ВТОРОЕ ИЗДАНИЕ

ЯЗЫК ПРОГРАММИРОВАНИЯ

БРАЙАН КЕРНИГАН
ДЕННИС РИТЧИ

Серия книг по программированию от Prentice Hall

ВИЛЬЯМС

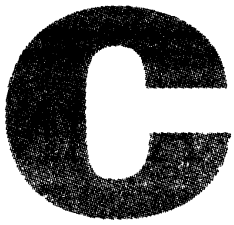
PH PTR

ЯЗЫК ПРОГРАММИРОВАНИЯ

C

второе издание

THE



PROGRAMMING LANGUAGE

Second Edition

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

AT & T Bell Laboratories
Murray Hill, New Jersey

Prentice Hall PTR, Upper Saddle River, New Jersey 07458

ЯЗЫК ПРОГРАММИРОВАНИЯ

C

второе издание,
переработанное и дополненное

БРАЙАН КЕРНИГАН
ДЕННИС РИТЧИ

Издательский дом "Вильямс"
Москва • Санкт-Петербург • Киев
2009

ББК 32.973.26-018.2.75
К36
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *В.Л. Бродового*

По общим вопросам обращайтесь в Издательский дом "Вильямс"
по адресу: info@williamspublishing.com, http://www.williamspublishing.com

Керниган, Брайан У., **Ритчи**, Деннис М.

К36 Язык программирования С, 2-е издание. : Пер. с англ. — М. : Издательский дом "Вильямс", 2009. — 304 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459- 0891-9 (рус.)

Классическая книга по языку С, написанная самими разработчиками этого языка и выдержавшая в США уже 34 переиздания! Книга является как практически исчерпывающим справочником, так и учебным пособием по самому распространенному языку программирования. Предлагаемое второе издание книги было существенно переработано по сравнению с первым в связи с появлением стандарта ANSI С, для которого она частично послужила основой. Книга не рекомендуется для чтения новичкам; для своего изучения она требует знания основ программирования и вычислительной техники.

Книга предназначена для широкого круга программистов и компьютерных специалистов. Может использоваться как учебное пособие для вузов.

ББК 32.973.26-018.2.75

ISBN 978-5-8459- 0891-9 (рус.)
ISBN 0-13-110362-8 (англ.)

Оглавление

Содержание

Предисловие

С момента выхода книги *Язык программирования С* в 1978 году в мире вычислительных технологий произошли революционные изменения. Большие компьютеры стали еще больше, а персональные приобрели такую мощь, что могут посоревноваться с суперкомпьютерами предыдущего десятилетия. За это время язык С также изменился, хотя и в небольшой степени. А главное — он вышел далеко за пределы своего первоначального узкого применения в виде языка операционной системы Unix.

Постоянно возрастающая популярность С, накопившиеся за годы изменения в языке, разработка компиляторов независимыми группами, которые не участвовали в его создании, — все это вместе показало, что назрела необходимость дать более точное и современное описание языка, чем то, которое содержалось в первом издании этой книги. В 1983 году Американский национальный институт стандартов (ANSI — *American National Standards Institute*) создал комитет, целью которого была разработка "четко сформулированного и системно-независимого определения языка С" при сохранении самого духа и стиля этого языка. Результатом стало появление стандарта ANSI языка С.

В этом стандарте даны формальные определения конструкций, которые лишь очерчивались, но не описывались строго в первом издании книги. В частности, это присваивание структур и перечислимые типы. В стандарте задается новая форма объявления функций, которая позволяет выполнить перекрестную проверку правильности объявления и фактического вызова. В нем определяется состав стандартной библиотеки функций, содержащей обширный набор средств для ввода-вывода, управления памятью, операций со строками и т.п. Стандарт дает точное определение тех средств, для которых в первоначальном описании языка такого определения дано не было, и в то же время указывает, какие аспекты языка остались системно-зависимыми.

Во втором издании книги *Язык программирования С* описывается язык С в том виде, в каком его определяет стандарт ANSI. Хотя мы и отмечаем те места языка, которые эволюционировали со временем, но все же мы решили писать почти исключительно о современном состоянии С. Как правило, это не очень существенно; наиболее заметное изменение — это новая форма объявления и определения функций. Современные компиляторы уже поддерживают большую часть нового стандарта.

Мы постарались сохранить краткость первого издания. Язык С невелик по объему, и нет большого смысла писать о нем толстые книги. Мы улучшили изложение ключевых вопросов — таких как указатели, являющиеся центральным моментом в программировании на С. Мы доработали первоначальные примеры, а также добавили новые в некоторые из глав. Например, рассказ о сложных объявлениях дополнен программой преобразования деклараций в текстовые описания и наоборот. Как и раньше, все примеры протестированы и отлажены непосредственно из текста, который подготовлен в электронном виде.

Приложение А является справочником по языку, но не описанием стандарта, хотя мы и попытались изложить все основные элементы стандарта в сжатом виде. Справочник задумывался как легкодоступное пособие для программистов, но отнюдь не как строгое определение языка для разработчиков компиляторов — эта роль по праву принадлежит собственно стандарту. Приложение Б содержит краткое описание средств стандартной библиотеки. Оно также задумывалось как справочник для программистов, но не для раз-

работчиков компиляторов. Приложение B содержит список изменений, внесенных в язык C со времени первого издания книги.

Как говорилось в предисловии к первому изданию, язык C "становится все удобнее по мере того, как приобретается опыт работы с ним". После нескольких десятилетий работы мы не изменили своего мнения. Надеемся, что эта книга поможет вам изучить C и пользоваться им как можно эффективнее.

Мы глубоко признательны нашим друзьям, без которых второе издание книги не вышло бы в свет. Джон Бентли (Jon Bentley), Дуг Гвин (Doug Gwyn), Дуг Макилрой (Doug McIlroy), Питер Нельсон (Peter Nelson) и Роб Пайк (Rob Pike) высказали вдумчивые замечания почти по каждой странице исходного текста. Мы также благодарны за тщательное прочтение рукописи таким людям, как Эл Ахо (Al Aho), Деннис Эллисон (Dennis Allison), Джо Кэмпбелл (Joe Campbell), Дж. Р. Эмлин (G. R. Emlin), Карен Фортганг (Karen Fortgang), Аллен Холуб (Allen Holub), Эндрю Хьюм (Andrew Hume), Дэйв Кристол (Dave Kristol), Джон Линдерман (John Linderman), Дэйв Проссер (Dave Prosser), Джин Спаффорд (Gene Spafford) и Крис Ван Вик (Chrys Van Wyk). Ряд ценных предложений выдвинули Билл Чезвик (Bill Cheswick), Марк Керниган (Mark Kernighan), Энди Кёниг (Andy Koenig), Робин Лэйк (Robin Lake), Том Ланден (Tom London), Джим Ридз (Jim Reeds), Кловис Тондо (Clovis Tondo) и Питер Вайнбергер (Peter Weinberger). Дэйв Проссер (Dave Prosser) самым подробным образом ответил на множество вопросов, касающихся стандарта ANSI. Для текущей компиляции и тестирования наших программ мы широко использовали транслятор C++ Бьерна Страуструпа (Bjarne Stroustrup), а для окончательного тестирования Дэйв Кристол (Dave Kristol) предоставил компилятор ANSI C. Большую помощь в типографском оформлении текста оказал Рич Дрекслер (Rich Drechsler).

Выражаем всем нашу искреннюю благодарность.

Брайан В. Керниган
Деннис М. Ритчи

Предисловие к первому изданию

С представляет собой язык программирования общего назначения, характеризующийся краткостью выражений, современными управляющими конструкциями и структурами данных, а также богатым набором операций. С — это язык не слишком высокого уровня, не слишком объемный и не приспособленный специально к какой-либо конкретной области приложений. Но зато отсутствие в нем каких-либо ограничений и большая общность делают его более удобным и эффективным для решения многих задач, чем языки, даже считающиеся по разным причинам более мощными.

Первоначально язык С был разработан для операционной системы Unix на ЭВМ DEC PDP-11 и реализован в этой системе Деннисом Ритчи (Dennis Ritchie). Операционная система, компилятор С и практически все основные прикладные программы для Unix (в том числе те, с помощью которых готовилась эта книга) написаны на С. Существуют профессиональные компиляторы и для других систем, в частности IBM System/370, Honeywell 6000 и Interdata 8/32. Тем не менее язык С не привязан к какой-либо конкретной аппаратной или системной платформе, и на нем легко писать программы, которые будут выполняться безо всяких изменений в любой системе, поддерживающей этот язык.

Эта книга предназначена для того, чтобы помочь читателю научиться программировать на С. Первая глава книги называется "Вводный урок", благодаря которой читатель может сразу взяться за дело, а затем следуют главы с подробным изложением всех основных средств языка, завершает книгу справочное руководство. Изложение большей частью построено на чтении, написании и исправлении примеров программ, а не на простом описании правил и конструкций. В основном рассматриваются примеры законченных и реально работающих программ, а не отдельные искусственные фрагменты. Все примеры были протестированы непосредственно из исходного текста, который подготовлен в электронном виде. Кроме демонстрации эффективного применения языка, мы постарались по возможности дать понятие о ряде полезных алгоритмов и принципов хорошего стиля и разумного написания программ.

Эта книга не представляет собой элементарное введение в программирование. Предполагается, что читатель знаком с основными понятиями программирования — такими как переменные, операторы присваивания, циклы и функции. Тем не менее даже начинающий программист скорее всего окажется в состоянии читать примеры и извлекать из них уроки языка, хотя помощь более знающего коллеги будет не лишней.

Судя по нашему опыту, язык С оказался легким в применении, выразительным и достаточно универсальным для целого ряда программных приложений. Он прост в изучении и по мере накопления опыта работы с ним становится все удобнее. Мы надеемся, что эта книга поможет вам эффективно работать с языком С.

Вдумчивая критика и предложения многих наших друзей и коллег очень помогли нам в написании этой книги и сделали процесс работы над ней очень приятным. В частности, Майк Бьянки (Mike Bianchi), Джим Блю (Jim Blue), Стью Фельдман (Stu Feldman), Дуг Макилрой (Doug McIlroy), Билл Рум (Bill Roome), Боб Розин (Bob Rosin) и Ларри Рослер (Larry Rosler) прочитали несколько версий книги самым тщательным образом. Мы глу-

боко признательны Элу Ахо (Al Aho), Стиву Борну (Steve Bourne), Дэну Дворак (Dan Dvorak), Чаку Хэйли (Chuck Haley), Дебби Хэйли (Debbie Haley), Марион Харрис (Marion Harris), Рику Холту (Rick Holt), Стиву Джонсону (Steve Johnson), Джону Мэши (John Mashey), Бобу Митцу (Bob Mitze), Ральфу Мухе (Ralph Muha), Питеру Нельсону (Peter Nelson), Эллиоту Пинсону (Elliot Pinson), Биллу Плоджеру (Bill Plauger), Джерри Спиваку (Jerry Spivack), Кену Томпсону (Ken Thompson) и Питеру Вайнбергеру (Peter Weinberger) за полезные замечания на разных этапах подготовки книги, а также Майку Леску (Mike Lesk) и Джо Оссанне (Joe Ossanna) за неоценимую помощь при макетировании.

Брайан В. Керниган
Деннис М. Ричи

Ждем ваших отзывов!

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 вам эта книга или нет, 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`

Адреса для писем из:

России:	127055, Москва, ул. Лесная, д. 43, стр. 1
Украины:	03150, Киев, а/я 152

Издательство выражает признательность Крупенько Никите Анатольевичу за присланные замечания и пожелания.

Введение

C является языком программирования общего назначения. Он всегда находился в самой тесной связи с операционной системой Unix, в которой и для которой он и разрабатывался, поскольку как сама система, так и большинство работающих в ней программ написаны именно на C. Тем не менее сам язык не привязан жестко к одной определенной системе или аппаратной платформе. И хотя C называют "языком системного программирования", поскольку на нем удобно писать компиляторы и операционные системы, он столь же удобен и для написания больших прикладных программ в самых разных областях применения.

Многие ключевые идеи C пришли из языка BCPL, разработанного Мартином Ричардсом (Martin Richards). BCPL оказал влияние на C опосредованно — через язык B, разработанный Кеном Томпсоном (Ken Thompson) в 1970 году для первой системы Unix на ЭВМ DEC PDP-7.

В языках BCPL и B отсутствует типизация данных. В отличие от них язык C предлагает широкий ассортимент типов. Фундаментальные типы включают в себя символьный, а также целый и вещественный (с плавающей точкой) нескольких различных размеров. Кроме того, существует целая иерархия производных типов данных, создаваемых с помощью указателей, массивов, структур и объединений. Из операндов и знаков операций формируются выражения; любое выражение, в том числе присваивание и вызов функции, может являться оператором. Благодаря указателям поддерживается механизм системно-независимой адресной арифметики.

В языке C имеются все основные управляющие конструкции, необходимые для написания хорошо структурированной программы: группировка операторов в блоки, принятие решения по условию (`if-else`), выбор одного из нескольких возможных вариантов (`switch`), циклы с проверкой условия завершения в начале (`while`, `for`) и в конце (`do`), а также принудительный выход из цикла (`break`).

Функции могут возвращать значения простых типов, структуры, объединения или указатели. Любую функцию можно вызывать рекурсивно. Локальные переменные функции обычно являются "автоматическими", т.е. создаются при каждом ее вызове. Определения (тела) функций нельзя вкладывать друг в друга, однако переменные можно объявлять в блочно-структурированном стиле. Функции программы на C могут находиться в отдельных файлах исходного кода, компилируемых также отдельно. Переменные могут быть внутренними по отношению к функции, внешними и при этом видимыми только в одном файле кода или же видимыми во всем пространстве программы.

На этапе препроцессорной обработки в тексте программы выполняются макроподстановки, включение дополнительных файлов исходного кода и условная компиляция.

C — это язык сравнительно "низкого" уровня. Ничего уничижительного в этом определении нет; это всего лишь значит, что язык C работает с теми же объектами, что и большинство компьютерных систем, а именно с символами, числами и адресами. Эти данные можно комбинировать разными способами с помощью арифметических и логических операций, которые реализованы реальными аппаратными и системными средствами.

В языке C нет операций для прямого манипулирования составными объектами, например строками символов, множествами, списками или массивами. Не существует операций для непосредственной работы с целым массивом строк, хотя структуру можно

скопировать как единое целое. Язык не содержит специальных средств распределения памяти — только статическое определение и стек, в котором хранятся локальные переменные функций; не предусмотрена ни системная куча (*heap*), ни сборка мусора (*garbage collection*). Наконец, в самом языке нет и средств ввода-вывода наподобие операторов `READ` или `WRITE`, а также отсутствуют встроенные механизмы обращения к файлам. Все эти операции высокого системного уровня выполняются путем явного вызова функций. В большинстве реализаций языка С имеется более или менее стандартный набор таких функций.

Аналогично, в С имеются только несложные, однопоточные управляющие структуры: проверки условий, циклы, блоки и подпрограммы, однако отсутствует мультипрограммирование, распараллеливание операций, синхронизация и сопрограммы.

Хотя отсутствие некоторых из перечисленных средств может показаться очень серьезным недостатком ("То есть я должен вызывать функцию, чтобы просто сравнить две строки символов?!"), тем не менее в том, чтобы сохранять набор базовых конструкций языка относительно небольшим, есть и свои преимущества. Поскольку С невелик, его описание занимает немного места, а изучение отнимает немного времени. Каждый программист вполне может знать, понимать и регулярно использовать практически всю базу языка.

Много лет определением языка служил справочник по нему, включенный в первое издание книги *Язык программирования С*. В 1983 году Американский национальный институт стандартов (ANSI) основал комитет для создания полного и современного определения языка С. В результате к концу 1988 года было завершено создание стандарта ANSI С. Большинство средств и возможностей этого стандарта поддерживается современными компиляторами.

Стандарт основан на первоначальном справочнике по языку. Сам язык изменился относительно мало; одной из целей его стандартизации было обеспечить совместимость с большинством уже написанных программ или по крайней мере добиться от компиляторов корректных предупреждений, если там будут встречаться новые средства.

Для большинства программистов самым важным новшеством оказался новый синтаксис объявления и определения функций. Объявление функции теперь может содержать описание ее аргументов; синтаксис определения должен соответствовать объявлению. Дополнительная информация сильно облегчает компиляторам работу по выявлению ошибок, причиной которых стала несогласованность типов аргументов. Судя по нашему опыту, это очень полезное дополнение к языку.

Есть и другие небольшие изменения. Присваивание структур и перечислимые типы, которые давно уже встречались в разных реализациях, официально стали частью языка. Операции с плавающей точкой теперь можно выполнять с одинарной точностью. Более четко определены свойства арифметических операций, особенно над переменными без знака. Препроцессор стал более сложным и развитым. Однако бо́льшая часть изменений не очень повлияет на работу программистов.

Второе существенное новшество, связанное с введением стандарта, — это определение стандартной библиотеки функций, включаемой в состав компилятора С. В эту библиотеку входят функции для обращения к операционной системе (например, для чтения и записи файлов), для форматированного ввода-вывода, распределения памяти, операций со строками и т.п. Совокупность стандартных заголовочных файлов обеспечивает единообразие обращения к объявлениям функций и типов данных. Программы, которые пользуются этой библиотекой для взаимодействия с операционной системой, являются

гарантированно совместимыми с разными системными средами. Большая часть библиотеки построена по принципам стандартной библиотеки ввода-вывода системы Unix. Эта библиотека описывалась еще в первом издании и с тех пор широко использовалась во многих других системах. Большинство программистов и здесь не столкнутся с существенными изменениями.

Поскольку типы данных и управляющие структуры языка С поддерживаются многими компьютерами непосредственно на аппаратном уровне, библиотека функций, необходимых для реализации автономных, полнофункциональных программ, имеет совсем крошечные размеры. Функции из стандартной библиотеки всегда вызываются только явным образом, так что их применения легко избежать, если это необходимо. Большинство функций написаны на С и полностью переносимы, за исключением некоторых подробностей работы операционных систем, которые скрыты на более низком уровне.

Хотя язык С соответствует по своей структуре аппаратным возможностям многих компьютеров, он не привязан к какой-либо конкретной машинной архитектуре. При некотором усердии на нем легко написать полностью переносимую программу, т.е. программу, которая будет работать безо всяких изменений на самых разных компьютерах. В стандарте вопросы переносимости четко определены и выведены на явный уровень. Стандартом предписывается использование набора констант, характеризующего архитектуру системы, в которой работает программа.

Языку С не свойствен строгий контроль типов, хотя по мере его эволюции развивались средства проверки соответствия типов. В исходной версии С компилятор выдавал предупреждения, но не запрещал взаимную замену указателей и целых переменных разных типов. Но от этого уже давно отказались, и теперь стандарт требует полной совместимости объявлений и явного выполнения преобразований типов. Хорошие компиляторы уже давно требовали от программиста того же самого. Еще одним шагом в этом направлении как раз и является новый стиль объявления функций. Компиляторы выдают предупреждения по поводу большинства ошибок, связанных с несоответствием типов, и не выполняется никакого автоматического преобразования несовместимых типов данных. И все же в С сохраняется традиционное отношение к программисту, как к человеку, который сам должен знать и решать, что ему делать; от него просто требуется выражать свои намерения в явной и четкой форме.

Как и у любых языков, у С есть свои недостатки: некоторые операции имеют нелогичный приоритет; некоторые синтаксические конструкции можно было бы организовать и получше. И тем не менее язык С доказал свою высочайшую эффективность и выразительность для самого широкого круга приложений.

Эта книга организована следующим образом. Глава 1 представляет собой краткое учебное пособие по основным конструкциям С. Цель этой главы — быстро ввести читателя в курс дела. Мы убеждены, что лучший способ изучить новый язык — это сразу начать писать программы на нем. В этом учебном пособии предполагается, что читатель имеет некоторые знания по основам программирования, поэтому мы не разжевываем, что такое компьютер, зачем нужен компилятор или что означает выражение `n=n+1`. Хотя мы и старались везде, где это возможно, демонстрировать полезные приемы программирования, все-таки книга не является пособием по структурам данных и алгоритмам. Если необходимо было сделать подобный выбор, мы всегда сосредоточивались на вопросах, связанных с самим языком.

В главах 2–6 более подробно и строго, чем в главе 1, рассматриваются различные аспекты языка С, хотя главное место в изложении по-прежнему занимают примеры закон-

ченных программ, а не изолированные фрагменты кода. В главе 2 рассматриваются базовые типы данных, операции и выражения. В главе 3 обсуждаются управляющие конструкции: `if-else`, `switch`, `while`, `for` и т.д. Глава 4 посвящена функциям и структуре программы: внешним переменным, области видимости, распределению кода по исходным файлам и т.п. Также в ней рассматривается работа с препроцессором. В главе 5 описаны указатели и операции адресной арифметики. В главе 6 рассматриваются структуры и объединения.

Глава 7 содержит описание стандартной библиотеки функций, представляющей собой стандартный интерфейс для взаимодействия с операционной системой. Эта библиотека определена стандартом ANSI и должна поддерживаться во всех системах, в которых реализован язык С, так что программы, которые пользуются ее функциями для ввода, вывода и других операций на уровне операционной системы, должны обладать свойством переносимости на другие платформы безо всяких изменений.

В главе 8 описывается взаимодействие программ на С и операционной системы Unix. Основное внимание уделяется вводу-выводу, файловой системе и распределению памяти. Хотя частично эта глава содержит только информацию, специфическую для систем Unix, программисты, работающие с другими системами, все равно найдут здесь для себя много полезного. Среди прочего это взгляд изнутри на реализацию одной из версий стандартной библиотеки, а также рекомендации по переносимости программ.

Приложение А является полным справочником по языку. Официальным документом по синтаксису и семантике языка С является сам стандарт ANSI. Однако стандарт как справочное пособие в основном предназначается для разработчиков компиляторов, в то время как справочник в этой книге описывает язык более доходчиво, кратко и без перегруженности официальными выражениями. В приложении Б содержится описание стандартной библиотеки, опять-таки предназначенное скорее для программистов-пользователей, чем для разработчиков компиляторов. В приложение В включена сводка изменений, внесенных в язык по сравнению с его исходной версией. Однако в случае возникновения каких-либо сомнений и разночтений программисту всегда следует опираться на стандарт и имеющийся в наличии компилятор.

Глава 1

Вводный урок

Начнем изложение с быстрого введения в язык С. Нашей целью будет продемонстрировать основные элементы языка в действии — в реально работающих программах — и при этом не утонуть в подробностях, правилах и исключениях. На этом этапе мы не претендуем на полноту и даже на строгость (хотя все примеры, насколько мы можем судить, написаны без ошибок). Мы намерены помочь читателю как можно быстрее преодолеть тот барьер, после которого уже можно писать практически полезные программы, и поэтому наше внимание будет сосредоточено на фундаментальных понятиях: переменных и константах, математических операциях, управляющих конструкциях, функциях, а также основах ввода-вывода. Те средства С, которые необходимы для написания больших программ, в этой главе пока не рассматриваются. Это указатели, структуры, большая часть богатого набора операций, несколько управляющих операторов и стандартная библиотека функций.

В таком подходе есть и свои недостатки. Самый значительный из них — это то, что ни одно конкретное средство языка не описано здесь во всей полноте. Поэтому вводный урок из-за своей краткости может в чем-то и дезориентировать читателя. Поскольку в примерах не используются все возможности С, они написаны не столь кратко и изящно, как это можно было бы сделать. Мы пытались сделать так, чтобы подобных недостатков было как можно меньше. Тем не менее они есть, и мы об этом предупреждаем. Еще один недостаток состоит в том, что в следующих главах вынужденно повторяется часть материала этой главы. Надеемся, что это повторение окажется скорее полезным, чем надоедливым.

Как бы то ни было, опытным программистам должно быть нетрудно перейти к своим практическим потребностям, отталкиваясь от материала этой главы. Начинающим следует воспользоваться изученным, чтобы написать небольшие учебные программы, аналогичные приведенным. И в том и в другом случае читатель может пользоваться материалом этой главы как основой, на которую можно наложить более подробные сведения, начинающиеся с главы 2.

1.1. Первые шаги

Самый лучший способ изучить новый язык программирования — это сразу начать писать на нем программы. Программировать на любом языке начинают с такой программы:

Вывести слова

```
hello, world
```

Это не так мало, как кажется. Чтобы преодолеть этот этап, необходимо уметь подготовить текст программы, успешно скомпилировать его, загрузить и запустить на выполнение, а затем выяснить, куда и как был выведен результат ее работы.

На языке С программа для вывода слов `"hello, world"` выглядит так:

```
#include <stdio.h>

main()
{
    printf("hello, world");
}
```

Процесс выполнения этой программы зависит от системы, в которой вы работаете. Например, в операционной системе Unix необходимо поместить программу в файл, имя которого имеет окончание `.c`, например `hello.c`. Затем ее следует скомпилировать с помощью следующей команды:

```
cc hello.c
```

Если не было сделано никаких случайных ошибок (таких как пропуск буквы или описка в слове), то компиляция пройдет без сообщений, и в результате будет создан выполняемый файл `a.out`. Теперь этот файл можно запустить на выполнение командой

```
a.out
```

В результате его выполнения будет выведена строка

```
hello, world
```

В других системах эта процедура будет отличаться. Обратитесь к справочнику или специалисту за подробностями.

Теперь рассмотрим саму программу. Программа на C независимо от ее размера состоит из *функций* и *переменных*. Функция содержит *операторы* — команды для выполнения определенных вычислительных операций, а в переменных хранятся числа и другие данные, используемые в этих операциях. Функции C аналогичны подпрограммам и функциям языка Fortran или процедурам и функциям языка Pascal. В нашем примере функция имеет имя `main`. Обычно функциям можно давать любые имена по своему желанию, но `main` — особый случай, поскольку программа начинает выполняться именно с начала этой функции. Это означает, что в любой программе на C обязательно должна присутствовать функция `main`.

Обычно из `main` для выполнения разных операций вызываются другие функции, некоторые из которых программист пишет сам, а другие содержатся в стандартных библиотеках. В первой строчке программы выводится следующее:

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

Это указание компилятору включить в программу информацию о стандартной библиотеке ввода-вывода. Эта же строка имеется в начале многих файлов исходного кода C. Стандартная библиотека описывается в главе 7 и приложении Б.

Один из методов передачи данных между функциями заключается в том, что одна функция вызывает другую и предоставляет ей список чисел, именуемых *аргументами*. Список заключается в круглые скобки и ставится после имени функции. В нашем примере `main` является функцией без аргументов, на что указывает пустой список в скобках.

Первая программа на C

```
#include <stdio.h>          включение информации о стандартной
                            библиотеке
main()                      определение функции main без аргументов
{                           операторы main заключаются в скобки
```

```
    printf("hello, world\n");  библиотечная функция printf выводит
}                              строку;
                               \n обозначает конец строки
```

Операторы функции заключаются в фигурные скобки. Функция `main` в данном примере содержит всего один оператор:

```
printf("hello, world\n");
```

Чтобы вызвать функцию, следует ввести ее имя, а затем список аргументов в круглых скобках. Так что здесь вызывается функция `printf` с аргументом `"hello, world\n"`. Функция `printf` включена в стандартную библиотеку и выводит свои аргументы на экран или на печать. В данном случае это один аргумент в виде строки текста в кавычках.

Последовательность символов в двойных кавычках, такая как `"hello, world\n"`, называется *символьной строкой* или *строковой константой*. В данный момент единственным применением таких строк для нас будет их передача в функцию `printf` и аналогичные функции вывода.

Комбинация символов `\n` в символьной строке является условным обозначением языка С для *символа конца строки*, который переводит вывод данных на левый край новой строки. Если выбросить `\n` (поэкспериментируйте для наглядности), то окажется, что после вывода символов переход на новую строку не произойдет. Чтобы включить такой переход в аргумент `printf`, необходимо воспользоваться комбинацией `\n`. Если попытаться использовать приведенную ниже конструкцию, компилятор С выдаст сообщение об ошибке:

```
printf ("hello, world
");
```

Функция `printf` никогда не вставляет конец строки автоматически, поэтому одну строку вывода можно составить за несколько вызовов этой функции. Наша первая программа могла бы иметь и такой вид:

```
#include <stdio.h>

main()
{
    printf("hello, ");
    printf("world");
    printf("\n");
}
```

Результат ее работы был бы тот же.

Обратите внимание, что комбинация `\n` представляет один символ. Для обозначения всех символов, которые трудно или невозможно изобразить на письме, используются *управляющие последовательности* (*escape sequences*), аналогичные `\n`. Среди прочих в С используются комбинации `\t` для обозначения знака табуляции, `\b` для возврата на один символ назад с затиранием (*backspace*), `\"` для двойной кавычки, `\\` для обратной косой черты. В разделе 2.3 представлен полный список таких последовательностей.

Упражнение 1.1. Запустите программу `"hello, world"` на выполнение в вашей системе. Поэкспериментируйте с ней, выбрасывая определенные фрагменты программы и наблюдая за сообщениями об ошибках, которые будут появляться в связи с этим.

Упражнение 1.2. Попытайтесь выяснить экспериментально, что происходит при передаче в функцию `printf` строки, содержащей управляющую последовательность `\`*c*, где *c* — некий символ, не входящий в вышеперечисленные комбинации.

1.2. Переменные и арифметические выражения

В следующей программе выводится таблица температур по Фаренгейту и их соответствий по шкале Цельсия. Для этого используется формула $°C = (5/9)(°F - 32)$. Результат работы программы имеет вид

```
0    -17
20   -6
40   4
60   15
80   26
100  37
120  48
140  60
160  71
180  82
200  93
220  104
240  115
260  126
280  137
300  148
```

Сама программа, как и предыдущая, состоит из определения всего одной функции под именем `main`. Она несколько длиннее, чем та, которая выводила на экран слова `"hello, world"`, но ненамного сложнее. В ней появляется ряд новых конструкций, таких как комментарии, объявления, переменные, арифметические выражения, циклы и форматированный вывод.

```
#include <stdio.h>

/* вывод таблицы температур по Фаренгейту и Цельсию
     для fahr = 0, 20, ..., 300 */
main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;      /* нижняя граница температур в таблице */
    upper = 300;    /* верхняя граница */
    step = 20;      /* величина шага */

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
```

```
        fahr = fahr + step;
    }
}
```

Следующие две строки программы называются *комментарием*:

```
/* вывод таблицы температур по Фаренгейту и Цельсию
    для fahr = 0, 20, ..., 300 */
```

В данном случае комментарий содержит краткое описание того, что делает программа. Любые символы, заключенные между `/*` и `*/`, игнорируются компилятором, поэтому такую конструкцию можно использовать по своему усмотрению для придания программе удобочитаемости. Комментарий может стоять везде, где допускается наличие пустой строки, табуляции или пробела.

В языке С все переменные необходимо объявить до их использования. Обычно это делается в начале функции до любых выполняемых операторов. Свойства переменных описываются в *объявлении*, которое состоит из наименования типа и списка переменных, например:

```
int fahr, celsius;
int lower, upper, step;
```

Тип `int` означает, что перечисленные далее переменные являются целочисленными, в отличие от `float` — вещественных чисел с плавающей точкой, которые могут содержать дробную часть. Диапазон значений как типа `int`, так и типа `float` зависит от аппаратной и системной платформы. Общеприняты как 16-разрядные (16-битовые) целые числа типа `int` с диапазоном от –32768 до +32767, так и 32-разрядные. Число типа `float` обычно представляется 32-битовой переменной, содержащей как минимум 6 значащих цифр и имеющей диапазон значений от 10^{-38} до 10^{+38}.

Помимо `int` и `float`, в языке С имеется ряд других элементарных типов данных, таких как

`char` — символ (один байт);

`short` — короткое целое число;

`long` — длинное целое число;

`double` — вещественное число двойной точности.

Длины этих типов данных также зависят от используемой системы. К тому же существуют *массивы*, *структуры* и *объединения* элементарных типов, *указатели* на них, а также возвращающие их *функции*. Все это будет изучено в свое время.

Непосредственные вычисления в программе перевода температур начинаются с *операторов присваивания*:

```
lower = 0;
upper = 300;
step = 20;
fahr = lower;
```

Эти операторы задают переменным их начальные значения. Операторы отделяются друг от друга точками с запятой.

Каждая строка таблицы вычисляется одним и тем же образом, поэтому здесь используется цикл, который выполняется по одному разу на каждую строку результата. Для этого вводится цикл `while`:

```
while (fahr <= upper) {
    ...
}
```

Цикл while работает следующим образом. Проверяется условие в скобках. Если оно справедливо (т.е. переменная fahr меньше или равна upper), то выполняется тело цикла — три оператора, заключенные в фигурные скобки. Затем условие проверяется снова, и если оно по-прежнему справедливо, то тело цикла выполняется опять. Как только проверка дает отрицательный результат (fahr превосходит по величине upper), цикл заканчивается и работа программы продолжается с того оператора, который стоит сразу после цикла. Поскольку в программе больше операторов нет, она на этом завершается.

Тело цикла while может состоять из одного или нескольких операторов, заключенных в фигурные скобки, как это видно в программе перевода температур, или из одного оператора без скобок:

```
while (i < j)
    i = 2 * i;
```

В любом случае оператор или операторы, выполняемые в цикле while, выделяются отступом в виде табуляции (в данном случае он эквивалентен четырем пробелам), так что на глаз сразу видно, что именно входит в цикл. Отступы подчеркивают логическую структуру программы. Хотя компиляторам С безразличен внешний вид программы, расстановка отступов и пробелов очень важна для того, чтобы сделать программу удобочитаемой. Рекомендуется писать по одному оператору в строке, окружать их пробелами и выделять пустыми строками для подчеркивания группировки. Расположение фигурных скобок не так важно, хотя многие программисты лелеют привязанность к той или иной их расстановке. Мы выбрали один из нескольких популярных стилей. Рекомендуется придерживаться в этом вопросе единой системы: выберите один стиль раз и навсегда.

Большая часть работы выполняется в теле цикла. Следующий оператор вычисляет температуру по Цельсию и помещает ее в переменную celsius:

```
celsius = 5 * (fahr-32) / 9;
```

Причина того, что использовалось умножение на 5 и деление на 9 вместо простого умножения на 5/9, состоит вот в чем. В языке С, как и во многих других языках, при делении целых чисел *усекается* (отбрасывается) дробная часть результата. Поскольку 5 и 9 являются целыми числами, частное 5/9 было бы усечено до нуля и все температуры по Цельсию также оказались бы равными нулю.

В приведенном примере также продемонстрированы некоторые возможности функции printf. Это функция форматированного вывода самого широкого назначения. Подробнее о ней рассказывается в главе 7. Ее первый аргумент — это строка выводимых символов, причем каждый знак процента (%) обозначает место, куда при выводе следует подставить следующий (второй, третий и т.д.) аргумент функции, а также форму, в которой эти следующие аргументы следует выводить. Например, %d указывает на целочисленный аргумент, так что следующий оператор осуществляет вывод двух целых значений переменных fahr и celsius с символом табуляции (\t) между ними:

```
printf ("%d\t%d\n", fahr, celsius);
```

Каждая конструкция с символом % в первом строковом аргументе функции printf должна иметь соответствие: второй, третий и т.д. аргумент; их количество и типы должны быть согласованы, иначе будут выданы неверные ответы.

Кстати говоря, функция printf не является частью определения языка С. В самом языке не определены стандартные конструкции ввода-вывода, так что printf — это просто полезная функция из стандартной библиотеки, к которой обычно имеют возможность обращаться программы на С. Однако поведение и свойства функции printf рег-

ламентированы в стандарте ANSI, поэтому она должна работать одинаково во всех библиотеках и компиляторах, соответствующих этому стандарту.

Чтобы сосредоточить внимание на самом языке C, мы отложим рассмотрение вопросов ввода-вывода до главы 7. В частности, пока не будем заниматься форматированным вводом. Если понадобится вводить числа, ознакомьтесь с описанием работы функции `scanf` в разделе 7.4. Эта функция аналогична `printf`, но она считывает данные из потока ввода, вместо того чтобы отправлять их в поток вывода.

Программа преобразования температур содержит несколько недочетов. Простейший из них состоит в том, что результат выглядит не очень красиво, поскольку числа не выровнены по правому краю. Это легко исправить, добавив в каждый элемент `%` при вызове функции `printf` ширину поля вывода. В итоге каждое число будет выравниваться по правому краю отведенного ему пространства. Например, с помощью следующего оператора первое число в каждой строке будет выводиться в поле шириной три цифры, тогда как второе — в поле шириной шесть цифр:

```
printf("%3d %6d\n", fahr, celsius);
```

Результат получится таким:

```
  0   -17
 20    -6
 40     4
 60    15
 80    26
100    37
...
```

Более серьезная проблема состоит в том, что, поскольку используется целочисленная арифметика, температура по Цельсию вычисляется не очень точно. Например, 0°F соответствует –17,8°C, а не –17. Чтобы получить более точные ответы, следует прибегнуть к вещественной арифметике. Для этого в программу нужно внести некоторые изменения. Вот ее вторая версия:

```
#include <stdio.h>

/* вывод таблицы температур по Фаренгейту и Цельсию
     для fahr = 0, 20, ..., 300;
     версия с вещественной арифметикой */
main()
{
    float fahr, celsius;
    int lower, upper, step;

    lower = 0;      /* нижняя граница температур в таблице */
    upper = 300;    /* верхняя граница */
    step = 20;      /* величина шага */

    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Эта версия программы в значительной мере совпадает с предыдущей, только переменные `fahr` и `celsius` в ней объявлены вещественными, а формула преобразования записана более естественно. Ранее выражение `(5/9)` нельзя было использовать из-за его усечения до нуля при целочисленном делении. А вот наличие десятичной точки в записи константы означает, что это вещественное число, поэтому `5.0/9.0` не усекается, будучи частным от деления двух вещественных чисел.

Если знак математической операции соединяет два целочисленных операнда, то выполняется целочисленная операция. Однако если один из операндов вещественный, а другой — целый, то перед выполнением операции целое число будет преобразовано в вещественное. Если записать выражение `fahr-32`, константа `32` будет автоматически преобразована к вещественному представлению. Но для удобства чтения программы всегда лучше записать вещественную константу с десятичной точкой, чтобы подчеркнуть ее вещественную природу, даже если она имеет целое значение без дробной части.

Подробно о правилах, которым подчиняется преобразование целых чисел в вещественные, речь пойдет в главе 2. А пока следует знать вот что. Кроме арифметических операций, мы видели переменные еще в операторах присваивания:

```
fahr = lower;
```

Также они использовались в проверке, которая стоит в заголовке цикла:

```
while (fahr <= upper)
```

В обоих этих случаях применяется тот же принцип: значение типа `int` преобразуется во `float` перед непосредственным выполнением операции.

Спецификация формата из функции `printf`, а именно `%3.0f`, указывает, что следует вывести вещественное число (в данном случае `fahr`) в поле шириной не менее 3 символов без десятичной точки и дробной части. Спецификация `%6.1f` задает вывод еще одного числа (`celsius`), которое должно выводить в поле как минимум 6 символов шириной с одной цифрой после десятичной точки. Вот как будет выглядеть результат:

```
  0  -17.8
 20   -6.7
 40    4.4
...
```

Ширину и точность представления в спецификации можно по желанию опустить. Так, `%6f` задает вывод в поле шириной не менее 6 символов; `%.2f` требует вывода двух цифр после точки, но не ограничивает ширину поля; `%f` задает вывод десятичного вещественного числа без ограничений.

`%d`	—	вывести аргумент как десятичное целое число.
`%6d`	—	вывести аргумент как десятичное целое число в поле шириной не менее 6 символов.
`%f`	—	вывести аргумент как вещественное число с плавающей точкой.
`%6f`	—	вывести аргумент как вещественное число в поле шириной не менее 6 символов.
`%.2f`	—	вывести аргумент как вещественное число с двумя цифрами после десятичной точки.
`%6.2f`	—	вывести аргумент как вещественное число в поле не короче 6 символов и с двумя цифрами после точки.

В числе прочих спецификаций функция `printf` понимает `%o` — вывод восьмеричного числа; `%x` — шестнадцатеричного числа; `%c` — отдельного символа; `%s` — строки, а также `%%` — собственно знака процента.

Упражнение 1.3. Модифицируйте программу преобразования температур так, чтобы она выводила заголовок над таблицей.

Упражнение 1.4. Напишите программу для перевода температур по Цельсию в шкалу Фаренгейта и вывода соответствующей таблицы.

1.3. Оператор `for`

Для каждой конкретной задачи можно написать множество разных версий программы, которая эту задачу решает. Попробуем написать другой вариант программы преобразования температур:

```
#include <stdio.h>

/* вывод таблицы температур по Фаренгейту и Цельсию */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Эта программа дает те же результаты, но выглядит совсем по-другому. Одно из самых больших отличий — устранение из программы почти всех переменных. Осталась только переменная `fahr`, тип которой изменен на `int`. Верхний и нижний пределы изменения температуры, а также шаг перебора фигурируют только в виде констант в операторе `for`, который, кстати, является новой конструкцией. Выражение, в котором вычисляется значение температуры по Цельсию, фигурирует третьим аргументом в функции `printf`, а не отдельно в операторе присваивания.

Это последнее отличие является примером общего правила: в любом контексте, где можно использовать переменную определенного типа, можно также употребить и более сложное выражение того же типа. Поскольку третьим аргументом функции `printf` должно быть вещественное число, выводимое по спецификации `%6.1f`, на этом месте может стоять любое вещественнозначное выражение.

Оператор `for` — это оператор цикла, обобщение оператора `while`. Сравните его форму с употреблявшимся ранее `while`, и его работа должна стать вполне понятной. В скобках содержится три смысловых блока, разделенных запятыми. Первый блок — это инициализация:

```
fahr = 0
```

Он выполняется один раз, до входа в собственно цикл. Вторая часть представляет собой проверку условия, управляющего выполнением цикла:

```
fahr < 300
```

Если проверка дает положительный результат, далее выполняется тело цикла (в данном случае единственный вызов функции `printf`). Затем выполняется третий блок заголовка, модификация или приращение управляющих циклом переменных:

```
fahr = fahr + 20
```

Цикл прекращается, как только условие перестает выполняться. Как и в случае while, тело цикла может состоять из одного оператора или группы операторов, заключенных в скобки. Блоки инициализации, условия и модификации могут представлять собой любые выражения.

Выбрать ли цикл for или while для конкретной задачи, зависит только от специфики задачи и диктуется соображениями удобства; вообще же они взаимозаменяемы. Цикл for обычно удобен в тех случаях, когда блоки инициализации и модификации состоят из одного оператора каждый и при этом имеют непосредственное отношение к телу цикла; в этом случае for дает более компактную запись, чем while, поскольку все управляющие элементы содержатся в одном месте.

Упражнение 1.5. Доработайте программу преобразования температур так, чтобы она выводила таблицу в обратном порядке, т.е. от 300 градусов до нуля.

1.4. Символические константы

Сделаем одно последнее замечание, прежде чем расстаться с программой преобразования температур навсегда. Наводнять программу явными числовыми константами типа 300 или 20 — это дурной тон, поскольку они практически ничего не говорят постороннему человеку, взявшемуся читать такую программу. К тому же эти константы трудно выискивать и изменять по всему тексту единообразно в случае необходимости. Один из подходов к решению проблемы числовых констант состоит в том, чтобы давать им значащие имена. Строка #define определяет *символическое имя* или *символическую константу* в виде строки символов:

```
#define   имя   текст для подстановки
```

Всякий раз, когда в программе встретится определенное таким образом *имя* (не в кавычках и не в составе другого имени), оно будет заменено соответствующим *текстом для подстановки*. *Имя* задается в той же форме, что и имя переменной, т.е. как последовательность букв и цифр, начинающаяся с буквы. *Текст для подстановки* может представлять собой последовательность любых символов, а не только цифр.

```
#include <stdio.h>

#define   LOWER  0       /* нижний предел диапазона */
#define   UPPER  300     /* верхний предел */
#define   STEP   20      /* размер шага */

/* вывод таблицы температур по Фаренгейту и Цельсию */
main()
{
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Величины LOWER, UPPER и STEP в этом случае являются символическими константами, а не переменными, поэтому они не фигурируют в объявлениях. Имена символических констант обычно записывают прописными буквами, чтобы легко отличать их от

имен переменных в нижнем регистре. Обратите внимание, что строка `#define` не заканчивается точкой с запятой.

1.5. Символьный ввод-вывод

В этом разделе мы рассмотрим семейство программ, выполняющих те или иные операции по обработке символьной информации. Со временем вы увидите, что многие программы являются всего лишь расширенными вариантами тех простых прототипов, которые будут приведены здесь.

Модель ввода-вывода, поддерживаемая стандартной библиотекой функций, очень проста. Текстовый ввод-вывод, независимо от его физического источника или места назначения, выполняется над потоками символов. *Поток символов* — это последовательность символов, разбитых на строки; каждая строка заканчивается специальным символом конца строки и может быть пустой или содержать некоторое количество символов. За то, чтобы привести каждый поток ввода или вывода в соответствие с этой моделью, отвечает стандартная библиотека, тогда как программисту на C, пользующемуся ей, нет нужды заботиться о том, как строки потока представляются вовне программы.

В стандартной библиотеке имеется ряд функций для чтения или записи одного символа за одну операцию; простейшими из них являются `getchar` и `putchar`. Каждый раз при вызове `getchar` эта функция считывает *следующий символ* текстового потока ввода и возвращает его в качестве своего значения. Таким образом, после выполнения приведенного ниже оператора переменная `c` будет содержать следующий символ входного потока:

```
c = getchar();
```

Обычно символы поступают с клавиатуры. Ввод из файлов рассматривается в главе 7.

Функция `putchar` при каждом вызове выводит один символ:

```
putchar(c);
```

Данный оператор выводит значение целочисленной переменной `c` в виде символа — как правило, на экран монитора. Вызовы `putchar` и `printf` можно чередовать как угодно — выводимые данные будут следовать в том порядке, в каком выполняются вызовы.

1.5.1. Копирование файлов

Имея в своем распоряжении функции `getchar` и `putchar`, можно написать удивительно много полезного программного кода, больше ничего не зная о вводе-выводе. Простейшим примером является программа, копирующая входной поток в выходной по одному символу:

```
прочитать символ
while (символ не является символом-конца-файла)
    вывести прочитанный символ
    прочитать символ
```

Если все это перевести на C, получится вот что:

```
#include <stdio.h>

/* копирование входного потока в выходной */
```

```
main()
{
    int c;

    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

Знак != означает операцию отношения "не равно".

Все, что на экране или клавиатуре выглядит как символ, в памяти компьютера, разумеется, представлено в виде набора битов. Для хранения таких символьных данных специально предназначен тип `char`, но можно использовать и любой другой целочисленный тип. Мы выбрали для этой цели тип `int` по неочевидной, но важной причине.

Существует проблема, как отличить конец входного потока от обычного символа данных. Функция `getchar` решает ее, возвращая при окончании потока ввода особое значение, которое нельзя перепутать ни с каким настоящим символом. Это значение называется `EOF` — сокращение от "end-of-file" (конец файла). Необходимо объявить переменную `c` принадлежащей к типу достаточного объема, чтобы вместить любое значение, возвращаемое из `getchar`. Тип `char` для этих целей не годится, поскольку переменная `c` должна вмещать как любой возможный символ, так и `EOF`. Вот почему мы используем тип `int`.

Символическая константа `EOF` — это целое число, определенное в файле `<stdio.h>`, но ее конкретное числовое значение не играет роли, коль скоро оно не совпадает ни с одним из возможных значений типа `char`. Использование символической константы гарантирует, что никакие компоненты программы не окажутся зависимыми от ее конкретного числового значения.

Опытные программисты на C написали бы программу копирования потоков короче и экономнее. В языке C любое присваивание является выражением и имеет значение:

```
c = getchar()
```

Значением этого выражения является значение переменной слева от знака равенства после выполнения присваивания. Отсюда следует, что выражение с присваиванием может фигурировать в составе других выражений. Если присваивание вводимого символа переменной `c` внести в проверку условия цикла `while`, программу копирования можно переписать в таком виде:

```
#include <stdio.h>

/* копирование входного потока в выходной; 2-я версия */
main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

В заголовке `while` вводится символ и присваивается переменной `c`, а затем выполняется проверка, не обозначает ли он конец файла. Если это не так, выполняется тело

цикла, в котором выводится символ. Затем цикл while повторяется. Когда достигается конец потока, цикл, а вместе с ним и функция main прекращают работу.

В этой версии программы операции ввода концентрируются в одном месте, поскольку остается всего один вызов getchar, а объем кода уменьшается. Программа записывается более компактно, и ее становится удобнее читать (при условии, что читатель знаком с данной устойчивой конструкцией). Такой стиль будет встречаться часто. Конечно, им можно чрезмерно увлечься и написать совершенно неудобочитаемый код, но мы постараемся избежать этой тенденции.

Скобки вокруг присваивания в условии необходимы. *Приоритет* операции != выше, чем операции =, т.е. в отсутствие скобок сравнение != выполнялось бы раньше, чем присваивание =. Таким образом, следующие два выражения будут эквивалентны:

```
c = getchar() != EOF
c = (getchar() != EOF)
```

При этом переменная c получает значение 0 или 1 в зависимости от того, встретился или нет конец файла при вызове getchar. В то же время от этого выражения ожидалось совсем другое. Более подробно поговорим об этом в главе 2.

Упражнение 1.6. Проверьте, что выражение getchar() != EOF действительно равно 1 или 0.

Упражнение 1.7. Напишите программу для вывода значения константы EOF.

1.5.2. Подсчет символов

Следующая программа подсчитывает символы; она похожа на программу копирования потоков:

```
#include <stdio.h>

/* подсчет символов во входном потоке; 1-я версия */
main()
{
    long nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

Здесь использована новая операция ++, представляющая собой *инкремент*, или увеличение на единицу. Вместо нее можно было бы записать nc = nc+1, но ++nc пишется короче и, как правило, работает быстрее. Существует и противоположная операция декремента (--) для уменьшения на единицу. Операции ++ и – могут быть префиксными (++nc) или постфиксными (nc++). В выражениях эти два способа записи дают разные результаты, как будет показано в главе 2. Но в любом случае как ++nc, так и nc++ инкрементируют переменную nc. Пока что будем придерживаться префиксной формы записи.

