код преобразуется к следующему виду:

```
int cd_sales; // 32-разрядное целое
scanf("%d", &cd_sales);
printf("Продажи компакт-дисков составили %10d единиц\n", cd_sales);
```

Если вы переносите код в систему с 16-разрядным `int`, то эта система может определять `int32_t` как `long`, `PRId32` — как `"ld"`, а `SCNd32` — как `"ld"`.

В оставшейся части этого раздела руководства перечислены расширенные типы вместе со спецификаторами формата и макросами, представляющими предельные значения этих типов.

Типы строгой ширины

Один набор `typedef` идентифицирует типы точного размера. Общая форма выглядит как `intN_t` для типов со знаком и `uintN_t` — для типов без знака, где `N` — означает количество разрядов. Помните, однако, что не все системы могут поддерживать все типы. Например, существуют системы, для которых минимальной ячейкой используемой памяти является 16 разрядов; такие системы не могут поддерживать типы `int8_t` и `uint8_t`. Макросы формата могут использовать либо `d`, либо `i` для типов со знаком, поэтому `PRIi8` и `SCNi8` работают. Для типов без знака вы можете подставлять `o`, `x` или `X` для `u`, чтобы получить спецификаторы `%o`, `%x`, или `%X` вместо `%u`. Например, вы можете использовать `PRIX32`, чтобы напечатать `uint32_t` в шестнадцатеричной форме. В табл. RS.VI.1 перечислены типы строгой ширины, спецификаторы формата и предельные значения.

Таблица RS.VI.1. Типы строгой ширины

Наименование типа	*Спецификатор printf()*	*Спецификатор scanf()*	*Минимальное значение*	*Максимальное значение*
int8_t	PRId8	SCNd8	INT8_MIN	INT8_MAX
int16_t	PRId16	SCNd16	INT16_MIN	INT16_MAX
int32_t	PRId32	SCNd32	INT32_MIN	INT32_MAX
int64_t	PRId64	SCNd64	INT64_MIN	INT64_MAX
uint8_t	PRIu8	SCNu8	0	UINT8_MAX
uint16_t	PRIu16	SCNu16	0	UINT16_MAX
uint32_t	PRIu32	SCNu32	0	UINT32_MAX
uint64_t	PRIu64	SCNu64	0	UINT64_MAX

Типы минимальной ширины

Типы минимальной ширины гарантируют минимальное количество разрядов размера. Эти типы существуют всегда. Например, система, которая не поддерживает 8-разрядные элементы, может определить `int_least_8` как 16-разрядный тип. В табл. RS.VI.2 перечислены типы минимальной ширины, спецификаторы формата и предельные значения.

Таблица RS.VI.2. Типы минимальной ширины

Наименование типа	*Спецификатор printf()*	*Спецификатор scanf()*	*Минимальное значение*	*Максимальное значение*
int_least8_t	PRILEASTd8	SCNLEASTd8	INT_LEAST8_MIN	INT_LEAST8_MAX
int_least16_t	PRILEASTd16	SCNLEASTd16	INT_LEAST16_MIN	INT_LEAST16_MAX
int_least32_t	PRILEASTd32	SCNLEASTd32	INT_LEAST32_MIN	INT_LEAST32_MAX
int_least64_t	PRILEASTd64	SCNLEASTd64	INT_LEAST64_MIN	INT_LEAST64_MAX
uint_least8_t	PRILEASTu8	SCNLEASTu8	0	UINT_LEAST8_MAX
uint_least16_t	PRILEASTu16	SCNLEASTu16	0	UINT_LEAST16_MAX
uint_least32_t	PRILEASTu32	SCNLEASTu32	0	UINT_LEAST32_MAX
uint_least64_t	PRILEASTu64	SCNLEASTu64	0	UINT_LEAST64_MAX

Быстрые типы минимальной ширины

Для определенной системы некоторые представления целых чисел могут быть быстрее, чем другие. Например, int_least16_t может быть реализован как short, однако в некоторых системах вычисления могут выполняться быстрее с типом int. Поэтому inttypes.h также определяет наиболее быстрые типы, представленные, как минимум, определенным количеством разрядов. Эти типы существуют всегда. В некоторых случаях выбор самого быстрого типа не столь однозначен; в таких случаях система специфицирует один из возможных типов. В табл. RS.VI.3 перечислены быстрые типы минимальной ширины, спецификаторы формата и предельные значения.

Таблица RS.VI.3. Быстрые типы минимальной ширины

Наименование типа	*Спецификатор printf()*	*Спецификатор scanf()*	*Минимальное значение*	*Максимальное значение*
int_fast8_t	PRIFASTd8	SCNFASTd8	INT_FAST8_MIN	INT_FAST8_MAX
int_fast16_t	PRIFASTd16	SCNFASTd16	INT_FAST16_MIN	INT_FAST16_MAX
int_fast32_t	PRIFASTd32	SCNFASTd32	INT_FAST32_MIN	INT_FAST32_MAX
int_fast64_t	PRIFASTd64	SCNFASTd64	INT_FAST64_MIN	INT_FAST64_MAX
uint_fast8_t	PRIFASTu8	SCNFASTu8	0	UINT_FAST8_MAX
uint_fast16_t	PRIFASTu16	SCNFASTu16	0	UINT_FAST16_MAX
uint_fast32_t	PRIFASTu32	SCNFASTu32	0	UINT_FAST32_MAX
uint_fast64_t	PRIFASTu64	SCNFASTu64	0	UINT_FAST64_MAX

Типы максимальной ширины

Иногда вам могут понадобиться целочисленные типы максимально доступного размера. В табл. RS.VI.4 перечислены эти типы. Фактически они могут быть шире, чем long long или unsigned long long, поскольку система может представлять дополнительные типы — более широкие, чем обязательные стандартные.