Программа подсчета символов накапливает счет в переменной типа long вместо int. Переменные типа long имеют длину не менее 32 бит. Хотя в некоторых системах типы int и long имеют одинаковую длину, в других тип int является 16-разрядным и

может хранить числа не более 32 767. Чтобы переполнить счетчик типа int, потребуется не такой уж длинный поток. Спецификация вывода %ld сообщает функции printf, что соответствующий аргумент имеет тип long — длинное целое число.

Можно работать со счетчиком даже большей емкости, если использовать для него тип double (аналог float, но с двойной точностью). Вместо while можно применить цикл for, чтобы проиллюстрировать другой способ записи циклов:

```
#include <stdio.h>

/* подсчет символов во входном потоке; 2-я версия */
main()
{
    double nc;

    for (nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

В функции printf спецификация %f используется и для float, и для double; конструкция %.0f подавляет вывод десятичной точки и дробной части, которая равна нулю.

Тело данного цикла for пусто, поскольку все операции выполняются при проверке условия и модификации переменных. Однако грамматика C требует, чтобы оператор for имел тело. Это требование удовлетворяется наличием *пустого оператора* — точки с запятой. У нас в программе он ставится в отдельной строке, чтобы выделить тело цикла.

Прежде чем закончить с программой подсчета символов, сделаем одно замечание. Если во входном потоке совсем нет символов, то проверка условия в while или for дает отрицательный результат при первом же вызове getchar, и программа выдает нуль, который и является правильным ответом. Это важно. В операторах while и for хорошо именно то, что проверка выполняется в начале цикла перед входом в его тело. Если делать в цикле нечего, то ничего и не делается, поскольку до операторов тела управление так и не доходит. Программы должны действовать корректно, встретив поток ввода нулевой длины. Операторы while и end помогают безошибочно обрабатывать разного рода предельные случаи.

1.5.3. Подсчет строк

Следующая программа подсчитывает количество строк во входном потоке. Как уже упоминалось, за представление потока ввода в виде последовательности строк, заканчивающихся специальными символами, отвечает стандартная библиотека. Таким образом, подсчет строк сводится к подсчету символов конца строки в потоке:

```
#include <stdio.h>

/* подсчет строк во входном потоке */
main()
{
    int c, nl;

    nl = 0;
```

```
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

Тело цикла `while` в этом случае состоит из оператора `if`, который отвечает за приращение счетчика `++nl`. В операторе `if` проверяется условие, стоящее в скобках, и если оно истинно, то выполняется оператор или группа операторов в фигурных скобках после условия. И снова подчиненность или вложенность операторов в программе подчеркнута с помощью отступов.

Двойной знак равенства (==) в языке C используется для записи отношения "равно" (как = в языке Pascal или `.EQ.` в Fortran). Этот символ используется для того, чтобы отличать проверку равенства от присваивания, которое в C выражается одним знаком равенства (=). Одно предостережение: новички в C часто пишут присваивание (=), имея в виду равенство (==). Как будет показано в главе 2, сложность состоит в том, что в результате обычно получается синтаксически правильное выражение, поэтому даже не будет выдано предупреждение.

Символ, записанный в одинарных кавычках, представляет числовое значение, равное коду символа в символьном наборе системы. Такой символ называется *символьной константой*, хотя на самом деле это еще один способ записи небольшого целого числа. Например, `'A'` является символьной константой; в символьном наборе ASCII ее код равен 65, и это — внутреннее числовое представление символа A. Разумеется, запись `'A'` следует предпочесть записи `65`: ее смысл более очевиден, и константа будет независимой от конкретного символьного набора.

Управляющие комбинации, используемые в строковых константах, разрешаются также и в символьных константах, так что, например, `'\n'` обозначает символ конца строки с ASCII-кодом, равным 10. Уясните себе важное различие: константа `'\n'` — это один символ, и в выражениях она является просто целым числом, тогда как `"\n"` — это строковая константа, которая по случайности содержит всего один символ. Тема строковых и символьных констант рассматривается более подробно в главе 2.

Упражнение 1.8. Напишите программу для подсчета пробелов, знаков табуляции и символов конца строки.

Упражнение 1.9. Напишите программу для копирования входного потока в выходной с заменой каждой строки, состоящей из одного или нескольких пробелов, одним пробелом.

Упражнение 1.10. Напишите программу для копирования входного потока в выходной с заменой знаков табуляции на `\t`, символов возврата назад (`Backspace`) на `\b`, а обратных косых черт — на `\\`. Это сделает табуляции и символы возврата легко читаемыми в потоке.

1.5.4. Подсчет слов

Четвертая программа из нашей серии полезных утилит для обработки текстовых потоков подсчитывает строки, слова и символы. Используется нестрогое определение слова как последовательности символов, не содержащей пробелов, табуляций и символов новой строки. Это сильно урезанный "скелет" программы `wc` для системы Unix.

```
#include <stdio.h>

#define IN  1     /* внутри слова */
#define OUT 0     /* снаружи слова */

/* подсчет строк, слов и символов во входном потоке */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Всякий раз, когда программа встречает первый символ слова, она прибавляет единицу к счетчику слов. Переменная `state` учитывает состояние потока: находимся ли мы внутри слова или нет. Вначале она устанавливается в состояние "вне слова", что соответствует значению OUT. Мы предпочитаем символические константы `IN` и OUT явным числовым 1 и 0, поскольку так программу удобнее читать. В маленьких программах наподобие приведенной это не так важно, однако в больших улучшение удобочитаемости, достигаемое за счет применения символических констант, более чем окупает те скромные усилия, которые нужно приложить, чтобы с самого начала писать в этом стиле. Вы скоро убедитесь сами, что вносить глобальные изменения в программы гораздо легче, если числовые константы фигурируют в них только под символическими именами.

Следующая строка обнуляет сразу три переменные:

```
nl = nw = nc = 0;
```

Это не какая-нибудь специальная форма присваивания, а прямое следствие того факта, что оператор присваивания является выражением, имеющим значение, и что присваивание выполняется справа налево. Можно было бы записать и так:

```
nl = (nw = (nc = 0));
```

Знак `||` обозначает логическую операцию ИЛИ:

```
if (c == ' ' || c == '\n' || c == '\t')
```

Поэтому данная строка означает "если `c` — пробел, или `c` — символ конца строки, или `c` — табуляция". (Вспомните, что `\t` является визуальным представлением невидимого символа табуляции.) Существует также знак `&&` для операции И; его приоритет выше, чем у `||`. Выражения, в которых аргументы соединяются знаками `&&` и `||`, вычисляются слева направо, причем вычисление всегда прекращается в том месте, в котором гарантированно достигается значение ПРАВДА или ЛОЖЬ. Так, если `c` — пробел, то нет

нужды выяснять, является ли этот символ еще и концом строки или табуляцией, и соответствующие сравнения просто не выполняются. Здесь это не особенно важно, а вот в более сложных ситуациях это имеет значение, как вы вскоре убедитесь.

В приведенном примере фигурирует также конструкция `else`, которая задает альтернативные операции на тот случай, если проверка условия в `if` дает отрицательный результат. Вот общая форма оператора `if`:

```
if (выражение)
    оператор1
else
    оператор2
```

Выполняется всегда только один оператор из двух, ассоциированных с конструкцией `if-else`. Если выражение равно ПРАВДА (TRUE), выполняется *оператор1*; в противном случае выполняется *оператор2*. Каждый из этих *операторов* может быть и блоком операторов в фигурных скобках. В программе подсчета слов после `else` стоит еще один оператор `if`, управляющий двумя операторами в фигурных скобках.

Упражнение 1.11. Как бы вы протестировали программу подсчета слов? Какого рода входной поток скорее всего выявит ошибки в программе (если таковые есть)?

Упражнение 1.12. Напишите программу для вывода входного потока по одному слову в строке.

1.6. Массивы

Напишем программу, которая бы подсчитывала количество каждой из цифр, количество символов пустого пространства (пробелов, табуляций, переходов на новую строку) и количество всех остальных символов во входном потоке. Задача довольно искусственная, но позволяет проиллюстрировать несколько аспектов языка С в одной программе.

Итак, у нас есть двенадцать возможных категорий входных данных. Удобно завести массив, чтобы держать в нем количество вхождений каждой цифры, вместо того чтобы объявлять десять отдельных переменных. Вот первая версия такой программы:

```
#include <stdio.h>

/* подсчет цифр, символов пустого пространства, остальных */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
```

```
            ++nother;

    printf("digits =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n",
        nwhite, nother);
}
```

Если задать программе в качестве исходных данных ее же исходный текст, получим следующий результат:

```
digits = 9 3 0 0 0 0 0 0 0 1, white space = 211, other = 365
```

Вот как объявляется массив `ndigit` из 10 целых чисел:

```
int ndigit[10];
```

В языке C индексы массива всегда начинаются с нуля, т.е. этот массив содержит элементы `ndigit[0]`, `ndigit[1]`, ..., `ndigit[9]`. Этот факт отражен в заголовках циклов `for`, которые инициализируют, а затем выводят массив.

Индексом может служить любое целочисленное выражение, в том числе целые переменные наподобие `i` или явные константы.

Данная программа по умолчанию полагается на наличие определенных свойств у символьного представления цифр. Так, следующий оператор определяет, является ли цифрой символ в переменной `c`:

```
if (c >= '0' && c <= '9') ...
```

Если это так, то вычисляется значение этой цифры:

```
c - '0'
```

Эти операции выполняются корректно, только если константы `'0'`, `'1'`, ... `'9'` следуют друг за другом строго в порядке возрастания их числовых значений. К счастью, это так во всех символьных наборах и кодировках.

По определению величины типа `char` являются всего лишь короткими целыми числами, так что в арифметических выражениях переменные и константы этого типа ведут себя так же, как `int`. Это и естественно, и удобно; например, `c - '0'` — это целочисленное выражение со значением между 0 и 9, соответствующее одному из символов от `'0'` до `'9'`, помещенному в переменную `c`. Следовательно, это выражение может служить индексом массива `ndigit`.

Решение о том, является ли символ цифрой, пробелом, табуляцией, концом строки или же другим символом, принимается в следующем фрагменте программы:

```
if (c >= '0' && c <= '9')
    ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nwhite;
else
    ++nother;
```

Такая конструкция часто встречается в программах там, где требуется принимать многовариантное решение. Вот ее общий вид:

```
if (условие1)
    оператор1
else if (условие2)
    оператор2
```

```
...
    ...
else
    операторN
```

Условия проверяются сверху вниз, пока одно из них не окажется выполненным. В этом случае программа переходит к выполнению соответствующего *оператора*, и работа всей конструкции завершается. (На месте каждого из *операторов* может стоять блок операторов в фигурных скобках.) Если не выполняется ни одно из условий, управление передается на *операторN* после последнего `else` (если он присутствует). Если же завершающего `else` и *оператора* после него нет, ничего не делается. Между начальным `if` и конечным `else` можно вставить сколько угодно групп такого вида:

```
else if (условие)
    оператор
```

Что касается стиля, то лучше форматировать эту конструкцию так, как было показано. Если каждый `if` сдвигать относительно предыдущего `else` вправо, то длинная цепочка проверок условий "уедет" за правый край страницы.

Еще один путь принятия решений путем выбора из многих вариантов предоставляет оператор `switch`, который рассматривается в главе 3. Этот способ особенно удобен, если проверка условия состоит в том, равно ли некоторое целочисленное или символьное выражение одной из констант заданного набора. Версия приведенной выше программы с оператором `switch` будет продемонстрирована в разделе 3.4.

Упражнение 1.13. Напишите программу для вывода гистограммы длин слов во входном потоке. Построить гистограмму с горизонтальными рядами довольно легко, а вот с вертикальными столбцами труднее.

Упражнение 1.14. Напишите программу для вывода гистограммы частот, с которыми встречаются во входном потоке различные символы.

1.7. Функции

Функция в C — это эквивалент понятий подпрограммы и функции в языке Fortran или процедуры и функции в языке Pascal. Функции — это удобный способ свести в одно место (инкапсулировать) некоторые вычислительные операции, а затем обращаться к ним много раз, не беспокоясь об особенностях реализации. Если функция написана правильно, нет нужды знать, *как* она работает; достаточно знать, *что* именно она делает. В языке C пользоваться функциями легко, удобно и эффективно. Часто можно увидеть, как какая-нибудь короткая функция определяется для того, чтобы ее вызвали один раз, — только потому, что это проясняет смысл и структуру кода.

До сих пор мы пользовались только такими функциями, как `printf`, `getchar` и `putchar`, написанными кем-то другим. Пора уже самим написать несколько собственных функций. Поскольку в языке C нет операции возведения в степень наподобие `**` в Fortran, воспользуемся случаем и продемонстрируем процесс определения функций, написав функцию `power(m,n)` для возведения целого числа `m` в целую положительную степень `n`. Например, значением `power(2,5)` будет `32`. Эта функция не слишком применима на практике, поскольку она работает только с положительными степенями не-

больших целых чисел, но для иллюстративных целей вполне подойдет. (В стандартной библиотеке имеется функция `pow(x,y)`, вычисляющая x^y.)

Ниже приведены функция `power` и программа, выполняющая ее вызов. Таким образом, здесь продемонстрирован весь процесс работы с функциями.

```
#include <stdio.h>

int power(int m, int n);

/* тестирование power - функции возведения в степень */
main()
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}

/* power: возводит base в n-ю степень; n >= 0 */
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Определение функции имеет следующую форму:

```
тип-возвращ-знач имя-функции(объявления параметров)
{
    объявления
    операторы
}
```

Определения функций могут следовать в любом порядке, а также находиться как в одном файле исходного кода, так и в нескольких; однако запрещается разбивать одну функцию между несколькими файлами. Если программа состоит из нескольких файлов исходного кода, необходимо выполнить больше операций по ее компилированию и загрузке, чем в случае одного файла. Но это уже особенности работы с операционной системой, а не атрибуты языка. Пока предположим, что обе функции находятся в одном файле, так что остается в силе все, что мы до сих пор говорили о выполнении программ на C.

Функция `power` вызывается из `main` дважды в следующей строке:

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

При каждом вызове передаются два аргумента, а возвращается одно целое число, которое и выводится согласно спецификации. В выражениях конструкция `power(2,i)` может употребляться точно так же, как обычные целые константы и переменные, например `2` или `i`. (Не все функции возвращают целые значения; подробнее об этом будет сказано в главе 4.)

Рассмотрим первую строку самой функции power:

```
int power(int base, int n)
```

В ней объявляются типы и имена параметров, а также тип результата, возвращаемого функцией. Имена, которые используются в power для параметров, локализованы в самой функции и невидимы за ее пределами, так что другие функции программы могут пользоваться теми же именами безо всяких конфликтов. Это верно также для переменных i и p: счетчик i в функции power никак не связан с переменной i в main.

Будем называть *параметром* переменную из заключенного в скобки списка в заголовке функции, а *аргументом* — значение, подставляемое при вызове функции. Иногда в том же смысле используют термины *формальный* и *фактический аргумент*.

Значение, вычисляемое функцией power, возвращается в main оператором return. После слова return может стоять любое выражение:

```
return выражение;
```

Функция не обязана возвращать значение. Если записать оператор return без выражения после него, то он передаст в вызывающую функцию управление, но не значение, точно так же, как это делается при достижении конца функции — закрывающей фигурной скобки.

Можно заметить, что оператор return стоит и в конце функции main. Поскольку main — такая же функция, как и другие, она может возвращать значение в ту точку, откуда ее вызывают. А это фактически та операционная среда, в которой выполняется программа. Обычно возвращение нуля обозначает нормальное завершение программы, а ненулевого значения — нестандартные или ошибочные условия завершения. В целях упрощения мы не ставили return в конце наших функций main до этого момента, но теперь начнем это делать, чтобы подчеркнуть, что программам полезно посылать отчет о корректности своего завершения в операционную среду.

Перед функцией main можно заметить такую строку:

```
int power(int m, int n);
```

Она сообщает, что power является функцией с двумя аргументами типа int, возвращающей также значение типа int. Эта декларация называется *прототипом функции* и обязана согласовываться по форме как с определением (заголовком) функции power, так и с любыми ее вызовами. Если определение функции отличается от ее прототипа или формы вызова, возникает ошибка.

Имена параметров согласовывать не обязательно. Вообще-то не обязательно даже указывать имена параметров в прототипе функции, поэтому предыдущий прототип можно было бы записать так:

```
int power(int, int);
```

Однако хорошо выбранные имена являются своеобразной документацией к программе, так что они будут часто использоваться.

Сделаем одно замечание касательно истории языка. Самые большие изменения в ANSI C по сравнению с более ранними версиями произошли как раз в форме объявления и определения функций. В первоначальном синтаксисе языка C функция power была бы записана следующим образом:

```
/* power: возводит base в n-ю степень; n >= 0 */
/*        версия в старом стиле */
power(base, n)
int base, n;
```

```
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Имена параметров приведены в скобках, а вот типы объявлены перед открывающей левой фигурной скобкой. Необъявленные параметры считаются имеющими тип `int`. (Тело функции не отличается от предыдущей версии.)

Объявление функции `power` в начале программы выглядело бы так:

```
int power();
```

Список параметров в старой версии не предусматривался, поэтому компилятор не мог проверить, корректно ли вызывалась функция `power`. Если еще учесть, что тип возвращаемого из `power` значения и так по умолчанию принимался `int`, все это объявление можно было бы просто опустить.

Новый синтаксис прототипов функций позволяет компилятору очень легко выявлять ошибки в количестве аргументов или их типах. Старый стиль объявлений и определений функций все еще работает в ANSI C, по крайней мере в переходный период, но мы настоятельно рекомендуем пользоваться только новой формой, если компилятор ее поддерживает.

Упражнение 1.15. Перепишите программу преобразования температур из раздела 1.2 так, чтобы само преобразование выполнялось функцией.

1.8. Аргументы: передача по значению

Один из аспектов функций C может оказаться новым для программистов, привыкших к другим языкам, в частности к Fortran. В языке C все аргументы функций передаются "по значению". Это означает, что вызываемая функция получает значения своих аргументов в виде временных переменных, а не оригиналов. Отсюда следуют несколько другие свойства функций, чем в языках, где происходит "передача по ссылке", — как в Fortran или в блоке `var` в языке Pascal, где вызывающая процедура имеет доступ к исходным аргументам, а не локальным копиям.

Основное различие между этими подходами заключается в том, что в C вызываемая функция не может модифицировать переменные в вызывающей функции; она вправе изменять только свои локальные, временные копии этих переменных.

Передача по значению — это не ограничение, а благо. Обычно благодаря ей программы пишутся компактнее и с меньшим количеством лишних переменных, поскольку в вызываемой функции параметры можно воспринимать как удобно инициализированные локальные переменные. Например, вот версия функции `power`, эффективно использующая это свойство:

```
/* power: возводит base в n-ю степень; n >= 0; 2-я версия */
int power(int base, int n)
```

```
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

Параметр n используется в качестве временной переменной для обратного отсчета в цикле for, идущем "вспять". Цикл заканчивается, когда n становится равной нулю. Потребность в переменной i отпадает. Операции, выполняемые над n внутри функции power, никак не влияют на исходный аргумент, с которым эта функция вызывалась.

По необходимости можно сделать так, чтобы функция модифицировала переменные в вызывающем модуле программы. Для этого вызывающая функция должна передать *адрес* переменной (в понятиях C — *указатель* на переменную), а вызываемая — объявить параметр указателем и обращаться к переменной косвенно, по ссылке через указатель. Работа с указателями рассматривается в главе 5.

С массивами происходит другая история. Если имя массива используется в качестве аргумента, то передаваемое в функцию значение как раз и является местонахождением (адресом) начала массива — копирования элементов массива не происходит. Применяя к переданному аргументу индекс, функция может обращаться к любому элементу массива и модифицировать его. Этому будет посвящен следующий раздел.

1.9. Массивы символов

Наиболее распространенный тип массивов в C — это массив символов. Чтобы проиллюстрировать работу с такими массивами и функции, предназначенные для этого, напишем программу, которая бы считывала набор строк и выводила самую длинную из них. Алгоритм программы довольно прост:

```
while (поступает следующая строка)
    if (она длиннее предыдущей самой длинной)
        сохранить ее
        сохранить ее длину
вывести самую длинную строку
```

Из алгоритма ясно, что программа естественным образом делится на составные части. Одна часть занимается вводом новой строки, другая — ее анализом, третья — сохранением, а оставшаяся — управлением работой всей программы.

Раз уж имеется такое четкое разделение обязанностей, программу надо написать так, чтобы это было отражено. Соответственно, вначале напишем отдельную функцию getline для считывания следующей строки входного потока. Попробуем написать ее так, чтобы она оказалась полезной и для других приложений. Как минимум, getline должна уметь возвращать сигнал о конце потока; но еще полезней было бы, если бы она возвращала длину введенной строки и 0 в том случае, если встретился конец файла. Нуль подходит в качестве сигнала конца файла, поскольку реальная строка данных не может иметь такую длину. В любой текстовой строке есть как минимум один символ; даже строка, содержащая только символ конца строки, имеет длину 1.

Когда встречается строка длиннее, чем предыдущая самая длинная строка, ее нужно где-то сохранить. Поэтому нужно написать вторую функцию, copy, которая бы копировала новую строку в некое надежное место.

Наконец, нужна главная программа для управления работой getline и copy. Вот что получается в результате:

```
#include <stdio.h>
#define MAXLINE 1000    /* максимальная длина строки в потоке */

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* вывод самой длинной строки в потоке */
main()
{
    int len;            /* длина текущей строки */
    int max;            /* текущая максимальная длина */
    char line[MAXLINE];     /* текущая введенная строка */
    char longest[MAXLINE];  /* самая длинная строка из введенных */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(longest, line);
        }
    if (max > 0)    /* была непустая строка */
        printf("%s", longest);
    return 0;
}

/* getline: считывает строку в s, возвращает ее длину */
int getline(char s[], int lim)
{
    int c, i;

    for (i = 0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}

/* copy: копирует строку 'from' в 'to'; длина to считается доста-
точной */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
```

```
        ++i;
}
```

Функции `getline` и `copy` объявлены в начале программы, которую будем считать хранящейся в одном файле.

Функции `main` и `getline` обмениваются информацией через пару аргументов и возвращаемое значение. В `getline` аргументы объявляются в следующей строке:

```
int getline(char s[], int lim)
```

Здесь сказано, что первый аргумент `s` — это массив, а второй, `lim`, — целое число. Размер массива необходимо указывать при его объявлении, чтобы выделить достаточно памяти. Но знать длину массива `s` в функции `getline` не обязательно, поскольку эта длина определяется в `main`. Из функции `getline` в функцию `main` с помощью оператора `return` возвращается числовое значение — точно так же, как это делалось в функции `power`. В этой же строке объявляется и тип возвращаемого значения — `int`; поскольку этот тип принимается по умолчанию, его можно опустить.

Некоторые функции возвращают полезные значения, тогда как другие, наподобие `copy`, только выполняют какие-то операции и ничего не возвращают. Тип возвращаемого значения в функции `copy` — `void`, что и означает его отсутствие.

Функция `getline` помещает в конец созданного ею массива символ `'\0'` (*нулевой символ*, числовой код которого равен нулю), чтобы обозначить конец строки символов. Этот подход принят в языке C за стандарт. Пусть, например, в программе на C фигурирует такая строковая константа:

```
"hello\n"
```

В памяти она будет размещена как массив символов, содержащий по порядку все символы строки и заканчивающийся `'\0'` для обозначения конца:

h	e	l	l	o	\n	\0

Спецификация формата `%s` в вызове `printf` предполагает, что соответствующий аргумент будет строкой, представленной именно в такой форме. Функция `copy` также рассчитывает на то, что ее аргумент-источник оканчивается на `'\0'`, причем она копирует этот символ во вторую строку-аргумент. (При всем этом подразумевается, что символ `'\0'` не может встречаться в нормальном тексте.)

Стоит заметить, что даже такая крохотная программа, как эта, содержит несколько нерешенных технических вопросов. Например, что делать функции `main`, если ей встречается строка длиннее предельной длины? Функция `getline` в этом отношении безопасна, поскольку прекращает ввод, как только массив заполнился, даже если еще не встретился символ конца строки. Проверяя длину строки и последний введенный символ, функция `main` может определить, не чрезмерную ли длину имеет строка, а затем обработать эту ситуацию по своему усмотрению. Для краткости мы эту проблему пока игнорируем.

Пользователь функции `getline` никак не может знать заранее, какой длины будет вводимая строка, поэтому `getline` выполняет проверку на переполнение. С другой стороны, пользователь `copy` уже знает (или может выяснить) длину строки, поэтому мы решили не добавлять в нее контроль ошибок.

Упражнение 1.16. Доработайте главный модуль программы определения самой длинной строки так, чтобы она выводила правильное значение для какой угодно длины строк входного потока, насколько это позволяет текст.

Упражнение 1.17. Напишите программу для вывода всех строк входного потока, имеющих длину более 80 символов.

Упражнение 1.18. Напишите программу для удаления лишних пробелов и табуляций в хвосте каждой поступающей строки входного потока, которая бы также удаляла полностью пустые строки.

Упражнение 1.19. Напишите функцию `reverse(s)`, которая переписывает свой строковый аргумент `s` в обратном порядке. Воспользуйтесь ею для написания программы, которая бы выполняла такое обращение над каждой строкой входного потока по очереди.

1.10. Внешние переменные

Переменные в функции `main`, такие как `line`, `longest` и т.п., являются закрытыми или локальными для `main`. Поскольку они объявлены внутри тела `main`, ни одна другая функция не может иметь к ним непосредственного доступа. То же самое справедливо и для переменных других функций; например, переменная `i` в `getline` никак не связана с `i` в функции `copy`. Каждая локальная переменная в функции начинает свое существование при вызове функции и прекращает его при выходе из нее. Вот почему такие переменные обычно называются *автоматическими*, как и в других языках. В дальнейшем в отношении подобных локальных переменных будет употребляться термин "автоматические", а в главе 4 вы узнаете о статическом (`static`) классе памяти, в котором локальные переменные сохраняют свои значения между вызовами функции.

Поскольку автоматические переменные создаются и уничтожаются в связи с вызовами функций и выходами из них, они не сохраняют свои значения между вызовами и должны инициализироваться явным образом при каждом входе в функцию. Если их не инициализировать, они будут содержать случайный "мусор".

В противоположность автоматическим переменным, можно определить переменные, которые будут *внешними* по отношению ко всем функциям, т.е. такие переменные, к которым любая функция программы сможет обращаться по имени. (Этот механизм — аналог блока COMMON в языке Fortran или переменных языка Pascal, объявленных в самом внешнем блоке программы.) Поскольку внешние переменные доступны во всей программе глобально, их можно использовать вместо аргументов для обмена данными между функциями. Более того, поскольку внешние переменные существуют непрерывно, а не появляются и исчезают, они сохраняют свои значения даже после выхода из функций, которые модифицируют их значения.

Внешняя переменная должна быть *определена* ровно один раз за пределами всех функций программы. При этом для нее выделяется память. Она также должна быть *объявлена* в каждой функции, которая к ней обращается, с указанием ее типа. Объявление может быть явным, с помощью оператора `extern`, а может быть и неявным — по контексту. Чтобы придать нашему изложению конкретность, перепишем программу определения самой длинной строки, сделав `line`, `longest` и `max` внешними переменными. При этом придется изменить способы вызова, объявления и тела всех трех функций:

```
#include <stdio.h>
#define MAXLINE 1000    /* максимальная длина строки в потоке */

int max;                /* текущая максимальная длина */
```

```
char line[MAXLINE];     /* текущая введенная строка */
char longest[MAXLINE];  /* самая длинная строка из введенных */

int getline(void);
void copy(void);

/* вывод самой длинной строки в потоке; специальная версия */
main()
{
    int len;
    extern int max;
    extern char longest[];

    max = 0;
    while ((len = getline()) > 0)
        if (len > max) {
            max = len;
            copy();
        }
    if (max > 0)    /* была непустая строка */
        printf("%s", longest);
    return 0;
}

/* getline: специальная версия */
int getline(void)
{
    int c, i;
    extern char line[];

    for (i = 0; i<MAXLINE-1
        && (c=getchar()) != EOF && c != '\n'; ++i)
            line[i] = c;
    if (c == '\n') {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return i;
}

/* copy: специальная версия */
void copy(void)
{
    int i;
    extern char line[], longest[];

    i = 0;
    while ((longest[i] = line[i]) != '\0')
        ++i;
}
```

Внешние переменные для функций `main`, `getline` и `copy` определяются в первых строках приведенного примера, в которых объявляется тип и инициализируется выделение памяти для этих переменных. Объявления внешних переменных синтаксически ни-

чем не отличаются от объявлений локальных, но находиться они должны за пределами функций, что и делает их внешними. Прежде чем функция сможет пользоваться внешней переменной, имя этой переменной должно стать ей известно. Единственный способ сделать это — записать в функции объявление со словом `extern` впереди; в остальном объявление будет таким же, как и раньше.

В определенных обстоятельствах объявление `extern` можно опустить. Если определение внешней переменной находится в файле исходного кода перед обращением к ней в какой-нибудь функции, то в этой функции объявление со словом `extern` не обязательно. Таким образом, операторы `extern` в функциях `main`, `getline` и `copy` излишни. Обычная практика — ставить все определения внешних переменных в начале файла исходного кода, а затем опускать все объявления `extern`.

Если программа содержится в нескольких файлах исходного кода, а переменная определена в файле *`file1`* и используется в файлах *`file2`* и *`file3`*, то в двух последних файлах обязательно должны присутствовать объявления `extern`, чтобы согласовать между собой обращения к переменной. Обычно все `extern`-объявления переменных и функций собирают во внешний файл, по традиции именуемый *заголовочным* (*header*), и подключают его в строке `#include` в начале каждого файла исходного кода. Для имен заголовочных файлов традиционно используется расширение `.h`. Например, функции стандартной библиотеки объявляются в заголовочных файлах вида `<stdio.h>`. Эта тема подробно раскрыта в главе 4, а сама библиотека рассматривается в главе 7 и приложении Б.

Поскольку специальные версии функций `getline` и `copy` не имеют аргументов, по логике их прототипы в начале файла должны выглядеть как `getline()` и `copy()`. Однако для совместимости со старыми программами на C в стандарте предполагается, что пустой список аргументов — это объявление в старом стиле и что компилятор отключает все проверки списка аргументов. Поэтому для описания действительно пустого списка аргументов следует использовать слово `void`. Это будет более подробно обсуждаться в главе 4.

Следует отметить, что слова *объявление* и *определение* в этом разделе употребляются с особой тщательностью, когда речь идет о внешних переменных. "Определение" — это место, в котором переменная фактически создается с выделением памяти для нее. "Объявление" — это место, где указывается тип и характеристики переменной, но никакого выделения памяти не происходит.

Кстати, существует тенденция делать внешними (`extern`) переменными все, что попадается на глаза, потому что это якобы упрощает обмен данными: списки аргументов становятся короче, а переменные всегда под рукой в нужный момент. Но, к сожалению, внешние переменные всегда "крутятся" под рукой и в ненужный момент. Слишком интенсивное использование внешних переменных чревато опасностями, поскольку в таких программах обмен данными далеко не очевиден — переменные могут модифицироваться неожиданно и даже непреднамеренно, а доработка кода сильно усложняется. Вторая версия программы определения самой длинной строки уступает первой в качестве частично по этим причинам, а частично потому, что общность двух полезных функций сильно ухудшается из-за внедрения в их тела прямых ссылок на обрабатываемые ими переменные.

Итак, мы рассмотрели все то, что можно условно назвать ядром языка C. Располагая этим небольшим набором элементарных "кирпичиков", уже можно писать программы существенной длины, и было бы очень неплохо, если бы вы побольше попрактиковались

именно на этом этапе. В следующих упражнениях уровень сложности несколько выше, чем у тех программ, которые приводились в разделах этой главы.

Упражнение 1.20. Напишите программу `detab`, которая бы заменяла символы табуляции во входном потоке соответствующим количеством пробелов до следующей границы табуляции. Предположим, что табуляция имеет фиксированную ширину `n` столбцов. Следует ли сделать `n` переменной или символическим параметром?

Упражнение 1.21. Напишите программу `entab`, которая бы заменяла пустые строки, состоящие из одних пробелов, строками, содержащими минимальное количество табуляций и дополнительных пробелов, — так, чтобы заполнять то же пространство. Используйте те же параметры табуляции, что и в программе `detab`. Если для заполнения места до следующей границы табуляции требуется один пробел или один символ табуляции, то что следует предпочесть?

Упражнение 1.22. Напишите программу для сворачивания слишком длинных строк входного потока в две или более коротких строки после последнего непустого символа, встречающегося перед `n`-м столбцом длинной строки. Постарайтесь, чтобы ваша программа обрабатывала очень длинные строки корректно, а также удаляла лишние пробелы и табуляции перед указанным столбцом.

Упражнение 1.23. Напишите программу для удаления всех комментариев из программы на C. Позаботьтесь о корректной обработке символьных констант и строк в двойных кавычках. Вложенные комментарии в C не допускаются.

Упражнение 1.24. Напишите программу для выполнения примитивной синтаксической проверки программ на C, таких как непарные круглые, квадратные и фигурные скобки. Не забудьте об одинарных и двойных кавычках, управляющих символах и комментариях. (Это сложная программа, если пытаться реализовать самый общий случай.)

Глава 2

Типы данных, операции и выражения

Фундаментальные объекты данных, с которыми работает программа, — это переменные и константы. Используемые в программе переменные перечисляются в объявлениях или декларациях, в которых указывается их тип, а также иногда их начальные значения. Манипуляции и преобразования над данными выполняются с помощью знаков операций. Переменные и константы объединяются в выражения, чтобы таким образом порождать новые значения. Тип объекта всегда определяет, какой набор значений может иметь этот объект и какие операции могут над ним выполняться. Все эти "кирпичики" программы и рассматриваются в этой главе.

В стандарте ANSI базовые типы и операции подверглись незначительным изменениям и дополнениям по сравнению с исходной версией языка. Теперь для всех целочисленных типов есть формы `signed` и `unsigned`, а также введены способы обозначения констант без знака и шестнадцатеричных констант. Операции с плавающей точкой могут выполняться с одинарной точностью, а для повышенной точности расчетов введен тип `long double`. Строковые константы можно соединять в одну (т.е. выполнять над ними конкатенацию) на этапе компиляции программы. Перечислимые типы наконец-то стали полноправной частью языка, получив формальное определение после многих лет существования де-факто. Объекты можно объявлять с модификатором `const`, который запрещает изменять их значения впоследствии. Правила автоматического преобразования арифметических типов были доработаны с целью расширения набора типов.

2.1. Имена переменных

Хотя в главе 1 об этом не говорилось, существуют некоторые ограничения на имена переменных и символических констант. Имя может состоять из букв и цифр, но начинаться должно с буквы. Знак подчеркивания (_) считается буквой и часто оказывается полезным для улучшения удобочитаемости длинных имен. Однако не следует начинать с него имена переменных, потому что такие имена часто используются библиотечными функциями. Буквы в верхнем и нижнем регистре различаются, так что `x` и `X` — это два разных имени. Традиционно в С принято записывать имена переменных строчными буквами, а имена символических констант — прописными.

Как минимум 31 символ из имени, внутреннего для программы, считается значащим. Для имен функций и внешних переменных это количество может быть меньше 31, потому что внешние имена могут использоваться ассемблерами и загрузчиками, над которыми у языка нет никакого контроля. В отношении внешних имен стандарт гарантирует уникальность только первых шести символов и одного регистра. Ключевые слова напо-

добие `if`, `else`, `int`, `float` и т.п. зарезервированы — их нельзя использовать в качестве имен переменных. Они должны записываться в нижнем регистре.

Разумно выбирать имена переменных так, чтобы они описывали назначение самих переменных и чтобы одно не могло легко превратиться в другое из-за опечатки. Существует тенденция использовать совсем короткие имена для локальных переменных, счетчиков циклов и т.п. и имена подлиннее для глобальных переменных.

2.2. Типы данных и их размеры

В языке С существует всего несколько базовых типов данных:

`char`	—	один байт, содержащий один символ из локального символьного набора;
`int`	—	целое число, обычно имеющее типовой размер для целых чисел в данной системе;
`float`	—	вещественное число одинарной точности с плавающей точкой;
`double`	—	вещественное число двойной точности с плавающей точкой.

Кроме того, существуют модификаторы, которые могут применяться к этим базовым типам. К целым числам применимы модификаторы `short` и `long`:

```
short int sb;
long int counter;
```

Слово `int` в таких объявлениях можно опустить, что обычно и делается.

Целью введения этих модификаторов было разграничить длины двух типов целых чисел для практических потребностей. Обычно `int` имеет стандартную для той или иной системы длину. Тип `short` часто имеет размер 16 бит, `long` — 32 бита, а `int` — или 16, или 32. Компилятору разрешено самостоятельно выбирать размер в соответствии с характеристиками аппаратуры и следующими ограничениями: числа типа `short` и `int` должны иметь длину не менее 16 бит, `long` — не менее 32 бит; тип `short` должен быть не длиннее `int`, а `int` — не длиннее `long`.

Модификатор `signed` ("со знаком") или `unsigned` ("без знака") может применяться к типу `char` или любому целочисленному. Числа типа `unsigned` всегда неотрицательны, а длина диапазона их значений равна степени двойки 2^n, где n — количество битов в машинном представлении типа. Например, если `char` имеет длину 8 бит, то переменные типа `unsigned char` могут иметь значения от 0 до 255, а `signed char` — от –128 до 127 (в системе с дополнением до двух). Являются ли переменные типа `char` знаковыми (`signed`) или беззнаковыми, зависит от конкретной системы, но выводимые на экран и печать символы всегда имеют положительные коды.

Тип `long double` обозначает число с плавающей точкой повышенной точности. Как и в случае целочисленных переменных, длины вещественных объектов зависят от реализации языка, так что из длин типов `float`, `double` и `long double` различными могут быть одна, две или три.

Стандартные заголовочные файлы `<limits.h>` и `<float.h>` содержат символические константы для всех этих размеров, а также других свойств системы и компилятора. Они рассматриваются в приложении Б.

Упражнение 2.1. Напишите программу для определения диапазонов переменных типов `char`, `short`, `int` и `long` (как `signed`, так и `unsigned`) путем вывода соответст-

вующих значений из заголовочных файлов, а также с помощью непосредственного вычисления. Для второго способа усложним задачу: определите еще и диапазоны вещественных типов.

2.3. Константы

Целочисленная константа наподобие `1234` имеет тип `int`. Длинную целую константу записывают со строчной или с прописной буквой `l` (`L`) на конце, например `123456789L`; если целое число слишком велико, чтобы поместиться в переменную типа `int`, оно также будет восприниматься как `long`. Константы без знака записываются с `u` или `U` на конце, а суффикс `ul` или `UL` обозначает длинную целую константу без знака.

Вещественные константы содержат десятичную точку (`123.4`), степенную часть (`1e-2`) или и то и другое. Они имеют тип `double`, если с помощью суффикса не указано другое. Суффикс `f` или `F` обозначает константу типа `float`, а `l` или `L` — константу типа `long double`.

Значение целого числа можно указывать в восьмеричной или шестнадцатеричной системе вместо десятичной. Если целая константа начинается с нуля (`0`), то она записана в восьмеричном представлении, а если с `0x` или `0X` — то в шестнадцатеричном. Например, десятичное число `31` можно записать как `037` в восьмеричной системе и `0x1f` или `0X1F` в шестнадцатеричной. В конце восьмеричных и шестнадцатеричных констант также можно ставить `L` для превращения их в длинные, а `U` — для лишения их знака. Так, число `0XFUL` является константой типа `unsigned long` с десятичным значением 15.

Символьная константа — это целое число в форме одного символа в кавычках, например `'x'`. Значением символьной константы является числовое значение кода символа в наборе, используемом в данной системе. Например, в наборе ASCII символьная константа `'0'` имеет код 48, который никак не связан с числом 0. Если записать `'0'` вместо числового кода 48, который зависит от конкретного символьного набора, мы получим более независимую от системы и удобочитаемую программу. Символьные константы участвуют в арифметических операциях на правах полноправных целых чисел, хотя чаще всех к ним применяются операции сравнения с другими символами.

Некоторые символы можно представить в символьных и строковых константах с помощью управляющих последовательностей наподобие `\n` (символа конца строки). Такие последовательности содержат два символа, но обозначают только один. Кроме того, любую битовую последовательность длиной один байт можно представить таким способом:

`'\`*ooo*`'`

где строка *ooo* может содержать от одной до трех восьмеричных цифр (`0...7`). Допускается также следующее представление:

`'\x`*hh*`'`

где *hh* — одна или две шестнадцатеричные цифры (`0...9`, `a...f`, `A...F`). Поэтому можно записать следующее в восьмеричном представлении:

```
#define VTAB '\013'     /* Вертикальная табуляция в ASCII */
#define BELL '\007'     /* Звуковой сигнал в ASCII */
```

В шестнадцатеричном представлении это будет выглядеть так:

```
#define VTAB '\xb'     /* Вертикальная табуляция в ASCII */
#define BELL '\x7'     /* Звуковой сигнал в ASCII */
```

Ниже приведен полный набор управляющих последовательностей, начинающихся с обратной косой черты.

`\a`	подача звукового сигнала	`\\`	обратная косая черта
`\b`	возврат назад и затирание	`\?`	вопросительный знак
`\f`	прогон страницы	`\'`	одинарная кавычка
`\n`	конец строки	`\"`	двойная кавычка
`\r`	возврат каретки	`\`*ooo*	восьмеричное число
`\t`	горизонтальная табуляция	`\x`*hh*	шестнадцатеричное число
`\v`	вертикальная табуляция		

Символьная константа `'\0'` представляет символ с нулевым кодом, т.е. нулевой символ. Часто пишут именно `'\0'`, а не `0`, чтобы подчеркнуть символьную природу того или иного выражения. Тем не менее числовое значение этой конструкции — просто нуль.

Константное выражение — это выражение, содержащее только константы. Такие выражения могут вычисляться в ходе компиляции, а не выполнения программы, и соответственно употребляться в любом месте, где допускается применение одной константы:

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

Еще один пример:

```
#define LEAP 1     /* в високосных годах */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Строковая константа, *строковый литерал*, или просто *литерал* — это последовательность из нескольких (в частном случае ни одного) символов, заключенных в двойные кавычки, например:

```
"I am a string"
```

А вот пример пустой строки:

```
""  /* пустая строка */
```

Кавычки не являются частью строки, а только ограничивают ее. Все управляющие последовательности, которые можно включать в символьные константы, допускаются и в строковых; `\"` представляет символ двойной кавычки. Строковые константы можно сцеплять (выполнять конкатенацию) в процессе компиляции. Так, записи в первой и второй строках эквивалентны:

```
"hello," "world"
"hello, world"
```

Этой возможностью удобно пользоваться, когда нужно распространить длинные строковые константы на несколько строк исходного кода.

Формально строковая константа является массивом символов. Во внутреннем представлении строки в конце присутствует нулевой символ `'\0'`, так что физический объем памяти для хранения строки превышает количество символов, записанных между кавычками, на единицу. Это означает, что на длину строки не накладываются никакие ограничения, но программа должна перебрать и проанализировать строку целиком, чтобы определить ее длину. Стандартная библиотечная функция `strlen(s)` возвращает количе-

ство символов в строке, переданной ей в качестве аргумента, не считая завершающего '\0'. Вот наша версия этой функции:

```
/* strlen:  возвращает длину строки s */
int strlen(char s[])
{
    int i;

    i = 0;
    while (s[i] != '\0')
        ++i;
    return i;
}
```

Функция strlen и другие функции для работы со строками объявлены в стандартном заголовочном файле <string.h>.

Будьте внимательны и не путайте символьные константы со строками, содержащими один символ: 'x' — это не то же самое, что "x". Первая величина — это целое число, используемое для представления кода буквы *x* в символьном наборе системы, тогда как вторая — массив символов, содержащий один символ (букву *x*) и завершающий нуль '\0'.

Существует еще один тип констант — *константы перечислимого типа*. Перечислимый тип (*перечисление*) определяется списком целочисленных символических констант, например:

```
enum boolean { NO, YES };
```

Первая константа в списке получает значение 0, вторая — 1 и т.д., если не указаны явные значения. Если заданы не все значения, те константы, для которых они не указаны, получат целые значения, следующие по порядку за последним из указанных. Это показано во втором из приведенных ниже примеров.

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
                    /* FEB = 2, MAR = 3 и т.д. */
```

Имена в различных перечислениях должны быть различными. Значения констант в одном перечислении не обязаны быть различными.

Перечисления — это удобный способ ассоциировать значения констант с символическими именами с тем преимуществом перед #define, что значения могут генерироваться автоматически. Хотя можно объявлять переменные типов enum, компиляторы не обязаны проверять, допустимые ли значения хранятся в таких переменных. Тем не менее переменные перечислимых типов предоставляют возможность такой проверки и потому часто бывают удобнее директив #define. Кроме того, отладчик может выводить значения перечислимых переменных в их символическом виде.

2.4. Объявления

Все переменные необходимо объявить до их использования, хотя некоторые объявления могут быть сделаны неявно — по контексту. В объявлении указываются тип и список из одной или нескольких переменных этого типа:

```
int lower, upper, step;
char c, line[1000];
```

Переменные можно распределять по объявлениям любым способом. Приведенные выше списки можно переписать и в таком виде:

```
int  lower;
int  upper;
int  step;
char c;
char line[1000];
```

Последняя запись занимает больше места, но в нее удобнее добавлять комментарии, а также вносить другие изменения.

Переменные можно инициализировать прямо в объявлениях. Если после имени поставить знак равенства и выражение, то значение этого выражения будет присвоено переменной при ее создании:

```
char  esc = '\\';
int   i = 0;
int   limit = MAXLINE + 1;
float eps = 1.0e-05;
```

Если переменная не автоматическая, то ее инициализация выполняется один раз — еще до начала выполнения программы. Инициализирующее выражение в этом случае должно быть константным. Автоматическая переменная, инициализируемая явным образом в объявлении, инициализируется всякий раз, когда управление передается в функцию или блок, где она объявлена. Ее инициализатором может быть любое выражение. Внешние и статические переменные по умолчанию инициализируются нулями. Автоматические переменные, для которых инициализирующие выражения не указаны явно, содержат случайные значения ("мусор").

К объявлению любой переменной можно добавить модификатор `const`, чтобы значение этой переменной впоследствии не менялось. В случае массива этот модификатор запрещает изменение любого из элементов.

```
const double e = 2.71828182845905;
const char msg[] = "warning: ";
```

Декларация `const` может употребляться с аргументами-массивами, указывая тем самым, что функция не изменяет передаваемый ей массив:

```
int strlen(const char[]);
```

Если предпринимается попытка изменить переменную, объявленную с модификатором `const`, ее результат будет зависеть от реализации языка.

2.5. Арифметические операции

В число двуместных арифметических операций входят +, -, *, / и взятие остатка от деления (%). При целочисленном делении дробная часть отбрасывается. Следующее выражение дает остаток от деления x на y и поэтому равно нулю, если x кратно y:

```
x % y
```

Например, год считается високосным, если его номер делится на 4, но не на 100, за исключением лет, которые делятся на 400 и по определению являются високосными. Вот как это проверить:

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    printf("%d is a leap year\n", year);
else
    printf("%d is not a leap year\n", year);
```

Операция % неприменима к числам типа float или double. Направление округления при операции / или знак результата при операции % для отрицательных аргументов зависят от системы, равно как и действия, предпринимаемые при переполнении или возникновении машинного нуля.

Операции + и - имеют одинаковый приоритет, который ниже, чем приоритет операций *, / и %. Приоритет трех последних, в свою очередь, ниже, чем приоритет одноместных операций + и -. Арифметические операции ассоциируются слева направо.

В табл. 2.1 приведены приоритеты и ассоциирование всех операций.

2.6. Операции отношения и логические операции

К операциям отношения принадлежат следующие:

```
>     >=     <     <=
```

Все они имеют одинаковый приоритет. Сразу после них по уровню приоритета следуют операции сравнения:

```
==     !=
```

Операции отношения и сравнения имеют более низкий приоритет, чем арифметические операции, так что выражение `i < lim-1` будет воспринято как `i < (lim-1)`, т.е. как и задумывалось.

Более интересны логические операции && и ||. Выражения, связанные этими знаками операций, вычисляются слева направо, и вычисление прекращается, как только установлена гарантированная истинность или ложность результата. Большинство программ на C использует это свойство. Например, так делается в цикле из функции ввода getline, приведенной в главе 1:

```
for (i=0; i<lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

Прежде чем вводить новый символ, необходимо проверить, достаточно ли места в массиве s, поэтому проверка i < lim-1 должна стоять *первой*. Более того, если эта проверка дает отрицательный результат, дальше не нужно считывать очередной символ.