Таблица RS.VI.4. Типы максимальной ширины

Наименование типа	*Спецификатор printf()*	*Спецификатор scanf()*	*Минимальное значение*	*Максимальное значение*
intmax_t	PRIdMAX	SCNdMAX	INTMAX_MIN	INTMAX_MAX
uintmax_t	PRIuMAX	SCNuMAX	0	UINTMAX_MAX

Целые, которые могут хранить указатели

Заголовочный файл inttypes.h (через включенный в него stdint.h) определяет два целочисленных типа, представленные в табл. RS.VI.5, которые могут корректно хранить указатели. То есть, если вы присвоите значение типа void * переменной одного из этих типов, а затем присвоите ее значение обратно указателю, то никакая информация не будет потеряна. Любой из этих типов, или все сразу, могут в системе отсутствовать.

Таблица RS.VI.5. Целочисленные типы, способные хранить указатели

Наименование типа	*Спецификатор printf()*	*Спецификатор scanf()*	*Минимальное значение*	*Максимальное значение*
intptr_t	PRIdPTR	SCNdPTR	INTPTR_MIN	INTPTR_MAX
uintptr_t	PRIuPTR	SCNuPTR	0	UINTPTR_MAX

Расширенные целочисленные константы

Вы можете обозначать длинные целые константы суффиксом L, например, 445566L. Как обозначить, что константа имеет тип int32_t? Для этого нужно воспользоваться макросом, определенным в inttypes.h. Например, выражение INT32_C(445566) расширяется до типа int32_t. По сути дела, этот макрос — просто приведение типа, то есть в данном случае — приведение к фундаментальному типу конкретной реализации int32_t.

Имена макросов формируются из имени типа, в котором _t заменено на _C и использован верхний регистр. Например, для объявления 1000 как константы типа uint_least64_t нужно использовать выражение UINT_LEAST64_C(1000).

Раздел VII. Расширенная поддержка символов

Изначально C не разрабатывался как интернациональный язык программирования. Его набор символов основан на более или менее стандартной клавиатуре, принятой в США. Однако всемирная популярность C привела к появлению некоторых расширений, поддерживающих различные более обширные наборы символов. В этом разделе руководства представлен обзор этих дополнений.

Триграфы

Некоторые клавиатуры не содержат всех символов, используемых в C. Для этих целей в C предусмотрено альтернативное представление некоторых символов в виде набора трехсимвольных последовательностей, называемых *триграф-последовательностями*, или просто *триграфами*. В табл. RS.VII.1 перечислены эти триграфы.

Таблица RS.VII.1. Триграфы

Триграф	*Символ*	*Триграф*	*Символ*	*Триграф*	*Символ*
??=	#	??(	[	??/	\
??)	]	??'	^	??<	{
??!	|	??>	}	??-	~

C заменяет все вхождения триграфов в файле исходного кода, даже внутри строк в кавычках, на соответствующие символы. То есть

```
??=include <stdio.h>
??=define LIM 100
int main()
??<
int q??(LIM??);
printf("Новости будут скоро.??/n");
...
??>
```

превращается в следующее:

```
#include <stdio.h>
#define LIM 100
int main()
{
   int q[LIM];
   printf("Новости будут скоро.\n");
   ...
}
```

Возможно, для активизации этого средства вам придется включить специальный флаг компилятора.

Диграфы

Учитывая громоздкость системы триграфов, стандарт C99 предусматривает также набор двухсимвольных комбинаций, называемых *диграфами*, которые могут использоваться вместо определенных знаков препинания C. В табл. RS.VII.2 перечислены эти диграфы.

Таблица RS.VII.2. Диграфы

Диграф	*Символ*	*Диграф*	*Символ*	*Диграф*	*Символ*
<:	[	:>	]	<%	{
%>	}	%:	#	%:%:	##

В отличие от триграфов, диграфы внутри строк в кавычках не имеют специального значения. То есть

```
%:include <stdio.h>
%:define LIM 100
int main()
<%
int q<:LIM:>;
printf("Новости будут скоро.:>");
...
%>
```

Ведет себя так же, как и следующий фрагмент:

```
#include <stdio.h>
#define LIM 100
int main()
{
   int q[LIM];
   printf("Новости будут скоро.:>"); // :> -- просто часть строки
   ...
} // :> -- то же, что и }
```

Альтернативная орфография: `iso646.h`

С помощью триграф-последовательностей вы можете записать операцию `||` как `??!??!`, что не выглядит особенно изящно. Стандарт C99 посредством заголовочного файла `iso646.h` представляет макросы, расширяемые в операторы, как показано в табл. RS.VII.3. В рамках стандарта эти макросы называются *альтернативной орфографией*.

Таблица RS.VII.3. Альтернативная орфография

Макрос	*Операция*	*Макрос*	*Операция*	*Макрос*	*Операция*
`and`	`&&`	`and_eq`	`&=`	`bitand`	`&`
`bitor`	`|`	`compl`	`~`	`not`	`!`
`not_eq`	`!=`	`or`	`||`	`or_eq`	`|=`
`xor`	`^`	`xor_eq`	`^=`		

Если вы включите заголовочный файл `iso646.h`, то операция вроде

```
if(x == M1 or x == M2)
   x and_eq 0XFF;
```

расширяется в

```
if(x == M1 || x == M2)
   x &= 0XFF;
```

Многобайтные символы

Стандарт описывает многобайтный символ как последовательность одного или более байт, представляющих элемент расширенного символьного набора, — как в исходном, так и в среде выполнения. Исходная среда — это та, где вы подготавливаете код;

среда выполнения — это та, в которой вы запускаете скомпилированную программу. Они могут различаться. Например, вы можете разрабатывать программу в одной среде с намерением запускать в другой. Расширенный набор символов — это надмножество базового набора символов, определенного в C.

Реализация может представлять расширенный набор символов, который позволяет, например, вводить клавиатурные символы, не входящие в базовый набор. Это может быть использовано в строковых литералах и символьных константах, а также может появляться в файлах. Реализации также могут представлять многобайтные эквиваленты символов из базового набора, которые могут применяться вместо триграфов или диграфов.

Немецкая реализация, например, может разрешить присутствие в строках символов с умляутами:

```
puts("eins zwei drei vier fünf");
```

Универсальные имена символов (UCN)

Многобайтные символы могут применяться в строках, но не могут в идентификаторах. Универсальные имена символов (Universal Character Names — UCN) — дополнение C99, позволяющее использовать символы из расширенного набора в качестве части имен идентификаторов. Система расширяет концепцию управляющих последовательностей для обеспечения возможности кодирования символов из стандарта ISO/IEC 10646. Этот стандарт разработан совместно Международной организацией по стандартизации (ISO) и Международной электротехнической комиссией (IEC), и представляет числовые коды для огромного списка символов.

Существуют две формы UIC-последовательностей. Первая — `\u hexquad`, где `hexquad` — последовательность четырех шестнадцатеричных цифр; например, `\u00F6`. Вторая — `\U hexquad hexquad`; например, `\U0000AC01`. Поскольку каждая шестнадцатеричная цифра соответствует четырем битам, форма `\u` может использоваться для кодов, представимых в 16-разрядных целых, а `\U` — для представления 32-разрядных целых.

Если в вашей системе реализованы UCN и она включает необходимые символы в расширенный символьный набор, то UCN могут применяться в строках, символьных константах и идентификаторах:

```
wchar_t value\u00F6\u00F8 = '\u00f6';
```

Расширенные символы

C99 посредством библиотек `wchar.h` и `wctype.h` представляет еще одну поддержку крупных символьных наборов — через расширенные символы. Эти заголовочные файлы определяют `wchar_t` как целочисленный тип; точное его определение зависит от реализации. Этот тип предназначен для хранения символов из расширенного символьного набора, который является надмножеством базового набора символов. По определению, тип `char` достаточен для работы с базовым набором символов. Типу `wchar_t` может потребоваться больше разрядов для обработки более широкого диапазона значений кодов. Например, `char` может быть 8-разрядным байтом, а `wchar_t` — 16-разрядным `unsigned short`.

Константы расширенных символов и строковые литералы обозначаются префиксом L, и вы можете пользоваться модификаторами %lc и %ls для отображения данных расширенных символов:

```
wchar_t wch = L'I';
wchar_t w_arr[20] = L"am wide! ";
printf("%lc %ls\n", wch, w_arr);
```

Если, к примеру, wchar_t реализован как 2-байтный элемент, то 1-байтный код 'I' должен быть сохранен в младшем байте wch. Символы, не входящие в стандартный набор, могут потребовать обоих байтов для размещения кода. Вы можете применять универсальные коды символов, например, для обозначения символов, значения кодов которых выходят за пределы диапазона char:

```
wchar_t w = L'\u00E2'; /* 16-разрядное значение кода */
```

Массив значений wchar_t может содержать строку расширенных символов, каждый элемент которой является отдельным кодом расширенного символа. Значение wchar_t со значением кода 0 — это wchar_t-эквивалент нулевого символа, называемый *расширенным нулевым символом*. Как можно догадаться, он служит ограничителем строк расширенных символов.