Аналогично, было бы неправильно проверять c на равенство символу конца файла до вызова getchar, поэтому вызов и присваивание должны выполняться перед анализом символа c.

Приоритет операции && выше, чем ||, но обе имеют приоритет ниже, чем операции сравнения и отношения, поэтому для выражений наподобие следующих не нужны дополнительные скобки:

```
i<lim-1 && (c=getchar()) != '\n' && c != EOF
```

Однако, поскольку приоритет операции != выше, чем присваивания, скобки нужны в следующем выражении:

```
(c=getchar()) != '\n'
```

В противном случае не был бы достигнут желаемый результат: присваивание переменной c и последующее сравнение ее с '\n'.

По определению числовое значение логического выражения или сравнения равно 1, если выражение истинно, и 0 — если ложно.

Одноместная операция отрицания (!) превращает ненулевой операнд в 0, а нулевой — в 1. Обычно она используется для того, чтобы записывать такие конструкции:

```
if (!valid)
```

Без нее этот оператор выглядел бы так:

```
if (valid == 0)
```

Трудно сказать, какая из двух форм лучше. Конструкции типа !valid удобно читать ("если не valid", т.е. "если не годится"), но более сложные могут оказаться малопонятными.

Упражнение 2.2. Напишите цикл, эквивалентный приведенному выше циклу for, не используя операции && и ||.

2.7. Преобразование типов

Если операнды некоторой операции имеют различные типы, они преобразуются к одному типу с использованием нескольких правил. В целом единственные корректные автоматические преобразования — это те, в которых более "узкий" тип преобразуется в более "широкий" без потери информации, например при преобразовании целого числа в вещественное в выражениях наподобие f + i. Выражения, которые вообще не имеют смысла, например использование числа типа float в качестве индекса массива, запрещены. Выражения, в которых может произойти частичная потеря информации, например помещение числа длинного типа в переменную более короткого типа или вещественного числа в целую переменную, могут привести к выдаче предупреждения, но не запрещены.

Переменные типа char — это короткие целые числа, поэтому их можно свободно использовать в арифметических выражениях. Эта возможность позволяет во многих случаях удобно работать с символьной информацией. Ниже приведен пример очень упрощенной реализации функции atoi, которая преобразует символьную строку цифр в ее смысловой эквивалент — целое число.

```
/* atoi:  преобразует строку s в целое число */
int atoi(char s[])
{
    int i, n;

    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

Как говорилось в главе 1, следующее выражение дает числовое значение цифрового символа, помещенного в `s[i]`, поскольку коды символов `'0'`, `'1'` и т.д. образуют непрерывную последовательность в порядке возрастания.

Еще один пример преобразования из `char` в `int` — это функция `lower`, которая переводит один символ в нижний регистр *в символьном наборе ASCII*. Если символ не является буквой в верхнем регистре, функция `lower` возвращает его без изменений.

```
/* lower:  преобразует символ c в нижний регистр; только ASCII */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
       return c + 'a' - 'A';
    else
       return c;
}
```

В наборе ASCII этот алгоритм работает, потому что соответствующие буквы верхнего и нижнего регистра расположены на фиксированных расстояниях друг от друга, и каждый алфавит образует непрерывную последовательность — между `A` и `Z` нет ничего, кроме букв в алфавитном порядке. Однако эта закономерность неверна, например для символьного набора EBCDIC, и в этом наборе приведенная функция преобразовывала бы не только буквы.

Стандартный заголовочный файл `<ctype.h>`, описанный в приложении Б, содержит объявления семейства функций, выполняющих независимые от символьного набора проверки и преобразования. Например, функция `tolower(c)` возвращает аналог символа `c` в нижнем регистре, если `c` — буква верхнего регистра, т.е. это системно-независимый аналог приведенной выше функции `lower`. Аналогично можно заменить следующую непереносимую между символьными наборами проверку:

```
c >= '0' && c <= '9'
```

Переносимый код будет выглядеть так:

```
isdigit(c)
```

Далее для этих целей всегда будут использоваться функции, определенные в `<ctype.h>`.

В преобразовании символов в целые числа есть одна тонкость. Стандарт языка не определяет, должны ли переменные типа `char` быть знаковыми или беззнаковыми. Может ли при преобразовании `char` в `int` получиться отрицательное число? Ответ зависит от конкретной системы, отражая различия в аппаратной и программной архитектуре. В некоторых системах символ `char`, крайний левый бит которого равен 1, преобразуется в отрицательное число (это называется "расширением знака"). В других тип `char` преоб-

разуется в int путем добавления нулей на левом краю числа, и поэтому результат получается всегда положительным.

Определение языка C гарантирует, что любой печатаемый и видимый на экране символ из стандартного набора системы никогда не может быть отрицательным, так что и в выражениях такие символы всегда положительны. А вот произвольные последовательности битов, помещаемые в переменные типа char, могут в некоторых системах оказаться отрицательными, а в других — положительными. Если необходимо поместить данные несимвольного характера в переменные типа char, то для лучшей переносимости задайте модификатор signed или unsigned.

Выражения отношения наподобие i > j и логические выражения, соединенные знаками && и ||, по определению имеют значение 1, если они истинны, и 0, если ложны. Рассмотрим присваивание:

```
d = c >= '0' && c <= '9'
```

В результате его переменная d получает значение 1, если c — цифра, и 0 в противном случае. Однако такие функции, как isdigit, могут возвращать любое ненулевое значение в качестве результата "истина". В условных конструкциях операторов if, while, for и т.д. "истина" означает просто "не нуль", поэтому особой разницы нет.

Неявные арифметические преобразования в основном работают так, как можно ожидать от них по логике вещей. В целом, если операция наподобие + или *, имеющая два операнда (двуместная), связывает два аргумента разных типов, то более "узкий" тип *расширяется* до более "широкого", прежде чем выполняется собственно операция. Результат будет принадлежать к более "широкому" типу. В разделе 6 приложения A приведены точные правила таких преобразований. Но если среди аргументов нет чисел типа unsigned, то можно обойтись следующим неформальным набором правил.

- Если один из операндов имеет тип long double, другой преобразуется в long double.
- В противном случае, если один из операндов — double, другой преобразуется в double.
- В противном случае, если один из операндов — float, другой преобразуется в float.
- В противном случае char и short преобразуются в int.
- Затем, если среди операндов есть long, другой преобразуется в long.

Заметьте, что числа типа float в выражениях не преобразуются в double автоматически; это — изменение по сравнению с исходным определением языка. В основном все математические функции, объявленные в заголовочном файле <math.h>, используют двойную точность. Основная причина, по которой иногда используется тип float, заключается в том, чтобы сэкономить память в больших массивах, или, реже, сэкономить время в системах, имеющих особенно затратную вещественную арифметику с двойной точностью.

Правила преобразования становятся сложнее, если в выражениях участвуют аргументы типа unsigned. Проблема состоит в том, что сравнение между величинами со знаком и без знака является системно-зависимым, потому что зависит от размеров целочисленных переменных. Для примера предположим, что тип int имеет длину 16 бит, а long — 32 бита. Тогда -1L < 1U, поскольку 1U, будучи числом типа int, расширяет-

ся до signed long. Но при этом -1L > 1UL, потому что -1L расширяется до unsigned long и становится большим положительным числом.

Преобразования происходят также в ходе присваивания; значение с правой стороны преобразуется в тип переменной, стоящей слева, и результат будет иметь именно этот тип.

Символ типа char преобразуется в целое число, причем с расширением знака или без, как было описано раньше.

Длинные целые числа преобразуются в более короткие или величины типа char путем отбрасывания лишних старших битов. Поэтому после приведенных ниже операций значение c остается неизменным.

```
int  i;
char c;

i = c;
c = i;
```

Это справедливо независимо от того, применяется или нет расширение знака. А вот присваивание в обратном порядке может привести к потере информации.

Если переменная x имеет тип float, а i — тип int, то преобразования выполняются как при присваивании x = i, так и при i = x. При преобразовании float в int отбрасывается дробная часть. При преобразовании double в float значение числа либо усекается, либо округляется — это зависит от конкретной реализации языка.

Поскольку аргументы функций представляют собой выражения, преобразования типов могут иметь место и при передаче аргументов в функции. Если у функции отсутствует прототип, значения типа char и short становятся int, а float становится double. Вот почему мы объявляли аргументы функции как int и double, хотя вызывалась она с аргументами типа char и float.

Наконец, в любом выражении можно принудительно выполнить явное преобразование типов. Для этого используется одноместная операция под названием *приведение типов*:

```
(имя-типа) выражение
```

Этим способом *выражение* преобразуется в указанный *тип* по правилам, приведенным выше. Смысл приведения типов состоит в том, что *выражение* как бы присваивается переменной заданного типа, которая потом и используется вместо всей конструкции. Например, библиотечная функция sqrt ожидает аргумента типа double и выдает бессмыслицу, если случайно получает аргумент неподходящего типа. (Функция sqrt объявлена в заголовочном файле <math.h>.) Если n — целочисленная переменная, ее можно вначале привести к типу double, а затем уже передавать в функцию:

```
sqrt((double) n)
```

Обратите внимание, что приведение типа *порождает* новое значение нужного типа, никак *не изменяя* исходную переменную. Операция приведения имеет такой же высокий приоритет, как и другие одноместные операции, сводный список которых приведен в табл. 2.1.

Если аргументы объявлены в прототипе функции (как это и должно быть), то при любом вызове функции выполняется принудительное приведение типов. Например, пусть объявлен следующий прототип функции sqrt:

```
double sqrt(double);
```

Тогда при следующем вызове целое число 2 будет автоматически преобразовано в вещественное 2.0 без необходимости явного приведения:

```
root2 = sqrt(2);
```

Стандартная библиотека содержит переносимую между системами версию генератора псевдослучайных чисел, а также функцию для инициализации этого генератора. В первой из них иллюстрируется приведение типов:

```
unsigned long int next = 1;

/* rand:  возвращает псевдослучайное число от 0 до 32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand:  устанавливает инициализатор для rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Упражнение 2.3. Напишите функцию `htoi(s)`, которая преобразует строку шестнадцатеричных цифр (учитывая необязательные элементы `0x` или `0X`) в ее целочисленный эквивалент. В число допустимых цифр входят десятичные цифры от `0` до `9`, а также буквы `a–f` и `A–F`.

2.8. Операции инкрементирования и декрементирования

В языке C существуют две необычные операции — инкрементирование и декрементирование переменных. Операция инкрементирования `++` добавляет к своему операнду единицу, а операция декрементирования — отнимает. Инкрементирование уже широко использовалось в книге, например в таких конструкциях:

```
if (c == '\n')
    ++nl;
```

Необычный аспект этих операций состоит в том, что они могут использоваться как префиксные (т.е. стоять перед переменной, как в `++n`) или как постфиксные (после переменной: `n++`). В обоих случаях переменная получает единичное приращение. Однако в выражении `++n` переменная инкрементируется *до того*, как используется, а в выражении `n++` — *после того*. Это означает, что в контексте использования самой переменной, а не только эффекта приращения, выражения `++n` и `n++` различны. Пусть, например, `n` равна 5. Тогда следующий оператор делает `x` равным 5:

```
x = n++;
```

А этот оператор присваивает `x` значение 6:

```
x = ++n;
```

При этом n в обоих случаях становится равным 6. Операции инкрементирования и декрементирования применимы только к переменным; выражения наподобие (i+j)++ недопустимы.

В контексте, где значение переменной не используется, а только увеличивается, префиксная и постфиксная формы эквивалентны:

```
if (c == '\n')
    nl++;
```

Но бывают и ситуации, когда специально используются свойства одной из них. Для примера рассмотрим функцию squeeze(s, c), которая удаляет все вхождения символа c из строки s.

```
/* squeeze:  удаляет все символы c из строки s */
void squeeze(char s[], int c)
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

Всякий раз, когда встречается символ, отличный от c, он копируется в текущую позицию j, и только затем j инкрементируется, чтобы подготовиться к приему следующего символа. Эта форма в точности эквивалентна следующей:

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

Еще один похожий пример можно взять из функции getline, написанной нами в главе 1. Там присутствовала конструкция

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

которую можно заменить более компактной записью:

```
if (c == '\n')
    s[i++] = c;
```

В качестве третьего примера рассмотрим стандартную функцию strcat(s, t), которая присоединяет строку t к концу строки s (выполняет конкатенацию). При этом подразумевается, что свободного места достаточно для хранения удлиненной строки. В нашем варианте strcat не возвращает никакого значения, тогда как стандартная библиотечная версия возвращает указатель на итоговую строку.

```
/* strcat:  присоединяет строку t к концу s;
            в s должно быть достаточно места */
void strcat(char s[], char t[])
{
    int i, j;

    i = j = 0;
```

```
    while (s[i] != '\0')     /* поиск конца s */
        i++;
    while ((s[i++] = t[j++]) != '\0')  /* копирование t */
        ;
}
```

По мере копирования символов из `t` в `s` к индексам `i` и `j` применяется постфиксное приращение `++`, чтобы сдвинуть их в следующие по порядку позиции для следующего прохода цикла копирования.

Упражнение 2.4. Напишите альтернативную версию функции `squeeze(s1,s2)`, которая бы удаляла из строки `s1` все символы, встречающиеся в *строке* `s2`.

Упражнение 2.5. Напишите функцию `any(s1,s2)`, возвращающую номер первой позиции в строке `s1`, в которой находится какой-либо из символов строки `s2`, либо `-1`, если строка `s1` не содержит ни одного символа из `s2`. (Стандартная библиотечная функция `strpbrk` делает то же самое, но возвращает указатель на найденную позицию.)

2.9. Поразрядные операции

В языке C имеется шесть операций битового (поразрядного уровня), применимых только к целочисленным аргументам, т.е. `char`, `short`, `int` и `long`, как имеющим, так и не имеющим знака.

`&`	—	поразрядное И.
`|`	—	поразрядное включающее ИЛИ.
`^`	—	поразрядное исключающее ИЛИ.
`<<`	—	сдвиг влево.
`>>`	—	сдвиг вправо.
`~`	—	одноместное поразрядное дополнение до единицы.

Поразрядная операция И (`&`) часто используется для обнуления некоторого набора битов. Например, следующий оператор обнуляет все биты переменной `n`, кроме младших семи:

```
n = n & 0177;
```

Поразрядная операция ИЛИ (`|`), напротив, используется для того, чтобы сделать отдельные биты единицами. Следующий оператор делает единицами все биты, которые равны 1 в константе `SET_ON`:

```
x = x | SET_ON;
```

Операция исключающего ИЛИ (`^`) помещает единицу в позиции, где в операндах стоят различные биты, и нуль в те позиции, в которых биты операндов совпадают.

Необходимо отличать поразрядные операции `&` и `|` от логических операций `&&` и `||`, которые предполагают вычисление результата типа "истина-ложь" слева направо. Например, если `x` равно 1, а `y` равно 2, то `x & y` равно нулю, тогда как `x && y` равно единице.

Операции сдвига `<<` и `>>` выполняют сдвиг своего левого операнда соответственно влево и вправо на количество разрядов (битовых позиций), заданных правым операндом, который обязательно должен быть положительным. Таким образом, выражение `x << 2` сдвигает значение `x` влево на две разрядные позиции, заполняя вакантные биты нулями. Получается то же самое, как если бы умножить число на 4. Сдвиг вправо числа типа

unsigned всегда заполняет освободившиеся биты нулями. Сдвиг вправо числа со знаком в некоторых системах приводит к заполнению этих битов значением знакового бита ("арифметический сдвиг"), а в других — нулями ("логический сдвиг").

Одноместная операция ~ дает поразрядное дополнение целого числа до единицы, т.е. преобразует каждый единичный бит в нулевой и наоборот. Например, следующий оператор делает последние шесть бит переменной x равными 0:

```
x = x & ~077;
```

Заметьте, что x & ~077 не зависит от длины слова и поэтому предпочтительнее, чем, например, x & 0177700, где предполагается, что x — 16-разрядная величина. Переход к переносимой форме не требует дополнительных вычислительных затрат, поскольку ~077 — константа, вычисляемая при компиляции.

В качестве иллюстрации некоторых поразрядных операций рассмотрим функцию getbits(x,p,n), которая возвращает n-битовое поле, начинающееся с позиции p и сдвинутое к правому краю, из переменной x. Предполагается, что позиция 0 находится на правом краю числа и что n и p — корректные положительные числа. Например, getbits(x,4,3) возвращает три бита в позициях 4, 3 и 2, выровненные по правому краю:

```
/* getbits:  извлекает n бит, начиная с p-й позиции */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

Выражение x >> (p+1-n) сдвигает нужное поле к правому краю слова. Величина ~0 состоит целиком из единиц; сдвиг ее влево на n позиций с помощью выражения ~0 << n заполняет правые n битов нулями. Взятие дополнения с помощью операции ~ порождает маску, в которой крайние правые n бит заполнены единицами.

Упражнение 2.6. Напишите функцию setbits(x,p,n,y) так, чтобы она возвращала аргумент x, в котором n битов, начиная с позиции p, равны n крайним правым битам аргумента y, а остальные биты не тронуты.

Упражнение 2.7. Напишите функцию invert(x,p,n), возвращающую свой аргумент x, в котором обращены n бит, начиная с позиции p (т.е. единичные биты заменены нулевыми и наоборот), а остальные не тронуты.

Упражнение 2.8. Напишите функцию rightrot(x,n), которая бы возвращала значение своего целого аргумента x после его вращения вправо на n двоичных разрядов.

2.10. Операции с присваиванием и выражения с ними

Операторы присваивания, в которых переменная из левой части повторяется в правой, часто можно записать в более компактной форме. Возьмем такое выражение:

```
i = i + 2
```

Его можно переписать в сжатом виде:

```
i += 2
```

Знак += обозначает одну из *операций с присваиванием*.

Большинство двуместных операций (т.е. операций наподобие + и -, у которых два аргумента — левый и правый) имеют соответствующие им операции с присваиванием *оп*=, где *оп* — одна из следующих операций:

```
+   -   *   /   %   <<   >>   &   ^   |
```

Пусть *выр1* и *выр2* — два выражения и имеется запись

```
выр1 оп= выр2
```

Это эквивалентно следующей записи:

```
выр1 = (выр1) оп (выр2)
```

При этом выражение *выр1* вычисляется только один раз. Обратите внимание на скобки вокруг выражения *выр1*. Рассмотрим, например, такой оператор:

```
x *= y + 1
```

Он эквивалентен следующему:

```
x = x * (y + 1)
```

А вот при отсутствии скобок в определении можно было бы ошибочно подумать, что полная форма выглядит так:

```
x = x * y + 1
```

Для примера рассмотрим функцию `bitcount`, которая подсчитывает количество единичных битов в своем целочисленном аргументе:

```
/* bitcount:  подсчитывает единицы в двоичной записи x */
int bitcount(unsigned x)
{
    int b;

    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            b++;
    return b;
}
```

Объявление аргумента `x` с модификатором `unsigned` гарантирует, что при сдвиге вправо освободившиеся биты будут заполнены нулями независимо от системы, в которой выполняется программа.

Помимо такого второстепенного преимущества, как краткость записи, операции с присваиванием имеют то преимущество, что они соответствуют способу человеческого мышления. Когда мы говорим "прибавить 2 к `i`", то имеем в виду "увеличить `i` на 2", а не "извлечь `i`, прибавить 2 и поместить результат обратно в `i`". Поэтому выражение `i += 2` предпочтительнее, чем `i = i + 2`. Кроме того, в левой части присваивания встречаются очень сложные выражения:

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

В этом случае операция, объединенная с присваиванием, делает код более легким для понимания, поскольку читателю не придется прикладывать немалые усилия, чтобы проверить, действительно ли два выражения в левой и правой части идентичны. Операции с присваиванием даже могут помочь компилятору генерировать более оптимальный код.

Мы уже говорили о том, что оператор присваивания имеет значение, как любое другое выражение, и поэтому может сам входить в выражения. Вот самый распространенный пример такого рода:

```
while ((c = getchar()) != EOF)
```

Другие операции с присваиванием (+=, -= и т.д.) тоже могут фигурировать в выражениях, хотя и встречаются в них не так часто.

Во всех таких случаях тип выражения с присваиванием соответствует типу левого операнда, а его значение равно значению этого операнда после присваивания.

Упражнение 2.9. Благодаря свойствам двоичной системы счисления выражение `x &= (x-1)` удаляет самый правый единичный бит в переменной `x`. Докажите это. Воспользуйтесь этим фактом для того, чтобы написать более быструю версию функции `bitcount`.

2.11. Условные выражения

Следующий код берет большее из двух чисел, `a` и `b`, и помещает его в переменную `z`:

```
if (a > b)
    z = a;
else
    z = b;
```

Эту и аналогичные конструкции можно записать другим способом, прибегнув к условному выражению с трехместной операцией `?:`. Условное выражение имеет следующий вид:

выраж1 `?` *выраж2* `:` *выраж3*

Вначале вычисляется выражение *выраж1*. Если оно не равно нулю (т.е. истинно), то вычисляется выражение *выраж2*, которое и становится значением всей условной конструкции. В противном случае вычисляется *выраж3*, значение которого считается значением всего условного выражения. Всегда вычисляется только одно из *выраж2* и *выраж3*. Итак, чтобы присвоить переменной `z` большее из двух значений, `a` или `b`, необходимо записать следующее:

```
z = (a > b) ? a : b;     /* z = max(a, b) */
```

Следует отметить, что условное выражение является полноправным выражением и может использоваться везде, где допускается применение выражений. Если *выраж2* и *выраж3* имеют различные типы, то тип результата определяется правилами преобразования, рассмотренными ранее в этой главе. Например, если `f` имеет тип `float`, а `n` — тип `int`, то следующее выражение будет иметь тип `float` независимо от положительности значения `n`:

```
(n > 0) ? f : n
```

Первое выражение этой условной конструкции не обязательно заключать в скобки, поскольку приоритет операции `?:` очень низкий — чуть выше, чем у присваивания. Однако скобки желательны, поскольку они подчеркивают условную часть выражения.

Условные выражения часто позволяют сделать код очень кратким. Например, следующий цикл выводит `n` элементов массива по десять в одной строке, отделяя каждый столбец пробелом и завершая каждую строку (в том числе последнюю) символом конца строки:

```
for (i = 0; i < n; i++)
    printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

Символ конца строки выводится после каждого десятого элемента, а также после n-го. После всех остальных выводится один пробел. Код может показаться запутанным, но он намного компактнее, чем соответствующая конструкция `if-else`. Вот еще один характерный пример:

```
printf("You have %d item%s.\n", n, n==1 ? "" : "s");
```

Упражнение 2.10. Перепишите функцию `lower`, которая преобразует буквы в верхнем регистре к нижнему, с использованием условного выражения вместо конструкции `if-else`.

2.12. Приоритет и порядок вычисления

В табл. 2.1 сведены правила определения приоритета и порядка ассоциирования операций, в том числе и тех, о которых еще не говорилось. Операции, перечисленные в одной строке, имеют одинаковый приоритет. Строки расположены в порядке убывания приоритета; например, `*`, `/` и `%` имеют одинаковый приоритет, более высокий, чем у одноместных операций `+` и `-`. "Знак операции" `()` обозначает вызов функции. Операции `->` и `.` используются для обращения к элементам структур. Они будут рассмотрены в главе 6 вместе с операцией `sizeof` (вычисление размера объекта). В главе 5 рассматриваются операции `*` (ссылка по указателю) и `&` (получение адреса объекта), а в главе 3 — оператор "запятая".

Обратите внимание, что приоритет поразрядных операций `&`, `^` и `|` меньше, чем у операций сравнения `==` и `!=`. Поэтому выражения, в которых анализируются отдельные биты, необходимо брать в скобки, чтобы получить корректные результаты:

```
if ((x & MASK) == 0) ...
```

Таблица 2.1. Приоритет и ассоциирование операций

Операции	Ассоциирование
`() [] -> .`	Слева направо
`! ~ ++ -- + - * & (`*тип*`) sizeof`	Справа налево
`* / %`	Слева направо
`+ -`	Слева направо
`<< >>`	Слева направо
`< <= > >=`	Слева направо
`== !=`	Слева направо
`&`	Слева направо
`^`	Слева направо
`|`	Слева направо
`&&`	Слева направо

Окончание табл. 2.1

Операции	Ассоциирование
`||`	Слева направо
`?:`	Справа налево
`= += -= *= /= %= &= ^= |= <<= >>=`	Справа налево
`,`	Слева направо

Как и в большинстве языков, в C в основном не указывается порядок вычисления операндов в операциях. (Исключениями являются `&&`, `||`, `?:` и ‘,’.) Например, в следующем операторе первой может вызываться как функция `f()`, так и функция `g()`:

```
x = f() + g();
```

Если в одной из функций, `f` или `g`, модифицируется переменная, от которой зависит работа другой функции, то итоговое значение `x` будет зависеть от порядка вызова. Чтобы гарантировать определенную последовательность операций, следует поместить промежуточный результат в какую-нибудь временную переменную.

Аналогичным образом, не определяется и порядок, в котором вычисляются аргументы функции при ее вызове. Поэтому следующий оператор может дать различные результаты при трансляции разными компиляторами:

```
printf("%d %d \n", ++n, power(2, n));  /* НЕПРАВИЛЬНО */
```

Результат будет зависеть от того, получает ли `n` приращение до или после вызова функции `power`. Чтобы решить проблему, достаточно записать так:

```
++n;
printf("%d %d \n", n, power(2, n));
```

Вызовы функций, вложенные операторы присваивания, операции инкрементирования и декрементирования имеют “побочные эффекты” — в ходе вычисления выражений могут непредсказуемо модифицироваться некоторые переменные. В любом выражении, в котором возможны побочные эффекты, присутствует зависимость от порядка использования переменных в вычислениях. Одна из типичных ситуаций представлена следующим оператором:

```
a[i] = i++;
```

Проблема связана с тем, что неизвестно, какое значение индекса `i` используется для обращения к массиву — старое или новое. Компиляторы могут интерпретировать этот оператор по-разному и выдавать различные результаты в зависимости от своей интерпретации. В стандарте такие вопросы намеренно оставлены открытыми. Когда именно побочным эффектам (присваивание переменным) следует иметь место внутри выражений — ответ на этот вопрос оставлен на усмотрение авторов компиляторов, поскольку наилучший порядок операций сильно зависит от аппаратно-системной архитектуры. (В стандарте все-таки определено, что побочные эффекты воздействуют на аргументы *до* вызова функций, но в приведенном выше примере с функцией `printf` не поможет и эта информация.)

Таким образом, из всего вышесказанного можно сделать вывод, что писать код, зависящий от порядка вычисления аргументов или операндов, — это плохой стиль в любом языке программирования. Разумеется, необходимо знать, каких вещей избегать, однако если вы вообще *не знаете*, *как* те или иные операции выполняются в разных системах, у вас не возникнет искушения воспользоваться свойствами конкретной реализации.

Глава 3

Управляющие конструкции

Управляющие операторы и конструкции языка задают порядок, в котором выполняются вычислительные операции программы. В примерах предыдущих глав уже демонстрировались самые необходимые управляющие конструкции. В этой главе будут описаны все остальные операторы управления, существующие в языке, а также сделаны уточнения, касающиеся уже известных.

3.1. Операторы и блоки

Выражение наподобие `x = 0`, `i++` или `printf(...)` становится *оператором*, если после него поставить точку с запятой:

```
x = 0;
i++;
printf(...);
```

В языке C точка с запятой является элементом оператора и его завершающей частью, а не разделителем операторов, как в языке Pascal.

Фигурные скобки, { и }, служат для группировки объявлений и операторов в *составные операторы*, или *блоки*, синтаксически эквивалентные одному оператору. Фигурные скобки, окружающие операторы тела функции, — это самый очевидный пример такого блока, а еще один — это скобки вокруг группы из нескольких операторов после `if`, `else`, `while` или `for`. (Переменные можно объявлять в любом блоке; об этом будет сказано подробнее в главе 4.) После правой скобки, закрывающей блок, точка с запятой не ставится.

3.2. Оператор `if-else`

Оператор `if-else` выражает процесс принятия альтернативных решений. Его формальный синтаксис таков:

```
if (выражение)
    оператор1
else
    оператор2
```

Часть, начинающаяся со слова `else`, необязательна. Вначале вычисляется *выражение*; если оно истинно (т.е. имеет ненулевое значение), то выполняется *оператор1*. Если оно ложно (т.е. имеет нулевое значение) и присутствует блок `else`, то выполняется *оператор2*.

Поскольку в операторе `if` всего-навсего анализируется числовое значение выражения, можно несколько упростить отдельные конструкции. Самый очевидный пример — упрощение следующей конструкции:

```
if (выражение != 0)
```

Более краткая форма выглядит так:

```
if (выражение)
```

Такая запись часто оказывается более естественной и понятной, хотя иногда может вызвать путаницу.

Поскольку блок `else` в операторе `if` необязателен, его отсутствие в серии вложенных `if` порождает двусмысленность. Но проблема решается благодаря тому, что `else` всегда ассоциируется с ближайшим предыдущим оператором `if` без `else`. Рассмотрим пример:

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

где `else` соответствует внутреннему `if`, что и показано с помощью отступа. Если программист имел в виду другое, он должен принудительно изменить принадлежность блока `else` с помощью фигурных скобок:

```
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
```

Особенно вредоносной эта двусмысленность бывает в таких ситуациях:

```
if (n >= 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
        }
else        /* ОШИБКА */
    printf("error - n is negative\n");
```

Отступ ясно показывает, чего хочет программист, но компилятор этого не понимает и в результате ассоциирует `else` с ближайшим вложенным `if`. Такие ошибки найти нелегко, поэтому рекомендуется всегда использовать фигурные скобки для выделения вложенных операторов `if`.

Кстати, заметьте, что после `z = a` стоит точка с запятой:

```
if (a > b)
    z = a;
else
    z = b;
```

По грамматическому определению после ключевого слова `if` стоит *оператор*, а выражение-оператор наподобие `z = a` всегда оканчивается точкой с запятой.

3.3. Конструкция `else-if`

Следующая конструкция встречается так часто, что заслуживает отдельного рассмотрения:

```
if (выражение)
    оператор
else if (выражение)
    оператор
else if (выражение)
    оператор
else if (выражение)
    оператор
else
    оператор
```

Эта последовательность операторов `if` представляет собой самый общий способ записи принятия многовариантного решения. Перечисленные *выражения* вычисляются по порядку; если какое-нибудь из них оказывается истинным, то выполняется ассоциированный с ним *оператор*, и цепочка проверки условий прерывается. Как всегда, *оператор* может представлять собой как отдельный оператор, так и блок в фигурных скобках.

Последний фрагмент данной конструкции (`else` без условия) обрабатывает вариант "ничто из перечисленного" — выполняет операции "по умолчанию" в случае, если ни одно из условий не удовлетворяется:

```
else
    оператор
```

Иногда этот вариант не нужен — для него не предусмотрены никакие операции. Тогда и завершающий фрагмент можно опустить или же воспользоваться им для обработки ошибок — перехвата "невозможной" ситуации.

Проиллюстрируем выбор решения из трех возможных на примере функции двоичного поиска (дихотомии), которая определяет, встречается ли некоторое число `x` в упорядоченном массиве `v`. Элементы `v` должны располагаться в порядке возрастания. Функция возвращает позицию в массиве (число от `0` до `n-1`), если `x` в нем встречается, или `-1`, если не встречается.

Вначале процедура двоичного поиска сравнивает значение `x` со средним элементом массива `v`. Если `x` меньше, чем средний элемент, дальнейший поиск выполняется в нижней части таблицы, в противном случае — в верхней. В любом из случаев следующий шаг состоит в том, чтобы сравнить `x` со средним элементом выбранной половины. Процесс деления диапазона пополам продолжается, пока не будет найдено требуемое значение или пока диапазон не станет пустым.

```
/* binsearch: поиск x в v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
```

```
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else    /* найдено соответствие */
            return mid;
    }
    return -1;  /* нет соответствия */
}
```

Основной вопрос, который здесь нужно выяснить, — является ли `x` меньшим, бо́льшим или равным среднему элементу `v[mid]` текущего диапазона. Для принятия таких решений удобно применять именно конструкцию `else-if`.

Упражнение 3.1. В нашем двоичном поиске каждый цикл содержит две проверки, тогда как достаточно было бы одной (зато взамен их потребовалось бы больше снаружи цикла). Напишите версию функции с одной проверкой внутри цикла и сравните быстродействие (затраты времени).

3.4. Оператор `switch`

Оператор `switch` используется для выбора одного из нескольких вариантов действий в зависимости от того, с какой из набора целочисленных *констант* совпадает значение некоторого выражения. В зависимости от найденного соответствия выполняется ветвление программы.

```
switch (выражение) {
    case константа-выраж: операторы
    case константа-выраж: операторы
    default: операторы
}
```

Каждый из вариантов (блоков `case`) имеет метку в виде константы с целочисленным значением или константного выражения. Если одна из меток совпадает со значением *выражения*, управление передается операторам после этой метки. Все выражения после `case` должны быть различными. Блок `default` выполняется в том случае, если не найдено ни одного соответствия в блоках `case`. Наличие блока `default` не обязательно; если его нет и не найдено ни одного соответствия, то не будут выполнены никакие операции. Блоки `case` и `default` могут следовать друг за другом в любом порядке.

В главе 1 рассматривалась программа для подсчета количества отдельных цифр, пробелов и всех остальных символов с использованием последовательности `if ... else ... if ... else`. Ниже эта программа переписана с использованием оператора `switch`.

```
#include <stdio.h>

main()  /* подсчет цифр, пробелов, других символов */
{
    int c, i, nwhite, nother, ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
        ndigit[i] = 0;
```

```
    while ((c = getchar()) != EOF) {
        switch (c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            ndigit[c-'0']++;
            break;
        case ' ':
        case '\n':
        case '\t':
            nwhite++;
            break;
        default:
            nother++;
            break;
        }
    }
    printf("digits =");
    for (i = 0; i < 10; i++)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n",
        nwhite, nother);
    return 0;
}
```

Оператор `break` инициирует немедленный выход из оператора `switch`. Поскольку блоки `case` — это, по сути, всего лишь метки, после выполнения кода в одном из них продолжается выполнение кода следующего (*насквозь* через блоки `case`), пока не будет предпринята какая-нибудь операция для выхода из `switch`. Для этого чаще всего используются операторы `break` и `return`. Оператор `break` также можно использовать для принудительного выхода из циклов `while`, `for` и `do`, о чем будет сказано позже.

Возможность сквозного выполнения блоков `case` имеет как достоинства, так и недостатки. Достоинство состоит в том, что несколько возможных вариантов (`case`) можно ассоциировать с одним и тем же набором операций, как это делалось в приведенном примере. Но при этом подразумевается, что для обычного, а не сквозного выполнения каждый блок `case` должен оканчиваться оператором `break`. Сквозной метод не способствует устойчивости работы программы, поскольку при ее доработке могут возникнуть разные побочные эффекты. За исключением использования нескольких меток для одной и той же операции, сквозного выполнения следует по возможности избегать, а все случаи, когда оно все же применяется, тщательно комментировать.

С точки зрения хорошего стиля программирования рекомендуется ставить оператор `break` даже в конце последнего блока (в данном случае `default`), хотя в этом нет необходимости. Со временем, когда в конец оператора `switch` добавится еще один блок `case`, этот нехитрый прием подстраховки спасет вас от лишних неприятностей.

Упражнение 3.2. Напишите функцию под именем `escape(s,t)`, которая бы преобразовывала символы наподобие конца строки и табуляции в управляющие последовательности языка C, такие как `\n` и `\t`, в процессе копирования строки `t` в строку `s`. Воспользуйтесь оператором `switch`. Напишите функцию также и для противоположной операции — преобразования символических управляющих последовательностей в фактические управляющие символы.

3.5. Циклы — while и for

Циклы while и for нам уже встречались. Первый из них устроен следующим образом:

```
while (выражение)
    оператор
```

Здесь вначале вычисляется *выражение*. Если оно не равно нулю, то выполняется *оператор*, а затем *выражение* вычисляется снова. Эти действия повторяются до тех пор, пока *выражение* не станет равным нулю. После этого управление передается в точку программы, следующую за *оператором*.

Оператор for имеет следующее устройство:

```
for (выраж1; выраж2; выраж3)
    оператор
```

Эта конструкция эквивалентна следующей:

```
выраж1;
while (выраж2) {
    оператор;
    выраж3;
}
```

Исключением является применение операции continue, которая рассматривается в разделе 3.7.

С точки зрения грамматики все три компонента в заголовке цикла for являются выражениями. Чаще всего *выраж1* и *выраж3* являются операторами присваивания или вызовами функций, а *выраж2* — выражением отношения или логическим выражением. Любую из трех частей можно опустить, хотя точки с запятыми должны остаться на своих местах. Если опустить *expr1* или *expr3*, то соответствующие операции не будут выполняться. Если же опустить проверку условия, *expr2*, то по умолчанию считается, что условие продолжения цикла всегда истинно, и следующая конструкция станет бесконечным циклом (зациклится):

```
for (;;) {
    ...
}
```

Подразумевается, что такой цикл должен прерываться другими способами, например с помощью операторов break или return.

Какой из двух циклов использовать, while или for, — это в значительной мере дело вкуса. Например, в следующем цикле нет ни инициализации, ни модификации переменных цикла, поэтому для него естественной является конструкция while:

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ;   /* пропуск символов пустого пространства */
```

Цикл for следует предпочесть тогда, когда есть простая инициализация и инкрементирование переменных цикла, поскольку все управляющие элементы цикла в этом случае удобно сгруппированы вместе в его заголовке. Вот наиболее очевидный вариант:

```
for (i = 0; i < n; i++)
    ...
```

Это стандартная конструкция (идиома) языка C для обработки n элементов массива — аналог оператора DO в языке Fortran или for в Pascal. Но эта аналогия не является

полной, поскольку счетчик и пределы цикла for в С можно изменять как угодно внутри самого цикла, причем переменная i сохраняет свое значение при выходе из цикла по любой причине. Поскольку компоненты цикла for представляют собой произвольные выражения, его возможности отнюдь не исчерпываются перебором арифметической прогрессии. Тем не менее перегружать инициализацию и инкрементирование в цикле for операциями, не относящимися к делу, — это плохой стиль. В заголовке цикла следует выполнять только операции по непосредственному управлению этим циклом.

Рассмотрим более объемистый пример — версию функции atoi для преобразования строки в ее числовой эквивалент. Этот вариант будет несколько более общим, чем в главе 2; он сможет пропускать символы пустого пространства в начале строки, а также необязательные знаки + и -. (В главе 4 будет демонстрироваться функция atof, которая выполняет аналогичное преобразование над вещественными числами.)

Структура программы отражает формат ее входных данных:

```
пропустить символы пустого пространства, если таковые есть
обработать знак, если он есть
прочитать целую часть и преобразовать ее в число
```

На каждом этапе выполняются свои операции, после чего строка готова к выполнению следующего. Обработка строки заканчивается, как только встретился первый символ, недопустимый в целом числе.

```
#include <ctype.h>

/* atoi: преобразует строку s в целое число; версия 2 */
int atoi(char s[])
{
    int i, n, sign;

    for (i = 0; isspace(s[i]); i++) /* пропуск пробелов и т.п. */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-') /* пропуск знака */
        i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

В стандартной библиотеке имеется более тщательно проработанная функция strtol для преобразования строк в длинные целые числа. (См. раздел 5 приложения Б)

Преимущества группировки всех управляющих элементов цикла в одном месте становятся еще более очевидными во вложенных циклах. Ниже приведена функция для сортировки массива целых чисел по алгоритму Шелла. Основная идея алгоритма, разработанного Д.Л. Шеллом (D.L. Shell) в 1959 году, заключается в том, что на первом этапе сортировки сравниваются далеко отстоящие друг от друга элементы, а не соседние, как в более простых методах. Это позволяет быстро ранжировать большие массы неупорядоченных элементов, чтобы на следующих этапах оставалось меньше работы. Интервал между сравниваемыми элементами постепенно уменьшается до единицы, и в этот момент сортировка сводится к обмену соседних элементов — методу пузырьков.

```
/* shellsort:  сортирует v[0]...v[n-1] в порядке возрастания */
void shellsort(int v[], int n)
{
```

```
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}
```

В этой функции — три вложенных цикла. Внешний цикл управляет расстоянием между сравниваемыми элементами, сжимая его в два раза (начиная с `n/2`) при каждом проходе, пока он не становится равным нулю. Средний цикл выполняет перебор по всем элементам. Внутренний цикл сравнивает все пары элементов, отделенных друг от друга текущим расстоянием `gap`, и меняет местами те из них, которые стоят в неправильном порядке. Поскольку `gap` постепенно уменьшается до нуля, все элементы постепенно упорядочиваются. Обратите внимание, что конструкция цикла `for` позволяет придать внешнему циклу ту же форму, что и внутренним, хотя он и не представляет собой арифметическую прогрессию.

Еще одним знаком операции языка C является запятая, которая чаще всего находит применение в операторе `for`. Пара выражений, разделенных запятой, вычисляется слева направо; тип и значение результата равны типу и значению правого операнда. Таким образом, в разделы заголовка оператора `for` можно вставлять по несколько выражений — например, для параллельной обработки двух индексов. Для иллюстрации приведем функцию `reverse(s)`, которая меняет порядок символов в строке `s` на обратный прямо на месте, без дополнительных буферных строк:

```
#include <string.h>

/* reverse:  обращает порядок символов в строке s */
void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

Однако запятые, которые разделяют аргументы функций, переменные в объявлениях и т.п., *не являются* знаками операций и не гарантируют порядок вычисления слева направо.

Операцию "запятая" следует использовать как можно реже и с осторожностью. Наиболее уместно ее применение в конструкциях, где объединяются в единое целое связанные друг с другом операции, — как, например, в теле цикла `for` в функции `reverse`, или в макросах, где многошаговые вычисления объединяются в одну операцию. Запятую вполне можно применить при обмене местами элементов в функции `reverse`, поскольку этот обмен можно представить себе единой операцией:

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
```

Упражнение 3.3. Напишите функцию `expand(s1,s2)`, которая бы разворачивала сокращенную запись наподобие `a-z` в строке `s1` в полный список `abc...xyz` в строке `s2`. Учитывайте буквы в любом регистре, цифры, а также записи вида `a-b-c`, `a-z0-9` и `-a-z`. Сделайте так, чтобы знаки `-` в начале и в конце строки воспринимались буквально, а не как символы развертывания.

3.6. Циклы — do-while

Как было сказано в главе 1, циклы `while` и `for` выполняют проверку условия в начале. В противоположность им третий вид циклов в C — конструкция `do-while` — проверяет условие в конце, *после* выполнения тела цикла. Таким образом, тело цикла всегда выполняется как минимум один раз.

Вот синтаксис оператора `do`:

```
do
    оператор
while (выражение);
```

Здесь сначала выполняется *оператор*, затем вычисляется *выражение*. Если оно истинно (не равно нулю), то снова выполняется *оператор*, и т.д. Как только выражение становится ложным, выполнение цикла прекращается. За исключением способа проверки выражения, цикл `do-while` аналогичен оператору `repeat-until` в языке Pascal.

Опыт показывает, что цикл `do-while` используется значительно реже, чем `while` и `for`. Тем не менее время от времени он оказывается полезным, как в приведенной ниже функции `itoa`, преобразующей целое число в строку символов (т.е. выполняющей операцию, противоположную `atoi`). Сделать это несколько сложнее, чем может показаться на первый взгляд, поскольку самый простой метод генерирования цифр расставляет их в обратном порядке. В этой функции мы решили сначала сгенерировать строку с обратным порядком цифр, а потом выполнить перестановку:

```
/* itoa:  преобразует число n в строку символов s */
void itoa(int n, char s[])
{
    int i, sign;

    if ((sign = n) < 0)  /* записываем знак */
        n = -n;          /* делаем число положительным */
    i = 0;
    do {       /* генерируем цифры в обратном порядке */
        s[i++] = n % 10 + '0';   /* извлекаем цифру */
    } while ((n /= 10) > 0);     /* удаляем ее */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

Здесь цикл do-while необходим (или по крайней мере уместен), поскольку в строку s в любом случае следует поместить хотя бы один символ, даже если n равно нулю. Также мы заключили в фигурные скобки единственный оператор, составляющий тело цикла do-while, хотя в этих скобках и не было необходимости, чтобы поспешный читатель не принял блок while в конце do за *заголовок* отдельного цикла while.

Упражнение 3.4. В представлении чисел с помощью дополнения до двойки наша версия функции itoa не умеет обрабатывать самое большое по модулю отрицательное число, т.е. значение n, равное $-(2^{\text{длина\_слова}} - 1)$. Объясните, почему это так. Доработайте функцию так, чтобы она выводила это число правильно независимо от системы, в которой она работает.

Упражнение 3.5. Напишите функцию itob(n,s,b), которая бы преобразовывала целое число n в его символьное представление в системе счисления с основанием b и помещала результат в строку s. Например, вызов itoa(n,s,16) должен представлять n в виде шестнадцатеричного числа.

Упражнение 3.6. Напишите версию itoa, принимающую три аргумента вместо двух. Третий аргумент будет минимальной шириной поля; преобразованное число следует дополнить пробелами слева, если оно недостаточно длинное, чтобы заполнить это поле.

3.7. Операторы break и continue

Иногда бывает удобно выйти из цикла другим способом, отличным от проверки условия в его начале или в конце. Оператор break вызывает принудительный выход из циклов for, while и do — аналогично выходу из оператора switch. Выход выполняется из ближайшего (самого внутреннего) цикла или оператора switch.