Для чтения расширенных символов можно использовать спецификаторы %lc и %ls:

```
wchar_t wch;
wchar_t w_arr[20];
puts("Введите свою научную степень:");
scanf("%lc", &wch);
puts("Введите свое имя и фамилию:");
scanff("%ls",w_arr);
```

Заголовочный файл wchar.h предоставляет и другую дополнительную поддержку широких символов. В частности, предусмотрены функции ввода-вывода расширенных символов, функции преобразования расширенных символов, функции манипуляции строками. Большей частью они являются эквивалентами расширенных символов для существующих функций стандартной библиотеки. Например, вы можете использовать fwprintf() и wprintf() для вывода, а fwscanf() и wscanf() — для ввода. Главное отличие этих функций состоит в том, что они требуют управляющих строк расширенных символов и работают с входными и выходными потоками расширенных символов. Например, следующий фрагмент отображает информацию в виде последовательность расширенных символов:

```
wchar_t * pw = L"Указывает на строку расширенных символов";
int dozen = 12;
wprintf(L"Элемент %d: %ls\n", dozen, pw);
```

Аналогично обстоят дела с функциями getwchar(), putwchar(), fgetws() и fputws(). Заголовок определяет макрос WEOF, который играет ту же роль, что EOF для байт-ориентированного ввода-вывода. Это требует наличия символа, который не соответствует ни одному нормальному символу из набора. Поскольку возможно, что все значения типа wchar_t являются допустимыми символами, библиотека определяет тип wint_t, который может допускать все возможные значения wchar_t плюс WEOF.

Предусмотрены эквиваленты функций из библиотеки string.h. Например, wcscpy(ws2, ws1) копирует строку расширенных символов, на которую указывает

ws1, в массив расширенных символов, на который указывает ws2. Аналогично, существует функция wcscmp() для сравнения двух широких строк и так далее.

Заголовочный файл wctype.h добавляет функции классификации символов. Например, iswdigit() возвращает true, если ее аргумент расширенных символов является десятичной цифрой, а iswblank() возвращает true, если ее аргумент — пробел. Стандартным значением пробела в расширенных символах является пробел, записанный как L' ', а знак горизонтальной табуляции записывается, как L'\t'.

Расширенные и многобайтные символы

Расширенные и многобайтные символы представляют собой два различных подхода для обработки расширенных символьных наборов. Многобайтный символ, например, может быть представлен одним, двумя, тремя и более байтами. Все расширенные символы имеют одинаковую ширину. Многобайтные символы могут использовать состояние сдвига (то есть байты, определяющие, как должны интерпретироваться последующие байты); расширенные символы не поддерживают состояния сдвига. Файл многобайтных символов может быть прочитан в обычный массив char с помощью стандартных функций ввода; файл расширенные символов должен быть прочитан в массив расширенных символов только посредством одной из специальных функций ввода расширенных символов.

C99 через библиотеку wchar.h предоставляет функции для преобразования между этими двумя представлениями. Функция mbrtowc() преобразует многобайтный символ в расширенный, а wcrtomb() — расширенный символ в многобайтный. Аналогично, функция mbstrtowcs() преобразует многобайтную строку в строку расширенных символов, а wcstrtombs() — строку расширенных символов в многобайтную.

Раздел VIII. Расширенные средства вычислений C99

Исторически сложилось так, что FORTRAN был первым языком, предназначенным для числовых научных и инженерных вычислений. Стандарт C90 привел методы вычислений C в более полное согласие с FORTRAN. Например, спецификации характеристик плавающей запятой, использованные в float.h, основаны на модели, разработанной комитетом по стандартизации FORTRAN. Стандарт C99 продолжает работу по расширению возможностей C для выполнения вычислительных работ.

Стандарт плавающей запятой IEC

Международная электротехническая комиссия (IEC) опубликовала стандарт вычислений с плавающей запятой (IEC 60559). Этот стандарт включает описание форматов плавающей запятой, точности, представления NaN, бесконечности, методик округления, преобразований, исключений, рекомендованных функций и алгоритмов и тому подобного. C99 принял этот стандарт в качестве руководства по реализации вычислений с плавающей запятой на языке C. Большая часть дополнений C99, касающихся средств работы с плавающей запятой, являются частью этих усилий. Речь идет о таких вещах, как заголовочный файл fenv.h и некоторые из новых математических функций.

Однако может случиться так, что конкретная реализация не отвечает требованиям IEC 60559; например, по причине того, что этого не позволяет установленное оборудование. Поэтому C99 определяет два макроса, которые могут использоваться в директивах препроцессора для проверки соответствия. Во-первых, макрос

```
__STDC_IEC_559__
```

условно определен как константа 1, если реализация отвечает спецификациям плавающей запятой IEC 60559. Во-вторых, макрос

```
__STDC_IEC_559_COMPLEX__
```

условно определен как константа 1, если реализация придерживается совместимой с IEC 60559 арифметики комплексных чисел.

Если в реализации эти макросы не определены, значит, нет никакой гарантии совместимости с IEC 60559.

Заголовочный файл `fenv.h`

Заголовочный файл `fenv.h` предоставляет средства взаимодействия со средой плавающей запятой. То есть он позволяет устанавливать *значения управляющего режима* плавающей запятой, которые определяют порядок выполнения вычислений с плавающей запятой, позволяют определять значения флагов состояния плавающей запятой, или *исключения*, которые сообщают информацию об эффектах от арифметических вычислений. Примером настроек управляющего режима может служить способ округления чисел. Примером флага состояния может быть флаг, устанавливаемый операциями, вызывающими переполнение плавающей запятой. Операция, устанавливающая флаг состояния, называется *возбуждением исключения*.

Флаги состояния и управляющие режимы имеют смысл, только если их поддерживает оборудование. Например, вы не можете изменить метод округления, если оборудование не позволяет этого делать.

Чтобы включить поддержку режимов и флагов, используется следующая директива препроцессора:

```
#pragma STDC FENV_ACCESS ON
```

Поддержка остается включенной до тех пор, пока программа не достигнет конца блока команд, содержащих указание компилятору, или, если указание компилятору является внешним — до конца файла единицы трансляции. Альтернативно вы можете использовать следующую директиву для отключения поддержки:

```
#pragma STDC FENV_ACCESS OFF
```

Можно также применить следующее указание компилятору:

```
#pragma STDC FENV_ACCESS DEFAULT
```

Оно восстановит состояние компилятора по умолчанию, которое зависит от реализации.

Данное средство важно для тех разработчиков, которые имеют дело с критичными вычислениями плавающей запятой, но представляет ограниченный интерес для большинства пользователей, поэтому в данном приложении мы не будем вдаваться в детали.

Указание компилятору STDC FP_CONTRACT

Некоторые арифметические процессоры с плавающей запятой могут объединять многооператорные выражения с плавающей запятой в единую операцию. Например, процессор может выполнить следующее выражение за один шаг:

```
x*y - z
```

Это повышает скорость вычислений, но может привести к снижению их предсказуемости. Указание компилятору STDC FP_CONTRACT позволяет включать и отключать это средство. Состояние по умолчанию зависит от реализации.

Чтобы отключить это средство для определенного вычисления, а затем включить его вновь, можете поступить следующим образом:

```
#pragma STDC FP_CONTRACT OFF
val = x * y - z;
#pragma STDC FP_CONTRACT ON
```

Дополнения к библиотеке math.h

Математическая библиотека C90 большей частью объявляет функции с аргументами double и типом возврата double, как показано ниже:

```
double sin(double);
double sqrt(double);
```

Помимо этого, библиотека C99 представляет версии всех этих функций для типов float и long double. Эти функции используют суффиксы f и l в своих именах:

```
float sinf(float);          /* float-версия sin() */
long double sinl(long double);   /* long double-версия sin() */
```

Наличие семейств функций с различными уровнями точности позволяет выбирать наиболее эффективную комбинацию типов и функций, подходящих в каждом конкретном случае.

В C99 также добавлен набор функций, часто используемых в научных, инженерных и математических вычислениях. Таблица RS.V.14, в которой перечислены double-версии всех математических функций, демонстрирует эти дополнения C99. Во многих случаях эти функции возвращают результаты, которые могут быть вычислены существующими функциями, однако новые функции делают это быстрее или с большей точностью. Например, log1p(x) представляет то же значение, что log(1 + x), но log1p(x) использует другой алгоритм, который дает более точный результат при малых значениях x. Поэтому вы должны использовать функцию log() для вычислений в большинстве случаев, а log1p(x) — в случаях с малыми значениями x, когда важна высокая точность.

В дополнение к этим функциям математическая библиотека определяет несколько констант и функций, связанных с классификацией чисел и их округлением. Например, значение может быть классифицировано как бесконечное, как NaN, нормальное, субнормальное и истинный ноль. (NaN — специальное значение, указывающее на то, что значение не является числом; например, asin(2.0) возвращает NaN, поскольку asin() может принимать аргументы от -1 до 1. Субнормальное число — это такое, абсолютная величина которого меньше, чем минимально допустимое значение, которое

можно выразить при максимальной точности.) Существуют также специализированные функции сравнения, которые ведут себя иначе, чем стандартные операции отношений, когда один или более аргументов являются ненормальными значениями.

Вы можете использовать схемы классификации C99 для обнаружения разного рода неправильностей в вычислениях. Например, макрос `isnormal()` из заголовочного файла `math.h` возвращает `true`, если его аргумент является нормальным числом. Ниже представлен пример кода, использующего эту функцию для прерывания цикла, когда число становится ненормальным:

```
#include <math.h> // для isnormal()
...
float num = 1.7e-19;
float numprev = num;
while (isnormal(num)) // пока num имеет полную точность float
{
   numprev = num;
   num /= 13.7f;
}
```

Короче говоря, теперь существует расширенная поддержка тонкого управления вычислений с плавающей запятой.

Поддержка комплексных чисел

Комплексное число — это число, состоящее из действительной и мнимой части. Действительная часть — обычное действительное число, которое может быть представлено типом с плавающей запятой. Мнимая часть представляет мнимое число. Мнимое число, в свою очередь, представляет собой величину, кратную корню квадратному из –1. В математике комплексные числа часто записываются в форме `4.2 + 2.0i`; где `i` символически представляет корень квадратный из –1.

В C99 поддерживаются три комплексных типа:

```
float _Complex
double _Complex
long double _Complex
```

Значение `float _Complex`, например, должно сохраняться с использованием того же способа организации памяти, что и двухэлементный массив `float`, где действительная часть сохраняется в первом элементе, а мнимая — во втором элементе.

Реализации C99 могут поддерживать следующие три мнимых типа:

```
float _Imaginary
double _Imaginary
long double _Imaginary
```

Включение заголовочного файла `complex.h` позволяет использовать имя `complex` вместо `_Complex` и `imaginary` вместо `_Imaginary`.

Арифметические операции, определенные для комплексных типов, следуют обычным правилам математики. Например, значение `(a+b*I)*(c+d*I)` равно `(a*c-b*d)+(b*c+a*d)*I`.