Приведенная ниже функция trim удаляет пробелы, символы табуляции и конца строки из "хвоста" символьной строки, используя оператор break для выхода из цикла, как только найден самый правый символ, не являющийся одним из перечисленных "пустых" символов.

```
/* trim:  удаляет символы пустого пространства из конца строки */
int trim(char s[])
{
    int n;

    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

Функция strlen возвращает длину строки. Цикл for начинает работу с конца строки и продвигается по ней назад, разыскивая первый символ, не являющийся одним из трех символов пустого пространства. Работа цикла заканчивается, как только такой символ найден или как только n станет отрицательным (т.е. пройдена вся строка). Рекомендуется проверить, правильно ли ведет себя функция даже в тех случаях, когда строка пустая или содержит только символы пустого пространства.

Оператор continue напоминает break, но используется реже; он передает управление на следующую итерацию (проход) ближайшего цикла for, while или do. В цикле while или do это означает немедленную проверку условия, тогда как в цикле for дополнительно выполняется инкрементирование. Оператор continue применим только к циклам, но не к оператору switch. Если поставить continue внутри switch, в свою очередь находящегося внутри цикла, то управление будет передано на следующий проход этого цикла.

Например, следующий фрагмент кода обрабатывает только неотрицательные элементы массива a; отрицательные просто пропускаются:

```
for (i = 0; i < n; i++) {
    if (a[i] < 0)   /* пропускаем отрицательные элементы */
        continue;
    ... /* обрабатываем положительные элементы */
}
```

Оператор continue часто используется там, где следующая за ним часть цикла слишком длинная и сложная, так что выделение ее отступами в условном операторе ухудшает удобочитаемость.

3.8. Оператор goto и метки

В языке C имеется оператор goto — бездонный источник потенциальных неприятностей — и метки для перехода с его помощью. В техническом плане оператор goto никогда не бывает необходим, и на практике почти всегда легче обходиться без него. В этой книге он вообще не используется.

Тем не менее есть несколько ситуаций, в которых и goto может найти свое место под солнцем. Наиболее распространенная — это необходимость прекратить работу управляющей структуры с большим количеством вложений, например выйти сразу из двух и более вложенных циклов. Оператор break тут не применим непосредственно, поскольку он обеспечивает выход только из внутреннего, ближайшего цикла. Поэтому получаем вот что:

```
    for ( ... )
        for ( ... ) {
            ...
            if (disaster)
                goto error;
        }
    ...
error:
    решить проблему
```

Такая конструкция бывает удобной, если код для обработки ошибок нетривиален, а сами ошибки могут происходить в разных местах.

Метка для перехода имеет ту же форму, что и имя переменной. После нее ставится двоеточие. Ее можно ставить перед любым оператором в той же функции, в которой находится соответствующий goto. Область действия метки — вся функция.

В качестве следующего примера рассмотрим задачу определения того, имеют ли массивы a и b хотя бы один общий элемент. Вот один из возможных вариантов решения:

```
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            if (a[i] == b[j])
                goto found;
    /* общие элементы не найдены */
    ...
found:
    /* есть общие элементы:  a[i] == b[j] */
```

Код, в котором есть оператор `goto`, всегда можно переписать без него, хотя, возможно, придется добавить дополнительную проверку или переменную. Например, решение той же задачи без `goto` выглядит так:

```
found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
            found = 1;
if (found)
    /* элемент найден:  a[i-1] == b[j-1] */
    ...
else
    /* нет общих элементов */
    ...
```

За немногими исключениями наподобие приведенного здесь примера код, основанный на переходах с помощью оператора `goto`, труднее понимать и дорабатывать, чем код без этих операторов. Хотя мы и не утверждаем этого категорически, все же оператор `goto` следует стараться употреблять как можно реже или вообще никогда.

Глава 4

Функции и структура программы

Функции не только помогают разбить большие вычислительные задачи на набор маленьких, но и позволяют программистам пользоваться разработками предшественников, а не начинать все заново. Хорошо написанные функции скрывают подробности своей работы от тех частей программы, которым их знать не положено, таким образом проясняя задачу в целом и облегчая процесс внесения исправлений и дополнений.

Язык С построен так, чтобы сделать использование функций максимально простым и эффективным. Обычно программы на С состоят из множества мелких функций, а не нескольких сравнительно больших. Программа может храниться в одном или нескольких файлах исходного кода, которые можно компилировать по отдельности, а затем собирать вместе, добавляя к ним предварительно скомпилированные функции из библиотек. Здесь этот процесс не будет освещаться подробно, поскольку его частности сильно меняются от системы к системе.

Объявление и определение функций — это тот раздел языка С, который претерпел самые заметные изменения с введением стандарта ANSI. В главе 1 говорилось о том, что теперь стало возможным объявлять типы аргументов при объявлении функции. Изменился также синтаксис определения функций с целью согласовать объявления с определениями. Благодаря этому компилятор может обнаружить гораздо больше ошибок, ранее ему недоступных. Более того, при правильном объявлении аргументов соответствующие приведения типов выполняются автоматически.

Сейчас стандарт задает более четкие правила области действия имен; в частности, выдвигается требование, чтобы каждый внешний объект определялся в программе ровно один раз. Более общей стала инициализация — теперь можно инициализировать автоматические массивы и структуры.

Усовершенствованию подвергся также и препроцессор С. В число новых возможностей препроцессора входят расширенный набор директив условной компиляции, новый метод создания символьных строк в кавычках из аргументов макросов, а также более совершенный контроль над процессом раскрытия макросов.

4.1. Основы создания функций

Для начала спроектируем и напишем программу, выводящую каждую из строк своих входных данных, в которой встречается определенный "шаблон" — заданная строка символов. (Это частный случай программы под названием `grep` из системы Unix.) Пусть, например, задан поиск строки `"ould"` в следующем наборе строк:

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
```

```
Would not we shatter it to bits - and then
Re-mould it nearer to the Heart's Desire![1]
```

На выходе программы получится следующее:

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits - and then
Re-mould it nearer to the Heart's Desire!
```

Задача естественным образом раскладывается на три части:

```
while (на вход поступает очередная строка)
    if (строка содержит заданный шаблон)
        вывести ее
```

Хотя, разумеется, весь код для решения этих задач можно поместить в функцию `main`, лучше будет все-таки разделить его на части и организовать в виде отдельных функций, чтобы воспользоваться структурированностью программы. С маленькими частями удобнее иметь дело, чем с большим целым, поскольку в функциях можно скрыть несущественные детали реализации, при этом избежав риска нежелательного вмешательства одной части программы в другую. К тому же отдельные части могут даже оказаться полезными для других задач.

Первую часть нашего алгоритма (ввод строки) реализует функция `getline`, написанная еще в главе 1, а третью (вывод результата) — функция `printf`, которую давно написали за нас. Таким образом, нам осталось только написать функцию, которая бы определяла, содержит ли текущая строка заданный шаблон-образец.

Эта задача будет решена с помощью функции `strindex(s,t)`, которая возвращает позицию (или индекс) в строке `s`, с которой начинается строка `t`, либо `-1`, если строка `s` не содержит `t`. Поскольку массивы в C начинаются с нулевой позиции, индексы могут быть нулевыми или положительными, а отрицательное значение наподобие `-1` удобно для сигнализации ошибки. Если впоследствии понадобится применить более расширенный алгоритм поиска образцов в тексте, в программе достаточно будет заменить только `strindex`, а остальной код не надо будет изменять. (В стандартной библиотеке есть функция `strstr`, аналогичная по своему назначению `strindex`, только она возвращает не индекс, а указатель.)

Имея перед собой общую схему алгоритма, запрограммировать его детали уже не так сложно. Ниже приведен полный текст программы, по которому можно судить, как все его части сведены воедино. Образец, который можно разыскивать в тексте, пока что представляется литеральной строкой, что никак нельзя считать достаточно общим механизмом реализации. Вскоре будет рассмотрен вопрос инициализации символьных массивов, а в главе 5 рассказывается, как сделать задаваемый образец параметром, который бы указывался при запуске программы. В программу также включена несколько модифицированная версия функции `getline`; возможно, вам будет поучительно сравнить ее с приведенной в главе 1.

```
#include <stdio.h>
#define MAXLINE 1000    /* максимальная длина входной строки */

int getline(char line[], int max);
```

[1] Когда к жизни Любовь меня в мир призвала,
Мне уроки любви она сразу дала,
Ключ волшебный сковала из сердца частичек
И к сокровищам духа меня привела. (*Омар Хайям, пер. Н. Тенигиной*)

```
int strindex(char source[], char searchfor[]);

char pattern[] = "ould";    /* образец для поиска */

/* поиск всех строк, содержащих заданный образец */
main()
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}

/* getline: считывает строку в s, возвращает ее длину */
int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex: возвращает индекс строки t в s, -1 при отсутствии */
int strindex(char s[], char t[])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

Каждое из определений функций имеет следующую форму:

```
тип-возвращ-знач имя-функции(объявления аргументов)
{
    объявления и операторы
}
```

Различные части этого определения могут отсутствовать; самая минимальная функция имеет вид

```
dummy() {}
```

Она ничего не делает и ничего не возвращает. Такого рода функция, не выполняющая никаких операций, часто бывает полезна как заменитель ("заглушка") в ходе разработки программы. Если тип возвращаемого значения опущен, по умолчанию подразумевается `int`.

Всякая программа является всего лишь набором определений переменных и функций. Функции обмениваются данными посредством передачи аргументов и возвращения значений, а также через внешние переменные. Функции могут следовать друг за другом в файле исходного кода в любом порядке, и текст программы можно разбивать на любое количество файлов, но при этом запрещено делить текст функции между файлами.

Оператор `return` — это механизм возвращения значений из вызываемой функции в вызывающую. После `return` может идти любое выражение:

`return` *выражение*

По мере необходимости *выражение* преобразуется в тип, возвращаемый функцией согласно ее объявлению и определению. Часто *выражение* заключают в круглые скобки, но это не обязательно.

Вызывающая функция имеет полное право игнорировать возвращаемое значение. Более того, после `return` вовсе не обязано стоять выражение; в этом случае в вызывающую функцию ничего не передается. Точно так же управление передается в вызывающую функцию без возврата значения, если достигнут конец тела функции, т.е. закрывающая правая фигурная скобка. Если функция возвращает значение из одного места и не возвращает из другого, это допускается синтаксисом языка, но может указывать на ошибку. Во всяком случае, если функция не возвращает значения, хотя по объявлению должна это делать, в передаваемых ею данных гарантированно будет содержаться "мусор" — случайные числа.

Программа поиска образцов в строках возвращает из `main` результат работы программы — количество найденных совпадений. Это число может использоваться операционной средой, которая вызвала программу.

Процедура компиляции и компоновки программы на C, хранящейся в нескольких файлах исходного кода, сильно отличается в разных системах. Например, в системе Unix все это делает команда `cc`, уже упоминавшаяся в главе 1. Предположим, что три разные функции программы хранятся в трех файлах: `main.c`, `getline.c` и `strindex.c`. Для компиляции и компоновки их в программу применяется следующая команда:

```
cc main.c getline.c strindex.c
```

Она помещает объектный код, полученный в результате компиляции, в файлы `main.o`, `getline.o` и `strindex.o`, а затем компонует их все в выполняемую программу `a.out`. Если, скажем, в файле `main.c` встретилась синтаксическая ошибка, этот файл можно будет перекомпилировать заново и скомпоновать с уже существующими объектными файлами с помощью такой команды:

```
cc main.c getline.o strindex.o
```

При использовании команды `cc` объектный код отличают от исходного благодаря применению разных расширений имен файлов — соответственно `.o` и `.c`.

Упражнение 4.1. Напишите функцию `strrindex(s,t)`, которая бы возвращала индекс самого правого вхождения строки `t` в `s`, либо `-1`, если такой строки в `s` нет.

4.2. Функции, возвращающие нецелые значения

До сих пор в наших примерах функции либо не возвращали никаких значений (void), либо возвращали числа типа int. А что если функция должна возвращать что-то другое? Многие функции для математических расчетов, такие как sqrt, sin или cos, возвращают числа типа double. Другие специализированные функции возвращают данные и других типов. Чтобы показать это на примере, напишем функцию atof(s) для преобразования строки символов s в вещественное число двойной точности, которое она изображает. Функция atof — это расширенная версия функции atoi, разные версии которой демонстрировались в главах 2 и 3. Она может обрабатывать знак и десятичную точку, а также различать, присутствует ли в числе целая и/или дробная часть. Наша версия не является высококлассной функцией преобразования, потому что иначе она бы заняла намного больше места. В стандартной библиотеке С функция atof имеется и объявлена в заголовочном файле <stdlib.h>.

Во-первых, следует явным образом объявить тип возвращаемого из atof значения, поскольку это не int. Имя типа ставится перед именем функции:

```
#include <ctype.h>

/* atof:  преобразование строки s в число типа double */
double atof(char s[])
{
    double val, power;
    int i, sign;

    for(i = 0; isspace(s[i]); i++)  /* пропуск пробелов */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for(val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10;
    }
    return sign * val / power;
}
```

Во-вторых, и это не менее важно, вызывающая функция должна знать, что atof возвращает нецелое число. Один из способов сделать это — объявить atof прямо в вызывающей функции. Такое объявление показано ниже в программе-калькуляторе — настолько примитивной, что с ее помощью едва ли можно подбить даже бухгалтерский баланс. Она считывает числа по одному в строке (перед числом может стоять знак), складывает их и выводит текущую сумму после ввода каждой строки:

```
#include <stdio.h>

#define MAXLINE 100

/* примитивный калькулятор */
main()
{
    double sum, atof(char []);
    char line[MAXLINE];
    int getline(char line[], int max);

    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%g\n", sum += atof(line));
    return 0;
}
```

В программе имеется следующее объявление:

```
double sum, atof(char []);
```

В нем говорится, что `sum` является переменной типа `double`, а `atof` — функцией, принимающей один аргумент типа `char []` и возвращающей значение типа `double`.

Функция `atof` должна иметь согласованные между собой объявление и определение. Если сама функция `atof` и ее вызов в функции `main` имеют несогласованные типы и находятся в одном файле исходного кода, то компилятор выдаст сообщение об ошибке. Однако если (что более вероятно) функция `atof` компилируется отдельно, то несоответствие типов замечено не будет — функция возвратит число типа `double`, а `main` воспримет его как `int`, и в результате получится бессмыслица.

В свете всего сказанного насчет соответствия объявлений и определений функций это может показаться странным. Но дело в том, что здесь отсутствует прототип функции и функция объявляется неявным образом — своим первым появлением в таком выражении, как

```
sum += atof(line)
```

Если в выражении встречается имя, которое ранее не объявлялось, и если после него стоят круглые скобки, оно по контексту объявляется функцией. По умолчанию считается, что эта функция возвращает число типа `int`, а относительно ее аргументов не делается никаких предположений. Более того, если объявление функции не содержит аргументов, то и в этом случае не делается никаких предположений относительно их типа и количества, как в этом примере:

```
double atof();
```

Проверка соответствия параметров для `atof` просто отключается. Это особое значение пустого списка аргументов введено для того, чтобы старые программы на C могли компилироваться новыми компиляторами. Однако использовать этот стиль в новых программах — дурной тон. Если функция принимает аргументы, объявляйте их. Если же она не принимает никаких аргументов, пишите `void`.

При наличии функции `atof`, объявленной должным образом, с ее помощью можно написать функцию `atoi` (для преобразования строки в целое число):

```
/* atoi:  преобразует строку s в целое число с помощью atof */
int atoi(char s[])
{
```

```
    double atof(char s[]);

    return (int) atof(s);
}
```

Обратите внимание на структуру объявления и на оператор `return`, обычная форма которого такова:

`return` *выражение;*

Стоящее в операторе *выражение* всегда преобразуется к возвращаемому типу перед выполнением оператора. Поэтому значение `atof`, имеющее тип `double`, автоматически преобразуется в тип `int` в том случае, если оно фигурирует в этом операторе, поскольку функция `atoi` возвращает тип `int`. При выполнении этой операции возможна частичная потеря информации, поэтому некоторые компиляторы в таких случаях выдают предупреждения. А вот при наличии явного приведения типов компилятору становится ясно, что операция выполняется намеренно, так что он не выдает предупреждений.

Упражнение 4.2. Усовершенствуйте функцию `atof` так, чтобы она понимала экспоненциальную запись чисел вида

```
123.45e-6
```

В этой записи после вещественного числа может следовать символ `e` или `E`, а затем показатель степени — возможно, со знаком.

4.3. Внешние переменные

Программа на C обычно состоит из набора внешних объектов, являющихся либо переменными, либо функциями. Прилагательное "внешний" здесь употребляется как противоположность слову "внутренний", которое относится к аргументам и переменным, определенным внутри функций. Внешние переменные определяются вне каких бы то ни было функций и поэтому потенциально могут использоваться несколькими функциями. Сами по себе функции — всегда внешние, поскольку в C нельзя определить функцию внутри другой функции. По умолчанию внешние переменные и функции имеют то свойство, что все ссылки на них по одним и тем же именам — даже из функций, скомпилированных отдельно, — являются ссылками на один и тот же объект программы. (Стандарт называет это свойство *внешним связыванием — external linkage*). В этом отношении внешние переменные являются аналогом переменных блока COMMON в языке Fortran или переменных в самом внешнем блоке в программе на Pascal. Позже будет показано, как определить внешние переменные и функции, чтобы они были доступны (видимы) только в одном файле исходного кода.

Поскольку внешние переменные имеют глобальную область видимости, они представляют собой способ обмена данными между функциями, альтернативный передаче аргументов и возврату значений. Любая функция может обращаться к внешней переменной по имени, если это имя объявлено определенным образом.

Если функциям необходимо передавать друг другу большое количество элементов данных, то внешние переменные удобнее и эффективнее, чем длинные списки аргументов. Однако, как было указано в главе 1, здесь не все так однозначно, поскольку от этого страдает структурированность программы и возникает слишком сильная перекрестная зависимость функций друг от друга.

Внешние переменные также бывают полезны из-за их более широкой области действия и более длительного времени жизни. Автоматические переменные являются внутренними для функций; они создаются при входе в функцию и уничтожаются при выходе из нее. С другой стороны, внешние переменные существуют непрерывно и хранят свои значения от одного вызова функции до другого. Поэтому если двум функциям необходимо иметь общий набор данных, но при этом ни одна из них не вызывает другую, бывает удобнее поместить общие данные во внешние переменные, чем передавать их туда-сюда в виде аргументов.

Рассмотрим эти вопросы более подробно на объемном примере. Задача состоит в том, чтобы написать программу-калькулятор с набором операций +, -, * и /. Калькулятор будет использовать обратную польскую, или бесскобочную, запись ввиду простоты ее реализации вместо обычной инфиксной. (Обратная польская запись применяется в некоторых карманных калькуляторах, а также в языках программирования Forth и Postscript.)

В обратной польской записи каждый знак операции стоит после своих операндов. Возьмем, например, следующее выражение:

```
(1 - 2) * (4 + 5)
```

В обратной польской записи оно выглядит так:

```
1 2 - 4 5 + *
```

Скобки здесь не нужны, поскольку запись расшифровывается однозначно, если известно, сколько операндов у каждой операции.

Реализация алгоритма проста. Каждый операнд помещается в стек; как только поступает знак операции, из стека извлекается нужное количество операндов (для двухместной операции — два), к ним применяется данная операция, и результат снова помещается в стек. В приведенном примере в стек вначале помещаются 1 и 2, а затем они заменяются их разностью (–1). Затем в стек помещаются 4 и 5, после чего они заменяются их суммой (9). Произведение –1 и 9, т.е. –9, заменяет оба вычисленных операнда в стеке. Как только встретился конец строки входного потока, из стека извлекается и выводится на экран верхнее значение.

Таким образом, программа организована в виде цикла, выполняющего операции над знаками и операндами по мере их поступления:

```
while (очередной знак или операнд - не конец файла)
    if (число)
        поместить операнд в стек
    else if (знак операции)
        извлечь операнд из стека
        выполнить операцию
        поместить результат в стек
    else if (конец строки)
        извлечь и вывести верх стека
    else
        ошибка
```

Операции помещения в стек и извлечения из него тривиальны, но после добавления проверки и исправления ошибок эти операции становятся достаточно объемными, чтобы поместить их в отдельные функции, а не повторять длинные фрагменты кода несколько раз. Необходимо также иметь отдельную функцию для извлечения следующего операнда или знака операции.

Осталось обсудить основное проектное решение, необходимое для разработки данной программы. Это вопрос о том, где разместить стек, т.е. какая из функций программы будет с ним работать. Один из возможных вариантов — держать его в функции main, передавая как его, так и текущую позицию в нем в функции, которые помещают и извлекают данные. Но функции main нет нужды знать что-либо о переменных, которые управляют стеком; ей нужно только уметь помещать и извлекать данные. Поэтому будем хранить стек и связанную с ним информацию во внешних переменных, к которым будут обращаться функции push и pop, но не функция main.

Превратить это словесное описание в код сравнительно просто. Если пока что ограничиться программой, состоящей из одного файла исходного кода, получится следующее:

```
#include-директивы
#define-директивы

объявления функций, используемых в main

main() { ... }

внешние переменные для функций push и pop

void push(double f) { ... }
double pop(void) { ... }

int getop(char s[]) { ... }

функции, вызываемые из getop
```

Ниже рассказывается, как все это можно распределить по двум или нескольким файлам исходного кода.

Функция main организована в виде цикла, содержащего объемистый оператор switch по типам операций и операндов. Это более типичный пример применения оператора switch, чем тот, который приводился в разделе 3.4.

```
#include <stdio.h>
#include <stdlib.h>   /* для объявления atof() */

#define MAXOP   100   /* максимальный размер операнда или знака */
#define NUMBER  '0'   /* сигнал, что обнаружено число */

int getop(char s[]);
void push(double);
double pop(void);

/* калькулятор с обратной польской записью */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch(type) {
        case NUMBER:
            push(atof(s));
```

```
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if(op2 != 0.0)
                push(pop() / op2);
            else
                printf("error: zero divisor\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("error: unknown command %s\n", s);
            break;
        }
    }
    return 0;
}
```

Поскольку сложение (+) и умножение (*) — коммутативные операции, порядок, в котором их аргументы извлекаются из стека, не имеет значения. А вот для вычитания (-) и деления (/) следует различать левый и правый операнды. Допустим, вычитание представлено следующим образом:

```
push(pop() - pop());    /* НЕПРАВИЛЬНО */
```

В этом случае порядок, в котором обрабатываются два вызова `pop()`, не определен. Чтобы гарантировать правильный порядок, необходимо извлечь первый операнд заранее во временную переменную, как это и сделано в функции `main`.

```
#define MAXVAL  100   /* максимальная глубина стека val */

int sp = 0;           /* следующая свободная позиция в стеке */
double val[MAXVAL];   /* стек операндов */

/* push:  помещает число f в стек операндов */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n", f);
}


/* pop:  извлекает и возвращает верхнее число из стека */
double pop(void)
```

```
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: stack empty\n");
        return 0.0;
    }
}
```

Переменная является внешней, если она определена вне каких бы то ни было функций. Поэтому сам стек и его индекс, которые совместно используются функциями `push` и `pop`, определены вне этих функций. Но функция `main` сама не обращается ни к стеку, ни к его индексу непосредственно, поэтому представление стека можно от нее спрятать.

Теперь займемся реализацией функции `getop`, которая доставляет на обработку очередной операнд или знак операции. Ее задача проста. Сначала необходимо пропустить пробелы и тому подобные символы. Если очередной символ не является цифрой или десятичной точкой, возвратить его. В противном случае накопить строку цифр (возможно, с десятичной точкой) и возвратить NUMBER — сигнал о том, что на вход поступило число.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop:  извлекает следующий операнд или знак операции */
int getop(char s[]);
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c;     /* не число */
    i = 0;
    if (isdigit(c))   /* накопление целой части */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')     /* накопление дробной части */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

Что такое `getch` и `ungetch`? Часто бывает, что программа не может определить, достаточно ли она прочитала данных, до тех пор, пока не прочитает их слишком много. Один из примеров — это накопление символов, составляющих число. Пока не встретится первый символ, не являющийся цифрой, ввод числа не завершен; но как только он встретится, он оказывается лишним, и программа не знает, как его обрабатывать.

Проблема была бы решена, если бы существовал способ вернуть назад ненужный символ. Тогда каждый раз, когда программа считывала бы на один символ больше, чем

нужно, она могла бы вернуть его в поток ввода, а остальные части программы продолжали бы работать с этим потоком так, как будто последней операции ввода не было вовсе. К счастью, возврат символа в поток довольно легко смоделировать, написав для этого пару специальных функций. Функция getch будет извлекать следующий символ из потока ввода, а функция ungetch — запоминать символы, возвращаемые в поток, чтобы при следующем вызове getch вначале вводились они, а потом уже те, которые действительно поступают из потока ввода.

Совместная работа этих функций организована просто. Функция ungetch помещает возвращенный символ в общий буфер — массив символов. Функция getch читает данные из буфера, если они там есть, или вызывает getchar, если буфер пустой. Необходимо также иметь индексную переменную, которая бы хранила позицию текущего символа в буфере.

Поскольку буфер и индекс совместно используются функциями getch и ungetch, а также должны сохранять свои значения между вызовами этих функций, их следует сделать внешними переменными. Запишем итоговый код для getch, ungetch и их общих переменных:

```
#define BUFSIZE 100

char buf[BUFSIZE];    /* буфер для ungetch */
int bufp = 0;         /* следующая свободная позиция в buf */

int getch(void) /* ввод символа (возможно, возвращенного в поток) */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c) /* возвращение символа в поток ввода */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}
```

Стандартная библиотека содержит функцию ungetc, которая возвращает в поток один символ; она будет рассматриваться в главе 7. Здесь же для иллюстрации более общего подхода в качестве буфера возврата используется целый массив, а не один символ.

Упражнение 4.3. При наличии базовой структуры программы усовершенствование калькулятора уже не представляет особых трудностей. Реализуйте операцию взятия остатка (%) и работу с отрицательными числами.

Упражнение 4.4. Добавьте в программу реализацию команд для вывода верхнего элемента стека без его удаления, для создания в стеке дубликата этого элемента и для обмена местами двух верхних элементов. Также реализуйте команду очистки стека.

Упражнение 4.5. Добавьте реализацию библиотечных математических функций sin, exp и pow. См. заголовочный файл <math.h> в приложении Б, раздел 4.

Упражнение 4.6. Добавьте команды для работы с переменными. (Можно использовать 26 различных переменных, если разрешить имена только из одной буквы.) Введите переменную, обозначающую последнее выведенное на экран число.

Упражнение 4.7. Напишите функцию `ungets(s)`, возвращающую в поток целую строку символов. Следует ли этой функции знать о существовании переменных `buf` и `bufp`, или ей достаточно вызывать `ungetch`?

Упражнение 4.8. Предположим, что в поток ввода никогда не будет возвращаться больше одного символа. Доработайте функции `getch` и `ungetch` соответственно.

Упражнение 4.9. Наши функции `getch` и `ungetch` не могут корректно обработать символ конца файла `EOF`, возвращенный в поток ввода. Подумайте, как эти функции должны реагировать на `EOF` в буфере, а затем реализуйте ваш замысел.

Упражнение 4.10. В качестве альтернативного подхода можно использовать функцию `getline` для считывания целых строк. Тогда `getch` и `ungetch` становятся ненужными. Перепишите программу-калькулятор с использованием этого подхода.

4.4. Область действия

Функции и внешние переменные, составляющие программу на C, не обязательно компилируются в одно и то же время. Исходный текст программы может храниться в нескольких файлах, и к тому же могут подключаться заранее скомпилированные функции из библиотек. В этой связи возникает ряд вопросов, на которые необходимо знать ответы.

- Как записать объявления переменных, чтобы они правильно воспринимались во время компиляции?
- Как организовать объявления так, чтобы при компоновке программы все ее части правильно объединились в одно целое?
- Как организовать объявления так, чтобы каждая переменная присутствовала в одном экземпляре?
- Как инициализируются внешние переменные?

Рассмотрим эти вопросы на примере, распределив код программы-калькулятора по нескольким файлам. С практической точки зрения эта программа слишком мала, чтобы ее стоило так разбивать, но зато это хорошая иллюстрация важных приемов, применяемых в больших программах.

Областью действия (видимости) имени называется часть программы, в пределах которой можно использовать имя. Для автоматической переменной, объявляемой в начале функции, областью видимости является эта функция. Локальные переменные с одинаковыми именами, объявленные в разных функциях, не имеют никакого отношения друг к другу. То же самое справедливо и для параметров функций, которые по сути являются локальными переменными.

Область действия внешней переменной или функции распространяется от точки, в которой она объявлена, до конца компилируемого файла. Например, пусть `main`, `sp`, `val`, `push` и `pop` определены в одном файле, в порядке, показанном выше, т.е.

```
main() { ... }

int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

double pop(void) { ... }
```

Тогда переменные sp и val можно использовать в функциях push и pop, просто обращаясь по именам, — никаких дальнейших объявлений внутри функций не нужно. Однако эти переменные невидимы в функции main, как и собственно функции push и pop.

Если же необходимо обратиться к внешней переменной до ее определения или если она определена в другом файле исходного кода, то обязательно нужно вставить объявление с ключевым словом extern.

Важно понимать различие между *объявлением* внешней переменной и ее *определением*. Объявление сообщает, что переменная обладает определенными свойствами (в основном типом), а определение выделяет место в памяти для ее хранения. Если следующие строки фигурируют вне функций, то они являются *определениями* внешних переменных sp и val:

```
int sp;
double val[MAXVAL];
```

Благодаря этому определению выделяется память для хранения переменных. Кроме того, это еще и объявление, действительное до конца файла. С другой стороны, следующие строки дают только *объявление*, также действительное до конца файла, согласно которому sp имеет тип int, а val является массивом типа double (его размер определяется в другом месте). При этом память не выделяется и переменные не создаются.

```
extern int sp;
extern double val[];
```

Во всех файлах, образующих исходный текст программы, должно быть в общей сложности не больше одного *определения* внешней переменной; в других файлах могут содержаться *объявления* со словом extern, чтобы оттуда можно было к ней обращаться. (В файле, содержащем определение переменной, также могут находиться и extern-объявления ее же.) Размеры массивов обязательно указываются в определении, но необязательно — в объявлении со словом extern.

Инициализация внешней переменной выполняется только в определении.

Хотя в данной программе такая организация ни к чему, функции push и pop можно было бы определить в одном файле, а переменные val и sp — определить и инициализировать в другом. В итоге для связывания программы в единое целое понадобились бы следующие объявления и определения:

в файле file1:

```
    extern int sp;
    extern double val[];

    void push(double f) { ... }

    double pop(void) { ... }
```

в файле file2:

```
    int sp = 0;
    double val[MAXVAL];
```

Поскольку объявления с extern в файле *file1* находятся впереди и вовне определений функций, они действительны во всех функциях; одного набора объявлений достаточно для всего файла. То же самое необходимо было бы записать, если бы определения переменных sp и val стояли после обращения к ним в одном и том же файле.

4.5. Заголовочные файлы

Теперь рассмотрим, как распределить программу-калькулятор по нескольким файлам исходного кода. Это необходимо было бы сделать, если бы каждый из компонентов программы имел намного большие размеры, чем сейчас. В таком случае функция `main` будет храниться в одном файле под именем `main.c`, функции `push` и `pop` вместе с их переменными — в другом файле, `stack.c`, а функция `getop` — в третьем файле, `getop.c`. Наконец, поместим функции `getch` и `ungetch` в четвертый файл, `getch.c`. Они будут отделены от остальных, потому что в настоящей, не учебной программе такие функции берутся из заранее скомпилированных библиотек.

Единственное, о чем стоит побеспокоиться, — это как разнести определения и объявления по разным файлам, сохранив возможность их совместного использования. Мы постарались сохранить централизованную структуру, насколько это было возможно, — собрать все необходимое в одном месте и использовать его на протяжении всей эволюции программы. Весь этот общий материал помещается в *заголовочный файл* `calc.h`, который подключается к файлам кода по мере необходимости. (Директива `#include` рассматривается в разделе 4.11.) В результате получается следующая программа:

`calc.h`:

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

`main.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

`getop.c`:

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
}
```

`getch.c`:

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

`stack.c`

```
#include <stdio.h>
#include <calc.h>
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

С одной стороны, хочется разрешить доступ из каждого файла только к самой необходимой информации; с другой стороны, уследить за множеством заголовочных файлов довольно нелегко. Нужно искать компромисс: добиться одного можно, только жертвуя другим. Пока объем программы не достиг некоторой средней величины, бывает удобнее хранить все, что необходимо для связи двух частей программы, в одном заголовочном файле. Именно такое решение было принято здесь. В программах, намного больших по объему, требуется более разветвленная организация и больше заголовочных файлов.

4.6. Статические переменные

Переменные `sp` и `val` в файле `stack.c`, а также `buf` и `bufp` в файле `getch.c` предназначены для внутреннего использования функциями в соответствующих файлах кода; всем остальным частям программы доступ к ним закрыт. Если объявление внешней переменной или функции содержит слово `static`, ее область действия ограничивается данным файлом исходного кода — от точки объявления до конца. Таким образом, внешние статические переменные — это механизм сокрытия имен наподобие `buf` и `bufp` в паре функций `getch-ungetch`, которые должны быть внешними, чтобы использоваться совместно, но не должны быть доступны за пределами указанных функций.

Статическое хранение переменной в памяти задается ключевым словом `static` в начале обычного объявления. Пусть в одном и том же файле компилируются две функции и две переменные:

```
static char buf[BUFSIZE]; /* буфер для ungetch */
static int  bufp = 0;     /* следующая свободная позиция в buf */

int getch(void) { ... }

void ungetch(int c) { ... }
```

Тогда никакая другая функция не сможет обратиться к переменным `buf` и `bufp`, и не возникнет конфликта, если в других файлах программы будут употребляться такие же имена. Аналогично можно скрыть и переменные `sp` и `val`, объявив их статическими, чтобы только функции `push` и `pop` могли ими пользоваться для операций со стеком.

Чаще всего внешними статическими объявляются переменные, но такое объявление применимо и к функциям. Обычно имена функций являются глобальными и видимыми в любой части программы. Но если функцию объявить статической (`static`), ее имя будет невидимо за пределами файла, в котором она объявлена.

Объявление `static` применимо и к внутренним переменным. Внутренние статические переменные являются локальными по отношению к конкретной функции, как и автоматические. Но в отличие от автоматических статические переменные продолжают существовать непрерывно, а не создаются и уничтожаются при вызове и завершении функции. Получается, что внутренние статические переменные являются средством постоянного хранения скрытой информации внутри одной функции.

Упражнение 4.11. Измените функцию `getop` так, чтобы ей не нужно было использовать `ungetch`. *Подсказка:* воспользуйтесь внутренней статической переменной.

4.7. Регистровые переменные

Объявление с ключевым словом `register` сообщает компилятору, что соответствующая переменная будет интенсивно использоваться программой. Идея заключается в том, чтобы поместить такие (*регистровые*) переменные в регистры процессора и добиться повышения быстродействия и уменьшения объема кода. Впрочем, компилятор имеет право игнорировать эту информацию.

Объявления со словом `register` выглядят следующим образом:

```
register int  x;
register char c;
```

Объявление `register` применимо только к автоматическим переменным и к формальным параметрам функций. В случае параметров оно выглядит так:

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

На практике существуют ограничения на употребление регистровых переменных, связанные со свойствами аппаратных устройств компьютера. В каждой функции только очень небольшое количество переменных (и только нескольких определенных типов) можно сделать регистровыми. Однако от лишних объявлений `register` не бывает никакого вреда, поскольку избыточные или неразрешенные объявления просто игнорируются компилятором. Не разрешается вычислять адрес регистровой переменной (эта тема рассматривается в главе 5), причем независимо от того, помещена ли она на самом деле в регистр процессора. Конкретные ограничения на количество и тип регистровых переменных зависят от системы и аппаратного обеспечения.

4.8. Блочная структура

Язык С не является блочно-структурным в том же смысле, как Pascal или другие языки, поскольку в нем функции нельзя определять внутри других функций. С другой стороны, внутри одной функции переменные можно определять в блочно-структурном стиле. Объявления переменных (в том числе инициализации) могут стоять после левой фигурной скобки, открывающей *любой* составной оператор, а не только после такой, которая открывает тело функции. Переменные, объявленные таким образом, блокируют действие любых переменных с теми же именами, объявленных во внешних блоках, и остаются в силе, пока не встретится соответствующая (закрывающая) правая скобка. Рассмотрим пример:

```
if (n > 0) {
    int i  /* объявляется новая переменная i */

    for (i = 0; i < n; i++)
        ...
}
```

В этом фрагменте кода область действия переменной i — та ветвь оператора if, которая соответствует "истине" в условии. Эта переменная не связана с любыми другими i за пределами блока. Автоматическая переменная объявляется и инициализируется в блоке всякий раз заново при входе в этот блок. Статическая переменная инициализируется только в первый раз при входе в блок.

Автоматические переменные, в том числе формальные параметры, также преобладают над внешними переменными и функциями с теми же именами, скрывая и блокируя действие их имен. Рассмотрим такое объявление:

```
int x;
int y;

f(double x)
{
    double y;
    ...
}
```

Внутри функции f любое употребление имени x будет относиться к параметру функции, имеющему тип double; за ее пределами x будет обозначать внешнюю переменную типа int. То же самое справедливо в отношении переменной y.

Стремясь программировать в хорошем стиле, лучше избегать имен переменных, которые перекрывают действие переменных из более внешних блоков, — слишком велика опасность путаницы и непредвиденных ошибок.

4.9. Инициализация

Инициализация мимоходом упоминалась уже много раз, но всегда как второстепенная тема по отношению к другим. Ранее уже рассматривались различные классы памяти для переменных, поэтому в данном разделе собраны некоторые правила инициализации, связанные с ними.

При отсутствии явной инициализации внешние и статические переменные гарантированно инициализируются нулями, а автоматические и регистровые получают неопределенные начальные значения ("мусор").

Скалярные переменные можно инициализировать прямо при объявлении, поставив после имени знак равенства и выражение:

```
int  x = 1;
char squote = '\'';
long day = 1000L * 60L * 60L * 24L;  /* миллисекунд в дне */
```

Инициализирующие выражения для внешних и статических переменных должны быть константными; инициализация выполняется один раз, фактически до начала выполнения программы. А для автоматических и регистровых переменных инициализация выполняется каждый раз при входе в соответствующий блок.

Инициализирующее выражение автоматической или регистровой переменной не обязано быть константным — оно может содержать любые значения, определенные ранее, даже вызовы функций. Например, инициализацию в программе двоичного поиска из раздела 3.3 можно записать так:

```
int binarysearch(int x, int v[], int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

Ранее она записывалась таким образом:

```
int low, high, mid;

low = 0;
high = n - 1;
```

Фактически инициализация автоматической переменной — это сокращенная запись для комбинации объявления и оператора присваивания. Какую из форм записи предпочесть — это, в общем-то, вопрос вкуса. Мы обычно употребляем явные операторы присваивания, поскольку инициализации в объявлениях труднее разглядеть и они дальше от точки использования той или иной переменной.

Чтобы инициализировать массив, необходимо поставить после его объявления и знака равенства список инициализирующих значений, разделенных запятыми, в фигурных скобках. Например, вот как инициализируется массив `days`, содержащий количество дней в каждом месяце года:

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Если не указывать размер массива, компилятор вычислит его сам, сосчитав инициализирующие значения, которых в данном случае 12.

Если указано меньше инициализирующих значений, чем заданный размер массива, то недостающие значения будут заменены нулями для внешних, статических и автоматических переменных. Если же их слишком много, возникнет ошибка. Не существует способа задать повторение инициализирующих значений в сокращенном виде, а также инициализировать какой-нибудь средний элемент массива, не инициализировав все предыдущие.

Массивы символов — особый случай; для их инициализации можно использовать строку вместо фигурных скобок и запятых:

```
char pattern[] = "ould";
```

Эта запись короче, чем следующая, но эквивалентна ей:

```
char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

В этом случае длина массива равна пяти (четыре символа плюс завершающий `'\0'`).

4.10. Рекурсия

Функции в C могут использоваться рекурсивно, т.е. функция может вызывать сама себя прямо или косвенно. Для примера рассмотрим задачу вывода числа в виде строки символов-цифр. Как уже говорилось, цифры генерируются в неправильном порядке: младшие цифры становятся известны раньше, чем старшие, однако выводить их надо как раз наоборот, начиная со старших.

У этой задачи есть два возможных решения. Одно заключается в том, чтобы помещать цифры в массив по мере генерирования, а затем вывести в обратном порядке. Это

делалось в функции itoa (см. раздел 3.6). Другой способ основан на рекурсии, при которой функция printd вначале вызывает сама себя для обработки старших (первых) цифр, а затем выводит самую последнюю. Эта функция также может не сработать с самым большим по модулю отрицательным числом.

```
#include <stdio.h>

/* printd:  вывод числа n десятичными цифрами */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

Когда функция рекурсивно вызывает сама себя, при каждом таком вызове создается новый набор автоматических переменных, независимый от предыдущего набора. Таким образом, при вызове printd(123) функция в первый раз получает аргумент n = 123, затем передает 12 во вторую копию printd, затем единицу — в третью копию, которая выводит эту единицу и возвращается на второй уровень. Там выводится двойка, и управление передается первому уровню, на котором выводится тройка и работа функции заканчивается.

Еще одним удачным примером рекурсии является быстрая сортировка — алгоритм сортировки, разработанный Ч.А.Р. Хоаром (C.A.R. Hoare) в 1962 г. Берется массив, выбирается один элемент, а все остальные делятся на два подмножества — элементы, бо́льшие выбранного или равные ему, и элементы, меньшие выбранного. Затем процедура применяется рекурсивно к каждому из двух подмножеств. Если в подмножестве меньше двух элементов, сортировка ему не требуется, и рекурсия на этом прекращается.

Наша версия быстрой сортировки — не самая быстрая из возможных, но зато одна из простейших. Для разбиения каждого подмножества используется его средний элемент.

```
/* qsort: сортировка v[left]...v[right] в порядке возрастания */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right) /* ничего не делать, если в массиве */
        return;        /* меньше двух элементов */
    swap(v, left, (left+right)/2);   /* переместить */
    last = left;                /* разделитель в v[0] */
    for (i = left+1; i<= right; i++) /* упорядочение */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last); /* вернуть разделитель на место */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Операция перемены элементов местами, `swap`, вынесена в отдельную функцию, поскольку она выполняется в `qsort` три раза.

```
/* swap:  обмен местами v[i] и v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Стандартная библиотека включает версию `qsort`, которая может сортировать объекты любого типа.

Рекурсия может оказаться несовместимой с экономией памяти, поскольку нужно где-то хранить стек обрабатываемых чисел. Не дает она также и выигрыша в быстродействии. Но рекурсивный код обычно компактнее, и его значительно легче писать и дорабатывать, чем его аналог без рекурсии. Рекурсия особенно удобна при работе с рекурсивно определенными структурами данных наподобие деревьев; интересный пример будет рассмотрен в разделе 6.5.

Упражнение 4.12. Примените идеи, реализованные в `printd`, чтобы написать рекурсивную версию функции `itoa` — преобразование целого числа в строку путем рекурсивного вызова.

Упражнение 4.13. Напишите рекурсивную версию функции `reverse(s)`, которая обращает порядок символов в строке прямо на месте, без дополнительного строкового буфера.

4.11. Препроцессор C

Некоторые средства языка C реализованы с помощью препроцессора, работа которого по сути является первым этапом компиляции. Два наиболее часто используемых средства — это директива `#include`, включающая все содержимое заданного файла в компилируемый код, и директива `#define`, заменяющая некий идентификатор заданной последовательностью символов. Другие средства, описанные в этом разделе, — это условная компиляция и макросы с аргументами.

4.11.1. Включение файлов

С помощью включения файлов можно легко распоряжаться наборами директив `#define` и объявлений (среди прочего). Включение файлов задается одной из следующих строк в исходном коде:

```
#include "имя_файла"
#include <имя_файла>
```

При обработке текста строка заменяется содержимым файла *имя_файла*. Если *имя_файла* указано в кавычках, поиск файла обычно начинается с того места, где находится исходный текст программы. Если его там нет или если имя файла заключено в угловые скобки < и >, то поиск файла продолжается по правилам, зависящим от реализации языка. Включаемый файл может в свою очередь содержать директивы `#include`.

Очень часто в начале файла исходного кода стоит сразу несколько директив `#include`, которые подключают необходимые наборы директив `#define` и объявлений `extern` или же загружают объявления-прототипы библиотечных функций из таких заголовочных файлов, как `<stdio.h>`. (Строго говоря, они даже не обязаны быть файлами; способ обращения к заголовочным модулям зависит от системы и реализации языка.)

Директива `#include` — это оптимальный способ собрать все объявления в нужном месте в большой программе. Этот способ гарантирует, что все файлы исходного кода будут включать одни и те же макроопределения и объявления внешних переменных, таким образом ликвидируя возможность появления особенно неприятных ошибок. Разумеется, при внесении изменений во включаемый файл все файлы, которые ссылаются на него, должны быть перекомпилированы.

4.11.2. Макроподстановки

Следующая конструкция задает макроопределение, или макрос:

```
#define  имя  текст-для-замены
```

Это — макроопределение простейшего вида; всякий раз, когда *имя* встретится в тексте программы после этого определения, оно будет заменено на *текст-для-замены*. Имя в `#define` имеет ту же форму, что и имена переменных, а текст для замены может быть произвольным. Обычно текст для замены умещается на одной строке, но если он слишком длинный, его можно продлить на несколько строк, поставив в конце каждой строки символ продолжения (\). Область действия имени, заданного в директиве `#define`, распространяется от точки его определения до конца файла исходного кода. В макроопределении могут использоваться предыдущие определения. Подстановка выполняется только для идентификаторов и не распространяется на такие же строки символов в кавычках. Например, если для имени `YES` задано макроопределение, подстановка не выполняется для `printf("YES")` или `YESMAN`.

Текст для подстановки может быть совершенно произвольным. Например, таким образом можно определить имя `forever`, обозначающее бесконечный цикл:

```
#define  forever  for (;;)   /* бесконечный цикл */
```

Можно также определять макросы с аргументами, чтобы изменять текст подстановки в зависимости от способа вызова макроса. Например, так определяется макрос с именем `max`:

```
#define  max(A, B)  ((A) > (B) ? (A) : (B))
```

Хотя внешне он напоминает вызов функции, на самом деле `max` разворачивается прямо в тексте программы путем подстановки. Вместо формальных параметров (`A` или `B`) будут подставлены фактические аргументы, обнаруженные в тексте. Возьмем, например, следующий макровызов:

```
x = max(p+q, r+s);
```

При компиляции он будет заменен на следующую строку:

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Если всегда задавать при вызове соответствующие по типу аргументы, нет необходимости иметь несколько различных макросов `max` для различных типов данных, как это было бы с функциями.

Если внимательно изучить способ подстановки макроса max, можно заметить ряд "ловушек". Так, выражения в развернутой форме макроса вычисляются дважды, и это даст неправильный результат, если в них присутствуют побочные эффекты наподобие инкремента-декремента или ввода-вывода. Например, в следующем макровызове большее значение будет инкрементировано два раза:

```
max(i++, j++)    /* НЕПРАВИЛЬНО */
```

Необходимо также аккуратно использовать скобки, чтобы гарантировать правильный порядок вычислений. Представьте себе, что произойдет, если следующий макрос вызвать в виде square(z+1):

```
#define  square(x)  x * x    /* НЕПРАВИЛЬНО */
```

И тем не менее макросы очень полезны. Один из практических примеров можно найти в файле <stdio.h>, в котором имена getchar и putchar часто определяются как макросы, чтобы избежать лишних вызовов функций для ввода отдельных символов.

Чтобы отменить определение имени, используется директива #undef. Обычно это делается для того, чтобы гарантировать, что имя функции действительно определяет функцию, а не макрос:

```
#undef getchar

int getchar(void) { ... }
```

Формальные параметры не заменяются аргументами, если они стоят в кавычках. Если же в тексте подстановки перед именем формального параметра стоит значок #, то эта комбинация заменяется строкой в кавычках, в которые вместо формального параметра подставляется аргумент. Эту возможность можно сочетать с конкатенацией строк, например для организации макроса отладочного вывода:

```
#define  dprint(expr)  printf(#expr " = %g\n", expr)
```

Вызовем его следующим образом:

```
dprint(x/y);
```

Макрос будет развернут в следующую строку:

```
printf("x/y" " = %g\n", x/y);
```

После конкатенации строк окончательный результат будет таким:

```
printf("x/y = %g\n", x/y);
```

Внутри фактического аргумента (в строке) каждый символ " заменяется на \", а каждый символ \ на \\, так что в результате получается правильная строковая константа.

Операция препроцессора ## предоставляет возможность сцепить фактические аргументы в одну строку в процессе раскрытия макроса. Если параметр в подставляемом тексте находится рядом со знаком ##, этот параметр заменяется фактическим аргументом, сам знак ## и окружающие его пробелы удаляются, а результат снова анализируется препроцессором. Например, макрос paste сцепляет два своих аргумента:

```
#define  paste(front, back)  front ## back
```

Выражение paste(name, 1) порождает идентификатор name1.

Правила использования вложенных знаков ## загадочны и малопонятны; подробности можно найти в приложении А.

Упражнение 4.14. Определите макрос swap(t,x,y), который обменивает местами значения двух аргументов типа t. (Примените блочную структуру.)

4.11.3. Условное включение

Существует возможность управлять самой препроцессорной обработкой, используя условные директивы, которые выполняются по ходу этой обработки. Это делает возможным условное включение фрагментов кода в программу в зависимости от условий, выполняющихся в момент компиляции.

В директиве `#if` выполняется анализ целочисленного выражения (которое не должно содержать операцию `sizeof`, приведение типов и константы, определенные через `enum`). Если выражение не равно нулю, в программу включаются все последующие строки вплоть до директивы `#endif`, `#elif` или `#else`. (Директива препроцессора `#elif` аналогична оператору `else if`.) Выражение `defined(`*имя*`)` в директиве `#if` равно единице, если *имя* определено с помощью `#define`, и нулю в противном случае.

Например, чтобы обеспечить включение файла `hdr.h` в текст программы не более одного раза, его содержимое следует окружить следующими условными директивами:

```
#if !defined(HDR)
#define HDR

/* здесь находится содержимое файла hdr.h */

#endif
```

При первом включении `hdr.h` определяется имя `HDR`; при последующих включениях обнаруживается, что имя уже определено, и текст пропускается вплоть до `#endif`. Этот подход можно применить, чтобы избежать многократного включения набора связанных друг с другом файлов. Используя указанные директивы во всех файлах набора, можно смело включать в каждый файл все заголовочные модули, которые ему требуются, не думая о степени их взаимосвязанности.

В следующем фрагменте кода анализируется имя `SYSTEM` и принимается решение, какую версию заголовочного файла включать в программу:

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

Директивы `#ifdef` и `#ifndef` — это специальные формы условной директивы для проверки того, определено то или иное имя или нет. Первый пример директивы `#if`, приведенный в начале раздела, можно было бы записать так:

```
#ifndef HDR
#define HDR

/* здесь находится содержимое файла hdr.h */

#endif
```

Глава 5

Указатели и массивы

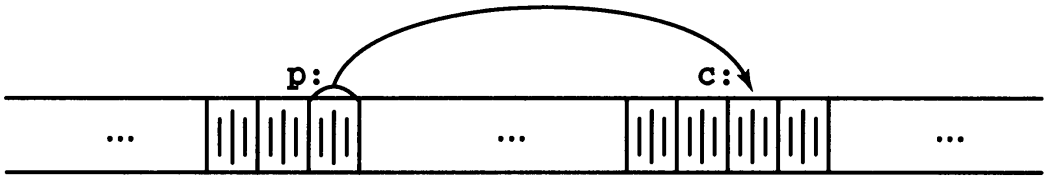
Указатель — это переменная, содержащая адрес другой переменной. Указатели очень широко используются в C — частично потому, что иногда определенные операции можно выполнить только с их помощью, а частично потому, что их использование обычно позволяет записать код более компактно и эффективно, чем другими способами. Указатели и массивы тесно связаны между собой; в этой главе рассматривается эта взаимосвязь и показывается, как ею пользоваться.

Об указателях говорят, что наряду с оператором `goto` — это верный способ запутать программу до невозможности. Это, несомненно, правда — если пользоваться ими бездумно. Совсем нетрудно так организовать работу с указателем, чтобы он указывал на что попало. Но при тщательном подходе с помощью указателей можно добиться большей ясности и простоты кода, чем без них. Именно это мы и постараемся продемонстрировать.

Основное изменение, внесенное в эту часть языка стандартом ANSI, — это четкая формулировка правил обращения с указателями, которые по сути сводят воедино то, что хорошие программисты уже давно делают, а хорошие компиляторы уже давно требуют от программистов. Кроме того, теперь стандартным объявлением нетипизированного указателя является `void *` (указатель на пустой тип `void`), а не `char *`.

5.1. Указатели и адреса

Начнем с простейшего представления о том, как организована память. Типичная компьютерная система содержит массив последовательно пронумерованных (адресуемых) ячеек памяти, с которыми можно работать по отдельности либо целыми непрерывными группами. Так, один байт может хранить переменную типа `char`, два соседних байта — целую переменную типа `short int`, а четыре байта — переменную типа `long`. Указатель представляет собой группу ячеек (две или четыре), которые могут содержать адрес. Например, если `c` — переменная типа `char`, а `p` — указатель, который на нее указывает, это можно представить следующим образом:



Одноместная (*унарная*) операция `&` дает адрес объекта. Поэтому поместить адрес переменной `c` в переменную `p` можно следующим образом:

```
p = &c;
```

В этом случае говорят, что `p` *указывает* на `c`. Операция `&` применима только к объектам, хранящимся в оперативной памяти: переменным и элементам массивов. Ее нельзя применить к выражениям, константам и регистровым переменным (`register`).

Одноместная операция * называется операцией *ссылки по указателю* (*indirection*) или *разыменования* (*dereferencing*). Применяя ее к указателю, получаем объект, на который он указывает. Предположим, что x и y — целые переменные, а ip — указатель на целую переменную. В следующем фрагменте кода (довольно искусственном) показано, как объявляется указатель и как используются операции & и *:

```
int x = 1, y = 2, z[10];
int *ip;          /* ip - указатель на int */

ip = &x;          /* ip теперь указывает на x */
y = *ip;          /* y теперь равно 1 */
*ip = 0;          /* x теперь равно 0 */
ip = &z[0];       /* ip теперь указывает на z[0] */
```

В объявлениях переменных x, y и z нет ничего нового по сравнению с уже изученным. Новое здесь только объявление указателя ip:

```
int *ip;
```

Форма этого объявления задумывалась как мнемоническая — специально для облегчения его понимания и использования. Объявление сообщает, что выражение *ip имеет тип int. Синтаксис объявления переменной копирует синтаксис выражений, в которых эта переменная может появляться. Нечто подобное имеет место и в отношении функций. Например:

```
double *dp, atof(char *);
```

Здесь говорится, что такие конструкции, как *dp и atof(s), в выражениях имеют тип double, а аргумент функции atof является указателем на char.

Следует также заметить, что любой указатель может указывать только на объекты одного конкретного типа данных, заданного при его объявлении. (Исключением является только "указатель на void", в котором может содержаться произвольный адрес без указания на тип данных; однако по указателям этого типа нельзя ссылаться и получать значения. Подробнее об этом рассказывается в разделе 5.11.)

Если ip указывает на целую переменную x, выражение *ip может фигурировать в любом контексте, в котором допускается x. Например, следующий оператор увеличивает *ip на 10:

```
*ip = *ip + 10;
```

Одноместные операции * и & имеют более высокий приоритет для своих операндов, чем арифметические операции. Рассмотрим присваивание:

```
y = *ip + 1
```

В этом выражении берется значение, на которое указывает ip, к нему прибавляется единица, и результат присваивается переменной y. Следующий оператор инкрементирует то значение, на которое указывает ip:

```
*ip += 1
```

Его можно записать еще в двух эквивалентных формах:

```
++*ip
(*ip)++
```

В последней записи скобки необходимы, поскольку без них инкрементировался бы указатель ip, а не то, на что он указывает, — одноместные операции * и ++ выполняются в последовательности справа налево.

Наконец, поскольку указатели сами являются переменными, их можно употреблять сами по себе, без разыменования. Пусть, например, `iq` — еще один указатель на значения типа `int` и пусть выполняется присваивание:

```
iq = ip
```

Эта операция копирует содержимое `ip` в `iq`, после чего `iq` указывает на то же самое, на что и `ip`.

5.2. Указатели и аргументы функций

Поскольку в языке С аргументы передаются в функции по значению, не существует прямого способа изменить локальную переменную вызывающей функции, действуя внутри вызываемой функции. Пусть, например, необходимо написать функцию обмена элементов местами под названием `swap` для использования в алгоритме сортировки. Чтобы поменять местами два элемента, хотелось бы обойтись такой записью:

```
swap(a, b);
```

Подумаем, что будет, если определить функцию `swap` следующим образом:

```
void swap(int x, int y)  /* НЕПРАВИЛЬНО */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Но аргументы передаются по значению, поэтому `swap` не окажет никакого влияния на аргументы `a` и `b` из вызывающей функции. Поменяются местами только *копии* `a` и `b`.