Заголовочный файл `complex.h` определяет некоторые макросы и несколько функций, принимающих аргументы в виде комплексных чисел и возвращающие результаты

в виде тех же комплексных чисел. В частности, макрос I представляет корень квадратный из –1. Это позволяет поступать следующим образом:

```
double complex c1 = 4.2 + 2.0 * I;
float imaginary c2= -3.0 * I;
```

Заголовочный файл complex.h содержит несколько прототипов комплексных функций. Многие из них являются комплексными эквивалентами функций math.h, но с префиксом c. Например, csin() возвращает комплексный синус комплексного аргумента. Другие функции касаются специфических свойств комплексных чисел. Например, creal() возвращает действительную часть комплексного числа, а cimag() — его мнимую часть в виде действительного числа. То есть для переменной z типа double complex истинно следующее:

```
z = creal(z) + cimag(z) * I;
```

Если вы знакомы с комплексными числами и нуждаетесь в их применении, вам стоит внимательно рассмотреть complex.h.

Если вы используете C++, вам следует иметь в виду, что заголовочный файл C++ complex предлагает другой способ работы с комплексными числами — на базе классов, что никак не связано со средствами complex.h языка C.

Раздел IX. Различия между C и C++

Большей частью C++ представляет собой надстройку над C — в том смысле, что корректные программы на C также являются корректными программами на C++. Главное отличие между C++ и C состоит в том, что C++ поддерживает множество дополнительных средств. Однако существует несколько областей, где правила C++ слегка отличаются от их эквивалентов в C. Эти отличия могут стать причиной того, что программа на C будет работать иначе, или даже вовсе не работать, если вы скомпилируете ее как программу C++. Именно эти различия и рассматриваются в этом разделе. Если вы компилируете свои программы на C компилятором, который поддерживает только C++, но не C, вам следует знать об этих различиях. Хотя они очень незначительно влияют на примеры, представленные ранее в книге, все же иногда они могут приводить к тому, что некоторые экземпляры корректного кода C вызывают сообщения об ошибках, если код компилируется как программа на C++.

Выпуск стандарта C99 усложняет ситуацию, поскольку в некоторых случаях он приближает C к C++. Например, он позволяет разносить объявления по телу кода и распознает вариант комментария //. В других отношениях C99 углубляет отличия от C++, например, добавляя массивы переменной длины и ключевое слово restrict. Хотя C99 все еще находится в зачаточном состоянии, разница между C90 и C99, C90 и C++ уже хорошо видна, равно как и разница между C99 и C++. Однако в конечном итоге C99 полностью заменит собой C90, поэтому в данном разделе мы обратимся к будущему и обсудим некоторые отличия между C99 и C++.

Прототипы функций

В C++ прототипирование функций обязательно, но таковым не является в C. Это различие проявляется, когда вы оставляете скобки пустыми в объявлении функции.

В C пустые скобки означают, что это предварительное прототипирование, а в C++ это значит, что функция не имеет параметров. То есть в C++ следующий прототип:

```
int slice();
```

означает то же самое, что и

```
int slice(void);
```

Например, следующая последовательность приемлема в устаревшем C, но считается ошибочной с C++:

```
int slice();
int main()
{
...
slice(20, 50);
...
}
int slice(int a, int b)
{
...
}
```

В C компилятор полагает, что вы используете старую форму объявления функций. В C++ компилятор полагает, что `slice()` — это то же самое, что `slice(void)`, но вы забыли объявить функцию `slice(int, int)`.

К тому же C++ позволяет объявлять более одной функции с одним и тем же именем, если их списки аргументов отличаются.

Константы char

Язык C рассматривает константы `char` как тип `int`, а C++ относит их к типу `char`. Например, рассмотрим следующий оператор:

```
char ch = 'A';
```

В C константа `'A'` сохраняется в области памяти размером в `int`; точнее говоря, код этого символа сохраняется в `int`. То же самое числовое значение также сохраняется в переменной `ch`, но здесь оно занимает только один байт памяти.

С другой стороны, язык C++ использует один байт для `'A'` так же, как и для `ch`. Это отличие не касается никаких примеров в тексте. Тем не менее, некоторые программы на C используют константы `char` типа `int`, применяя при этом символьную нотацию для представления целочисленных значений. Например, если система поддерживает 4-байтный `int`, вы можете написать на C следующее:

```
int x = 'ABCD'; /* нормально для C с 4-байтным int, но неверно для C++*/
```

Значение `'ABCD'` — это 4-байтный `int`, в котором первый байт сохраняет символьный код буквы `'A'`, второй — символьный код `'B'` и так далее. Обратите внимание, что `'ABCD'` — это не то же самое, что `"ABCD"`. Первое значение — просто забавный способ представления значения `int`, а второе — строка, которая соответствует адресу 5-байтного участка памяти.

Рассмотрим следующий код:

```
int x = 'ABCD';
char c = 'ABCD';
printf("%d %d %c %c\n", x, 'ABCD', c, 'ABCD');
```

В нашей системе получился такой вывод:

```
1094861636 1094861636 D D
```

Этот пример иллюстрирует, что если вы рассматриваете 'ABCD' как int, то это будет 4-байтное целое значение, но если вы трактуете его как char, то программа обращает внимание лишь на финальный байт. Попытка напечатать 'ABCD' со спецификатором формата %s в нашей системе приводит к аварийному завершению программы, поскольку числовое значение 'ABCD' (1094861636) соответствует некорректному адресу памяти.