Чтобы добиться нужного эффекта, необходимо передать из вызывающей программы в функцию *указатели* на переменные, которые нужно изменить:

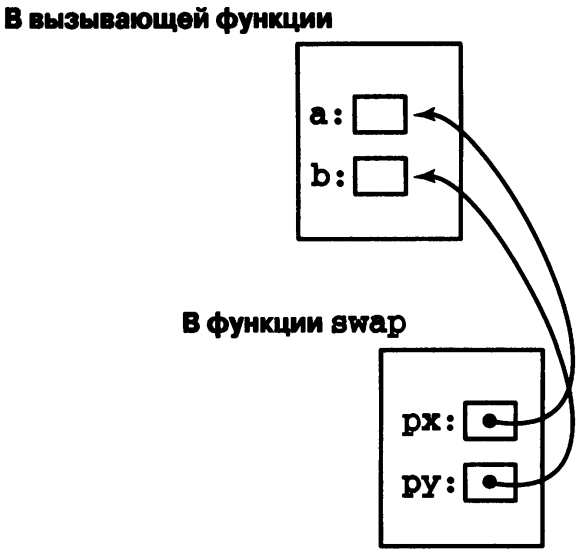
```
swap(&a, &b);
```

Поскольку операция `&` дает адрес переменной, выражение `&a` является указателем на `a`. В самой функции `swap` параметры следует объявить указателями, а к требуемым операндам обращаться косвенно по этим указателям.

```
void swap(int *px, int *py)  /* обмен местами *px и *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

Это можно условно показать на рисунке.



Благодаря аргументам-указателям функция может обращаться к объектам в вызвавшей ее функции, в том числе модифицировать их. Для примера рассмотрим функцию `getint`, которая выполняет преобразование неформатированного входного потока, разбивая цепочку символов на целые числа и вводя по одному числу за один вызов. Функция `getint` должна возвращать обнаруженное ею число, а также сигнализировать о конце файла, если в потоке больше нет входных данных. Эти данные должны поступать разными путями, поскольку каково бы ни было числовое значение управляющего символа EOF, оно может встретиться в потоке и как обычное целое число.

Одно из возможных решений — это возвращать из `getint` сигнал конца файла как ее функциональное значение, а для передачи считанного и преобразованного целого числа использовать аргумент-указатель. Эту же схему использует и функция `scanf` — см. раздел 7.4.

В следующем цикле целочисленный массив заполняется числами, полученными с помощью вызовов `getint`:

```
int n, array[SIZE], getint(int *);

for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
    ;
```

При каждом вызове `getint` элемент массива `array[n]` получает значение, взятое из входного потока; при этом инкрементируется индекс `n`. Учтите, что в функцию `getint` необходимо передать адрес элемента `array[n]`, поскольку нет другого способа получить из нее введенное значение.

Наша версия `getint` возвращает следующее: код EOF, если встретился конец потока; нуль, если следующий элемент данных не является числом; положительное число, если в потоке встретилось нормальное целое число.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getint:  считывает очередное целое число
            из входного потока в *pn */
int getint(int *pn)
{
    int c, sign;
```

```
    while (isspace(c = getch()))    /* пропуск пробелов */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c);   /* не цифра */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for(*pn = 0; isdigit(c); c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

На протяжении всей функции `getint` выражение `*pn` используется как заурядная целая переменная. Здесь также применяются функции `getch` и `ungetch` (описанные в разделе 4.3) для возвращения лишнего символа назад в поток.

Упражнение 5.1. Функция `getint` в ее нынешнем виде воспринимает плюс или минус без последующей цифры как разрешенное представление нуля. Измените функцию так, чтобы она возвращала символ назад в поток.

Упражнение 5.2. Напишите функцию `getfloat`, аналог `getint` для вещественных чисел. Данные какого типа должна возвращать функция `getfloat`?

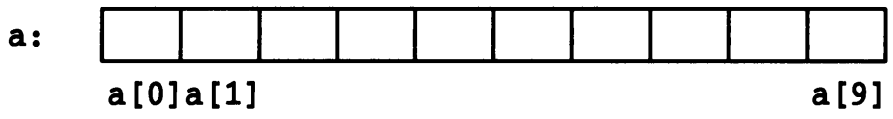
5.3. Указатели и массивы

В языке C существует столь тесная связь между указателями и массивами, что их удобно изучать совместно. Любую операцию, выполняемую с помощью индексации массива, можно проделать с применением указателей, причем код с применением указателей обычно работает быстрее, но для непосвященных выглядит более запутанно.

Следующее объявление задает *массив* длиной 10 элементов:

```
int a[10];
```

По сути, это блок из десяти последовательных объектов с именами `a[0]`, `a[1]`, ..., `a[9]`:



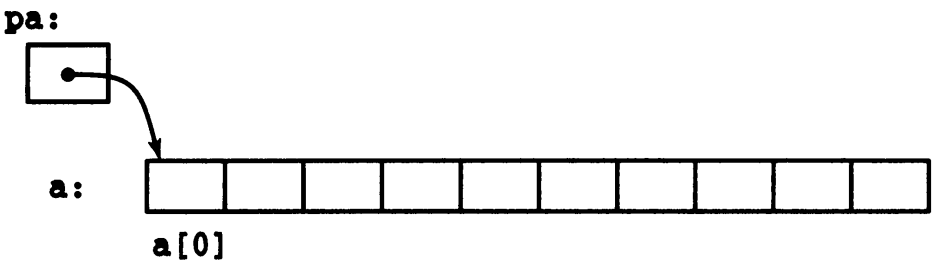
Запись `a[i]` обозначает `i`-й элемент массива. Пусть `pa` — указатель целого типа, объявленный следующим образом:

```
int *pa;
```

Рассмотрим следующее присваивание:

```
pa = &a[0];
```

После выполнения этого оператора `pa` начинает указывать на нулевой элемент массива `a`, т.е. `pa` содержит адрес `a[0]`:



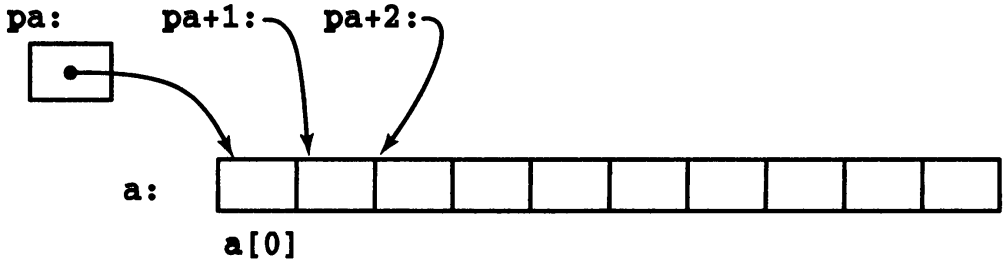
А теперь следующий оператор копирует содержимое элемента `a[0]` в переменную `x`:

```
x = *pa;
```

Если `pa` указывает на какой-либо элемент массива, то по определению `pa+1` указывает на следующий элемент, `pa+i` — на `i`-й элемент после `pa`, а `pa-i` — на `i`-й элемент перед `pa`. Итак, допустим, что `pa` указывает на `a[0]`. Тогда следующее выражение равно содержимому элемента `a[1]`:

```
*(pa+1)
```

Соответственно, `pa+i` — это адрес элемента `a[i]`, а `*(pa+i)` — содержимое ячейки `a[i]`.



Сказанное справедливо для любого массива `a` независимо от типа и размера его элементов. Смысл прибавления единицы к указателю (или, если обобщить, любой операции *адресной арифметики*) состоит в том, что `pa+1` указывает на следующий объект, а `pa+i` — на `i`-й объект после `a`.

Между обращением к массиву по индексам и адресной арифметикой существует самое близкое родство. По определению значение переменной или выражения типа "массив" является адресом нулевого элемента массива. Выполним такое присваивание:

```
pa = &a[0];
```

После него переменные `pa` и `a` будут иметь одинаковые значения. Поскольку имя массива является синонимом для местоположения его первого элемента, присваивание `pa = &a[0]` можно записать и таким образом:

```
pa = a;
```

На первый взгляд может показаться удивительным, что ссылку на элемент `a[i]` можно записать в виде `*(a+i)`. Вычисляя выражение `a[i]`, компилятор C сам преобразует его в форму `*(a+i)`, поскольку две эти формы тождественны. Применяя к обеим частям этого тождества операцию `&`, получаем, что `&a[i]` и `a+i` также тождественны, т.е. `a+i` — это адрес `i`-го элемента после `a`. С другой стороны, если `pa` — указатель, его можно употреблять в выражениях с индексом; `pa[i]` будет идентично `*(pa+i)`. Короче говоря, выражение в виде обращения к массиву по индексу эквивалентно ссылке по указателю со смещением.

Однако есть и одно различие между именем массива и указателем, которое следует иметь в виду. Указатель является переменной, так что выражения `pa=a` и `pa++` вполне допустимы. А вот конструкции наподобие `a=pa` и `a++` не разрешены.

Если в функцию передается имя массива, по сути туда поступает адрес первого элемента. В вызванной функции этот аргумент является локальной переменной, так что имя массива в виде параметра функции — это указатель, т.е. переменная, содержащая адрес. Этим фактом можно воспользоваться, чтобы написать другую версию функции `strlen`, которая вычисляет длину строки символов:

```
/* strlen:  возвращает длину строки s */
int strlen(char *s)
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

Поскольку `s` — указатель, инкрементировать его вполне допустимо. Выражение `s++` не оказывает никакого влияния на символьную строку во внешней функции, вызвавшей `strlen`, а всего лишь инкрементирует локальную копию указателя, действительную только внутри `strlen`. Это означает, что `strlen` можно вызывать любым из следующих способов:

```
strlen("hello, world");  /* строковая константа */
strlen(array);           /* массив char array[100]; */
strlen(ptr);             /* указатель char *ptr; */
```

Следующие две формы эквивалентны, если употребляются в виде формальных параметров функции:

```
char s[];
char *s;
```

Мы предпочитаем вторую форму, потому что она более явно выражает тот факт, что параметр является указателем. Когда в функцию передается имя массива, функция имеет право сама решать, обходиться с ним как с массивом или как с указателем. Если это удобно и не портит удобочитаемость, можно даже комбинировать оба способа.

В функцию можно передать и всего лишь часть массива, сообщив ей указатель на начало требуемого подмассива. Пусть `f` — некоторая функция, тогда следующие два эквивалентных вызова передают в нее адрес подмассива, начинающегося с элемента `a[2]`:

```
f(&a[2])
f(a+2)
```

В самой функции `f` параметр может быть объявлен любым из двух способов:

```
f(int arr[]) { ... }
f(int *arr) { ... }
```

Таким образом, с точки зрения функции `f` совершенно неважно, указывает ли параметр на целый массив или его часть.

Если есть уверенность в том, что соответствующие элементы существуют, можно индексировать массив и в противоположном направлении. Выражения `p[-1]`, `p[-2]` и т.д. синтаксически правильны и относятся к элементам, находящимся в памяти перед `p[0]`. Разумеется, неправильно обращаться таким образом к объектам, лежащим вне границ массива.

5.4. Адресная арифметика

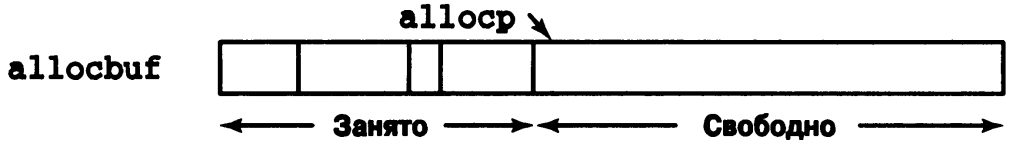
Если p — указатель на некоторый элемент массива, то выражение p++ инкрементирует p так, чтобы он указывал на следующий элемент, а p+=i переводит указатель на i элементов вперед. Эти и аналогичные конструкции представляют собой простейшие операции адресной (указательной) арифметики.

Язык C в его подходе к операциям с адресами отличается последовательностью и логичностью; одним из его преимуществ является тесная взаимосвязь между массивами, указателями и адресной арифметикой. Проиллюстрируем это, написав примитивные функции для управления распределением памяти. Их будет две. Первая, alloc(n), должна возвращать указатель p на n последовательно идущих ячеек памяти длиной один символ, которые могут затем использоваться вызывающей программой для хранения символьной информации. Вторая функция, afree(p), освобождает память, выделенную таким образом, для последующего повторного использования. Эти функции названы "примитивными", поскольку afree всегда должна вызываться в порядке, противоположном вызовам alloc. Таким образом, память, распределяемая функциями alloc и afree, имеет стековую организацию — "последним пришел, первым вышел". В стандартной библиотеке имеются аналогичные функции malloc и free, у которых таких ограничений нет; в разделе 8.7 будет показано, как их можно реализовать.

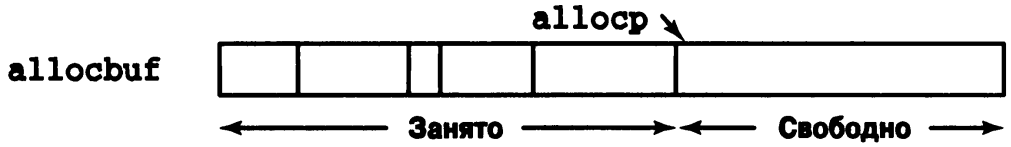
Самый простой вариант реализации alloc — это раздача ею по запросам небольших фрагментов большого текстового массива, который мы назовем allocbuf. Этот массив будет закрытым — доступным только функциям alloc и afree. Поскольку эти функции оперируют не индексами, а указателями, другим модулям программы вообще незачем знать имя этого массива, который можно объявить статическим (static) в файле исходного кода, содержащем alloc и afree, и таким образом сделать массив невидимым снаружи. В практических приложениях этот массив может и вовсе не иметь имени — например, он может быть создан вызовом функции malloc или получен по запросу к операционной системе, которая в ответ выдает указатель на неименованный блок памяти.

В каждый момент необходимо знать, какая часть массива allocbuf уже израсходована. Мы используем указатель allocp для указания на следующий свободный элемент. Когда у функции alloc запрашивают n символов, она проверяет, осталось ли в allocbuf достаточно места. Если да, то alloc возвращает текущее значение allocp (т.е. начало свободного блока) и затем увеличивает его на n, чтобы теперь указатель указывал на начало следующей свободной области. Если свободного места нет, alloc возвращает нуль. Функция afree просто делает allocp равным аргументу p, если p указывает на место внутри allocbuf.

До вызова alloc:



После вызова alloc:




```
#define ALLOCSIZE 10000 /* объем имеющейся памяти */

static char allocbuf[ALLOCSIZE]; /* буфер памяти для alloc */
static char *allocp = allocbuf;  /* следующая свободная позиция */

char *alloc(int n)  /* возвращает указатель на n символов */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* есть место */
        allocp += n;
        return allocp - n; /* старый p */
    } else  /* недостаточно места в буфере */
        return 0;
}

void afree(char *p)  /* освобождение памяти по адресу p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp= p;
}
```

В целом указатель инициализируется точно так же, как любая другая переменная, хотя обычно имеет смысл инициализировать указатели только нулями или выражениями с участием адресов ранее определенных данных соответствующего типа. Рассмотрим такое объявление:

```
static char *allocp = allocbuf;
```

В нем переменная `allocp` объявляется указателем на символьные данные и инициализируется адресом начала массива `allocbuf`, который при запуске программы является указателем на следующую свободную ячейку. Это объявление можно записать и так:

```
static char *allocp = &allocbuf[0];
```

Здесь используется тот факт, что имя массива — это одновременно адрес его нулевого элемента.

А вот как проверяется, достаточно ли места в буфере, чтобы удовлетворить запрос на n элементов:

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* есть место */
```

Если места достаточно, то новая позиция указателя `allocp` может сдвинуться максимум на элемент, следующий сразу за концом массива `allocbuf`. Если запрос можно удовлетворить, функция `alloc` возвращает указатель на начало блока символов

(обратите внимание на объявление самой функции). Если места не хватает, функция `alloc` должна как-то сигнализировать об этом. Язык C гарантирует, что 0 никогда не бывает адресом данных, поэтому возвращение нуля можно использовать как сигнал аварийного завершения — в данном случае нехватки места в буфере памяти.

Указатели и целые числа не употребляются вместе как взаимозаменяемые значения. Нуль — единственное исключение. Нулевую константу можно присваивать указателю, и указатель можно сравнивать с простым числовым нулем. Часто вместо нуля используется символическая константа NULL, которая более четко показывает, что это не просто число, а специальное значение указателя. Константа NULL определена в заголовочном файле `stdio.h`. В дальнейшем в соответствующих случаях всегда будет употребляться NULL.

Следующие условные операторы демонстрируют некоторые важные особенности адресной арифметики:

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* есть место */

if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

Первое, что видно из этих конструкций, — это то, что при определенных обстоятельствах указатели можно сравнивать. Если `p` и `q` указывают на элементы одного и того же массива, то для них корректно работают сравнения `==`, `!=`, `<`, `>`, `>=` и т.п. Например, если `p` указывает на элемент, имеющий меньший номер в массиве, чем элемент `q`, то справедливо соотношение

```
p < q
```

Любой указатель можно сравнивать с нулем, и это сравнение всегда будет иметь смысл. Но если указатели не указывают на один и тот же массив, то арифметические операции и сравнения с их участием не будут определены. (Есть одно исключение: в адресной арифметике можно применять также первый элемент после конца массива.)

Второе наблюдение уже было сделано ранее — к указателям можно прибавлять целые числа:

```
p + n
```

Эта конструкция обозначает адрес `n`-го объекта после того, на который указывает `p`. Это всегда справедливо независимо от типа данных, на которые указывает `p`; число `n` масштабируется по размеру соответствующих объектов, который определяется объявлением `p`. Например, если тип `int` имеет длину четыре байта, `n` будет прибавляться с коэффициентом 4.

Вычитание указателей также возможно: если `p` и `q` указывают на элементы одного и того же массива, и при этом `p < q`, то выражение `q-p+1` дает количество элементов от `p` до `q` включительно. Этот факт можно использовать для того, чтобы написать еще одну версию функции `strlen`:

```
/* strlen:  возвращает длину строки s */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

В этом объявлении p инициализируется значением s, т.е. устанавливается на начало строки. В цикле while по очереди анализируется каждый символ, пока не будет найден завершающий '\0'. Поскольку p указывает на символьные данные, выражение p++ всякий раз продвигает указатель на один символ вперед, а p-s дает общее количество символов, на которое он в итоге продвигается, т.е. длину строки. (Количество символов в строке может быть настолько большим, что не поместится в переменную типа int. В заголовочном файле <stddef.h> определяется тип ptrdiff_t, достаточно большой для того, чтобы содержать разность любых двух указателей со знаком. Однако, чтобы соблюсти технику безопасности, лучше всего принять size_t в качестве типа возвращаемого из strlen значения. Именно так сделано в стандартной библиотечной версии этой функции. Тип size_t — это целочисленный тип без знака; результаты этого типа дает операция sizeof.)

Адресная арифметика устроена единообразно: если нужно было бы работать с числами типа float, которые занимают больше места, чем char, и если бы p был указателем на данные типа float, то операция p++ продвигала бы указатель на следующее число. Таким образом, можно написать аналогичную версию alloc, оперирующую числами типа float вместо char, заменив везде char на float в тексте alloc и afree. Все манипуляции с указателями будут автоматически выполняться с учетом размера объектов.

К числу разрешенных операций с указателями относятся:

- присваивание указателей одного типа;
- сложение или вычитание указателя и целого числа;
- вычитание или сравнение указателей, указывающих на один и тот же массив данных;
- присваивание нуля или сравнение с ним.

Любые другие операции с адресами являются ошибками и не разрешены. Нельзя складывать два указателя, умножать, делить, сдвигать или применять маски к указателям, прибавлять к ним числа типа float или double. За исключением указателей типа void * нельзя даже присвоить указатель одного типа указателю другого без явного приведения типов.

5.5. Символьные указатели и функции

Строковые константы в языке C — это по сути массивы символов. Например:

```
"I am a string"
```

Во внутреннем представлении строки она заканчивается нулевым символом '\0', чтобы программа могла найти ее конец. Таким образом, фактическая длина строки в памяти на один символ длиннее, чем то, что написано в двойных кавычках.

Вероятно, самое популярное применение строковых констант — это их передача в функции в качестве аргументов.

```
printf("hello, world\n");
```

Если такая строка фигурирует в программе, все операции с ней выполняются посредством указателя. Функция `printf` получает указатель на начало массива символов. Итак, обращение к строковой константе выполняется по указателю на ее первый элемент.

Строковые константы не обязаны быть только аргументами функций. Пусть переменная `pmessage` объявлена следующим образом:

```
char *pmessage;
```

Тогда следующий оператор присваивает ей указатель на массив символов:

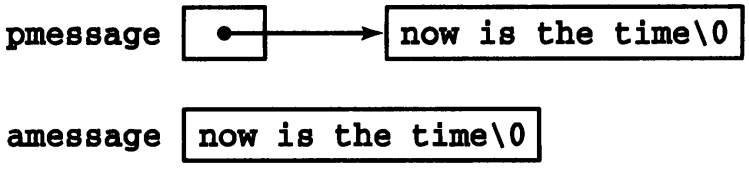
```
pmessage = "now is the time"
```

При этом *не выполняется* копирование строки. В языке С нет операций для манипулирования именно строкой символов как единым целым; вместо этого применяются операции с указателями.

Между следующими двумя определениями есть важное различие:

```
char amessage[] = "now is the time";  /* массив */
char *pmessage = "now is the time";   /* указатель */
```

В первом из них `amessage` — это массив, ровно такой длины, чтобы в него поместилась инициализирующая строка символов и завершающий `'\0'`. Отдельные символы в строке можно изменять, но переменная `amessage` всегда будет указывать на один и тот же участок памяти. А вот `pmessage` — это указатель, который после инициализации указывает на строковую константу; впоследствии этот указатель можно изменить так, чтобы он указывал в другое место, но попытка изменить содержимое строки после этого даст неопределенный результат.



Проиллюстрируем дополнительные аспекты указателей и массивов на примере двух полезных функций, адаптированных из стандартной библиотеки. Первая из них — это `strcpy(s, t)`, которая копирует строку `t` в строку `s`. Было бы неплохо иметь возможность просто написать `s=t`, но при этом скопируется только указатель, а не символы. Чтобы скопировать сами символы, нужен цикл. Вначале рассмотрим версию с массивом:

```
/* strcpy:  копирует строку t в s; версия с индексированием массива
*/
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

Ниже для сравнения приводится версия той же функции `strcpy` с указателями.

```
/* strcpy:  копирует строку t в s; 1-я версия с указателями */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
```

```
        s++;
        t++;
    }
}
```

Поскольку аргументы передаются по значениям, `strcpy` имеет право использовать параметры `s` и `t` по своему усмотрению. Здесь эти указатели уже инициализированы с самого начала; они передвигаются по своим массивам, копируя по одному символу за раз, пока не встретится символ `'\0'`, завершающий строку `t`.

На практике функция `strcpy` была бы написана не так. Опытные программисты на C предпочли бы такую запись:

```
/* strcpy:  копирует строку t в s; 2-я версия с указателями */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

Здесь инкрементирование указателей `s` и `t` включено непосредственно в проверку условия цикла. Значение выражения `*t++` — это символ, на который указывал `t` до инкрементирования; постфиксная операция `++` не изменяет `t`, пока по нему не получен нужный символ. Аналогично, сначала символ помещается по старому (не инкрементированному) адресу в `s`, а уже затем `s` инкрементируется. Этот же символ сравнивается с `'\0'` для контроля завершения цикла. В итоге результат всей этой операции таков, что символы копируются из `t` в `s`, включая и завершающий `'\0'`.

Чтобы выполнить самое последнее сокращение, заметим, что сравнение с `'\0'` излишне, поскольку проверяемое условие состоит всего-навсего в том, равно ли некоторое выражение нулю. Поэтому в итоге функция записывается так:

```
/* strcpy:  копирует строку t в s; 3-я версия с указателями */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

На первый взгляд эта запись кажется невразумительной, но на самом деле она достаточно удобна, и эту программную идиому следует освоить, поскольку в программах на C она встречается часто.

Функция `strcpy` из стандартной библиотеки (`<string.h>`) возвращает в качестве значения указатель на строку, в которую выполняется копирование.

Второй функцией, которую мы рассмотрим, будет `strcmp(s,t)`. Она сравнивает символьные строки `s` и `t`, возвращая отрицательное, нулевое или положительное значение, если строка `s` соответственно меньше, равна или больше, чем строка `t` (в алфавитном смысле). Возвращаемое значение получается вычитанием символов в первой позиции, в которой `s` и `t` не совпадают.

```
/* strcmp:  возвращает <0 при s<t, 0 при s==t, >0 при s>t */
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
```

```
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

А вот версия `strcmp` с применением указателей:

```
/* strcmp:  возвращает <0 при s<t, 0 при s==t, >0 при s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

Поскольку операции `++` и `--` бывают как постфиксными, так и префиксными, встречаются и другие их комбинации со знаком `*`, хотя несколько реже. Например:

```
*--p
```

Это выражение декрементирует указатель `p` до того, как извлечь символ, на который он указывает. Вообще говоря, следующая пара операторов представляет стандартные идиомы для помещения данных в стек и извлечения их оттуда (см. раздел 4.3):

```
*p++ = val;   /* помещение val в стек */
val = *--p;   /* извлечение val из стека */
```

Заголовочный файл `<string.h>` содержит объявления всех стандартных функций, упомянутых в этом разделе, а также целого ряда других библиотечных функций для работы со строками.

Упражнение 5.3. Напишите свою версию функции `strcat`, продемонстрированной в главе 2, с применением указателей. Напоминаем, что `strcat(s,t)` копирует строку `t` в конец строки `s`.

Упражнение 5.4. Напишите функцию `strend(s,t)`, которая бы возвращала 1, если строка `t` присутствует в конце строки `s`, и 0 в противном случае.

Упражнение 5.5. Применяя указатели, напишите ваши версии библиотечных функций `strncpy`, `strncat` и `strncmp`, которые обрабатывают не более чем `n` первых символов своих строковых аргументов. Например, функция `strncpy(s,t,n)` копирует не более `n` символов из строки `t` в строку `s`.

Упражнение 5.6. Перепишите программы из примеров и упражнений предыдущих глав, применяя указатели вместо индексов массивов. В частности, полезно будет поработать с такими функциями, как `getline` (главы 1 и 4), `atoi`, `itoa` и их версиями (главы 2–4), `reverse` (глава 3), а также `strindex` и `getop` (глава 4).

5.6. Массивы указателей и указатели на указатели

Поскольку указатели сами по себе являются переменными, их можно хранить в массивах, как и переменные других типов. Продемонстрируем это на примере программы,

сортирующей набор текстовых строк в алфавитном порядке. Это будет сильно усеченная и упрощенная версия программы `sort` из системы Unix.

В главе 3 была представлена функция сортировки массива целых чисел по Шеллу (Shell), а в главе 4 приведена более совершенная функция быстрой сортировки. Здесь будут использоваться те же алгоритмы, с тем различием, что теперь мы будем иметь дело со строками текста. Эти объекты имеют различную длину, и их нельзя сравнить или переместить одной элементарной операцией в отличие от целых чисел. Необходимо представление данных, которое бы помогло эффективно и удобно работать со строками текста переменной длины.

И здесь на сцену выходит массив указателей. Если сортируемые строки располагаются в одном длинном символьном массиве вплотную — начало одной к концу другой, то к каждой строке можно обращаться по указателю на ее первый символ. Сами же указатели можно поместить в массив. Две строки можно сравнить, передав указатели на них в функцию `strcmp`. Если необходимо поменять местами две строки, стоящие в неправильном порядке, то обмениваются только указатели на них в массиве указателей, а не сами строки текста.

Благодаря этому устраняются сразу две взаимосвязанные проблемы: сложность управления памятью и дополнительный расход ресурсов на перемещение строк символов.

Процесс сортировки состоит из трех этапов:

```
считать все строки из входного потока
отсортировать строки
вывести их в отсортированном порядке
```

Как обычно, лучше всего будет разделить программу на функции, которые соответствуют этому естественному делению задачи, а на функцию `main` возложить обязанности по управлению ими. Давайте пока оставим в стороне процесс сортировки, сосредоточившись на структуре данных и вводе-выводе.

Функция ввода должна накапливать и сохранять символы каждой строки, а также строить массив указателей на строки. Ей также придется подсчитывать количество поступающих строк, поскольку это необходимо знать при сортировке и выводе. Функция ввода может справиться только с ограниченным количеством строк, поэтому она должна возвращать некое нереальное количество — например, `-1` — в качестве сигнала о том, что входных данных слишком много.

Функция вывода должна всего лишь вывести строки в том же порядке, в каком они расставлены в массиве указателей.

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

#define MAXLINES 5000  /* максимальное количество сортируемых строк */

char *lineptr[MAXLINES]; /* указатели на строки */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
```

```
void qsort(char *lineptr[], int left, int right);

/* сортировка строк входного потока */
main()
{
    int nlines;     /* количество введенных строк */

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("error: input too big to sort\n");
        return 1;
    }
}

#define MAXLEN 1000   /* максимальная длина входной строки */
int getline(char *, int);
char *alloc(int);

/* readlines:  считывание строк из входного потока */
int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
    char *p, line[MAXLEN];

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || (p = alloc(len)) == NULL)
            return -1;
        else {
            line[len-1] = '\0'; /* удаление конца строки */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}

/* writelines:  вывод строк в выходной поток */
void writelines(char *lineptr[], int nlines)
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

Функция `getline` берется из раздела 1.9.

Основной новый элемент в этом коде — объявление `lineptr`. Это массив указателей на данные типа `char`, общей длиной `MAXLINES` элементов:

```
char *lineptr[MAXLINES];
```

Таким образом, `lineptr[i]` является указателем на символьные данные, а `*lineptr[i]` — символом, на который он указывает, или первым символом i-й хранящейся в массиве строки.

Поскольку `lineptr` является именем массива, с ним можно обращаться как с указателем таким же образом, как в предыдущих примерах. Тогда функцию `writelines` можно переписать так:

```
/* writelines:  вывод строк в выходной поток */
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}
```

Первоначально `*lineptr` указывает на первую строку, а каждое инкрементирование передвигает его на следующий указатель на строку; при этом отсчет строк ведется уменьшением `nlines` до нуля.

Справившись с вводом и выводом, можно приступать к сортировке. Алгоритм быстрой сортировки из главы 4 придется немножко переделать: нужно внести изменения в объявления, а сравнение выполнять функцией `strcmp`. Сам алгоритм остается в точности тем же, и это позволяет думать, что он будет работать так же хорошо, как и раньше.

```
/* qsort:  сортировка v[left]...v[right] в порядке возрастания */
void qsort(char *v[], int left, int right)
{

    int i, last;
    void swap(char *v[], int i, int j);

    if (left >= right) /* ничего не делать, если в массиве */
        return;        /* меньше двух элементов */
    swap(v, left, (left+right)/2);
    last = left;
    for (i = left+1; i<= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Аналогично, функция `swap` требует лишь самой тривиальной доработки:

```
/* swap:  обмен местами v[i] и v[j] */
void swap(char *v[], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Поскольку отдельные элементы `v` (фактически `lineptr`) — указатели на символьные данные, переменная `temp` должна иметь тот же тип, чтобы в нее и из нее можно было копировать указатели.

Упражнение 5.7. Перепишите функцию `readlines` так, чтобы строки помещались в массив, предоставленный функцией `main`, без вызова `alloc` для распределения памяти. Насколько быстрее получится программа?

5.7. Многомерные массивы

В языке С есть возможность работать с многомерными прямоугольными массивами, хотя на практике они используются гораздо реже, чем массивы указателей. В этом разделе мы рассмотрим некоторые свойства таких массивов.

Проанализируем задачу преобразования даты из дня месяца в порядковый день года и наоборот. Например, 1-е марта — это 60-й день невисокосного года или 61-й день високосного. Определим две функции для такого преобразования: `day_of_year` будет преобразовывать месяц и день в день года, а `month_day` — день года в месяц и день. Поскольку вторая из функций вычисляет два значения, соответствующие аргументы (месяц и день) будут представлены указателями. Так, после выполнения следующего оператора `m` станет равной 2, а `d` — 29 (двадцать девятое февраля):

```
month_day(1988, 60, &m, &d);
```

Этим функциям понадобится одна и та же справочная информация, а именно таблица количества дней в каждом месяце. Поскольку количество дней в месяцах отличается для високосного и обычного года, их лучше занести отдельно в строки двумерного массива, чем вычислять на ходу в программе, сколько дней в феврале того или иного года. В итоге получаем следующий массив и функции для выполнения преобразования:

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year:  вычисление дня года по месяцу и дню */
int day_of_year(int year, int month, int day)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}

/* month_day:  вычисление месяца и даты по дню года */
void month_day(int year, int yearday, int *pmonth, int *pday)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; yearday > daytab[leap][i]; i++)
        yearday -= daytab[leap][i];
    *pmonth = i;
    *pday = yearday;
}
```

Напомним, что арифметическое значение логического выражения — например, такого, которое присваивается переменной `leap`, — равно нулю (ложь) или единице (истина). Поэтому его можно использовать в качестве индекса массива `daytab`.

Массив `daytab` должен быть внешним по отношению к функциям `day_of_year` и `month_day`, чтобы они обе могли им пользоваться. Его элементы имеют тип `char`; этим подчеркивается, что переменные данного типа можно использовать для хранения небольших целых чисел вместо символов.

Переменная `daytab` — это первый случай использования двумерного массива. В языке C двумерный массив — это фактически одномерный массив, каждый элемент которого в свою очередь является массивом. Поэтому его индексы записываются так:

```
daytab[i][j]     /* [строка][столбец] */
```

А следующая запись является ошибочной:

```
daytab[i,j]      /* НЕПРАВИЛЬНО */
```

За исключением этого отличия в записи, с двумерным массивом можно обращаться точно так же, как и в других языках программирования. Элементы располагаются в памяти по строкам, так что правый индекс — номер столбца — изменяется быстрее при обращении к элементам в порядке их размещения.

Массив инициализируется списком значений в фигурных скобках; каждая строка многомерного массива инициализируется соответствующим вложенным подсписком. Массив `daytab` начинается со столбца нулей, чтобы месяцы нумеровались в естественном порядке от 1 до 12, а не от 0 до 11. Поскольку экономия памяти в данном случае не актуальна, так работать удобнее, чем вычислять номера месяцев по индексам.

Если двумерный массив необходимо передать в функцию, среди ее параметров должно быть количество столбцов. Количество строк необязательно, поскольку фактически передается, как и раньше, указатель на массив строк, каждая длиной 13 элементов типа `char`. Другими словами, это указатель на объект, представляющий собой массив из 13 элементов типа `char`. Так что если массив `daytab` нужно было бы передать в функцию `f`, объявление этой функции выглядело бы следующим образом:

```
f(char daytab[2][13]) { ... }
```

Его можно записать и так:

```
f(char daytab[][13]) { ... }
```

Здесь используется то, что количество строк не имеет значения при передаче такого массива. Еще один вариант таков:

```
f(char (*daytab)[13]) { ... }
```

В этом объявлении сообщается, что параметр функции — указатель на массив из 13 элементов. Наличие круглых скобок здесь обязательно, поскольку квадратные скобки (`[]`) имеют более высокий приоритет, чем знак ссылки (`*`). Без скобок это было бы объявление массива из 13 указателей на данные типа `char`:

```
char *daytab[13]
```

Обобщая, можно сказать, что длину первого измерения (диапазона индексов) массива можно не указывать, а вот наличие остальных обязательно.

В разделе 5.12 рассматриваются аналогичные и другие виды сложных объявлений.

Упражнение 5.8. В функциях `day_of_year` и `month_day` нет никакого контроля ошибок в данных. Устраните эту недоработку.

5.8. Инициализация массивов указателей

Рассмотрим такую задачу: написать функцию `month_name(n)`, которая возвращает указатель на строку символов, содержащую название n-го месяца. Это идеальный случай для применения внутреннего статического массива. Функция `month_name`, приведенная ниже, содержит закрытый от внешнего доступа массив символьных строк, и при вызове возвращает указатель на соответствующую строку. В этом разделе демонстрируется инициализация массива имен.

В целом синтаксис аналогичен предыдущим рассмотренным инициализациям:

```
/* month_name:  возвращает имя n-го месяца */
char *month_name(int n)
{
    static char *name[] = {
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };

    return (n < 1 || n > 12) ? name[0] : name[n];
}
```

Объявление массива символьных указателей, `name`, аналогично объявлению `lineptr` в примере с сортировкой. Инициализируется он списком символьных строк, каждая из которых помещается в соответствующую позицию массива. Символы i-й строки помещаются в какое-то место в памяти, а указатель на них — в элемент `name[i]`. Поскольку размер массива явно не указан, компилятор сам подсчитывает инициализирующие значения и таким образом вычисляет требуемую длину.

5.9. Указатели и многомерные массивы

Начинающие программисты на C часто путаются в различиях между двумерными массивами и массивами указателей, такими как `name` в приведенном выше примере. Рассмотрим два объявления-определения:

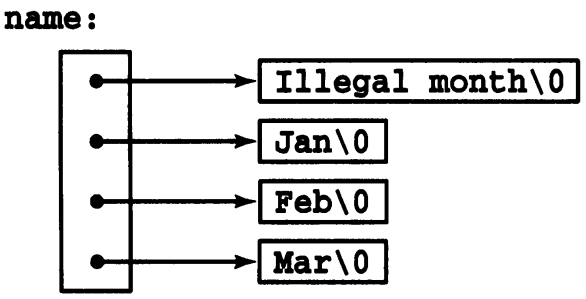
```
int a[10][20];
int *b[10];
```

Как `a[3][4]`, так и `b[3][4]` — вполне законные обращения к одному элементу данных типа `int`. При этом `a` — полноправный двумерный массив; для него выделено 200 ячеек памяти размера `int`, и для поиска элемента `a[`*строка*`][`*столбец*`]` используется перевод координат прямоугольного массива в линейный адрес по формуле `20 ×` *строка* `+` *столбец*. А вот для переменной `b` ее определение всего лишь выде-

ляет память для 10 указателей без их инициализации; инициализация должна выполняться явным образом — статически или программно. Предполагая, что каждый элемент b действительно указывает на массив из двадцати элементов, получаем те же 200 ячеек типа int плюс десять ячеек для указателей. Важное преимущество массива указателей состоит в том, что строки массива могут иметь различную длину, т.е. каждый элемент b не обязан указывать на вектор из 20 элементов. Одни указатели могут указывать на два элемента, другие — на пятьдесят, а третьи — вообще ни на что.

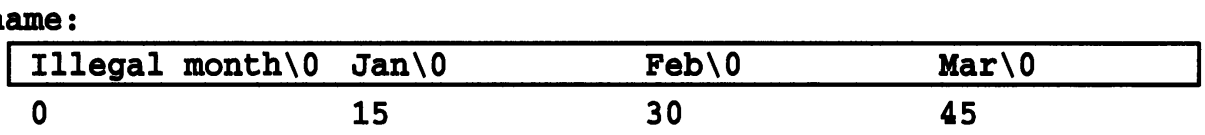
Хотя мы сейчас обсуждаем этот вопрос на примере целочисленных данных, самое популярное применение массивов указателей — это хранение символьных строк различной длины, как в функции month_name. Пусть имеется массив указателей со следующим объявлением и символическим изображением:

```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };
```



Сравните это объявление и изображение с аналогичным представлением двумерного массива:

```
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```



Упражнение 5.9. Перепишите функции day_of_year и month_day с применением указателей вместо индексирования массивов.

5.10. Аргументы командной строки

В системных средах, поддерживающих язык С, существует способ передавать в программу аргументы или параметры командной строки при запуске программы на выполнение. При вызове функции main она получает два аргумента. Первый, который обычно называют argc (от *argument count — счетчик аргументов*), содержит количество аргументов командной строки, с которыми была запущена программа. Второй, обычно под именем argv (от *argument vector — вектор аргументов*), указывает на массив символьных строк, содержащих сами аргументы, — по одному в строке. Для манипулирования этими символьными строками, естественно, используется многоуровневая система указателей.

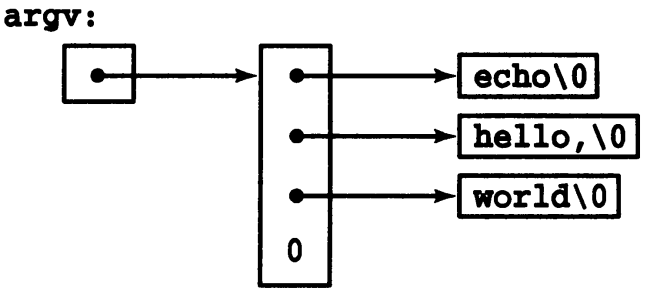
Самая простая демонстрация работы с аргументами командной строки — это программа echo, рассматриваемая ниже. Программа выводит свои аргументы в одной строке, разделяя их пробелами. Пусть, например, программа запущена на выполнение таким образом:

```
echo hello, world
```

Тогда она выдаст на экран следующее:

```
hello, world
```

По определению `argv[0]` содержит имя, под которым запускается программа, поэтому `argc` всегда не меньше 1. Если счетчик `argc` равен 1, то после имени программы нет никаких аргументов командной строки. В приведенном выше примере `argc` равен 3, а `argv[0]`, `argv[1]` и `argv[2]` содержат соответственно `"echo"`, `"hello,"` и `"world"`. Первый необязательный аргумент хранится в `argv[1]`, а последний — в `argv[argc-1]`. Кроме того, стандарт требует, чтобы элемент `argv[argc]` был нулевым указателем.



Первая версия программы `echo` воспринимает `argv` как массив символьных указателей:

```
#include <stdio.h>

/* echo:  вывод аргументов командной строки; 1-я версия */
main(int argc, char *argv[])
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

Поскольку `argv` — это указатель на массив указателей, можно работать с ним как с указателем, а не как с массивом. В следующем варианте программы используется инкрементирование `argv`, являющегося указателем на указатель на `char`. При этом `argc` уменьшается до нуля.

```
#include <stdio.h>

/* echo:  вывод аргументов командной строки; 2-я версия */
main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

Поскольку `argv` — это указатель на начало массива строковых аргументов, увеличение его на единицу (`++argv`) приводит к тому, что он начинает указывать на элемент

argv[1]. Каждое последующее инкрементирование передвигает указатель на следующий аргумент; *argv соответственно уже непосредственно указывает на строку аргумента. Одновременно декрементируется счетчик argc; как только он становится равным нулю, вывод аргументов заканчивается.

Оператор вызова printf можно было бы записать и так:

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

Отсюда видно, что в качестве строки формата printf также может использоваться и выражение.

В качестве второго примера внесем некоторые усовершенствования в программу поиска по текстовому образцу из раздела 4.1. Напомним, что в тот раз образец был "зашит" прямо в программу, что явно неудобно с точки зрения ее универсальности. Следуя принципам устройства программы grep из системы Unix, изменим нашу программу так, чтобы образец для поиска задавался первым аргументом командной строки.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find:  вывод строк, содержащих образец из 1-го аргумента */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;

    if (argc != 2)
        printf("Usage: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
            }
    return found;
}
```

Стандартная библиотечная функция strstr(s,t) возвращает указатель на первую найденную подстроку t в строке s или NULL, если не найдено ни одной. Функция объявлена в файле <string.h>.

Теперь эту базовую модель можно доработать, чтобы продемонстрировать еще несколько конструкций с указателями. Предположим, нужно добавить возможность запуска программы с двумя необязательными параметрами. Один из них должен приказывать "вывести все строки, *кроме* тех, которые соответствуют образцу", а второй — "добавить в начало каждой строки ее порядковый номер".