Смысл использования значений, подобных 'ABCD', состоит в том, что так можно установить каждый байт в int независимо, поскольку каждый символ в точности соответствует одному байту. Однако более удачный подход, не зависящий от конкретных кодов символов, предусматривает использование шестнадцатеричных значений целочисленных констант, памятуя о том факте, что каждое двузначное шестнадцатеричное число описывает один байт. В главе 15 эта техника описана более подробно. (Ранние версии C не представляли шестнадцатеричной системы обозначения, что, вероятно, объясняет, почему сначала была разработана техника множественных символьных констант).

Модификатор const

В C глобальные идентификаторы const имеют внешнее связывание, однако в C++ они имеют внутреннее связывание. Другими словами, в C++ объявление

```
const double PI = 3.14159;
```

эквивалентно следующему объявлению в C:

```
static const double PI = 3.14159;
```

предполагая, что обе они находятся вне функций. Правила C++ нацелены на то, чтобы упростить применение const в файлах заголовков. Если константа имеет внутреннее связывание, каждый файл, включающий заголовок, получает свою собственную копию константы. Если же константа имеет внешнее связывание, то в одном файле должно присутствовать определяющее объявление, а все остальные должны использовать ссылочное объявление с ключевым словом extern.

Между прочим, C++ может использовать ключевое слово extern, чтобы обеспечить внешнее связывание константного значения, поэтому оба языка могут создавать константы как с внешним, так и с внутренним связыванием. Разница состоит в том, какое из них применяется по умолчанию.

Одно дополнительное свойство ключевого слова const в C++ связано с тем, что оно может быть использовано для объявления размера обычного массива:

```
const int ARSIZE = 100;
double loons[ARSIZE]; /* в C++ то же самое, что double loons[100]; */
```

Вы можете воспользоваться тем же объявлением и в условиях действия стандарта C99, но в этом случае создается массив переменной длины.

В C++, но не в C, вы можете использовать константные значения для инициализации других константных значений:

```
const double RATE = 0.06;          // правильно в C++ и C
const double STEP = 24.5;          // правильно в C++ и C
const double LEVEL = RATE * STEP; // правильно ц C++, неправильно в C
```

Структуры и объединения

После того, как вы объявили структуру или объединение, используя дескриптор, в C++ вы можете использовать его как имя типа:

```
struct duo
{
   int a;
   int b;
};
struct duo m; /* правильно в C++ и C */
duo n;        /* неправильно в C, правильно в C++ */
```

В результате имя структуры может конфликтовать с именем переменной. Например, следующая программа компилируется как программа на C, но не может быть успешно скомпилирована как программа на C++, поскольку C++ интерпретирует `duo` в операторе `printf()` как тип структуры, а не как внешнюю переменную:

```
#include <stdio.h>
float duo = 100.3;
int main(void)
{
   struct duo { int a; int b;};
   struct duo y = { 2, 4};
   printf ("%f\n", duo); /* правильно в C, неправильно в C++ */
   return 0;
}
```

На C и на C++ вы можете объявлять одну структуру внутри другой:

```
struct box
{
   struct point {int x; int y; } upperleft;
   struct point lowerright;
};
```

В C вы можете использовать обе структуры позже, однако C++ требует специальной формы записи для вложенной структуры:

```
struct box ad;     /* правильно в C++ и C              */
struct point dot; /* правильно в C, неправильно в C++ */
box::point dot;   /* неправильно в C, правильно в C++ */
```

Перечисления

Язык C++ более строго относится к использованию перечислений, нежели C. В частности, почти единственное, что можно делать с переменными `enum` — это присваивать им константы `enum` и сравнивать их с другими значениями. Вы не можете при-

сваивать значения int переменным enum без явного приведения типа, и вы не можете инкрементировать переменные enum. Следующий код иллюстрирует эти утверждения:

```
enum sample {sage, thyme, salt, pepper};
enum sample season;
season = sage;              /* правильно в C и C++              */
season = 2;                 /* предупреждение в C, ошибка в C++ */
season = (enum sample) 3;   /* правильно в C и C++              */
season++;                   /* правильно в C, ошибка в C++      */
```

К тому же C++ позволяет опустить слово enum при объявлении переменной:

```
enum sample {sage, thyme, salt, pepper};
sample season; /* неправильно в C, правильно в C++ */
```

Как и в случае со структурами и объединениями, это может привести к конфликтам, если переменная и тип enum имеют одинаковые имена.

Указатель на void

В языке C++, как и в C, можно присвоить указатель любого типа указателю на void, тем не менее, в отличие от C, вы не можете присваивать указатель на void указателю на другой тип, если только не используется явное приведение типа. Следующий код иллюстрирует это:

```
int ar[5] = {4, 5, 6,7, 8};
int * pi;
void * pv;
pv = ar;          /* правильно в C и C++ */
pi = pv;          /* правильно в C, неправильно в C++ */
pi = (int * ) pv; /* правильно в C и C++ */
```

Другое отличие C++ заключается в том, что вы можете присваивать адрес объекта класса-наследника указателю базового класса, однако это касается средства, отсутствующего в C.

Булевские типы

В C++ булевский тип называется bool, а true и false представляют собой ключевые слова. В языке C булевский тип имеет название _Bool, но только включение заголовочного файла stdbool.h делает доступными bool, true и false.

Альтернативная орфография

В C++ альтернативная орфография or для || и так далее обеспечивается ключевыми словами. В C99 ее обеспечивают макросы, и для того, чтобы сделать ее доступной, потребуется включить заголовочный файл iso646.h.

Поддержка расширенных символов

В C++ wchar_t — это встроенный целочисленный тип, а wchar_t — ключевое слово. В C99 тип wchar_t определен в нескольких заголовочных файлах (stddef.h, stdlib.h, wchar.h, wctype.h).

C++ обеспечивает поддержку ввода-вывода расширенных символов через заголовочный файл `iostream`, в то время как C99 предоставляет отдельный пакет поддержки ввода-вывода через заголовочный файл `wchar.h`. К тому же C99 поддерживает многобайтные символы и преобразования их в расширенные символы и обратно, тогда как C++ — нет.

Комплексные типы

Язык C++ поддерживает комплексные типы через класс `complex`, реализованный в заголовочном файле `complex`. Язык C имеет встроенные комплексные типы и поддерживает их через заголовочный файл `complex.h`. Эти два подхода совершенно различны, к тому же несовместимы друг с другом. Версия C отражает в большей степени практические нужды вычислительного сообщества.

Встраиваемые функции

Стандарт C99 добавил поддержку встраиваемых функций — средства, давно существующего в языке C++. Однако C99 реализует их более гибко. В C++ встроенные функции по умолчанию имеют внутреннее связывание. Если в C++ встроенная функция появляется более чем в одном файле, то она должна иметь одно и то же определение, используя одни и те же лексемы. Например, один файл не может иметь определение с параметром типа `int`, а другой — определение с параметром типа `int32_t`, даже несмотря на то, что `int32_t` — это `typedef` для `int`. В противоположность этому, C разрешает организацию подобного рода. К тому же язык C, как описано в главе 15, разрешает смешивать встроенные и внешние определения, что не разрешено в C++.

Средства C99, которых нет в C++

Хотя традиционно считается, что язык C в большей или меньшей степени является подмножеством C++, стандарт C99 добавляет некоторые средства, которые в C++ отсутствуют. Ниже перечислены некоторые из них:

- Выделенные инициализаторы.
- Составные инициализаторы.
- Ограниченные указатели.
- Массивы переменной длины.
- Гибкие элементы массивов.
- Типы `long long` и `unsigned long long`.
- Переносимые целочисленные типы (`inttypes.h` и `stdint.h`).
- Универсальные имена символов.
- Дополнительные функции математической библиотеки.
- Доступ к среде плавающей запятой через `fenv.h`.
- Предопределенные идентификаторы, такие как `__func__`.
- Макросы с переменным количеством аргументов.

Возможно, некоторые из этих средств, например, тип `long long`, станут частью расширений C++, а некоторые будут включены в следующую редакцию стандарта C++.

ПРИЛОЖЕНИЕ В

Набор символов ASCII

Символы сохраняются в памяти компьютеров с использованием числовых кодов. В США наиболее часто используется кодировка ASCII (American Standard Code for Information Interchange — американский стандартный код для обмена информацией). Язык C позволяет представить большинство одиночных символов напрямую, заключая символ в одинарные кавычки, например `'A'` для символа *A*. Кроме того, одиночный символ можно представить с использованием его восьмеричного или шестнадцатеричного кода, перед которым должен находиться обратный слеш, например, `'\012'` и `'\0xa'` соответствуют символу перевода строки (LF). Управляющие последовательности подобного рода также могут быть частью строки, скажем, такой: `"Добро пожаловать,\012уважаемый"`.

В представленной ниже таблице символ ^, используемый как префикс, обозначает клавишу <Ctrl>.

Десятичное представление	*Восьмеричное представление*	*Шестнадцатеричное представление*	*Двоичное представление*	*Символ*	*Имя в коде 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, табуляция	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

Десятичное представление	*Восьмеричное представление*	*Шестнадцатеричное представление*	*Двоичное представление*	*Символ*	*Имя в коде ASCII*
16	020	0x10	00010000	^P	DLE
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	

Десятичное представление	*Восьмеричное представление*	*Шестнадцатеричное представление*	*Двоичное представление*	*Символ*	*Имя в коде ASCII*
54	066	0x36	00110110	6	
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	[	

Десятичное представление	*Восьмеричное представление*	*Шестнадцатеричное представление*	*Двоичное представление*	*Символ*	*Имя в коде ASCII*
92	0134	0x5c	01011100	\	
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, стирание	

Предметный указатель

E

F

H

I

L

M

N

O

P

R

S

T

U

V

W

А

В

Д

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ц

Я

Научно-популярное издание

Стивен Прата

Язык программирования C
Лекции и упражнения, 5-е издание

Верстка *Т.Н. Артеменко*
Художественный редактор *В.Г. Павлютин*

ООО "И. Д. Вильямс"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 02.10.2012. Формат 70×100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 77,4. Уч.-изд. л. 51,25
Тираж 1000 экз. Заказ №

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9-я линия, 12/28