В программах на C в системе Unix принято, чтобы необязательные аргументы или ключи начинались со знака "минус". Например, для поиска по обратному алгоритму можно ввести ключ -x (от *except* — *кроме*), а для вывода нумерации строк — ключ -n (от *number* — *номер*). В этом случае следующая командная строка задает вывод всех строк, в которых нет заданного образца, с номерами перед ними:

```
find -x -n образец
```

Кроме того, нужно разрешить ставить необязательные аргументы в любом порядке; к тому же основная часть программы не должна зависеть от количества задаваемых аргументов. Для большего удобства пользователей желательно также, чтобы аргументы-ключи можно было комбинировать:

```
find -nx образец
```

Ниже приведена итоговая программа.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: вывод строк, содержащих образец из последнего аргумента */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    long lineno = 0;
    int c, except = 0, number = 0, found = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                printf("find: illegal option %c\n", c);
                argc = 0;
                found = -1;
                break;
            }
    if (argc != 1)
        printf("Usage: find -x -n pattern\n");
    else
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((strstr(line, *argv)! = NULL)! = except){
                if (number)
                    printf("%ld:", lineno);
                printf("%s", line);
                found++;
            }
        }
    return found;
}
```

Счетчик `argc` декрементируется, а указатель `argv` инкрементируется перед каждым необязательным аргументом-ключом. В конце цикла, если не возникло никаких ошибок, счетчик `argc` сообщает, сколько еще аргументов остались необработанными, а `argv` указывает на первый из них. Таким образом, `argc` должен быть равен 1, а `*argv` —

указывать на заданный образец. Заметьте, что `*++argv` указывает на строку-аргумент, так что `(*++argv)[0]` является его первым символом. (Другая эквивалентная форма выглядела бы как `**++argv`.) Обращение по индексу `[]` выше по приоритету, чем инкремент `++` и ссылка `*`, поэтому нужно заключить аргумент в круглые скобки; без них выражение воспримется как `*++(argv[0])`. Кстати, именно эта форма и используется во внутреннем цикле, задача которого — пройти по конкретной строке аргумента. Во внутреннем цикле как раз выражение `*++argv[0]` и инкрементирует указатель `argv[0]`.

Только в редких случаях используются выражения с адресной арифметикой еще более сложные, чем приведенные выше; в таких случаях удобнее будет разбить вычисления на два-три этапа.

Упражнение 5.10. Напишите программу `expr`, которая бы вычисляла выражение в обратной польской записи из командной строки; каждый операнд и знак операции должен быть отдельным аргументом. Например, следующая командная строка задает вычисление выражения `2×(3+4)`:

```
expr 2 3 4 + *
```

Упражнение 5.11. Усовершенствуйте программы `entab` и `detab` (написанные в качестве упражнения в главе 1) так, чтобы они принимали в качестве аргументов командной строки список позиций табуляции. Если аргументы не заданы, должен использоваться заранее заданный стандартный список.

Упражнение 5.12. Сделайте так, чтобы программы `entab` и `detab` понимали следующую сокращенную запись:

```
entab -m +n
```

Это означает "вставить табуляции каждые *n* столбцов, начиная со столбца *m*". Выберите удобные (для пользователя) значения по умолчанию.

Упражнение 5.13. Напишите программу `tail` для вывода последних `n` строк ее входного потока данных. Пусть значение `n` по умолчанию будет равно, скажем, десяти, но сделайте так, чтобы его можно было изменить необязательным аргументом. Пусть следующая командная строка задает вывод последних *n* строк:

```
tail -n
```

Программа должна вести себя устойчиво и рационально даже с самыми нелепыми значениями *n* и содержимым входного потока. Напишите ее так, чтобы она максимально экономно использовала память: строки должны храниться таким образом, как в программе сортировки из раздела 5.6, а не в двумерном массиве фиксированного размера.

5.11. Указатели на функции

В языке C функция сама по себе не является переменной, но зато можно определить указатели на функции, которые разрешено присваивать, хранить в массивах, передавать в функции, возвращать из функций и т.п. Продемонстрируем это на примере модификации программы сортировки, ранее разработанной в этой главе. Новая программа будет работать так: если в командной строке задан необязательный аргумент `-n`, она будет сортировать входящие строки не по алфавиту, а по числовому значению.

Алгоритм сортировки часто состоит из трех частей: процедуры сравнения, определяющей способ упорядочения любой пары объектов; процедуры обмена местами, изменяющей порядок следования такой пары; собственно процедуры сортировки, выполняющей сравнение и обмен, пока объекты не окажутся упорядоченными. Сама сортировка независима от вида операций сравнения и обмена, поэтому можно задать сортировку по различным критериям, передавая в алгоритм разные функции сравнения и обмена. Именно этот подход принят в нашей новой программе сортировки.

Алфавитное сравнение двух строк выполняется функцией `strcmp`, как и раньше. Понадобится также функция `numcmp`, сравнивающая две строки по критерию числового значения и возвращающая результаты того же типа, что и `strcmp`. Эти функции объявлены перед `main`, и в `qsort` передается указатель на ту из них, какая требуется в текущий момент. Обработка ошибок в аргументах здесь опущена, чтобы сосредоточиться на главном.

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

#define MAXLINES 5000     /* максимальное количество строк */
char *lineptr[MAXLINES];  /* указатели на строки */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(void *lineptr[], int left, int right,
           int (*comp)(void *, void *));
int numcmp(char *, char *);

/* сортировка строк из входного потока */
main(int argc, char *argv[])
{
    int nlines;           /* количество считанных строк */
    int numeric = 0;      /* 1, если числовая сортировка */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void **) lineptr, 0, nlines-1,
          (int (*)(void*,void*))(numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("input too big to sort\n");
        return 1;
    }
}
```

При вызове функции `qsort` имена `strcmp` и `numcmp` являются указателями. Поскольку это функции, знак `&` перед ними не нужен, точно так же, как он не был нужен перед именами массивов.

Функция `qsort` написана таким образом, чтобы обрабатывать данные любого типа, а не только символьные строки. Как указано в прототипе функции, `qsort` принимает массив указателей, два целых числа и функцию с двумя аргументами-указателями, имеющими нетипизированную форму `void *`. Любой указатель можно привести к типу `void *`

и обратно без потери информации, поэтому можно смело вызывать qsort, приводя его аргументы к этому типу. Аргумент-указатель на функцию сравнения приводится к нужному типу довольно хитроумным выражением. Фактическое внутреннее представление данных редко зависит от наличия такого приведения, но все же лучше убедить компилятор, что в вызове все правильно.

```
/* qsort:  сортировка v[left]...v[right] в порядке возрастания */
void qsort(void *v[], int left, int right,
           int (*comp)(void *, void *))
{
    int i, last;
    void swap(void *v[], int i, int j);

    if (left >= right) /* ничего не делать, если в массиве */
        return;        /* меньше двух элементов */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i<= right; i++)
        if ((*comp)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}
```

Внимательно изучите объявления в этой функции. Четвертый параметр qsort объявлен как

```
int (*comp)(void *, void *)
```

Здесь сообщается, что comp — это указатель на функцию, принимающую два аргумента типа void * и возвращающую число типа int.

Использование comp в функции согласовано с объявлением:

```
if ((*comp)(v[i], v[left]) < 0)
```

Поскольку comp — указатель на функцию, *comp — это сама функция, а вызов ее выполняется соответственно таким образом:

```
(*comp)(v[i], v[left])
```

Круглые скобки нужны для установления правильных связей между компонентами выражения; без них получилось бы объявление функции, возвращающей указатель на int, т.е. нечто совершенно другое:

```
int *comp(void *, void *)     /* НЕПРАВИЛЬНО */
```

Функция strcmp, сравнивающая две строки, уже демонстрировалась ранее. Поэтому ниже приводится только новая функция numcmp, сравнивающая две строки по критерию их числовых значений (если число не занимает всю строку, берется только ее начало, имеющее числовое значение). Числовые значения вычисляются путем вызова функции atof.

```
#include <stdlib.h>

/* numcmp: сравнение строк s1 и s2 по числовым значениям */
int numcmp(char *s1, char *s2)
{
```

```
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

Функция `swap`, обменивающая местами два указателя, идентична той, которая уже рассматривалась в этой главе, с тем исключением, что указатели теперь имеют тип `void *`:

```
void swap(void *v[], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

В программу сортировки можно добавить еще много всяких возможностей. Реализация некоторых из них представляет собой нелегкую и интересную задачу.

Упражнение 5.14. Доработайте программу сортировки так, чтобы она обрабатывала ключ `-r`, требующий сортировки в обратном порядке (по убыванию). Сделайте так, чтобы ключ `-r` работал и при наличии ключа `-n`.

Упражнение 5.15. Добавьте в программу ключ `-f`, по которому верхний и нижний регистр при сортировке не различаются; например, `a` и `A` считаются одинаковыми символами.

Упражнение 5.16. Добавьте в программу ключ `-d` (от *directory order — упорядочение каталожного типа*), при наличии которого сравнение должно выполняться только по буквам, цифрам и пробелам. Обеспечьте совместимость этого ключа с `-f`.

Упражнение 5.17. Добавьте в программу возможность обработки полей, т.е. выполнения сортировки по полям внутри строк, причем каждое поле должно сортироваться с независимым набором параметров. (Так, предметный указатель оригинала этой книги был отсортирован с ключом `-df` по категориям и с ключом `-n` по номерам страниц.)

5.12. Сложные объявления

Язык C иногда обвиняют в чрезмерно сложном синтаксисе объявлений — в частности, тех, в которых участвуют указатели на функции. Синтаксис языка построен так, чтобы объявление и использование переменных были максимально согласованы. Для простых случаев это получается вполне удачно, а вот для сложных может возникать путаница, поскольку объявления не всегда читаются слева направо, да еще и бывают перегружены скобками. Хорошим примером может служить различие между этими двумя объявлениями:

```
int *f();      /* f - функция, возвращающая указатель на int */
int (*pf)();   /* pf - указатель на функцию, возвращающую int */
```

Хотя знак * обозначает префиксную операцию, он имеет более низкий приоритет, чем функциональные круглые скобки, поэтому для правильного комбинирования частей объявления необходимы еще и дополнительные скобки.

На практике редко встречаются очень сложные объявления, но тем не менее нужно понимать их, а по необходимости и писать. Один из удачных подходов — синтезировать объявления постепенно, с помощью оператора typedef, который рассматривается в разделе 6.7. А в качестве альтернативы в этом разделе будет представлена пара программ для перевода объявлений с языка С в словесное описание и наоборот. Словесное описание будет читаться строго слева направо.

Первая программа, dcl, более сложная. Она преобразует объявление на языке С в словесное описание, как в следующих примерах:

```
char **argv
    argv:  pointer to pointer to char
   (argv:  указатель на указатель на char)
int (*daytab)[13]
    daytab:  pointer to array[13] of int
   (daytab:  указатель на массив[13] типа int)
int *daytab[13]
    daytab:  array[13] of pointer to int
   (daytab:  массив[13] указателей на int)
void *comp()
    comp:  function returning pointer to void
   (comp:  функция, возвращающая указатель на void)
void (*comp)()
    comp:  pointer to function returning void
   (comp:  указатель на функцию, возвращающую void)
char (*(*x())[])()
    x: function returning pointer to array[] of
    pointer to function returning char
   (x: функция, возвращающая указатель на массив[]
    указателей на функцию, возвращающую char)
char (*(*x[3])())[5]
    x: array[3] of pointer to function returning
    pointer to array[5] of char
   (x: массив[3] указателей на функцию, возвращающую
    указатель на массив[5] типа char)
```

Программа dcl основана на грамматике, определяющей объявления, которая подробно и точно описана в приложении А, раздел А.8.5. Здесь приводится упрощенная форма:

```
обвл:        необяз. * прям-обвл
прям-обвл:   имя
             (обвл)
             прям-обвл()
             прям-обвл[необяз. размер]
```

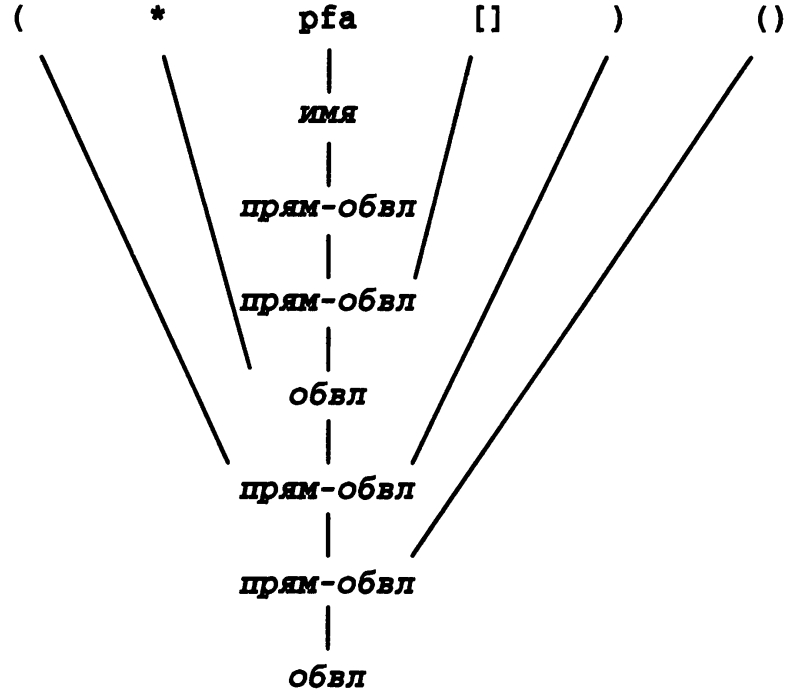
Перескажем все это словами. Объект *обвл* (объявление) является *прям-обвл* (прямым объявлением), перед которым могут стоять знаки *. В свою очередь, *прям-обвл* представляет собой *имя*, или *обвл* в круглых скобках, или *прям-обвл* с круглы-

ми скобками после него, или *прям-обвл* с квадратными скобками после него, в которых может указываться необязательный размер.

С помощью этой грамматики можно выполнять синтаксический анализ объявлений. Для примера рассмотрим следующий элемент объявления:

```
(*pfa[])()
```

Идентификатор `pfa` — это *имя*, а следовательно, *прям-обвл*. Тогда `pfa[]` — тоже *прям-обвл*. В таком случае `*pfa[]` по определению является *обвл*, а `(*pfa[])` — *прям-обвл*. Тогда `(*pfa[])()` — также *прям-обвл* и, следовательно, *обвл*. Этот синтаксический анализ можно проиллюстрировать приведенным ниже деревом.



Сердцевина программы `dcl` — это пара функций `dcl` и `dirdcl`, которые раскладывают объявление согласно приведенной выше грамматике. Поскольку грамматика определена рекурсивно, функции также вызывают друг друга рекурсивно, распознавая соответствующие фрагменты объявления. Поэтому программа называется синтаксическим анализатором по методу *рекурсивного спуска*.

```
/* dcl: синтаксический анализ объявления */
void dcl(void)
{
    int ns;

    for (ns = 0; gettoken() == '*'; )  /* подсчет знаков * */
        ns++;
    dirdcl();
    while (ns-- > 0)
        strcat(out, " pointer to");
}

/* dirdcl:  синтаксический анализ прямого объявления */
void dirdcl(void)
{
    int type;

    if(tokentype == '(') {              /* ( обвл ) */
```

```
        dcl();
        if (tokentype != ')')
            printf("error: missing )\n");
    } else if (tokentype == NAME)    /* имя переменной */
        strcpy(name, token);
    else
        printf("error: expected name or (dcl)\n");
    while ((type=gettoken()) == PARENS || type == BRACKETS)
        if (type == PARENS)
            strcat(out, " function returning");
        else {
            strcat(out, " array");
            strcat(out, token);
            strcat(out, " of");
        }
}
```

Поскольку эти программы всего лишь демонстрационные, они не отличаются "дуракоустойчивостью" и работают лишь со значительными ограничениями. Функции `dcl` и `dirdcl` могут справиться лишь с простыми типами данных наподобие `char` и `int`. Они не воспринимают объявления типов аргументов в функциях, а также модификаторы наподобие `const`. Также их запутывают лишние пробелы. Обработка ошибок налажена не слишком хорошо, поэтому и неправильные объявления также сбивают программу с толку. Соответствующие усовершенствования можно внести в качестве упражнения.

Далее приведены глобальные переменные и главная программа:

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

#define MAXTOKEN  100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);

int gettoken(void);
int tokentype;              /* тип последней лексемы */
char token[MAXTOKEN];       /* последняя введенная лексема */
char name[MAXTOKEN];        /* имя идентификатора */
char datatype[MAXTOKEN];    /* тип данных = char, int и т.п. */
char out[1000];             /* строка результата */

main()  /* преобразование объявлений в словесную форму */
{
    while (gettoken() != EOF) {    /* 1-я лексема в строке - */
        strcpy(datatype, token);  /* тип данных */
        out[0] = '\0';
        dcl();       /* анализ остальной части строки */
        if (tokentype != '\n')
            printf("syntax error\n");
        printf("%s: %s %s\n", name, out, datatype);
```

```
    return 0;
}
```

Функция gettoken пропускает пробелы и табуляции, затем находит следующую лексему (смысловой элемент объявления) в потоке. Под *лексемой* (*token*) будем понимать имя, пару круглых скобок, пару квадратных скобок (возможно, с числом между ними) или любой другой одиночный символ.

```
int gettoken(void)  /* считывает очередную лексему */
{
    int c, getch(void);
    void ungetch(int);
    char *p = token;

    while ((c = getch()) == ' ' || c == '\t')
        ;
    if (c == '(') {
        if ((c = getch()) == ')') {
            strcpy(token), "()");
            return tokentype = PARENS;
        } else {
            ungetch(c);
            return tokentype = '(';
        }
    } else if (c == '[') {
        for (*p++ = c; (*p++ = getch()) != ']'; )
            ;
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (isalpha(c)) {
        for (*p++ = c; isalnum(c = getch()); )
            *p++ = c;
        *p = '\0';
        ungetch(c);
        return tokentype = NAME;
    } else
        return tokentype = c;
}
```

Функции getch и ungetch рассматривались в главе 4.

Противоположная операция проще в реализации, особенно если не бояться генерировать лишние круглые скобки. Ниже приведена программа undcl, которая преобразует словесные описания наподобие "x — функция, возвращающая указатель на массив указателей на функцию, возвращающую char", в соответствующие объявления:

```
char (*(*x())[])()
```

Словесные описания должны предварительно представляться в таком виде:

```
x () * [] * () char
```

Сокращенный синтаксис входного потока позволяет вновь воспользоваться функцией gettoken. Программа undcl использует те же внешние переменные, что и dcl.

```
/* undcl:  перевод словесных описаний в объявления */
main()
{
```

```
    int type;
    char temp[MAXTOKEN];

    while (gettoken() != EOF) {
        strcpy(out, token);
        while ((type = gettoken()) != '\n')
            if (type == PARENS || type == BRACKETS)
                strcat(out, token);
            else if (type == '*') {
                sprintf(temp, "(*%s)", out);
                strcpy(out, temp);
            } else if (type == NAME) {
                sprintf(temp, "%s %s", token, out);
                strcpy(out, temp);
            } else
                printf("invalid input at %s\n", token);
        printf("%s\n", out);
    }
    return 0;
}
```

Упражнение 5.18. Доработайте `dcl` так, чтобы программа справлялась с обработкой ошибок во входных данных.

Упражнение 5.19. Доработайте `undcl` так, чтобы программа не добавляла в объявления лишних круглых скобок.

Упражнение 5.20. Усовершенствуйте `dcl` так, чтобы программа обрабатывала объявления с типами аргументов функций, модификаторами наподобие `const` и т.п.

Глава 6

Структуры

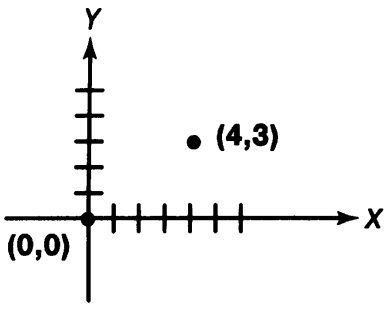
Структура — это совокупность нескольких переменных, часто различных типов, сгруппированных под единым именем для удобства обращения. (В некоторых языках — например, в Pascal — структуры называются записями.) С помощью структур удобно организовывать сложные данные — в частности, в больших программах, поскольку благодаря им группу связанных друг с другом переменных можно рассматривать и обрабатывать как единое целое, а не как разрозненный набор элементов.

Традиционным примером структуры является список работников какого-нибудь предприятия: в нем каждый сотрудник описывается набором таких атрибутов, как имя, адрес, номер полиса социального страхования, размер зарплаты и т.д. Некоторые из этих атрибутов сами могут быть структурами: например, имя состоит из нескольких компонентов, как и адрес, и даже зарплата. Еще один пример, более типичный для С, взят из компьютерной графики: точка описывается парой координат, прямоугольник — парой точек и т.д.

Главное изменение, внесенное стандартом ANSI в работу со структурами, — это определение присваивания структур. Теперь структуры можно копировать, присваивать, передавать в функции и возвращать из функций. Многие компиляторы поддерживают эти возможности уже много лет, но в настоящее время они наконец-то строго определены. Автоматические структуры и массивы теперь также можно инициализировать.

6.1. Основы работы со структурами

Для начала создадим несколько структур, подходящих для работы с графикой. Основным графическим объектом является точка, которая описывается своей координатой x и координатой y, причем обе координаты — целые числа.



Две координатные компоненты можно организовать в виде структуры, объявленной следующим образом:

```
struct point {
    int x;
    int y;
};
```

Ключевое слово `struct` начинает объявление структуры, состоящее из списка объявлений элементов в фигурных скобках. После слова `struct` может стоять необязательный идентификатор, именуемый *меткой структуры* (здесь это `point`). Метка обозначает конкретный структурный тип; впоследствии ее можно использовать для краткости, опуская все, что находится в скобках при объявлении структур того же вида.

Переменные, перечисленные в объявлении структуры, называются ее *членами*, *элементами* или *полями*. Элемент структуры или ее метка может иметь то же имя, что и обыкновенная переменная (не член структуры) безо всякого конфликта, поскольку они всегда отличаются по контексту. Более того, в разных структурах можно использовать одни и те же имена элементов, хотя с точки зрения хорошего стиля так можно делать только тогда, когда речь идет о близкородственных объектах.

Объявление со словом `struct` фактически вводит новый тип данных. После правой скобки, завершающей список элементов, может стоять список переменных, как и после имени любого элементарного типа:

```
struct { ... } x, y, z;
```

Эта запись имеет ту же синтаксическую форму, что и любая рассмотренная ранее:

```
int x, y, z;
```

Каждый из этих двух операторов объявляет переменные `x`, `y`, `z` определенного именованного типа и выделяет для них место в памяти.

Объявление структуры, после которого нет списка переменных, не выделяет никакой памяти для объектов, а просто описывает форму или "шаблон" структуры. Если в объявлении присутствует метка, ее можно использовать позже при определении экземпляров структуры. Например, при наличии объявления структуры `point` можно позже определить переменную `pt`, имеющую тип `struct point`:

```
struct point pt;
```

Структуру можно инициализировать, поставив после ее определения список значений-констант:

```
struct point maxpt = { 320, 200 };
```

Автоматические структуры также можно инициализировать путем присваивания или вызова функции, которая возвращает структуру соответствующего типа.

Обращение к элементам структуры (при включении их в выражение и т.п.) выполняется с помощью следующей конструкции:

имя-структуры.элемент

Знак операции обращения к элементу структуры — точка — ставится между именем структуры и именем элемента. Например, вывести координаты точки `pt` можно так:

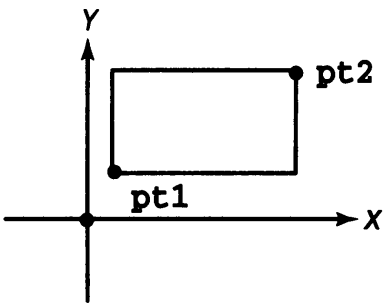
```
printf("%d,%d", pt.x, pt.y);
```

Чтобы вычислить расстояние от начала координат (0,0) до `pt`, запишем следующее:

```
double dist, sqrt(double);

dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Структуры можно вкладывать друг в друга. Так, одно из возможных представлений прямоугольника — это пара точек, обозначающих два противоположных по диагонали угла.




```
struct rect {
    struct point pt1;
    struct point pt2;
}
```

Структура `rect` содержит две структуры `point`. Пусть объявлена структурная переменная `screen`:

```
struct rect screen;
```

Тогда следующее выражение будет обозначать координату `x` элемента `pt1` этой структуры:

```
screen.pt1.x
```

6.2. Структуры и функции

Разрешенными операциями над структурами являются копирование или присваивание структуры как целого, взятие ее адреса операцией `&`, а также обращение к ее элементам. Копирование и присваивание включают в себя также передачу аргументов в функции и возвращение значений из функций. Структуры нельзя сравнивать между собой. Структуру можно инициализировать списком констант-инициализаторов для всех ее полей; автоматическую структуру также можно инициализировать присваиванием.

Рассмотрим вопросы работы со структурами на примере — напишем ряд функций для манипулирования точками и прямоугольниками. Тут есть как минимум три возможных подхода к решению: передавать отдельные компоненты структур, целые структуры или указатели на них. В каждом подходе есть свои сильные и слабые места.

Первая функция, `makepoint`, принимает два целых числа и возвращает структуру `point`:

```
/* makepoint:  делает структуру point из компонентов x и y */
struct point makepoint(int x, int y)
{
    struct point temp;

    temp.x = x;
    temp.y = y;
    return temp;
}
```

Заметьте, что между именами аргументов и членов структуры нет никакого конфликта; наоборот, использование одинаковых обозначений подчеркивает родственную природу данных.

Функцию makepoint можно использовать для динамической инициализации любой структуры или для передачи аргументов-структур в функции:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```

Следующим шагом будет создание набора функций для арифметических операций над этими структурами. Вот пример одной из них:

```
/* addpoint:  сложение координат двух точек */
struct point addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Здесь структурами являются как аргументы, так и возвращаемое значение. Вместо того чтобы помещать результат во временную переменную, мы инкрементировали компоненты структуры p1, чтобы подчеркнуть тот факт, что параметры-структуры передаются по значениям, как и любые другие параметры.

Рассмотрим еще один пример. Функция ptinrect анализирует, находится ли точка внутри прямоугольника. При этом условно считается, что прямоугольнику принадлежат его левая и нижняя сторона, но не принадлежат правая и верхняя.

```
/* ptinrect:  возвращает 1, если p в r, 0 в противном случае */
int ptinrect(struct point p, struct rect r)
{
    return p.x >= r.pt1.x && p.x < r.pt2.x
        && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

Здесь предполагается, что прямоугольник представлен в стандартной форме, в которой координаты pt1 меньше, чем координаты pt2. Следующая функция возвращает прямоугольник, гарантированно приведенный к канонической форме:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
/* canonrect:  приведение координат прямоугольника
               к каноническому виду */
struct rect canonrect(struct rect r)
{
    struct rect temp;

    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max(r.pt1.y, r.pt2.y);
    return temp
}
```

Если в функцию необходимо передать большую структуру, это лучше сделать, передав указатель на нее, а не копию всех ее данных. Указатели на структуры обладают все-

ми свойствами указателей на обычные переменные. Следующее объявление показывает, что pp является указателем на структуру типа `struct point`:

```
struct point *pp;
```

Если pp указывает на структуру `point`, то `*pp` — это сама структура, а `(*pp).x` и `(*pp).y` — ее поля. Вот как, например, можно было бы обратиться к элементам структуры в программе:

```
struct point origin, *pp;

pp = &origin;
printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

Скобки в обращении к полям необходимы, поскольку приоритет операции обращения выше, чем ссылки по указателю, т.е. выражение `*pp.x` означало бы на самом деле `*(pp.x)`, что синтаксически неправильно, поскольку `x` — не указатель.

Указатели на структуры используются так часто, что для удобства записи ссылок по ним введено дополнительное обозначение. Пусть `p` — указатель на структуру, тогда ссылка на элементы структуры выполняется следующим образом:

`p->`*`элемент-структуры`*

Знак этой операции состоит из знаков "минус" и "больше" без пробела между ними. Поэтому вместо приведенного выше примера можно записать так:

```
printf("origin is (%d,%d)\n", pp->x, pp->y);
```

Как точка, так и знак `->` анализируются при компиляции слева направо. Пусть объявлены следующие структура и указатель:

```
struct rect r, *rp = &r;
```

Тогда следующие четыре выражения эквивалентны:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Операции обращения к структурам (точка и `->`) наряду с круглыми скобками для вызовов функций и квадратными для индексов массивов находятся на самой вершине иерархии приоритетов, а потому применяются к своим аргументам всегда в первую очередь. Пусть имеется объявление:

```
struct {
    int len;
    char *str;
} *p;
```

Тогда выражение `++p->len` инкрементирует `len`, а не `p`. По умолчанию скобки в нем расставляются следующим образом: `++(p->len)`. С помощью скобок можно изменить порядок применения операций: выражение `(++p)->len` инкрементирует `p` до обращения к `len`, а `(p++)->len` инкрементирует его после обращения. (В последнем случае скобки не обязательны.)

Аналогично, выражение `*p->str` дает значение, на которое указывает `str`; `*p->str++` инкрементирует `str` после обращения к тому, на что указывает это поле (как и `*s++`); `(*p->str)++` инкрементирует то, на что указывает `str`; наконец, `*p++->str` инкрементирует `p` после обращения к данным, на которые указывает `str`.

6.3. Массивы структур

Напишем программу, подсчитывающую, сколько раз встречается в исходных данных каждое ключевое слово языка С. Для этого понадобится массив символьных строк, содержащих эти слова, и массив целых чисел для результатов подсчета. Один из возможных подходов — организовать два параллельных массива `keyword` и `keycount`:

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

Но сам факт, что эти массивы параллельны и взаимосвязаны, наводит на мысль о другой организации данных — массиве структур. Каждое ключевое слово и количество этих слов в тексте образуют взаимосвязанную пару:

```
char *word;
int count;
```

Таких пар много, и их можно организовать в массив. Следующее объявление создает структурный тип `key`, определяет массив `keytab` структур этого типа и выделяет для них место в памяти:

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

Каждый элемент этого массива является структурой. Это объявление можно записать еще и таким образом:

```
struct key {
    char *word;
    int count;
};

struct key keytab[NKEYS];
```

Поскольку массив структур `keytab` содержит постоянный набор имен, его удобно сделать внешней переменной и инициализировать раз и навсегда при определении. Инициализация массива структур аналогична изученной ранее: после определения ставится список инициализирующих значений в фигурных скобках.

```
struct key {
    char *word;
    int count;
} keytab[] = {
    "auto", 0,
    "break", 0,
    "case", 0,
    "char", 0,
    "const", 0,
    "continue", 0,
    "default", 0,
    /* ... */
    "unsigned", 0,
    "void", 0,
    "volatile", 0,
    "while", 0,
};
```

Инициализирующие значения перечислены парами соответственно порядку полей в структурах. Более строго было бы также заключить инициализаторы для каждой "строки" или структуры в фигурные скобки:

```
{ "auto", 0 },
{ "break", 0 },
{ "case", 0 },
...
```

Однако в этих внутренних скобках нет никакой необходимости, поскольку инициализирующие значения — простые числовые константы или текстовые строки, причем присутствуют все из них. Как обычно, количество элементов в массиве `keytab` вычисляется автоматически по количеству инициализирующих значений, поэтому скобки `[]` после `keytab` можно оставить пустыми.

Программа подсчета ключевых слов начинается с определения массива `keytab`. Главная программа считывает исходные данные, вызывая функцию `getword`, задача которой — доставлять из потока по одному слову за раз. Каждое слово разыскивается в таблице `keytab` с помощью варианта функции двоичного поиска, разработанной в главе 3. Список ключевых слов должен быть отсортирован внутри таблицы в порядке возрастания.

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

#define MAXWORD 100

int getword(char *, int);
int binsearch(char *, struct key *, int);

/* подсчет ключевых слов языка C */
main()
{
    int n;
    char word[MAXWORD];

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n",
                keytab[n].count, keytab[n].word);
    return 0;
}

/* binsearch:  поиск слова среди tab[0]...tab[n-1] */
int binsearch(char *word, struct key tab[], int n)
{
    int cond;
    int low, high, mid;

    low = 0;
```

```
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if ((cond = strcmp(word, tab[mid].word)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return -1;
}
```

Несколько позже будет продемонстрирована и функция `getword`; пока же достаточно сказать, что при каждом вызове `getword` считывается слово, которое помещается в массив с тем же именем, что и ее первый аргумент.

Величина `NKEYS` обозначает количество ключевых слов в таблице `keytab`. Хотя их можно было бы подсчитать вручную, намного легче и безопаснее сделать это автоматически, особенно если список может подвергнуться изменениям. Один из возможных вариантов — завершить список инициализирующих значений нулевым указателем, а затем пройти таблицу `keytab` в цикле, производя подсчет, пока не встретится конец.

Но это, собственно, намного больше, чем реально требуется, поскольку размер массива полностью определен уже на этапе компиляции. Размер всего массива равен длине одного элемента, умноженной на количество элементов, поэтому количество элементов соответственно составляет:

размер `keytab` / *размер* `struct key`

В языке C есть одноместная операция `sizeof`, выполняемая при компиляции с помощью которой можно вычислить размер любого объекта. Следующие два выражения дают в результате целое число, равное размеру заданного объекта или типа в байтах:

`sizeof` *объект*
`sizeof(`*имя типа*`)`

Строго говоря, операция `sizeof` дает целое значение без знака, тип которого называется `size_t` и определен в заголовочном файле `<stddef.h>`. Объектом операции может быть переменная, массив или структура. Имя типа может быть именем базового типа наподобие `int` или `double` либо производного сложного типа, например структуры или указателя.

В нашем случае количество ключевых слов равно размеру массива, деленному на размер одного элемента. Это равенство используется в директиве `#define` для определения величины `NKEYS`:

```
#define  NKEYS  (sizeof keytab / sizeof(struct key))
```

А вот еще один способ, основанный на делении длины массива на длину его же первого элемента:

```
#define  NKEYS  (sizeof keytab / sizeof keytab[0])
```

Здесь имеется то преимущество, что в записи не нужно ничего менять при изменении типа массива.

Операцию `sizeof` нельзя применять в директиве `#if`, поскольку препроцессор не понимает имена типов. А вот выражение в `#define` не вычисляется препроцессором, поэтому его вполне можно там употреблять.

Теперь вернемся к функции getword. Мы написали более общую функцию, чем требуется для этой программы, но она не слишком сложная. Функция getword считывает следующее "слово" из потока ввода, причем словом считается либо строка букв и цифр, начинающаяся с буквы, либо одиночный символ, не являющийся символом пустого пространства. Значением, возвращаемым из функции, является первый символ слова, EOF в случае конца файла или сам символ, если он не алфавитный.

```
/* getword:  считывает очередное слово или символ из потока ввода */
int getword(char *word, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = word;

    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
    for ( ; --lim > 0; w++)
        if (!isalnum(*w = getch())) {
            ungetch(*w);
            break;
        }
    *w = '\0';
    return word[0];
}
```

Функция getword пользуется функциями getch и ungetch, разработанными в главе 4. Как только совокупность алфавитно-цифровых символов заканчивается, оказывается, что функция getword уже зашла на один символ дальше, чем нужно. Вызов ungetch помещает этот символ назад во входной поток для обработки следующим вызовом. В getword также используются функции, объявленные в стандартном заголовочном файле <ctype.h>: isspace для пропуска пустого пространства, isalpha для распознавания букв и isalnum для распознавания букв и цифр.

Упражнение 6.1. Наша версия getword не умеет корректно обрабатывать знаки подчеркивания, строковые константы, комментарии и управляющие строки препроцессора. Усовершенствуйте эту функцию соответствующим образом.

6.4. Указатели на структуры

Чтобы продемонстрировать некоторые особенности, связанные с указателями на структуры и массивами структур, напишем программу подсчета слов еще раз, но теперь с указателями вместо индексирования массивов.

Внешнее объявление keytab не подвергнется никаким изменениям, а вот функции main и binsearch потребуют переработки:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* подсчет ключевых слов языка C;  */
main()
{
    char word[MAXWORD];
    struct key *p;

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((p=binsearch(word, keytab, NKEYS)) != NULL)
                p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s\n", p->count, p->word);
    return 0;
}

/* binsearch:  поиск слова среди tab[0]...tab[n-1] */
struct key *binsearch(char *word, struct key *tab, int n)
{
    int cond;
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;

    while (low < high) {
        mid = low + (high-low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}
```

Здесь есть несколько примечательных моментов. Во-первых, в объявлении функции `binsearch` необходимо сообщить, что она возвращает указатель на `struct key`, а не на целое число. Этот факт указывается как в прототипе функции, так и в определении `binsearch`. Если `binsearch` находит слово, то она возвращает указатель на него; если нет — то `NULL`.

Во-вторых, обращение к элементам `keytab` теперь выполняется по указателям. Это требует внесения существенных изменений в `binsearch`.

Теперь переменные `low` и `high` инициализируются указателями на начало таблицы и на точку непосредственно перед ее концом.

Средний элемент теперь уже нельзя вычислить так просто, как раньше, поскольку складывать указатели нельзя:

```
mid = (low+high) / 2  /* НЕПРАВИЛЬНО */
```

А вот вычитать указатели можно, так как выражение `high-low` дает количество элементов. В итоге указатель `mid` на средний элемент между `low` и `high` вычисляется следующим образом:

```
mid = low + (high-low) / 2
```

Самая важное изменение, которое необходимо внести в алгоритм, — это сделать так, чтобы в ходе его работы никогда не генерировался некорректный указатель и не предпринималась попытка обращения к элементу за пределами массива. Проблема состоит в том, что как `&tab[-1]`, так и `&tab[n]` находятся за пределами массива `tab`. Первый из указателей категорически запрещен к употреблению, а по второму нельзя ссылаться. Тем не менее стандарт языка гарантирует, что адресная арифметика с участием первого элемента после конца массива (т.е. `&tab[n]`) будет работать корректно.

В функции `main` записано следующее:

```
for (p = keytab; p < keytab + NKEYS;     p++)
```

Если `p` — указатель на структуру, то арифметические операции над `p` выполняются с учетом размера этой структуры, так что выражение `p++` инкрементирует `p` настолько, чтобы получить следующий элемент массива структур, и очередная проверка прекращает работу цикла как раз вовремя.

Не следует предполагать, что длина структуры равна сумме длин ее элементов. Различные объекты по-разному выравниваются по машинным словам, поэтому внутри структуры могут быть неименованные "дыры". Например, если тип `char` имеет длину 1 байт, а `int` — 4 байта, то следующая структура может фактически занимать в памяти восемь байт, а не пять:

```
struct {
    char c;
    int i;
};
```

Надо отметить, что операция `sizeof` всегда дает правильное значение размера объекта.

И еще одно побочное замечание по форматированию текста программы. Если функция возвращает сложный тип данных наподобие указателя на структуру, ее имя часто бывает трудно различить в тексте или найти в текстовом редакторе:

```
struct key *binsearch(char *word, struct key *tab, int n)
```

Поэтому иногда используют другую форму записи:

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

Это исключительно дело вкуса; выберите одну форму и придерживайтесь ее.

6.5. Структуры со ссылками на себя

Предположим, необходимо решить более сложную, чем предыдущая, задачу подсчета частоты, с которой *любое* слово встречается во входном потоке. Поскольку список слов заранее неизвестен, их нельзя предварительно удобно отсортировать и применить двоич-

ный поиск. И все-таки не хочется выполнять линейный поиск (перебор) каждого вновь поступающего слова в таблице уже имеющихся; такая программа выполнялась бы слишком долго. (Строго говоря, время выполнения зависело бы квадратично от количества поступающих слов.) Как же можно организовать слова так, чтобы эффективно справиться с любым их списком?

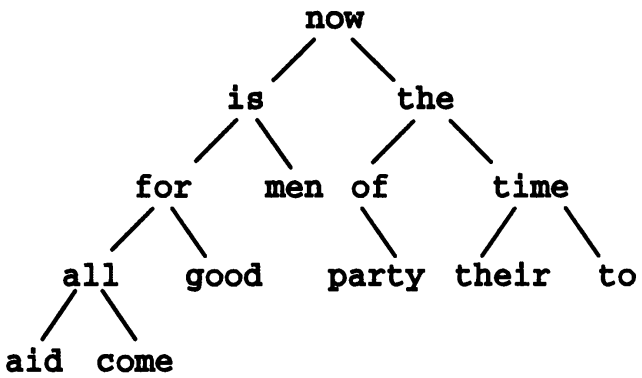
Одно из возможных решений — это в каждый момент поддерживать порядок в списке уже поступивших слов и ставить каждое новое слово в соответствующую позицию. Однако это не следует делать, сдвигая слова в линейном массиве, поскольку такая операция заняла бы много времени. Вместо этого применяется структура данных, именуемая *двоичным деревом*.

Дерево содержит один узел для каждого слова, отличающегося от всех других; каждый такой узел содержит следующую информацию:

- указатель на текст слова;
- частота его употребления в тексте;
- указатель на левый дочерний узел;
- указатель на правый дочерний узел.

Узел не может иметь больше двух дочерних узлов, но может иметь один или не иметь ни одного.

Узлы должны быть организованы так, чтобы в каждом из них левое поддерево содержало только слова, "меньшие" по алфавиту, чем слово в узле, а правое поддерево — только "бо́льшие". Ниже приведено двоичное дерево для предложения "*now is the time for all good men to come to the aid of their party*"[1], построенное в порядке следования слов.



Чтобы выяснить, есть ли вновь поступившее слово в дереве, начинают с корня и сравнивают новое слово с тем, которое помещено в этот корень. Если они совпадают, ответ на вопрос получен. Если новое слово меньше, чем корень дерева, следует продолжить поиск в левом поддереве; в противном случае — в правом поддереве. Если в выбранном направлении нужного дочернего узла нет, значит, нового слова вообще нет в дереве, и пустое место, до которого мы добираемся при поиске, — это как раз место для нового слова. Этот процесс имеет рекурсивный характер, поскольку поиск с началом в любом узле включает в себя поиск по одному из его дочерних узлов. Соответственно, для вставки слов и вывода наиболее естественно будет применить рекурсивные функции.

---

[1] В переводе — "*настало время всем добрым людям прийти на помощь своей стороне*". Это предложение составил американский учитель Чарльз Э. Уэллер (Charles E. Weller) как упражнение по машинописи, поскольку оно содержит почти все буквы латинского алфавита. Фраза часто употребляется для тестирования различных технологий обработки текста. — *Примеч. ред.*

Вернемся к описанию узла. Его удобно представить в виде структуры с четырьмя компонентами:

```
struct tnode {          /* узел дерева: */
    char *word;           /* указатель на текст слова */
    int count;            /* частота употребления */
    struct tnode *left;   /* левый дочерний узел */
    struct tnode *right;  /* правый дочерний узел */
};
```

Рекурсивное объявление узла может показаться сомнительным, но оно вполне правильно. Структуре не разрешается иметь в качестве компонента экземпляр самой себя, но здесь объявляется указатель на `tnode`, а не сама структура:

```
struct tnode *left;
```

Иногда встречается и такая разновидность структур, ссылающихся на себя, как структуры, ссылающиеся друг на друга. Вот как это делается:

```
struct t {
    ...
    struct s *p;    /* p указывает на структуру s */
};
struct s {
    ...
    struct t *q;    /* q указывает на структуру t */
};
```

Исходный код программы оказывается удивительно коротким (не считая двух-трех вспомогательных функций, уже написанных ранее). В основной части программы функция `getword` считывает слова из потока ввода, которые затем добавляются в дерево с помощью функции `addtree`.

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

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

/* программа подсчета частоты слов */
main()
{
    struct tnode *root;
    char word[MAXWORD];

    root = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);
    return 0;
}
```

Функция `addtree` является рекурсивной. Слово поступает из функции `main` на самый верхний уровень (корень) дерева. На каждом этапе это слово сравнивается со сло-

вом, находящимся в узле, и "просачивается" вниз — в правое или левое поддерево — путем рекурсивного вызова addtree. В конце концов либо слово совпадает с одним из уже имеющихся в дереве (в этом случае увеличивается на единицу частота его употребления), либо путь по дереву заканчивается нулевым указателем. В последнем случае необходимо создать новый узел и добавить его к дереву. Когда создается новый узел, addtree возвращает указатель на него, который фиксируется в родительском узле.

```
struct tnode *talloc(void);
char *strdup(char *);

/* addtree:  добавление узла со словом w в узел p или ниже */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL) {     /* поступило новое слово */
        p = talloc();    /* создается новый узел */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;      /* повторяющееся слово */
    else if (cond < 0)   /* меньшее - в левое поддерево */
        p->left = addtree(p->left, w);
    else                 /* большее - в правое поддерево */
        p->right = addtree(p->right, w);
    return p;
}
```

Память для нового узла выделяется функцией talloc, которая возвращает указатель на свободное пространство, достаточное для хранения узла дерева. Новое слово копируется в скрытое место функцией strdup. (Эти функции будут рассмотрены более подробно несколько позже.) Затем инициализируется счетчик слова и создаются два нулевых указателя на дочерние узлы. Эта часть кода выполняется только над листьями дерева при добавлении новых узлов. Мы опустили контроль ошибок в значениях, возвращаемых из strdup и talloc; вообще говоря, это неразумно, и здесь мы были не правы.

Функция treeprint выводит дерево в отсортированном порядке; в каждом узле она выводит левое поддерево (все слова, меньшие заданного), затем само слово, затем правое поддерево (все слова, бо́льшие заданного). Если вам не вполне понятен принцип этого рекурсивного алгоритма, "проиграйте" функцию treeprint на показанном выше дереве.

```
/* treeprint:  вывод дерева p в алфавитном порядке */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

Практическое примечание: если дерево станет "несбалансированным", потому что слова поступают не в случайном порядке, время работы программы может сильно увеличиться. В самом худшем случае — если все слова уже упорядочены — программа вы-

полняет довольно затратную симуляцию линейного поиска. Есть обобщения двоичного дерева, которые не подвержены этому недостатку, но здесь мы не будем уделять им внимания.

Прежде чем закончить рассмотрение этого примера, сделаем отступление по поводу проблемы распределения памяти. Очевидно, было бы желательно иметь в программе одну функцию распределения памяти, пусть даже ей придется обслуживать различные виды объектов. Но если одна и та же функция должна обрабатывать запросы, скажем, на указатели типа `char` и указатели типа `struct tnodes`, сразу же возникают два вопроса. Первый: как удовлетворить требование большинства реальных систем к выравниванию объектов некоторых типов по границам машинных слов (например, целые числа часто выравниваются по четным адресам)? Второй: каким образом объявить функцию так, чтобы она могла возвращать различные типы указателей?

Требования к выравниванию обычно удовлетворить нетрудно за счет некоторых избыточных затрат памяти. Для этого нужно сделать так, чтобы функция распределения памяти всегда возвращала указатель, удовлетворяющий *всем* возможным требованиям к выравниванию. Функция `alloc` из главы 5 не гарантирует никакого конкретного выравнивания, поэтому возьмем для наших целей стандартную библиотечную функцию `malloc`, которая это делает. В главе 8 будет показан один из способов реализации `malloc`.

Вопрос объявления типа для функции наподобие `malloc` не так уж прост в любом языке, который серьезно относится к контролю соответствия типов. В языке C правильно будет объявить, что функция `malloc` возвращает указатель на `void`, а затем явно привести полученный указатель к нужному типу. Функция `malloc` и ее аналоги объявлены в заголовочном файле `<stdlib.h>`. Таким образом, функцию `talloc` можно записать в следующем виде:

```
#include <stdlib.h>

/* talloc:  создание узла дерева типа tnode */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

Функция `strdup` просто копирует свой строковый аргумент в надежное место памяти, выделенное функцией `malloc`:

```
char *strdup(char *s)   /* создание дубликата строки s */
{
    char *p;

    p = (char *) malloc(strlen(s)+1);  /* +1 для '\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

Функция `malloc` возвращает `NULL`, если она не может выделить участок памяти; в таком случае `strdup` передает это значение дальше, возлагая обработку ошибки на вызывающую функцию.

Память, выделенную вызовом `malloc`, можно освободить для последующего повторного использования вызовом функции `free` (см. подробнее главы 7 и 8).

Упражнение 6.2. Напишите программу, которая бы считывала текст программы на C и выводила в алфавитном порядке каждую группу имен переменных, которые совпадают по первым шести символам, но могут отличаться дальше. Не принимайте во внимание слова внутри строк и комментариев. Сделайте так, чтобы число 6 можно было вводить как параметр командной строки.

Упражнение 6.3. Напишите программу анализа перекрестных ссылок, которая бы выводила список всех слов документа, а для каждого слова — список номеров строк, в которых оно встречается. Удалите несущественные слова наподобие артиклей (в английском тексте), союзов, частиц и т.п.

Упражнение 6.4. Напишите программу, выводящую все различные слова из ее входного потока, отсортированные в порядке убывания частоты употребления. Перед каждым словом выведите его частоту.

6.6. Поиск по таблице

В этом разделе будет написана содержательная часть программного пакета для поиска информации в таблицах, с помощью которого мы проиллюстрируем различные аспекты работы со структурами. Этот код достаточно типичен для функций работы с таблицами символов, встречающихся в макропроцессорах и компиляторах. Для примера рассмотрим директиву `#define`:

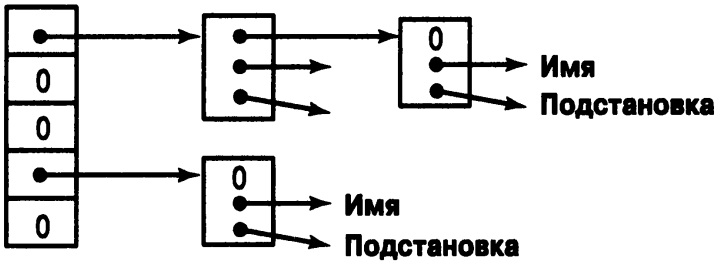
```
#define  IN  1
```

Как только в тексте программы встречается такая директива, имя `IN` и его подстановка `1` записываются в таблицу. Позже, всякий раз, когда имя `IN` встретится в тексте, его следует заменить на `1`, например:

```
state = IN;
```

Всю работу с именами и текстами подстановки выполняют две функции. Первая, `install(s,t)`, записывает имя `s` и текст подстановки `t` в таблицу; `s` и `t` — простые текстовые строки. Функция `lookup(s)` разыскивает строку `s` в таблице и возвращает указатель на место, в котором она ее нашла, или `NULL`, если не нашла.

Алгоритм основан на поиске в хэш-таблице: поступающее имя конвертируется в небольшое неотрицательное число, которое используется как индекс в массиве указателей. Каждый элемент массива указывает на начало связанного списка блоков, описывающих имена, которые соответствуют данному хэш-коду. Если какому-либо хэш-коду не соответствуют никакие имена, возвращается `NULL`.



Блок в этом списке представляет собой структуру, содержащую указатели на имя, на подставляемый вместо него текст и на следующий блок в списке. Если указатель на следующий блок равен `NULL`, это соответствует концу списка.

```
struct nlist { /* запись таблицы: */
    struct nlist *next;   /* следующая запись в цепочке */
    char *name;           /* имя в #define */
    char *defn;           /* подставляемый текст */
};
```

Массив указателей объявляется просто:

```
#define HASHSIZE  101

static struct nlist *hashtab[HASHSIZE]; /* таблица указателей */
```

Хэш-функция, используемая функциями `lookup` и `install`, добавляет очередной символ из строки к замысловатой шифрованной комбинации предыдущих символов, после чего возвращает остаток от деления на размер массива. Это не самая лучшая хэш-функция из возможных, но зато она короткая и быстрая.

```
/* hash:  формирование хэш-кода для строки s */
unsigned hash(char *s)
{
    unsigned hashval;

    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}
```

Использование беззнакового типа гарантирует, что хэш-код никогда не будет отрицательным.

Процедура хэш-кодирования генерирует начальный индекс в массиве `hashtab`. Если строка вообще есть в таблице, она непременно окажется в списке блоков, начинающихся с указанного элемента. Поиск выполняется функцией `lookup`. Если `lookup` находит уже имеющийся в таблице элемент, она возвращает указатель на него; если не находит, возвращает `NULL`.

```
/* lookup:  поиск элемента s в таблице hashtab */
struct nlist *lookup(char *s)
{
    struct nlist *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np;  /* элемент найден */
    return NULL;        /* элемент не найден */
}
```

Цикл `for` в функции `lookup` — это стандартная идиоматическая конструкция для перебора элементов связанного списка:

```
for (ptr = head; prt != NULL; prt = prt->next)
    ...
```

Функция `install` определяет с помощью `lookup`, не существует ли уже в таблице добавляемое в нее имя; если это так, новое определение заменит старое. Если имени еще не существует, создается новая запись. Функция `install` возвращает указатель `NULL`, если по какой-либо причине недостаточно места для новой записи.

```
struct nlist *lookup(char *);
char *strdup(char *);
```

```
/* install:  помещает запись "имя+определение"
        (name, defn) в таблицу hashtab
*/
struct nlist *install(char *name, char *defn)
{
    struct nlist *np;
    unsigned hashval;

    if ((np = lookup(name)) == NULL) {  /* имя не найдено */
        np = (struct nlist *) malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
        hashval = hash(name);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    } else      /* уже есть в таблице */
        free((void *) np->defn);  /* удаление старого определения */
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}
```

Упражнение 6.5. Напишите функцию `undef`, удаляющую имя и его определение из таблицы, обслуживаемой функциями `lookup` и `install`.

Упражнение 6.6. Напишите простую версию обработчика директивы `#define` (без аргументов), корректно работающую с текстами программ на С. В качестве основы используйте функции этого раздела. Полезными могут оказаться также функции `getch` и `ungetch`.

6.7. Определение новых типов

В языке С есть такое специальное средство для определения имен новых типов данных, как оператор `typedef`. Например, чтобы ввести синоним типа `int` под названием `Length`, следует записать:

```
typedef int Length;
```

После этого имя `Length` можно использовать в объявлениях, приведениях типов и т.п. точно так же, как имя самого типа `int`:

```
Length  len, maxlen;
Length  *lengths[];
```

Аналогично, следующее объявление делает имя `String` синонимом указателя на символьные данные (`char *`):

```
typedef char *String;
```

Теперь можно выполнять объявление и приведение типов переменных с его использованием:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Обратите внимание, что имя нового типа, объявляемое в typedef, стоит не сразу после ключевого слова, а на месте имени переменной. Синтаксически ключевое слово typedef можно считать аналогом идентификатора класса памяти — например, extern, static и т.п. Новые типы, определяемые с помощью typedef, начинаются с прописной буквы, чтобы можно было их легко различить.

Возьмем более сложный пример. Можно определить новые типы для дерева и его отдельного узла, описанных ранее в этой главе:

```
typedef struct tnode *Treeptr;

typedef struct tnode {    /* узел дерева: */
    char *word;             /* указатель на текст слова */
    int count;              /* частота употребления */
    struct tnode *left;     /* левый дочерний узел */
    struct tnode *right;    /* правый дочерний узел */
} Treenode;
```

Таким путем создаются два новых ключевых слова для типов данных. Один тип называется Treenode (это структура), а второй — Treeptr (указатель на структуру). Функция talloc в этом случае приобретет вид.

```
Treeptr talloc(void)
{
    return (Treeptr) malloc(sizeof(Treenode));
}
```

Следует подчеркнуть, что объявление typedef, собственно говоря, не создает совершенно нового типа; с его помощью определяется новое имя для некоторого уже существующего типа данных. Нет в нем и никаких смысловых тонкостей: переменные, объявленные таким образом, имеют точно такие же свойства, как и переменные, тип которых выписан явно, без "псевдонима". Фактически оператор typedef очень напоминает директиву #define с тем исключением, что, поскольку он анализируется компилятором, он может допускать такие текстовые подстановки, которые препроцессору не по силам. Например:

```
typedef int (*PFI)(char *, char *);
```

Здесь определяется тип PFI — "указатель на функцию от двух аргументов типа char *, возвращающую int". Этот тип можно использовать, например, в таком контексте (см. программу сортировки в главе 5):

```
PFI strcmp, numcmp;
```

Для применения оператора typedef есть две основные побудительные причины, помимо стилевых и вкусовых предпочтений. Первая — это возможность параметризировать программу с целью улучшения переносимости. Если используемые в программе типы данных могут зависеть от системы и аппаратной конфигурации компьютера, их можно определить через typedef, а затем при переносе заменять только эти определения. Так, часто с помощью typedef определяются условные имена для целочисленных типов, чтобы уже в конкретной системе подставить short, int или long. В качестве примера можно привести такие типы из стандартной библиотеки, как size_t и ptrdiff_t.

Вторая причина, побуждающая применять typedef, — это возможность улучшить удобочитаемость, сделать программу самодокументированной. Название типа Treeptr может оказаться более информативным, чем просто указатель на какую-то сложную структуру.

6.8. Объединения

Объединение — это переменная, которая может содержать объекты различных типов и размеров (но не одновременно); при этом удовлетворение требований к размеру и выравниванию возлагается на компилятор. С помощью объединений можно работать с данными различных типов в пределах одного участка памяти, не привнося в программу элементы низкоуровневого, машинно-зависимого программирования. Объединения аналогичны записям с вариантами в языке Pascal.

Рассмотрим пример, который мог бы встретиться в модуле управления таблицей символов какого-нибудь компилятора. Предположим, что константы в компилируемой программе могут иметь тип `int`, `float` или указатель на `char`. Значение конкретной константы должно храниться в переменной соответствующего типа. В то же время для ведения таблицы символов удобно, чтобы такие значения занимали одинаковый объем памяти и хранились в одном месте независимо от типа. Именно для этого и существуют объединения — переменные, способные хранить данные любого из нескольких типов. Их синтаксис выглядит следующим образом:

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

Переменная `u` будет иметь достаточную длину, чтобы содержать данные самого длинного из трех типов; конкретный размер зависит от системы и реализации. Переменной `u` можно присваивать данные любого типа, а затем использовать их в выражениях (строго по правилам работы с конкретным типом). Извлекать можно данные только того типа, какие были помещены при последнем обращении к переменной. Следить и помнить, какие именно данные были помещены в объединение, — это забота программиста; если поместить значение одного типа, а извлечь его как значение другого, результат будет системно-зависимым и трудно предсказуемым.

Обращение к элементам объединения выполняется так же, как к элементам структур, одним из следующих способов:

имя-объединения.элемент
указатель-на-объединение->элемент

Пусть в переменной `utype` хранится информация о типе данных, находящихся в текущий момент в объединении. Тогда можно представить себе такой код:

```
if (utype == INT)
    printf("%d\n", u.ival);
else if (utype == FLOAT)
    printf("%f\n", u.fval);
else if (utype == STRING)
    printf("%s\n", u.sval);
else
    printf("bad type %d in utype\n", utype);
```

Объединения могут употребляться в структурах и массивах, и наоборот. Способ обращения к члену объединения в структуре (или к члену структуры в объединении) полностью идентичен обращению к элементу вложенной структуры. Пусть, например, определен следующий массив структур:

```
struct {
    char *name;
    int flags;
    int utype;
    union {
        int ival;
        float fval;
        char *sval;
    } u;
} symtab[NSYM];
```

Тогда к элементу `ival` нужно обращаться так:

```
symtab[i].u.ival;
```

А к первому символу строки `sval` обращение выполняется одним из следующих способов:

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

Фактически объединение является структурой, в которой все элементы имеют нулевое смещение от ее начала, сама она имеет достаточную длину, чтобы в нее поместился самый длинный элемент, и при этом выравнивание выполняется правильно для всех типов данных в объединении. Над объединениями разрешено выполнять те же операции, что и над структурами: присваивать или копировать как единое целое, брать адрес и обращаться к отдельным элементам.

Объединение можно инициализировать только данными того типа, который имеет его первый элемент. Таким образом, описанное выше объединение `u` можно инициализировать только целыми числами.

В программе распределения памяти в главе 8 будет продемонстрировано, как можно использовать объединение для принудительного выравнивания переменной по границе участка памяти.

6.9. Битовые поля

Если место в памяти — на вес золота и каждый байт на счету, приходится упаковывать несколько объектов в одно машинное слово. Характерный пример — набор однобитовых сигнальных флажков-переключателей в таких программах, как компиляторы с их таблицами символов. Возможность обращаться к отдельным фрагментам машинного слова также бывает необходима для работы с заданными извне форматами данных, такими как интерфейсы аппаратных устройств.

Представьте себе фрагмент компилятора, управляющий таблицей символов. С каждым идентификатором в программе ассоциируется определенная информация — например, является он ключевым словом или нет, является ли он внешним и/или статическим и т.п. Самый компактный способ закодировать такую информацию — это поместить ее в виде однобитовых переключателей в одну переменную типа `char` или `int`.

Обычный способ сделать это — определить набор "масок", соответствующих битовым позициям в переменной. Например:

```
#define KEYWORD  01
#define EXTERNAL 02
#define STATIC   04
```

Это можно сделать и в другой форме:

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

Здесь числовые константы должны быть степенями двойки. После этого обращение к отдельным битам сводится к сдвигу, применению маски и взятию дополнения. Все эти операции описаны в главе 2

Некоторые операции встречаются так часто, что превратились в устойчивые конструкции (идиомы). Вот как включаются (устанавливаются равными единице) биты признаков EXTERNAL и STATIC в переменной flags:

```
flags |= EXTERNAL | STATIC;
```

А отключаются они (устанавливаются равными нулю) таким образом:

```
flags &= ~(EXTERNAL | STATIC);
```

Следующее выражение истинно, если оба бита отключены:

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

Эти идиомы осваиваются вполне легко; тем не менее им есть и альтернатива. В языке C имеется возможность определения полей внутри машинного слова непосредственно, а не с помощью поразрядных операций. Внутри системно-зависимой единицы памяти, которую мы будем называть "словом", можно задать *битовое поле* (*bit-field*) — совокупность идущих подряд битов. Синтаксис определения и использования полей основан на структурах. Например, приведенный набор директив #define для таблицы символов можно было бы заменить одним определением трех полей:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern  : 1;
    unsigned int is_static  : 1;
} flags;
```

Здесь определяется переменная flags, содержащая три однобитовых поля. Число после двоеточия задает ширину поля в битах. Поля объявлены как unsigned int, чтобы гарантированно быть величинами без знака.

Обращение к отдельным полям выполняется так же, как в структурах: flags.is_keyword, flags.is_extern и т.п. По своим свойствам поля являются небольшими целыми числами и могут употребляться в выражениях в этом качестве. Поэтому предыдущие примеры можно переписать в более естественном виде. Вот пример включения битов (установки их в единицу):

```
flags.is_extern = flags.is_static = 1;
```

А это отключение битов (установка в нуль):

```
flags.is_extern = flags.is_static = 0;
```

Так выполняется анализ значений:

```
if (flags.is_extern == 0 && flags.is_static == 0)
    ...
```

Практически все, что связано с битовыми полями, является системно-зависимым. Например, только в конкретной реализации определяется, могут ли поля перекрывать границы слов. Поля не обязаны иметь имена; безымянные поля (двоеточия с размером по-

сле них) часто используются для пропуска и резервирования отдельных битов. Для принудительного выравнивания по границе следующего слова можно использовать специальное значение длины поля, равное 0.

В некоторых системах поля выстраиваются слева направо, а в других — справа налево. Это означает, что хотя с помощью полей удобно работать с внутренними структурами данных, в случае обработки внешних структур следует тщательно подходить к вопросу о том, с какого конца начинать. Программы, зависимые от такого порядка, не переносимы между системами. Поля можно объявлять только с типом `int`; с целью переносимости лучше явно указывать модификатор `signed` или `unsigned`. Совокупность полей — не массив, и у них нет адресов, поэтому операция `&` к ним неприменима.

Глава 7

Ввод-вывод

Средства ввода-вывода не являются частью собственно языка С, поэтому до сих пор наше изложение касалось их только вскользь. А ведь программы могут взаимодействовать со своим системным окружением гораздо более сложными способами, чем было показано ранее. В этой главе рассматривается стандартная библиотека — набор функций для ввода и вывода данных, обработки строк, управления памятью, математических вычислений и других операций, необходимых в программах на С. Из всей стандартной библиотеки мы сосредоточимся в основном на функциях ввода-вывода.

В стандарте ANSI все библиотечные функции определены совершенно точно, так что они существуют в эквивалентной форме в любой системе, поддерживающей язык С. Программы, в которых взаимодействие с системой осуществляется исключительно с помощью средств из стандартной библиотеки, можно переносить из одной среды в другую безо всяких изменений.

Объявления библиотечных функций распределены по добрым полутора десяткам заголовочных файлов. Некоторые из них уже упоминались — это `<stdio.h>`, `<string.h>`, `<ctype.h>`. Вся библиотека здесь рассматриваться не будет, поскольку нам интереснее писать свои программы на С, чем изучать ее. Подробное описание библиотеки приведено в приложении Б.

7.1. Стандартные средства ввода-вывода

Как было сказано в главе 1, библиотека реализует простую модель текстового ввода и вывода. Текстовый поток состоит из последовательности строк; каждая строка оканчивается специальным символом конца строки. Если система работает не совсем таким образом, библиотека делает все необходимое, чтобы программе казалось именно так. Например, средства стандартной библиотеки преобразуют символы возврата каретки и перевода строки в единый символ конца строки при вводе и выполняют обратное преобразование при выводе.

Простейший механизм ввода состоит в том, чтобы считывать по одному символу за раз из *стандартного потока ввода* (*standard input*), обычно представляемого клавиатурой, с помощью функции `getchar`:

```
int getchar(void)
```

Функция `getchar` при каждом ее вызове возвращает следующий символ из потока или `EOF` в случае конца файла. Символическая константа `EOF` определена в `<stdio.h>`. Обычно ее значение равно `-1`, но в выражениях следует писать `EOF`, чтобы не зависеть от конкретного значения.

Во многих средах можно заменить стандартный поток ввода с клавиатуры на файл, используя символ < для перенаправления ввода. Пусть, например, программа prog пользуется функцией getchar для ввода. Тогда следующая команда заставит prog считывать символы из файла infile, а не с клавиатуры:

```
prog <infile
```

Переключение ввода произойдет таким образом, что и сама программа prog никакой разницы не заметит. Например, строка "<infile" не будет включена в число аргументов командной строки программы в массиве argv. Переключение ввода незаметно также и в том случае, если входные данные поступают из другой программы через *конвейер* (*pipe*). В некоторых системах можно задать такую команду:

```
otherprog | prog
```

В результате будут запущены на выполнение две программы, otherprog и prog, причем стандартный поток вывода программы otherprog будет перенаправлен в стандартный поток ввода prog.

Ниже приведена простейшая функция вывода.

```
int putchar(int)
```

Оператор putchar(c) помещает символ c в *стандартный поток вывода* (*standard output*), который по умолчанию соответствует экрану монитора. Функция putchar возвращает выведенный символ или EOF в случае ошибки. И в этом случае поток можно перенаправить в файл с помощью директивы *>имя_файла*. Если программа prog пользуется для вывода функцией putchar, то следующая команда перенаправит ее вывод в файл outfile:

```
prog >outfile
```

Если в системе поддерживаются конвейеры, можно перенаправить поток вывода программы prog в поток ввода программы anotherprog:

```
prog | anotherprog
```

Данные, выводимые функцией printf, также поступают в стандартный поток вывода. Вызовы putchar и printf можно чередовать как угодно — данные будут появляться в потоке в том порядке, в каком делаются вызовы функций.

Каждый файл исходного кода, в котором выполняется обращение к библиотечным функциям ввода-вывода, должен содержать следующую строку до первого вызова таких функций:

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

Если имя файла заключено в угловые скобки (<>), заголовочный файл разыскивается в определенном стандартном месте (например, в системах Unix это каталог /usr/include).

Многие программы считывают данные из одного потока ввода и записывают результаты также в один поток; для таких программ вполне достаточно ввода-вывода с помощью getchar, putchar и printf — по крайней мере, на первых порах. В частности, это справедливо, если используется перенаправление данных с выхода одной программы на вход другой. Для примера рассмотрим программу lower, которая переводит поступающие данные в нижний регистр:

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

```
main()  /* lower:  перевод выходных данных в нижний регистр */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

Функция `tolower` определена в файле `<ctype.h>`; она переводит букву из верхнего регистра в нижний, а любой другой символ оставляет в исходном виде. Как уже упоминалось, "функции" наподобие `getchar` и `putchar` из `<stdio.h>` и `tolower` из `<ctype.h>` часто являются макросами, благодаря чему удается избежать излишних затрат на вызов функций для каждого символа. В разделе 8.5 будет показано, как это делается. Независимо от того, как функции из файла `<ctype.h>` реализованы в конкретной системе, программы с их использованием благодаря им защищены от особенностей того или иного символьного набора.

Упражнение 7.1. Напишите программу для перевода входных данных из верхнего регистра в нижний или наоборот в зависимости от того, буквами какого регистра набрано имя программы при ее запуске (это имя помещается в `argv[0]`).

7.2. Форматированный вывод и функция `printf`

Функция вывода `printf` преобразует данные из внутренних форматов в выводимые символы. В предыдущих главах она широко использовалась без формального определения. Здесь рассматриваются самые типичные ее применения, хотя и не дается полное определение; за полными сведениями обращайтесь к приложению Б.

`int printf(char *format, `*`arг`*`1, `*`arг`*`2, ...)`

Функция `printf` преобразует, форматирует и выводит свои аргументы в стандартный поток вывода согласно строке формата — `format`. Она возвращает количество выведенных символов.

Строка формата содержит два типа объектов: обычные символы, которые копируются в поток вывода, и спецификации формата, каждая из которых вызывает преобразование и вывод очередного аргумента функции `printf`. Каждая спецификация формата начинается с символа `%` и заканчивается символом типа преобразования. Между `%` и символом типа могут стоять по порядку следующие элементы.

- Знак "минус", задающий выравнивание выводимого аргумента по левому краю отведенного поля вывода.
- Число, задающее минимальную ширину поля. Преобразованный аргумент выводится в поле, ширина которого не меньше, чем заданная. Если необходимо, поле дополняется пробелами слева (или справа, если это задано) до указанной длины.
- Точка, отделяющая ширину поля от точности представления.

- Число (точность представления), задающее максимальное количество символов при выводе строки, или количество цифр в вещественном числе после десятичной точки, или минимальное количество цифр для целого числа.
- Буква h, если целое число следует вывести как короткое (`short`), или буква `l`, если как длинное (`long`).

Символы преобразования приведены в табл. 7.1. Если после знака `%` стоит не спецификация формата, а что-то другое, результат будет непредсказуемым.

Таблица 7.1. Основные спецификации формата функции `printf`

Символ	Тип аргумента и способ вывода
`d, i`	`int`; десятичное целое число
`o`	`int`; восьмеричное целое число без знака (без нуля впереди)
`x, X`	`int`; шестнадцатеричное целое число без знака (без `0x` или `0X` в начале) с применением соответственно цифр `abcdef` и `ABCDEF` для 10, ..., 15
`u`	`int`; десятичное целое число без знака
`c`	`int`; отдельный символ
`s`	`char *`; выводятся символы из строки, пока не встретится `'\0'`, или в количестве, заданном параметром точности
`f`	`double`; `[-]`*m*`.`*dddddd*, где количество цифр на месте букв *d* задается параметром точности (по умолчанию 6)
`e, E`	`double`; `[-]`*m*`.`*ddddde±xx* или `[-]`*m*`.`*ddddd*`E±`*xx*, где количество цифр на месте букв *d* задается параметром точности (по умолчанию 6)
`g, G`	`double`; эквивалентно `%e` или `%E`, если показатель степени меньше `-4`, или больше, или равен заданной точности; в противном случае эквивалентно `%f`. Нули и десятичная точка в конце числа не выводятся
`p`	`void *`; указатель (точное представление зависит от конкретной системы и реализации языка)
`%`	преобразование аргументов не выполняется, выводится символ `%`

Ширину или точность можно указать в виде символа `*`. В этом случае нужное значение будет вычисляться по следующему аргументу (который должен иметь тип `int`). Например, вывод не более `max` символов из строки `s` выполняется так:

```
printf("%.*s", max, s);
```

Почти все спецификации формата уже иллюстрировались примерами в предыдущих главах. Одно из исключений состоит в применении параметра точности к строкам. Следующая таблица показывает, как работают разные спецификации при выводе строки `"hello, world"` (12 символов). Каждое поле вывода окружено двоеточиями, чтобы было видно его длину:

```
:%s:          :hello, world:
:%10s:        :hello, world:
:%.10s:       :hello, wor:
:%-10s:       :hello, world:
:%.15s:       :hello, world:
:%-15s:       :hello, world   :
:%15.10s:     :     hello, wor:
:%-15.10s:    :hello, wor     :
```

Предупреждение: функция printf по своему первому аргументу вычисляет, сколько аргументов будет дальше и какого они типа. Если количество или типы аргументов не соответствуют строке формата, возникнет путаница, и будет выдан неверный результат. Следует также иметь в виду разницу между этими двумя вызовами:

```
printf(s);          /* ОШИБКА, ЕСЛИ s СОДЕРЖИТ % */
printf("%s", s);    /* БЕЗОПАСНО */
```

Функция sprintf выполняет те же преобразования, что и printf, но помещает результат вывода в символьную строку:

```
int sprintf(char *string, char *format, apr1, apr2, ...)
```

Функция sprintf форматирует аргументы *apr*1, *apr*2 и т.д. согласно строке формата format, как и раньше, но результат помещается не в поток вывода, а в строку string. Предполагается, что string имеет достаточную для этого длину.

Упражнение 7.2. Напишите программу, выводящую данные, которые поступают к ней на вход, в удобном представлении. Как минимум, программа должна выводить непечатаемые символы в виде восьмеричных или шестнадцатеричных кодов (как привычнее пользователю) и разбивать слишком длинные строки текста.

7.3. Списки аргументов переменной длины

В этом разделе приводится реализация сильно урезанной, "скелетной" версии printf, которая демонстрирует, как пишутся переносимые функции с обработкой списков аргументов переменной длины. Поскольку здесь нас в основном интересует работа со списком аргументов, функция minprintf будет сама анализировать строку формата и аргументы, а для преобразования данных к выводимой форме вызывать функцию printf.

Вот как на самом деле объявляется функция printf:

```
int printf(char *fmt, ...)
```

Здесь конструкция ... означает, что количество и типы аргументов могут меняться. Такая конструкция может стоять только в конце списка аргументов. Наша функция minprintf объявляется следующим образом:

```
void minprintf(char *fmt, ...)
```

В отличие от printf, она не возвращает количество выведенных символов.

Сложность заключается в том, как же перебрать список аргументов в функции minprintf, когда у него даже нет имени. В стандартном заголовочном файле <stdarg.h> на этот случай имеется набор макроопределений, дающих способ перебора списка. Реализация этого заголовочного файла системно-зависима, однако интерфейс всегда единообразный.

Для объявления переменной, ссылающейся по очереди на каждый аргумент, имеется тип va_list. В функции minprintf эта переменная имеет имя ap (сокращение от *argument pointer* — указатель на аргумент). Макрос va_start инициализирует ap так, чтобы переменная указывала на первый безымянный аргумент. Этот макрос нужно вы-

звать один раз перед тем, как обращаться к переменной ap. Функция должна иметь как минимум один аргумент с именем; последний именованный аргумент используется макросом va_start для инициализации своей работы.

Каждый вызов va_arg возвращает один аргумент и передвигает указатель ap на следующий. Чтобы определить, какого типа аргумент нужно возвращать и на сколько передвигать указатель, va_arg использует заданное ему имя типа. Наконец, макрос va_end выполняет необходимые завершающие операции. Его необходимо вызвать до возвращения из функции.

Все перечисленное служит основой для нашей упрощенной версии printf:

```
#include <stdarg.h>

/* minprintf:  ограниченная версия printf
   со списком аргументов переменной длины */
void minprintf(char *fmt, ...)
{
    va_list ap;   /* указатель на безымянные аргументы */
    char *p, *sval;
    int ival;
    double dval;

    va_start(ap, fmt); /* установить ap на 1-й аргумент без имени */
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 'd':
            ival = va_arg(ap, int);
            printf("%d", ival);
            break;
        case 'f':
            dval = va_arg(ap, double);
            printf("%f", dval);
            break;
        case 's':
            for (sval = va_arg(ap, char *); *sval; sval++)
            putchar(*sval);
            break;
        default:
            putchar(*sval);
            break;
        }
    }
    va_end(ap);   /* завершающие операции */
}
```

Упражнение 7.3. Доработайте minprintf так, чтобы она реализовала больше возможностей printf.

7.4. Форматированный ввод и функция scanf

Функция scanf — это аналог printf для целей ввода. Она реализует многие аналогичные преобразования формата в противоположную сторону. Вот ее объявление:

```
int scanf(char *format, ...)
```

Функция scanf считывает символы из стандартного потока ввода, интерпретирует их в соответствии со строкой-спецификацией format и помещает результат в остальные аргументы. Строка формата будет рассмотрена ниже. Остальные аргументы, *каждый из которых должен быть указателем*, определяют, куда нужно поместить поступившие входные данные. Как и в случае printf, в этом разделе дается краткий обзор основных возможностей, а не исчерпывающее описание.

Функция scanf прекращает работу, как только закончится строка формата или как только входные данные придут в противоречие с управляющей ими спецификацией. Она возвращает количество успешно прочитанных, преобразованных и помещенных в указанные места элементов входных данных. Это значение можно использовать для учета и контроля введенных данных. В конце файла возвращается EOF; обратите внимание, что это не 0, который означает, что очередной введенный символ не соответствует текущей спецификации в строке формата. При следующем вызове scanf считывание потока продолжается со следующего символа после последнего введенного и преобразованного.

Существует также функция sscanf, которая считывает из строки, а не из стандартного потока ввода:

```
int sscanf(char *string, char *format, apr1, apr2, ...)
```

Эта функция считывает из строки string согласно строке формата format и помещает полученные результаты в *apr*1, *apr*2. Эти аргументы должны быть указателями.

Строка формата обычно содержит спецификации формата, используемые для управления преобразованием входных данных. В строке формата могут содержаться следующие элементы:

- пробелы или табуляции, которые игнорируются при вводе;
- обыкновенные символы (не %), которые должны соответствовать ожидаемым очередным непустым символам в потоке ввода;
- спецификации формата, состоящие из символа %; необязательного символа запрета присваивания *; необязательного числа — максимальной ширины поля; необязательного символа h, l или L, обозначающего длину представления числа, и собственно символа типа/формата.

Спецификация формата регулирует преобразование очередного элемента данных (*поля ввода*) в потоке к заданному типу и форме. Как правило, результат помещается в переменную, на которую указывает соответствующий аргумент. Если присваивание запрещено символом *, то элемент данных пропускается, и никакого присваивания не происходит. Поле ввода по определению представляет собой строку непустых символов, заканчивающуюся либо символом пустого пространства, либо на заданной ширине (если таковая указана). Отсюда следует, что функция scanf при вводе данных пропускает концы строк, поскольку они относятся к символам пустого пространства. (Символы пус-

того пространства — это пробел, табуляция, конец строки, возврат каретки, вертикальная табуляция, перевод страницы.)

Спецификация формата задает также и смысловую интерпретацию поля ввода. Соответствующий аргумент должен быть указателем, поскольку этого требует семантика вызова функций в C по значению. Символы-спецификации формата приведены в табл. 7.2.

Таблица 7.2. Основные спецификации формата функции `scanf`

Символ	Входные данные и тип аргумента
`d`	Десятичное целое число; `int *`
`i`	Целое число; `int *`. Число может записываться в восьмеричной (0 в начале) или шестнадцатеричной (`0x` или `0X` в начале) системе
`o`	Восьмеричное целое число (с нулем или без нуля впереди); `int *`
`u`	Десятичное целое число без знака; `unsigned int *`
`x`	Шестнадцатеричное целое число (с или без `0x`/`0X` в начале); `int *`
`c`	Символ; `char *`. Очередной символ из потока ввода (по умолчанию один) помещается в указанное место. Обычный пропуск пустого пространства отменяется; чтобы считать следующий непустой символ, используйте спецификацию `%1c`
`s`	Строка символов (без кавычек); `char *`, указатель на массив символов, достаточно большой для помещения строки с завершающим `'\0'`, который добавляется автоматически
`e, f, g`	Вещественное число, с необязательным знаком, необязательной десятичной точкой и необязательной экспоненциальной частью; `float *`
`%`	Символ процента; присваивание не выполняется

Перед спецификациями формата `d`, `i`, `o`, `u` и `x` может стоять символ `h`, означающий, что в списке аргументов находится указатель на `short`, а не на `int`. Вместо `h` может также стоять буква `l`, которая означает, что соответствующий аргумент — указатель на данные типа `long`. Аналогично, перед спецификациями `e`, `f` и `g` может стоять буква `l`, означающая, что в списке аргументов на соответствующем месте находится указатель на `double`, а не на `float`.

В качестве первого примера перепишем примитивный калькулятор из главы 4, выполняя ввод данных с помощью `scanf`:

```
#include <stdio.h>

main()  /* примитивный калькулятор */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return v;
}
```

Предположим, необходимо вводить строки данных, в которых содержатся даты в следующем формате:

```
25 Dec 1988
```

Оператор scanf для этого случая будет выглядеть так:

```
int day, year;
char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

Знак & с именем monthname употреблять не надо, поскольку имя массива само по себе является указателем.

В строке формата scanf могут фигурировать обычные, не управляющие символы; они должны в точности соответствовать символам во входном потоке. Поэтому считывать дату в формате мм/дд/гг из входного потока можно таким оператором:

```
int day, month, year;

scanf("%d/%d/%d", &month, &day, &year);
```

Функция scanf игнорирует пробелы и табуляции в строке формата. Более того, при поиске данных во входном потоке она пропускает вообще все символы пустого пространства (пробелы, табуляции, концы строк и т.д.). Считывание потока, формат которого не фиксирован, лучше выполнять по одной строке, которую затем разбирать функцией sscanf. Например, пусть необходимо считывать строки, которые могут содержать дату в любом из приведенных выше форматов. Тогда можно записать следующее:

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("valid: %s\n", line);   /* 25 Dec 1988 */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("valid: %s\n", line);   /* мм/дд/гг */
    else
        printf("invalid: %s\n", line); /* неправильная строка */
}
```

Вызовы функции scanf можно чередовать с вызовами других функций ввода. При следующем вызове любой из функций ввода считывание потока продолжится с первого символа, еще не считанного scanf.

Повторяем предупреждение в последний раз: аргументы функций scanf и sscanf *должны быть указателями*. Самая распространенная ошибка в этой связи — приведенная ниже конструкция.

```
scanf("%d", n);
```

Правильно следует записывать так:

```
scanf("%d", &n);
```

Обычно эта ошибка не выявляется при компиляции.

Упражнение 7.4. Напишите мини-реализацию функции scanf по аналогии с minprintf из предыдущего раздела.

Упражнение 7.5. Перепишите постфиксный калькулятор из главы 4 так, чтобы для ввода и преобразования чисел в нужную форму использовалась функция scanf и/или sscanf.

7.5. Доступ к файлам

До сих пор во всех примерах использовался ввод из стандартного потока ввода и вывод в стандартный поток вывода; эти потоки определены для программы по умолчанию операционной системой.

Следующий шаг — написать программу, которая бы обращалась к файлу данных, *не подключенному* к программе заранее. Одна из программ, демонстрирующих потребность в такого рода операциях, называется `cat`. Она соединяет набор именованных файлов (выполняет *конкатенацию*) в один массив данных и выводит его в стандартный поток вывода. Программа `cat` используется для вывода файлов на экран, а также применяется для подготовки входных данных для программ, которые сами не могут обращаться к файлам по именам. Например, следующая команда выводит содержимое файлов `x.c` и `y.c` (и больше ничего) в стандартный поток вывода:

```
cat x.c y.c
```

Вопрос состоит в том, как же организовать чтение данных из файлов с именами, т.е. как подключить имена внешних файлов, задаваемые пользователем, к операторам программы, считывающим данные.

Правила для этого просты. Перед тем как читать из файла или записывать в файл, его необходимо *открыть* с помощью библиотечной функции `fopen`. Эта функция берет внешнее имя наподобие `x.c` или `y.c`, выполняет некоторые подготовительные операции, обменивается запросами с операционной системой (подробности сейчас не важны) и в итоге возвращает указатель, который в дальнейшем используется для чтения или записи.

Этот *файловый указатель* указывает на структуру, содержащую информацию о файле: местонахождение буфера, текущую символьную позицию в буфере, характер операций (чтение или запись), наличие ошибок и состояния конца файла. Пользователям не обязательно знать все детали, поскольку среди различных определений в файле `<stdio.h>` имеется объявление структуры под именем `FILE`. Единственное объявление, которое требуется для файлового указателя, — это создание его экземпляра таким образом, как показано ниже.

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

Здесь говорится, что `fp` указываст на структуру `FILE`, а функция `fopen` возвращает указатель на `FILE`. Обратите внимание, что `FILE` — это название типа, а не метка структуры, поскольку оно определено с помощью оператора `typedef`. (Подробности возможной реализации `fopen` в системе Unix приведены в разделе 8.5.)

Вызов функции `fopen` в программе выполняется следующим образом:

```
fp = fopen(name, mode);
```

Первый аргумент `fopen` — это символьная строка, содержащая имя файла. Второй аргумент — *режим открытия*; это также символьная строка, указывающая способ использования файла. Допускаются режимы чтения `"r"` (от `"read"`), записи `"w"` (`"write"`) и добавления данных в конец файла `"a"` (`"append"`). В некоторых системах делается различие между текстовыми и двоичными файлами; для работы с двоичными файлами следует добавить в строку режима букву `"b"` (от `"binary"`).

Если файл, открываемый для записи или добавления в конец, не существует, предпринимается попытка его создания. Открытие существующего файла для записи стирает

все его содержимое, а открытие для добавления — сохраняет. Попытка читать из несуществующего файла является ошибкой; могут быть и другие причины для ошибки — например, отсутствие допуска на чтение из данного файла. Если произошла ошибка, функция fopen возвращает NULL. (Если необходимо, произошедшую ошибку можно определить точнее; см. описание функций обработки ошибок в конце раздела 1 приложения Б.)

Следующее, что необходимо для работы, — это средства чтения или записи файла после его открытия. Тут на выбор имеется несколько вариантов, из которых самые простые — это функции getc и putc. Функция getc возвращает очередной символ из файла; чтобы знать, из какого именно, она должна получить указатель на файл:

```
int getc(FILE *fp)
```

Итак, функция getc возвращает очередной символ из потока, на который указывает fp; при ошибке или в случае конца файла возвращается EOF.

Для вывода используется функция putc:

```
int putc(int c, FILE *fp)
```

Функция putc записывает символ c в файл fp и возвращает записанный символ или EOF в случае ошибки. Как и getchar/putchar, функции getc и putc могут фактически быть макросами.

Когда программа на C запускается на выполнение, операционная система автоматически открывает три файла и создает файловые указатели на них. Это стандартный поток ввода, стандартный поток вывода и стандартный поток ошибок. Соответствующие указатели называются stdin, stdout и stderr; они объявлены в файле <stdio.h>. Обычно stdin ассоциирован с клавиатурой, а stdout и stderr — с экраном монитора, но stdin и stdout можно ассоциировать с файлами или конвейерами (перенаправить), как описано в разделе 7.1.

Функции getchar и putchar можно определить через getc, putc, stdin и stdout следующим образом:

```
#define getchar()    getc(stdin)
#define putchar(c)   putc((c), stdout)
```

Для форматированного файлового ввода-вывода можно пользоваться функциями fscanf и fprintf. Они аналогичны scanf и printf, только их первым аргументом служит файловый указатель, а строка формата стоит на втором месте:

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

Справившись с предварительной подготовкой, мы теперь готовы приступить к программе cat, выполняющей конкатенацию файлов. Общий принцип ее устройства применяется во многих программах: если есть аргументы командной строки, они интерпретируются как имена файлов данных и обрабатываются по порядку; если нет, обрабатывается стандартный поток ввода.

```
#include <stdio.h>

/* cat:  сцепление файлов по порядку, версия 1 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
```

```
    if (argc == 1)  /* нет аргументов; стандартный поток */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                printf("cat: can't open %s\n", *argv);
                return 1;
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    return 0;
}

/* filecopy:  копирование файла ifp в файл ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}
```

Файловые указатели `stdin` и `stdout` являются объектами типа `FILE *`. Это константы, а *не переменные*, поэтому им нельзя присваивать какие-либо значения.

Обратную к `fopen` роль играет функция `fclose`:

```
int fclose(FILE *fp)
```

Она разрывает связь между внешним именем и файловым указателем, установленную функцией `fopen`, тем самым освобождая указатель для другого файла. Поскольку в большинстве операционных систем имеется ограничение на количество одновременно открытых программой файлов, полезно закрывать файлы и освобождать их указатели, как только файлы становятся ненужными. Именно это делается в программе `cat`. Есть и еще одна причина применить `fclose` к открытому файлу — в результате очищается и сбрасывается в файл буфер, в котором функция `putc` накапливает выходные данные. Функция `fclose` вызывается автоматически для каждого открытого файла во время нормального завершения программы. (Можно даже закрыть `stdin` и `stdout`, если они не нужны. Их можно затем переназначить библиотечной функцией `freopen`.)

7.6. Обработка ошибок. Поток `stderr` и функция `exit`

Обработка ошибок в программе `cat` далека от идеальной. Беда в том, что если обращение к одному из файлов по какой-то причине не удалось, соответствующая диагностика выводится в конце сцепленного потока данных. Это нормально, если поток выводится на экран, но неудобно в том случае, если поток направляется в файл или в другую программу через конвейерный механизм.

Чтобы справиться с этой ситуацией, программе назначается второй поток вывода под именем `stderr` (*standard error — стандартный поток ошибок*) точно таким же образом, как `stdin` и `stdout`. Данные, выведенные в `stderr`, обычно появляются на экране, даже если стандартный поток вывода перенаправлен в другое место.

Перепишем программу `cat` так, чтобы сообщения об ошибках в ней выводились в стандартный поток ошибок.

```
#include <stdio.h>

/* cat:  сцепление файлов по порядку, версия 2 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog = argv[0];   /* имя программы для сообщений */

    if (argc == 1)  /* нет аргументов; стандартный поток */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr, "%s: can't open %s\n",
                    prog, *argv);
                exit(1);
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    if (ferror(stdout) {
        fprintf(stderr, "%s: error writing stdout\n", prog);
        exit(2);
    }
    exit(0);
}
```

Программа сигнализирует об ошибках двумя способами. Во-первых, диагностика ошибок посылается функцией `fprintf` в поток `stderr` и гарантированно появляется на экране, а не исчезает где-то в недрах файла или конвейера. В сообщения включается имя программы из `argv[0]`, чтобы при использовании данной программы параллельно с другими можно было легко определить источник ошибки.

Во-вторых, в программе используется стандартная библиотечная функция `exit`, прекращающая работу программы. Аргумент `exit` становится доступным тому процессу, который запустил программу на выполнение, так что успешность или ошибочность ее завершения может проанализировать другая программа — та, которая вызвала ее в качестве подчиненного процесса. Принято, что возвращаемое значение 0 сигнализирует о нормальном завершении; различные ненулевые значения обычно обозначают аварийные ситуации. Функция `exit` вызывает `fclose` для закрытия каждого открытого файла, очищая тем самым буферы вывода.

В программе `main` конструкция `return` *выражение* эквивалентна выражению `exit(`*выражение*`)`. Функция `exit` имеет то преимущество, что ее можно вызвать из других функций и вызовы ее можно найти в тексте программой поиска по образцу наподобие приведенной в главе 5.

Функция `ferror` возвращает ненулевое число, если произошла ошибка в потоке `fp`:

```
int ferror(FILE *fp)
```

Хотя ошибки при выводе очень редки, они все же случаются (например, при заполнении всего дискового пространства), поэтому программа профессионального коммерческого качества должна проверять и такие случаи.

Функция `feof(FILE *)` похожа на `ferror` — она возвращает ненулевое число, если в указанном файле достигнут его конец.

```
int feof(FILE *fp)
```

Мы пока не обращаем внимание на то, с какими кодами завершения заканчиваются наши небольшие иллюстративные программы, но в любой серьезной программе в таких случаях необходимо возвращать разумные и полезные значения.

7.7. Ввод-вывод строк

В стандартной библиотеке имеется функция ввода `fgets`, аналогичная функции `getline`, разработанной и использованной в предыдущих главах:

```
char *fgets(char *line, int maxline, FILE *fp)
```

Функция `fgets` считывает очередную строку (вместе с символом ее конца) из файла `fp` в массив символов `line`; в общей сложности читается не более `maxline-1` символов. Получившаяся строка завершается нулевым символом `'\0'`. Обычно `fgets` возвращает `line`; в конце файла или в случае ошибки она возвращает `NULL`. (Наша функция `getline` возвращает более полезную информацию — длину строки; если она равна нулю, был достигнут конец файла.)

Что касается вывода, строка записывается в файл функцией `fputs` (строка не обязана содержать символ конца):

```
int fputs(char *line, FILE *fp)
```

Функция `fputs` возвращает `EOF` в случае ошибки и 0 в противном случае.

Библиотечные функции `gets` и `puts` аналогичны `fgets` и `fputs`, но работают с потоками `stdin` и `stdout`. Функция `gets` удаляет завершающий символ `'\n'`, а `puts` добавляет его, что часто вызывает путаницу.

Чтобы показать, что в функциях наподобие `fgets` и `fputs` нет ничего особенного, приведем их полные тексты из стандартной библиотеки в нашей системе.

```
/* fgets:  считывание не более n символов из iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}
```

```
/* fputs:  вывод строки s в файл iop */
int fputs(char *s, FILE *iop)
{
    int c;

    while (c = *s++)
        putc(c, iop);
    return ferror(iop) ? EOF : 0;
}
```

В стандарте указано, что `ferror` возвращает ненулевое число в случае ошибки; `fputs` возвращает `EOF` в случае ошибки и неотрицательное значение в противном случае.

С помощью `fgets` очень легко переписать нашу функцию `getline`:

```
/* getline:  считывает строку и возвращает ее длину */
int getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
        return 0;
    else
        return strlen(line);
}
```

Упражнение 7.6. Напишите программу для сравнения двух файлов с выводом первой строки, в которой они отличаются.

Упражнение 7.7. Доработайте программу поиска по образцу из главы 5 так, чтобы она брала исходные данные как из набора именованных файлов, так и из стандартного потока ввода (в том случае, если не заданы аргументы командной строки). Следует ли выводить имя файла в случае, когда найдено соответствие?

Упражнение 7.8. Напишите программу, выводящую данные из набора файлов. Каждый новый файл должен начинаться с новой страницы и заголовка в виде его имени; должна вестись сквозная для всех файлов нумерация страниц.

7.8. Различные функции

В стандартной библиотеке имеется широкое разнообразие всевозможных функций. В этом разделе дается краткий обзор самых полезных из них. Дополнительные функции и более подробные описания можно найти в приложении Б.

7.8.1. Операции со строками

Ранее уже упоминались такие функции для работы со строками, как `strlen`, `strcpy`, `srtcat` и `strcmp`, объявленные в файле `<string.h>`. Далее в списке `s` и `t` имеют тип `char *`, а `c` и `n` — тип `int`.

`strcat(s,t)`	присоединяет `t` в хвост к `s`
`strncat(s,t,n)`	присоединяет `n` символов из `t` в хвост к `s`
`strcmp(s,t)`	дает отрицательное, нуль или положительное число при `s < t`, `s == t`, `s > t` соответственно

`strncmp(s,t,n)`	то же, что `strcmp`, но выполняется над первыми `n` символами
`strcpy(s,t)`	копирует `t` в `s`
`strncpy(s,t,n)`	копирует не более `n` символов `t` в `s`
`strlen(s)`	вычисляет длину `s`
`strchr(s,c)`	возвращает указатель на первый символ `c` в `s` или `NULL`, если символ не найден
`strrchr(s,c)`	возвращает указатель на последний символ `c` в `s` или `NULL`, если символ не найден

7.8.2. Анализ, классификация и преобразование символов

Несколько функций из файла `<ctype.h>` выполняют анализ и преобразование символов. Далее считается, что `c` — число типа `int`, которое можно интерпретировать либо как `unsigned char`, либо как `EOF`. Все функции возвращают `int`.

`isalpha(c)`	не нуль, если `c` — алфавитный символ; 0, если нет
`isupper(c)`	не нуль, если `c` — буква в верхнем регистре; 0, если нет
`islower(c)`	не нуль, если `c` — буква в нижнем регистре; 0, если нет
`isdigit(c)`	не нуль, если `c` — цифра; 0, если нет
`isalnum(c)`	не нуль, если выполняется `isalpha(c)` или `isdigit(c)`; 0, если нет
`isspace(c)`	не нуль, если `c` — пробел, табуляция, конец строки, возврат каретки, перевод страницы, вертикальная табуляция
`toupper(c)`	возвращает символ `c`, приведенный к верхнему регистру
`tolower(c)`	возвращает символ `c`, приведенный к нижнему регистру

7.8.3. Функция `ungetc`

В стандартной библиотеке имеется довольно ограниченная версия функции `ungetch`, написанной нами в главе 4. Эта функция называется `ungetc`:

```
int ungetc(int c, FILE *fp)
```

Она помещает символ `c` назад в файл `fp` и возвращает либо `c`, либо `EOF` в случае ошибки. Для каждого файла гарантирован возврат только одного символа. Функцию `ungetc` можно использовать совместно с любой из функций ввода `scanf`, `getc` или `getchar`.

7.8.4. Выполнение команд

Функция `system(char *)` выполняет команду, содержащуюся в символьной строке `s`, а затем возобновляет выполнение текущей программы. Допустимое содержимое строки `s` сильно зависит от конкретной операционной среды. Ниже приведен тривиальный пример из системы Unix.

```
system("date");
```

Этот оператор запускает на выполнение программу date; она выводит текущую дату и время в стандартный поток вывода. Функция system возвращает целочисленный код завершения выполненной команды, который зависит от характеристик системы. В системе Unix код завершения представляет собой значение, передаваемое с помощью функции exit.

7.8.5. Управление памятью

С помощью функций malloc и calloc выполняется динамическое распределение блоков памяти. Функция malloc возвращает указатель на n байт неинициализированной памяти или NULL, если запрос на память нельзя выполнить:

```
void *malloc(size_t n)
```

Функция calloc возвращает указатель на участок памяти, достаточный для размещения n объектов заданного размера size или NULL, если запрос на память невыполним. Память инициализируется нулями.

```
void *calloc(size_t n, size_t size)
```

Указатель, возвращаемый функциями malloc и calloc, выровнен в памяти надлежащим образом, но его еще нужно привести к нужному типу:

```
int *ip;

ip = (int *) calloc(n, sizeof(int));
```

Функция free(p) освобождает участок памяти, на который указывает указатель p, первоначально полученный вызовом функции malloc или calloc. Порядок освобождения выделенных участков памяти не регламентируется. Однако если указатель не был получен с помощью malloc или calloc, то его освобождение является грубой ошибкой.

Обращение по указателю после его освобождения — также ошибка. Часто встречается неправильный фрагмент кода, в котором элементы списка освобождаются в цикле:

```
for (p = head; p != NULL; p = p->next) /*НЕПРАВИЛЬНО*/
    free(p);
```

Правильным было бы записать все, что нужно перед освобождением:

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
}
```

В разделе 8.7 показана реализация функции распределения памяти наподобие malloc, в которой выделенные блоки можно освобождать в любом порядке.

7.8.6. Математические функции

В заголовочном файле <math.h> объявлено больше двадцати математических функций; ниже перечислены наиболее часто используемые из них. Каждая принимает один или два аргумента типа double и возвращает результат типа double.

sin(x)	синус x, x в радианах
cos(x)	косинус x, x в радианах
atan2(y,x)	арктангенс отношения y/x в радианах

`exp(x)`	экспоненциальная функция (e в степени x)
`log(x)`	натуральный логарифм x (x > 0)
`log10(x)`	десятичный логарифм x (x > 0)
`pow(x,y)`	x в степени y
`sqrt(x)`	квадратный корень из x (при условии x >= 0)
`fabs(x)`	абсолютное значение x

7.8.7. Генерирование случайных чисел

Функция `rand()` вычисляет последовательность псевдослучайных целых чисел в диапазоне от 0 до RAND_MAX — величины, определенной в файле <stdlib.h>. Вот один из способов генерировать случайные вещественные числа, бо́льшие или равные нулю, но меньшие единицы:

```
#define frand() ((double) rand() / (RAND_MAX+1.0))
```

Учтите, что если в библиотеке вашего компилятора уже есть функция для вычисления случайных вещественных чисел, то она, скорее всего, имеет лучшие статистические свойства, чем приведенная.

Функция `srand(unsigned)` устанавливает для `rand` инициализирующее значение. Системно-независимая реализация `rand` и `srand`, предлагаемая стандартом, приведена в разделе 2.7.

Упражнение 7.9. Такие функции, как `isupper`, можно реализовать с позиций экономии времени, а можно с позиций экономии места. Исследуйте обе возможности.

Глава 8

Интерфейс системы Unix

Операционная система Unix предоставляет свои служебные средства программам через *системные вызовы* (*system calls*), которые по сути являются встроенными функциями системы, вызываемыми из пользовательских программ. В этой главе описано, как пользоваться некоторыми наиболее важными системными вызовами в программах на языке C. Если вы работаете с системой Unix, эта информация пригодится вам непосредственно, поскольку часто бывает необходимо применять системные вызовы с максимальной эффективностью или обращаться к каким-то средствам, которых нет в стандартной библиотеке. Но даже если вы пользуетесь языком C в другой операционной системе, изучение этих примеров поможет вам глубже понять программирование на C — хотя конкретные подробности зависят от системы, аналогичный код можно найти повсюду. Так как библиотека ANSI C в основном построена на идеях Unix, приведенный в этой главе код может помочь вам также в понимании работы библиотечных средств.

Тематику этой главы можно сгруппировать в три больших раздела: ввод-вывод, файловая система и управление памятью. Для изучения первых двух подразумевается некоторое знакомство с внешними характеристиками системы Unix.

В главе 7 изучался интерфейс ввода-вывода, общий для всех операционных систем в силу стандарта. В любой конкретной системе функции стандартной библиотеки пишутся на основе средств, предоставляемых системой. В следующих нескольких разделах рассматриваются системные вызовы Unix для ввода и вывода, а затем демонстрируется, как с их помощью можно реализовать средства стандартной библиотеки.

8.1. Дескрипторы файлов

В операционной системе Unix весь ввод-вывод осуществляется путем чтения и записи файлов, поскольку все периферийные устройства, даже клавиатура и экран монитора, являются файлами единой файловой системы. Это означает, что любая передача данных между программой и периферийными устройствами управляется через единообразный интерфейс.

В самом общем случае, перед тем как читать или записывать файл, необходимо проинформировать об этом намерении операционную систему. Этот процесс называется *открытием* файла. Если будет выполняться запись в файл, может оказаться необходимым создать его или удалить его предыдущее содержимое. Система проверяет, имеете ли вы право на это (существует ли файл и есть ли у вас допуск к нему), и, если все в порядке, возвращает в программу небольшое неотрицательное целое число, именуемое *дескриптором файла* (*file descriptor*). Независимо от того, будет ли выполняться ввод или вывод, для идентификации файла вместо его имени будет применяться дескриптор. (Дескриптор аналогичен файловому указателю из стандартной библиотеки; в системе MS-DOS также используется нечто похожее.) Вся информация об открытом файле нахо-

дится в ведении операционной системы; прикладная программа обращается к файлу только по его дескриптору.

Поскольку ввод-вывод с применением клавиатуры и экрана очень распространен, для удобства его выполнения приняты некоторые дополнительные меры. Когда программа запускается на выполнение в командном интерпретаторе (системной выполняющей среде или оболочке), автоматически открываются три файла с дескрипторами 0, 1 и 2. Они называются стандартными потоками ввода, вывода и ошибок. Если программа вводит данные только из файла 0, а выводит в 1 и 2, ей нет нужды беспокоиться об открытии файлов.

Пользователь программы может перенаправить ввод-вывод в файл и из файла с помощью символов-команд < и >:

```
prog <infile >outfile
```

В этом случае среда изменяет автоматически ассоциированные с дескрипторами 0 и 1 файлы на заданные. Обычно дескриптор 2 всегда остается ассоциированным с экраном, чтобы на него можно было выводить сообщения об ошибках. Примерно то же самое имеет место в случае ввода-вывода через конвейеры. В любом случае ассоциацию дескрипторов файлов выполняет операционная среда, а не программа. Программа понятия не имеет, откуда берутся ее входные данные и куда поступают выходные; она знает только, что входной поток соответствует дескриптору 0, а два выходных — 1 и 2.

8.2. Ввод-вывод низкого уровня — функции `read` и `write`

При вводе и выводе используются системные вызовы `read` и `write`, к которым программы на C обращаются через две функции с теми же именами. У обеих функций первый аргумент — дескриптор файла. Второй аргумент представляет собой символьный массив из программы, в который или из которого поступают данные. Третий аргумент — количество байт, которые необходимо переслать.

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

При каждом вызове возвращается количество переданных байт. При чтении возвращаемое количество байт может быть меньше, чем было запрошено. Нулевое количество означает, что достигнут конец файл, а -1 обозначает какую-либо ошибку. При записи возвращается количество записанных байт; если оно не равно количеству, посланному на запись, то произошла ошибка.

В ходе одного вызова можно записать или прочитать любое количество байт. Наиболее распространенные случаи — это один символ за раз ("небуферизованный ввод-вывод") либо количество наподобие 1024 или 4096, которое соответствует размеру физического блока данных на периферийном устройстве. Большими блоками передавать данные эффективнее, поскольку при этом выполняется меньше системных вызовов.

Из всего вышесказанного следует, что можно написать простую программу для копирования входного потока в выходной — эквивалент программы копирования файлов из главы 1. Эта программа сможет копировать любые потоки, поскольку ввод и вывод можно перенаправить на любой файл или устройство.

```
#include "syscalls.h"

main()  /* копирование потока ввода в поток вывода */
{
    char buf[BUFSIZ];
    int n;

    while ((n = read(0, buf, BUFSIZ)) > 0)
        write(1, buf, n);
    return 0;
}
```

Здесь функциональные прототипы системных вызовов считаются собранными в файл `syscalls.h`, который будет включаться в программы этой главы. Имя этого файла не является стандартным.

Параметр `BUFSIZ` также определен в файле `syscalls.h`; его размер выбирается подходящим для конкретной системы. Если размер файла не кратен `BUFSIZ`, в какой-то момент при вызове `read` будет прочитано меньше байт (и затем записано вызовом `write`), а следующий вызов возвратит значение 0.

Целесообразно рассмотреть вопрос, как можно использовать `read` и `write` для конструирования функций более высокого уровня — `getchar`, `putchar` и т.п. Например, далее приводится версия `getchar`, выполняющая небуферизованный ввод путем считывания из стандартного потока по одному символу за раз.

```
#include "syscalls.h"

/* getchar:  небуферизованный ввод одиночного символа */
int getchar(void)
{
    char c;

    return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
}
```

Переменная `c` должна иметь тип `char`, поскольку `read` требует указателя на символ. Приведение `c` к типу `unsigned char` в операторе `return` устраняет потенциальную проблему с расширением знака.

Вторая версия `getchar` выполняет ввод большими буферами, но выдает символы по запросам по одному за раз.

```
#include "syscalls.h"

/* getchar:  версия с простой буферизацией */
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0) {   /* буфер пустой */
        n = read(0, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

Если бы эти версии getchar компилировались с включением файла <stdio.h>, необходимо было бы отключить определение имени getchar с помощью директивы #undef на тот случай, если оно введено как макрос.

8.3. Функции open, creat, close, unlink

При работе с файлами, не являющимися стандартными потоками ввода, вывода и ошибок, необходимо открыть их явным образом, чтобы иметь возможность читать и записывать. Для этого существуют два системных вызова — open и creat (именно так, а не create).

Функция open очень похожа на fopen, которая рассматривалась в главе 7, но она возвращает не файловый указатель, а файловый дескриптор, представляющий собой всего-навсего целое число типа int. В случае ошибки возвращается -1.

```
#include <fcntl.h>

int fd;
int open(char *name, int flags, int perms);

fd = open(name, flags, perms);
```

Как и в случае fopen, аргумент name — это символьная строка, содержащая имя открываемого файла. Второй аргумент, flags, имеет тип int и указывает способ открытия файла. Ниже приведены некоторые основные значения, которые он может принимать.

O_RDONLY	открытие только для чтения
O_WRONLY	открытие только для записи
O_RDWR	открытие для чтения и записи

Эти константы определены в файле <fcntl.h> в версии системы Unix под названием System V и в файле <sys/file.h> в версии BSD (Berkeley).

Существующий файл открывается для чтения следующим образом:

```
fd = open(name, O_RDONLY, 0);
```

Во всех дальнейших применениях функции open аргумент perms всегда будет равен 0.

Попытка открыть несуществующий файл является ошибкой. Для создания новых файлов или перезаписи существующих заново используется системный вызов creat:

```
int creat(char *name, int perms);

fd = creat(name, perms);
```

При этом возвращается дескриптор файла, если создание прошло успешно, или -1, если нет. Если файл уже существует, creat усекает его до нулевой длины, тем самым уничтожая имеющееся содержимое. Таким образом, создавать существующий файл с помощью creat не является ошибкой.

Если файл не существует, creat создает его с набором допусков, определяемых аргументом perms. В файловой системе Unix с файлом ассоциируется девять битов служебной информации о допусках к нему. Они регулируют чтение, запись и запуск этого

файла его владельцем, группой, к которой принадлежит владелец, и всеми остальными пользователями.

Для иллюстрации покажем упрощенную версию программы cp из среды Unix, которая копирует один файл в другой. Наша версия копирует всего один файл, не допускает имени каталога в качестве второго аргумента, и сама устанавливает допуски вместо копирования их.

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666  /* Чтение/запись для владельца, группы, остальных */

void error(char *, ...);

/* cp:  копирование файла f1 в f2 */
main(int argc, char *argv[])
{
    int f1, f2, n;
    char buf[BUFSIZ];

    if (argc != 3)
        error("Usage: cp from to");
    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("cp: can't create %s, mode %03o",
            argv[2], PERMS);
    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error on file %s", argv[2]);
    return 0;
}
```

Эта программа создает файл для записи с всегда фиксированными допусками 0666. С помощью системного вызова stat, описанного в разделе 8.6, можно определить режим допусков существующего файла и установить такой же для его копии.

Обратите внимание, что функция error вызывается с переменными списками аргументов аналогично printf. Реализация error показывает пример использования еще одного члена семейства printf. Существует библиотечная функция vprintf, которая во всем похожа на printf, но вместо переменного списка аргументов принимает один аргумент, инициализированный вызовом макроса va_start. Аналогичным образом vfprintf и vsprintf повторяют возможности соответственно fprintf и sprintf.

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

/* error:  вывод сообщения об ошибке и останов программы */
void error(char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, "error: ");
    vfprintf(stderr, fmt, args);
```

```
    fprintf(stderr, "\n");
    va_end(args);
    exit(1);
}
```

Существует предельное количество (обычно около 20) файлов, которые программа может одновременно держать открытыми. Соответственно любая программа, которой необходимо обрабатывать сразу много файлов, должна быть готова к повторному использованию файловых дескрипторов. Функция `close(int fd)` разрывает связь между файловым дескриптором и открытым файлом и освобождает дескриптор для последующей работы с другим файлом. Она соответствует функции `fclose` из стандартной библиотеки с той разницей, что при этом нет никакого очищаемого буфера. Выход из программы с помощью функции `exit` или возвращение из тела `main` оператором `return` автоматически закрывает все открытые файлы.

Функция `unlink(char *name)` удаляет заданное имя файла из файловой системы. Она соответствует стандартной библиотечной функции `remove`.

Упражнение 8.1. Перепишите программу `cat` из главы 7 с использованием системных вызовов `read`, `write`, `open` и `close` вместо их эквивалентов из стандартной библиотеки. Проведите эксперименты по сравнению быстродействия двух версий этой программы.

8.4. Прямой доступ к файлу и функция `lseek`

Ввод и вывод обычно выполняются последовательно: каждая операция `read` или `write` выполняется в той позиции файла, на которой остановилась предыдущая операция. Однако по мере необходимости файл можно считывать или записывать в произвольном порядке. Системный вызов `lseek` позволяет передвигаться по файлу, не считывая и не записывая никаких данных:

```
long lseek(int fd, long offset, int origin);
```

Эта функция устанавливает указатель текущей позиции в файле с дескриптором `fd` на расстояние `offset` байт от места, заданного параметром `origin`. Следующая операция чтения или записи будет выполняться в этой позиции. Аргумент `origin` может принимать значения 0, 1 или 2, указывая тем самым, что смещение `offset` следует отсчитывать соответственно от начала, текущей позиции или конца файла. Например, чтобы дописать данные в конец файла (для чего в командной оболочке Unix служит команда перенаправления `>>`, а в функции `fopen` — аргумент `"a"`), следует перед записью установить указатель позиции на конец файла:

```
lseek(fd, 0L, 2);
```

Чтобы переместиться в начало ("перемотать" файл), нужно записать следующее:

```
lseek(fd, 0L, 0);
```

Обратите внимание на аргумент `0L`; его также можно записать в виде `(long)0` или просто `0`, если `lseek` объявлена должным образом.

С помощью функции lseek можно обращаться с файлами более-менее как с большими массивами, ценой некоторого снижения скорости доступа. Например, следующая функция считывает заданное количество байт из произвольного места файла. Она возвращает количество считанных байтов или -1 в случае ошибки.

```
#include "syscalls.h"

/* get:  считывание n байт из позиции pos */
int get(int fd, long pos, char *buf, int n)
{
    if (lseek(fd, pos, 0) >= 0)  /* подход к pos */
        return read(fd, buf, n);
    else
        return -1;
}
```

Из lseek возвращается значение типа long, дающее новую текущую позицию в файле, или -1 в случае ошибки. Стандартная библиотечная функция fseek аналогична lseek, но ее первый аргумент должен иметь типа FILE *, а в случае ошибки она возвращает ненулевое число.

8.5. Пример реализации функций fopen и getc

Продемонстрируем, как некоторые из описанных средств работают вместе, на примере реализации стандартных библиотечных функций fopen и getc.

Напомним, что файлы в стандартной библиотеке описываются файловыми указателями, а не дескрипторами. Файловый указатель — это указатель на структуру, содержащую набор данных о файле. Набор данных включает в себя указатель на буфер, через который файл можно читать большими фрагментами; счетчик количества символов, оставшихся в буфере; указатель на очередную символьную позицию в буфере; дескриптор файла; различные флаги, задающие режим чтения и записи, текущие ошибки и т.д.

Структура данных, описывающая файл, определена в файле <stdio.h>, который необходимо включить с помощью директивы #include в любой файл исходного кода, где используются функции и средства стандартной библиотеки ввода-вывода. Функции самой библиотеки также подключают этот файл. В следующем фрагменте типичного файла <stdio.h> имена, предназначенные для использования только функциями самой библиотеки, начинаются со знака подчеркивания, чтобы уменьшить вероятность их совпадения с идентификаторами прикладных программ. Это соглашение используется всеми стандартными библиотечными функциями.

```
#define NULL      0
#define EOF       (-1)
#define BUFSIZ    1024
#define OPEN_MAX  20  /* максимальное количество открытых файлов */

typedef struct _iobuf {
    int  cnt;         /* сколько осталось символов */
    char *ptr;        /* следующая символьная позиция */
```

```
    char *base;       /* местонахождение буфера */
    int  flag;        /* режим доступа к файлу */
    int fd;           /* дескриптор файла */
} FILE;
extern FILE _iob[OPEN_MAX];

#define stdin   (&_iob[0])
#define stdout  (&_iob[1])
#define stderr  (&_iob[2])

enum _flags {
    _READ   = 01,     /* файл открыт для чтения */
    _WRITE  = 02,     /* файл открыт для записи */
    _UNBUF  = 04,     /* файл без буферизации */
    _EOF    = 010,    /* в файле достигнут конец */
    _ERR    = 020     /* произошла ошибка */
};

int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define feof(p)     (((p)->flag & _EOF) != 0)
#define ferror(p)   (((p)->flag & _ERR) != 0)
#define fileno(p)   ((p)->fd)

#define getc(p)    (--(p)->cnt >= 0 \
                ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define putc(x,p) (--(p)->cnt >= 0 \
                ? *(p)->ptr++ = (x) : _flushbuf((x),p))

#define getchar()   getc(stdin)
#define putchar(x)  putc((x), stdout)
```

В большинстве случаев макрос `getc` уменьшает на единицу количество непрочитанных символов, передвигает указатель вперед и возвращает прочитанный символ. (Напоминаем, что слишком длинная директива `#define` продолжается на следующей строке с помощью обратной косой черты.) Если счетчик символов становится отрицательным, `getc` вызывает функцию `_fillbuf` для пополнения буфера, заново инициализирует содержимое структуры, а затем возвращает полученный символ. Символы возвращаются в формате `unsigned`, чтобы гарантировать их положительность.

Хотя детали здесь не обсуждаются, все же мы включили в текст определение `putc`, чтобы показать, что этот макрос устроен и работает примерно так же, как `getc`, вызывая функцию `_flushbuf` в случае заполненного буфера. Включены также определения макросов для обращения к индикаторам ошибок и конца файла, а также для получения файлового дескриптора.

Теперь можно написать функцию `fopen`. Большая часть этой функции посвящена открытию файла и установке указателя в нужное место, а также заполнению битов-флагов состояния. Функция `fopen` не занимается выделением памяти для буфера; эта задача возложена на функцию `_fillbuf`, которая вызывается при первом чтении из файла.

```
#include <fcntl.h>
#include "syscalls.h"
```

```
#define PERMS 0666    /* разрешены чтение и запись */
                      /* для владельца, группы, остальных  */

/* fopen:  открытие файла с возвращением файлового указателя */
FILE *fopen(char *name, char *mode)
{
    int fd;
    FILE *fp;

    if (*mode != 'r' && *mode != 'w' && *mode != 'a')
        return NULL;
    for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
        if ((fp->flag & (_READ | _WRITE)) == 0)
            break;          /* найдено свободное место для файла */
    if (fp >= _iob + OPEN_MAX) /* нет свободных мест для файла */
        return NULL;

    if (*mode == 'w')
        fd = creat(name, PERMS);
    else if (*mode == 'a') {
        if ((fd = open(name, O_WRONLY, 0)) == -1)
            fd = creat(name, PERMS);
        lseek(fd, 0L, 2);
    } else
        fd = open(name, O_RDONLY, 0);
    if (fd == -1);          /* указанное имя недоступно */
        return NULL;
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag = (*mode == 'r') ? _READ : _WRITE;
    return fp;
}
```

Эта версия `fopen` не работает со всеми возможными режимами доступа к файлу, имеющимися в стандарте, хотя их реализация и не потребовала бы много дополнительного кода. В частности, наша функция `fopen` не распознает режим `"b"` для двоичного доступа к файлу, поскольку система Unix не делает такого различия, а также режим `"+"` для комбинированного чтения и записи.

При первом вызове `getc` с конкретным файлом счетчик непрочитанных символов равен нулю, поэтому автоматически вызывается `_fillbuf`. Если функция `_fillbuf` обнаруживает, что файл не открыт для чтения, она немедленно возвращает `EOF`. В противном случае она пытается выделить буфер памяти (если чтение должно буферизироваться).

Как только буфер создан, `_fillbuf` вызывает `read` для его заполнения, устанавливает счетчик и указатели и возвращает символ из начала буфера. При последующих вызовах `_fillbuf` буфер уже будет в готовом виде.

```
#include "syscalls.h"

/* _fillbuf:  создание и заполнение буфера ввода */
int _fillbuf(FILE *fp)
{
    int bufsize;
```

```
    if ((fp->flag&(_READ|_EOF|_ERR)) != _READ)
        return EOF;
    bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
    if (fp->base == NULL)      /* буфера еще нет */
        if ((fp->base = (char *) malloc(bufsize)) == NULL)
            return EOF;        /* не удается создать буфер */
    fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, bufsize);
    if (--fp->cnt < 0) {
        if (fp->cnt == -1)
            fp->flag |= _EOF;
        else
            fp->flag |= _ERR;
        fp->cnt = 0;
        return EOF;
    }
    return (unsigned char) *fp->ptr++;
}
```

Единственное, что осталось проработать, — инициализацию всех файловых операций. Следует определить массив `_iob` и инициализировать его потоками `stdin`, `stdout` и `stderr`:

```
FILE _iob[OPEN_MAX] = {    /* stdin, stdout, stderr: */
    { 0, (char *) 0, (char *) 0, _READ, 0 },
    { 0, (char *) 0, (char *) 0, _WRITE, 1 },
    { 0, (char *) 0, (char *) 0, _WRITE | _UNBUF, 2 }
};
```

Инициализация полей `flag` файловых структур показывает, что `stdin` будет использоваться для чтения, `stdout` — для записи, а `stderr` — для записи без буферизации.

Упражнение 8.2. Перепишите функции `fopen` и `_fillbuf` с применением битовых полей вместо явных операций над битами. Сравните объем кода и быстродействие.

Упражнение 8.3. Спроектируйте и напишите функции `_flushbuf`, `fflush` и `fclose`.

Упражнение 8.4. Следующая стандартная библиотечная функция эквивалентна `lseek`:

```
int fseek(FILE *fp, long offset, int origin)
```

Различие состоит в том, что ее аргумент `fp` — это файловый указатель, а не файловый дескриптор, и возвращаемое ею значение типа `int` сообщает статус завершения, а не текущую позицию. Напишите функцию `fseek`. Сделайте так, чтобы она согласовывала свою работу с буферизацией, выполняемой другими библиотечными функциями.

8.6. Пример получения списка файлов в каталоге

Иногда бывает необходимо получить через файловую систему информацию не о содержимом файла, а *о самом файле*. Примером может служить программа `ls` из системы Unix для вывода списка файлов в каталоге с сопутствующей информацией о них, такой как размеры, допуски и т.д. Аналогичные функции выполняет команда `dir` в системе MS-DOS.

Поскольку каталог в Unix представляет собой просто файл, программе `ls` нужно всего лишь прочитать этот файл и извлечь из него нужные имена. Но для получения другой информации — например, размера файла — необходимо пользоваться системными вызовами. В других системах даже для получения имен файлов в каталоге не обойтись без системного вызова; именно так обстоит дело в MS-DOS. Хотелось бы обеспечить доступ к этой информации сравнительно системно-независимым способом, пусть даже реализация окажется существенно основанной на средствах конкретной системы.

Проиллюстрируем сказанное, написав программу под названием `fsize`. Эта программа — особая разновидность `ls`, выводящая размеры всех файлов, перечисленных в списке ее аргументов командной строки. Если один из файлов оказывается каталогом, `fsize` применяет сама себя рекурсивно к этому каталогу. Если аргументов нет вообще, программа работает с текущим каталогом.

Начнем с краткого обзора устройства файловой системы Unix. *Каталогом* (*directory*) в ней называется файл, содержащий список имен файлов и информацию об их местоположении. Эта информация представляет собой ссылку на другую таблицу — список *файловых индексов*. Файловый индекс (*inode*) представляет собой структуру данных, содержащую всю информацию о файле, кроме его имени. Каждый пункт списка файлов в каталоге обычно состоит из имени файла и соответствующего номера индекса.

К сожалению, формат и точное содержание файла-каталога меняется от одной версии системы к другой. Поэтому разделим задачу на составные части и попытаемся изолировать непереносимые фрагменты. На внешнем уровне определяется структура `Dirent` и три функции, `opendir`, `readdir` и `closedir`, которые обеспечивают системно-независимый доступ к имени и номеру индекса в списке каталога. Функция `fsize` будет написана с применением этого интерфейса. Затем будет показано, как реализовать эти средства в системах, имеющих ту же структуру каталогов, что и системы Unix Version 7 и System 5. Проработку вариантов оставляем читателю в качестве упражнения.

Структура `Dirent` содержит номер индекса и имя. Максимальная длина компоненты, соответствующей имени, равна `NAME_MAX` и зависит от реализации системы. Функция `opendir` возвращает указатель на структуру `DIR`, аналогичную `FILE`, которая используется в `readdir` и `closedir`. Вся эта информация собрана в файле `dirent.h`:

```
#define NAME_MAX   14  /* самая длинная компонента имени */
                       /* системно-зависимая */

typedef struct {       /* универсальный пункт списка: */
    long ino;                  /* номер индекса */
    char name[NAME_MAX+1];     /* имя + '\0' */
} Dirent;

typedef struct {       /* минимум: без буферизации и т.п. */
    int fd;              /* файловый дескриптор каталога */
    Dirent d;            /* пункт списка файлов */
} DIR;

DIR *opendir(char *dirname);
Dirent *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

Системный вызов `stat` принимает имя файла в качестве аргумента и возвращает всю информацию из индекса этого файла или `-1` в случае ошибки. Следующий фрагмент ко-

да заполняет структуру stbuf информацией из индекса, соответствующей заданному имени файла:

```
char *name;
struct stat stbuf;
int stat(char *, struct stat *);

stat(name, &stbuf);
```

Структура, описывающая возвращаемое из stat значение, определена в <sys/stat.h> и обычно выглядит так:

```
struct stat     /* информация о файле, возвращаемая stat */
{
     dev_t      st_dev;   /* устройство индекса */
     ino_t      st_ino;   /* номер индекса */
     short      st_mode;  /* биты режима */
     short      st_nlink; /* количество ссылок на файл */
     short      st_uid;   /* идентификационный номер владельца */
     short      st_gid;   /* идентификационный номер группы владельца */
     dev_t      st_rdev;  /* для специальных файлов */
     off_t      st_size;  /* размер файла в символах */
     time_t     st_atime; /* время последнего обращения */
     time_t     st_mtime; /* время последней модификации */
     time_t     st_ctime; /* время последней модификации индекса */
};
```

Большинство значений разъяснены в комментариях. Типы dev_t и ino_t определены в файле <sys/types.h>, который также следует подключить.

Поле st_mode содержит набор флаговых битов, описывающих свойства файла. Определения флагов также даны в файле <sys/stat.h>; нам понадобится только та часть, которая имеет отношение к типам файлов:

```
#define   S_IFMT   0160000  /* тип файла: */
#define   S_IFDIR  0040000  /* каталог */
#define   S_IFCHR  0020000  /* символьный специальный */
#define   S_IFBLK  0060000  /* блочный специальный */
#define   S_IFREG  0100000  /* обычный */

/* ... */
```

Теперь мы готовы к написанию программы fsize. Если режим, полученный от stat, соответствует файлу, а не каталогу, то размер сразу же оказывается под рукой, и его можно непосредственно вывести. Однако если файл является каталогом, тогда необходимо дополнительно прочитать из него список файлов по одному за раз. К тому же каталог может содержать подкаталоги, так что обработка будет носить рекурсивный характер.

Главный модуль программы обрабатывает аргументы командной строки; он передает каждый аргумент по очереди в функцию fsize.

```
#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h>      /* флаги чтения и записи */
#include <sys/types.h>  /* определения типов */
#include <sys/stat.h>   /* структура, возвращаемая stat */
#include "dirent.h"
```

```
void fsize(char *);

/* вывод размеров файлов */
main(int argc, char **argv)
{
    if (argc == 1)      /* по умолчанию текущий каталог */
        fsize(".");
    else
        while (--argc > 0)
            fsize(*++argv);
    return 0;
}
```

Функция `fsize` выводит размер файла. Однако если файл является каталогом, `fsize` вначале вызывает функцию `dirwalk` для обработки всех файлов каталога. Обратите внимание, как для определения того, является ли файл каталогом, используются флаги `S_IFMT` и `S_IFDIR` из файла `<sys/stat.h>`. Наличие скобок обязательно, поскольку приоритет операции & ниже, чем ==.

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));

/* fsize:  вывод размера файла name */
void fsize(char *name)
{
    struct stat stbuf;

    if (stat(name, &stbuf) == -1) {
        fprintf(stderr, "fsize: can't access %s\n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        dirwalk(name, fsize);
    printf("%8ld %s\n", stbuf.st_size, name);
}
```

Функция `dirwalk` имеет обобщенное назначение; ее задача — применить заданную функцию к каждому файлу в каталоге. Она открывает каталог, перебирает все имеющиеся в нем файлы, вызывает заданную функцию с каждым из них, а затем закрывает каталог и завершается. Поскольку функция `fsize` вызывает `dirwalk` в каждом каталоге, эти две функции связаны между собой рекурсивным вызовом.

```
#define MAX_PATH 1024

/* dirwalk:  применение fcn ко всем файлам каталога dir */
void dirwalk(char *dir, void (*fcn)(char *))
{
    char name[MAX_PATH];
    Dirent *dp;
    DIR *dfd;

    if ((dfd = opendir(dir)) == NULL) {
        fprintf(stderr, "dirwalk: can't open %s\n", dir);
        return;
    }
    while ((dp = readdir(dfd)) != NULL) {
```

```
        if (strcmp(dp->name, ".") == 0
         || strcmp(dp->name, "..") == 0)
            continue;   /* пропустить себя и родительский */
        if (strlen(dir)+strlen(dp->name)+2 > sizeof(name))
            fprintf(stderr, "dirwalk: name %s %s too long\n",
                dir, dp->name);
        else {
            sprintf(name, "%s/%s", dir, dp->name);
            (*fcn)(name);
        }
    }
    closedir(dfd);
}
```

При каждом вызове функции `readdir` возвращается указатель на информацию об очередном файле или NULL, если файлов больше нет. Каждый каталог всегда содержит пункты списка, соответствующие ему самому, с именем " . ", и родительскому каталогу с именем " . . ". Их необходимо пропустить, иначе программа зациклится.

Вплоть до этого уровня код был независим от того, в каком формате хранится информация о каталогах. Следующий наш шаг — написать минимальные, урезанные версии функций `opendir`, `readdir` и `closedir` для конкретной системы. Приведенные ниже варианты написаны для Unix Version 7 и System V; они используют формат информации о каталогах из заголовочного файла `<sys/dir.h>`, который выглядит так:

```
#ifndef DIRSIZ
#define DIRSIZ  14
#endif
struct direct   /* запись в каталоге */
{
    ino_t d_ino;          /* номер индекса */
    char  d_name[DIRSIZ]; /* в длинном имени нет '\0' */
};
```

Некоторые версии системы разрешают намного более длинные имена и имеют более сложную структуру каталогов.

Тип `ino_t` определяется через `typedef` и обозначает номер в списке файловых индексов. В системе, которой мы обычно пользуемся, он совпадает с `unsigned short`, но информацию об этом нельзя внедрять в программу, поскольку в разных системах тип может отличаться. Поэтому лучше использовать `typedef`. Полный набор "системных" типов можно найти в файле `<sys/types.h>`.

Функция `opendir` открывает каталог, проверяет, что файл является каталогом (на этот раз системным вызовом `fstat`, аналогичным `stat` с той лишь разницей, что он применяется к файловому дескриптору), размещает в памяти структуру каталога и записывает информацию:

```
int fstat(int fd, struct stat *);

/* opendir:  открывает каталог для вызовов readdir */
DIR *opendir(char *dirname)
{
    int fd;
    struct stat stbuf;
    DIR *dp;
```

```
    if ((fd = open(dirname, O_RDONLY, 0)) == -1)
     || fstat(fd, &stbuf) == -1
     || (stbuf.st_mode & S_IFMT) != S_IFDIR
     || (dp = (DIR *) malloc(sizeof(DIR))) == NULL)
        return NULL;
    dp->fd = fd;
    return dp;
}
```

Функция `closedir` закрывает файл каталога и освобождает место в памяти:

```
/* closedir:  закрытие каталога, открытого opendir */
void closedir(DIR *dp)
{
    if (dp) {
        close(dp->fd);
        free(dp);
    }
}
```

Наконец, функция `readdir` с помощью функции `read` считывает каждую запись о файле в каталоге. Если какая-либо ячейка в каталоге не используется в текущий момент (файл удален), то номер индекса равен нулю, и данная позиция пропускается. В противном случае номер индекса и имя помещаются в статическую структуру, указатель на которую возвращается пользователю. При каждом вызове затирается информация, полученная предыдущим вызовом.

```
#include <sys/dir.h>   /* местная структура каталога */

/* readdir:  чтение записей в каталоге по порядку */
Dirent *readdir(DIR *dp)
{
    struct direct dirbuf; /* местная структура каталога */
    static Dirent d;      /* переносимая структура */

    while (read(dp->fd, (char *) &dirbuf, sizeof(dirbuf))
                    == sizeof(dirbuf)) {
        if (dirbuf.d_ino == 0)    /* ячейка не занята */
            continue;
        d.ino = dirbuf.d_ino;
        strncpy(d.name, dirbuf.d_name, DIRSIZ);
        d.name[DIRSIZ] = '\0';  /* гарантированный конец */
        return &d;
    }
    return NULL;
}
```

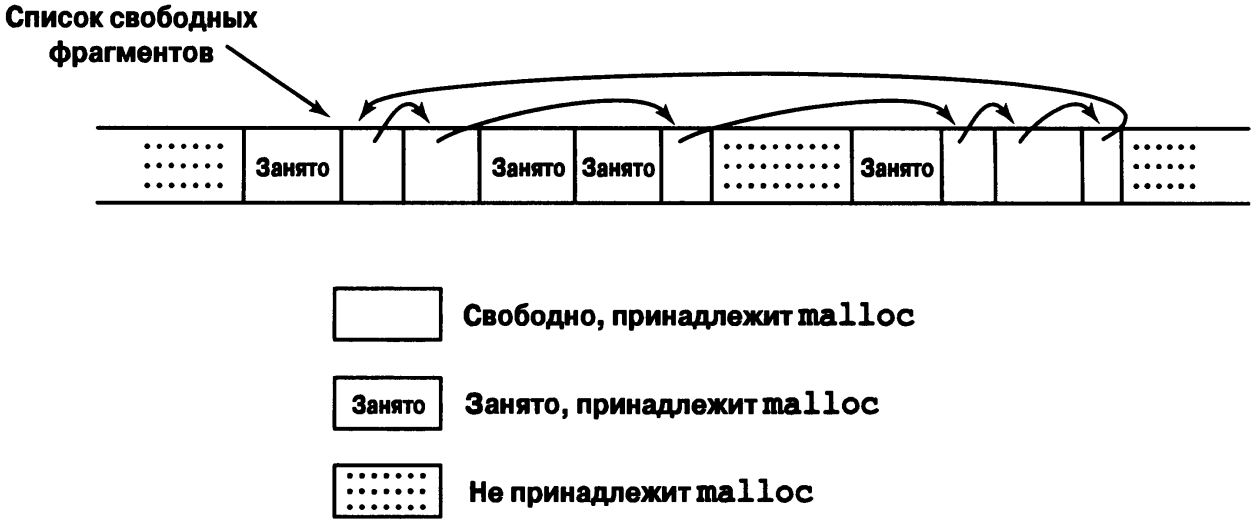
Хотя программа `fsize` достаточно узко специализирована, она все-таки иллюстрирует несколько важных идей. Во-первых, многие программы не являются "системными программами"; они просто пользуются информацией, которую хранит и предоставляет операционная система. В таких программах очень важно, чтобы способ представления информации был целиком описан в стандартных заголовочных файлах, которые подключаются к программе; объявления и определения не должны включаться в текст программы непосредственно. Во-вторых, при известной сноровке можно разработать интерфейс к системно-зависимым объектам, который сам по себе будет относительно независим от системы. Функции стандартной библиотеки могут служить хорошим примером этого.

Упражнение 8.5. Переработайте программу `fsize` так, чтобы она выводила другую информацию из файловых индексов.

8.7. Пример распределения памяти

В главе 5 была представлена функция распределения памяти с очень ограниченными возможностями, основанная на работе со стеком. Версия, которая будет представлена здесь, не имеет ограничений. Вызовы функций `malloc` и `free` могут выполняться в любом порядке; `malloc` обращается к операционной системе, запрашивая у нее дополнительную память по мере необходимости. Эти функции иллюстрируют некоторые соображения, касающиеся написания системно-зависимого кода в сравнительно системно-независимой манере, а также демонстрируют реальное практическое применение структур, объединений и операторов `typedef`.

Вместо выделения памяти из массива фиксированной длины, объявленного еще при компиляции, функция `malloc` будет запрашивать память у операционной системы по мере возможности. Поскольку в ходе других операций в программе также может запрашиваться память без помощи этой функции распределения памяти, управляемое `malloc` пространство может оказаться не непрерывным. Таким образом, свободная память будет храниться в виде списка свободных блоков. Каждый блок содержит свой размер, указатель на следующий блок и сам участок памяти. Блоки хранятся в порядке возрастания адресов, а последний блок (с самым старшим адресом) ссылается на первый.



Когда поступает запрос на память, выполняется перебор списка свободных блоков, пока не обнаруживается достаточно большой блок. Этот алгоритм кратко называется "первый подходящий" — в отличие от "наилучшего подходящего", в котором отыскивается наименьший из блоков, подходящих по размеру. Если блок имеет точно такой размер, как запрашивается, он отсоединяется от списка и передается пользователю. Если блок имеет больший размер, он делится на части, и пользователь получает столько, сколько просил, а остаток остается в списке. Если не удается найти достаточно большой блок, у операционной системы запрашивается очередной большой фрагмент памяти, который подключается к списку.

При освобождении также выполняется поиск по списку свободных блоков: отыскивается место для вставки освобождаемого блока. Если освобождаемый блок вплотную гра-

ничит со свободным блоком с какой-либо из двух сторон, то он сливается с ним в единый блок большего размера, чтобы не слишком фрагментировать память. Определить, как граничат блоки, нетрудно, поскольку список свободных блоков организован по возрастанию адресов.

Одна проблема, на наличие которой мы намекали в главе 5, заключается в том, чтобы обеспечить правильное выравнивание блока памяти, получаемого от `malloc`, для хранения помещаемых в нее объектов. Различные системы устроены по-разному, но в каждой есть самый "требовательный" тип данных: если элемент данных этого типа можно поместить по определенному адресу, то любые другие элементы также можно поместить туда. В некоторых системах самый требовательный тип — `double`, тогда как в других достаточно `int` или `long`.

Свободный блок содержит указатель на следующий блок в цепочке, размер блока и собственно участок свободной памяти; управляющая информация в начале называется "заголовком". Чтобы упростить выравнивание, все блоки по длине кратны размеру заголовка, а сам заголовок выровнен надлежащим образом. Это делается с помощью объединения, содержащего желаемую структуру заголовка и экземпляр самого требовательного типа данных, которым мы назначили `long`:

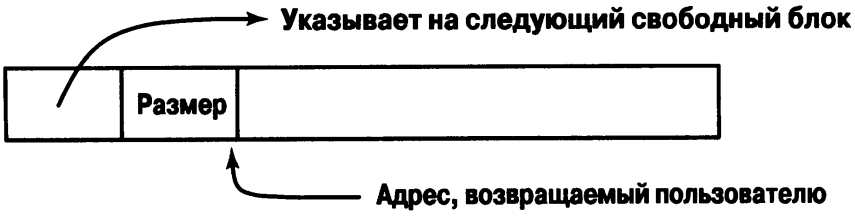
```
typedef long Align;  /* для выравнивания по границе long */

union header {       /* заголовок блока */
    struct {
        union header *ptr; /* следующий блок, если есть */
        unsigned size;     /* размер этого блока */
    } s;
    Align x;         /* принудительное выравнивание блоков */
};

typedef union header Header;
```

Поле `Align` никогда не используется; его наличие просто выравнивает каждый заголовок принудительно по границе самого требовательного типа.

В функции `malloc` запрашиваемое количество памяти (в символах) округляется до размера, кратного длине заголовка. Выделяемый участок содержит на один такой блок больше, чем нужно, — для помещения заголовка. Именно эта длина и записывается в поле `size` заголовка. Указатель, возвращаемый из `malloc`, указывает на свободное пространство, а не на заголовок. Пользователь может делать с полученным участком памяти все, что захочет, но если что-либо будет записано за пределами выделенной памяти, весь список скорее всего будет испорчен.



Поле длины, `size`, необходимо, поскольку блоки, распределяемые функцией `malloc`, не обязаны занимать сплошную область, — поэтому вычислить их длину с помощью адресной арифметики невозможно.

Для начала работы используется переменная `base`. Если `freep` равен `NULL`, как это имеет место при первом вызове `malloc`, создается вырожденный список свободных

блоков. Он содержит один блок нулевого размера и указывает сам на себя. В любом случае затем выполняется поиск по списку. Поиск свободного блока подходящей длины начинается в точке (`freep`), в которой был найден последний запрошенный блок; такая схема позволяет поддерживать единообразие работы со списком. Если найден слишком длинный блок, пользователю возвращается его "хвост"; благодаря этому в заголовке исходного блока остается исправить только размер. Во всех случаях указатель, возвращаемый пользователю, указывает на свободное пространство в блоке, которое начинается через один блок после заголовка.

```
static Header base;        /* пустой список для начала */
static Header *freep = NULL;          /* начало списка */

/* malloc:  функция распределения памяти */
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *morecore(unsigned);
    unsigned nunits;

    nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
    if ((prevp = freep) == NULL) {  /* списка еще нет */
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {  /* достаточный размер */
            if (p->s.size == nunits)  /* в точности */
                prevp->s.ptr = p->s.ptr;
            else {              /* отрезаем "хвост" */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p+1);
        }
        if (p == freep)  /* ссылается на сам список */
            if ((p = morecore(nunits)) == NULL)
                return NULL;    /* не осталось памяти */
    }
}
```

Функция `morecore` запрашивает и получает память от операционной системы. Детали реализации такого запроса меняются от системы к системе. Запрос памяти у системы — это сравнительно трудоемкая операция, и ее не стоит выполнять при каждом вызове `malloc`. Вот почему функция `morecore` запрашивает сразу как минимум `NALLOC` блоков; полученный большой блок потом будет "нарезаться" по мере необходимости. После установки поля размера `morecore` добавляет полученную память в общий фонд, вызывая функцию `free`.

Системный вызов Unix под названием `sbrk(n)` возвращает указатель на `n` дополнительных байт памяти. Эта функция возвращает `-1`, если места в памяти недостаточно, хотя возвращать `NULL` в этом случае было бы удачнее. Число `-1` следует привести к типу `char *`, чтобы можно было сравнивать его с возвращаемым значением.

Приведение типов делает функцию сравнительно устойчивой к различиям представлений указателей в разных системах. Правда, здесь сделано предположение о том, что указатели на различные блоки, возвращаемые из `sbrk`, можно сравнивать и получать при этом правильные ответы. Стандарт этого не гарантирует, поскольку разрешает сравнивать указатели только в пределах массива. Поэтому данная версия `malloc` переносима только между системами, в которых сравнение произвольных указателей имеет смысл и реализовано корректно.

```
#define NALLOC  1024 /* минимально запрашиваемое количество блоков */

/* morecore:  запрос дополнительной памяти у системы */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1)  /* места в памяти нет */
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *)(up+1));
    return freep;
}
```

И последней в нашем наборе функций идет `free`. Она перебирает список свободных блоков, начиная с `freep`, в поисках места для вставки освобождаемого блока. Такое место может найтись либо между двумя блоками, либо на одном из концов списка. В любом случае, если освобождаемый блок примыкает вплотную к одному из соседей, соседние блоки объединяются в один. Единственное, о чем следует тщательно позаботиться, — это о том, чтобы указатели указывали на правильные места и были заданы правильные размеры.

```
/* free:  помещение блока ap в список свободных блоков */
void free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)ap - 1;    /* указ. на заголовок */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break; /* освобождаемый блок в начале или в конце */

    if (bp + bp->s.size == p->s.ptr) { /* к верхнему соседу */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) {         /* к нижнему соседу */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
```

```
        p->s.ptr = bp;
    freep = p;
}
```

Хотя распределение памяти — по самой своей сути всегда системно-зависимая операция, приведенный код иллюстрирует, как эту зависимость можно сделать управляемой и локализовать в очень малой части программы. Выравнивание регулируется с помощью `typedef` и объединений (при условии, что `sbrk` возвращает правильный указатель). Приведение типов, применяемое к указателям, сделано явным и даже приспособлено к неудачному системному интерфейсу. Хотя подробности реализации приведенных функций относятся к распределению памяти, общий подход можно применять и к решению других проблем.

Упражнение 8.6. Стандартная библиотечная функция `calloc(n,size)` возвращает указатель на `n` объектов размера `size` и инициализирует участок памяти нулями. Напишите функцию `calloc`, либо просто вызывая `malloc`, либо доработав ее.

Упражнение 8.7. Функция `malloc` принимает запрос на участок памяти определенного размера, не проверяя разумность этого размера; в функции `free` предполагается, что освобождаемый блок содержит корректно заданное поле размера. Доработайте эти функции так, чтобы они более тщательно проверяли возможные ошибки.

Упражнение 8.8. Напишите функцию `bfree(p,n)`, которая бы освобождала произвольный блок `p` из `n` символов и добавляла его в список свободных блоков, которым управляют функции `malloc` и `free`. С помощью `bfree` пользователь всегда сможет добавить к списку свободных блоков любой статический или внешний массив.

Приложение А

Справочное руководство по языку С

А.1. Введение

Данное руководство описывает язык С в том виде, в каком он определен проектом документа "American National Standard for Information Systems — Programming Language C, X3.159-1989" ("Американский государственный стандарт информационных систем — язык программирования С, X3.159-1989"), поданного на утверждение в комитет ANSI 31 октября 1988 года. Справочник является интерпретацией предложенного стандарта, но не самим стандартом, хотя нами были приложены все усилия, чтобы сделать его надежным пособием по языку.

Данный документ в основном следует схеме изложения стандарта, который в свою очередь построен аналогично первой редакции этой книги, хотя детали могут отличаться. Представленная здесь грамматика основного ядра языка полностью совпадает со стандартом, если не считать отличий в наименованиях нескольких правил, а также отсутствия строгих определений лексических единиц препроцессора.

> В данном справочнике повсеместно встречаются комментарии, оформленные так, как этот абзац. Чаще всего эти комментарии поясняют различия между стандартом ANSI языка С и его определением, данным в первой редакции этой книги (или усовершенствованиями, внесенными впоследствии в различных компиляторах).

А.2. Лексические соглашения

Программа состоит из одной или нескольких *единиц трансляции* (*translation units*), хранящихся в виде файлов. Она проходит несколько этапов трансляции, описанных в разделе А.12. На начальных этапах осуществляются лексические преобразования низкого уровня, выполняются директивы, заданные в программе строками, начинающимися с символа #, обрабатываются и раскрываются макроопределения. По завершении работы препроцессора (раздел А.12) программа представляется в виде последовательности *лексем* (*tokens*).

А.2.1. Лексемы

Существует шесть классов лексем: идентификаторы, ключевые слова, константы, строковые литералы, знаки операций и прочие разделители. Пробелы, горизонтальные и

вертикальные табуляции, символы конца строки, символы прогона страницы и комментарии (имеющие общее название "символы пустого пространства") игнорируются; они рассматриваются компилятором только как разделители лексем. Такие символы необходимы, чтобы отделить друг от друга соседние идентификаторы, ключевые слова и константы.

Если входной поток уже разбит на лексемы до некоторого заданного символа, то следующей лексемой будет самая длинная строка символов, которые могут составлять лексему.

A.2.2. Комментарии

Комментарий начинается с символов `/*`, а заканчивается символами `*/`. Комментарии нельзя вкладывать друг в друга, а также помещать внутри строковых или символьных литеральных констант.

A.2.3. Идентификаторы

Идентификатор — это последовательность букв и цифр. Первым символом должна быть буква; знак подчеркивания (_) считается буквой. Буквы нижнего и верхнего регистров различаются. Идентификаторы могут иметь любую длину; для внутренних идентификаторов значимыми являются не менее 31 первого символа; в некоторых реализациях принято большее количество значимых символов. К внутренним идентификаторам относятся имена препроцессорных макросов и все остальные имена, не имеющие внешних связей (раздел A.11.2). На идентификаторы с внешними связями могут накладываться более строгие ограничения: в конкретных реализациях языка могут восприниматься не более шести первых символов, а также не различаться буквы верхнего и нижнего регистров.

A.2.4. Ключевые слова

Следующие идентификаторы зарезервированы в качестве ключевых слов и никаким другим способом использоваться не могут:

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

В некоторых реализациях зарезервированы также слова `fortran` и `asm`.

> Ключевые слова `const`, `signed` и `volatile` впервые появились в стандарте ANSI; `enum` и `void` — не использовались в первом издании книги, но уже вошли в употребление; ранее зарезервированное слово `entry` нигде не использовалось и поэтому больше не является ключевым словом.

A.2.5. Константы

Существует несколько видов констант. Каждая имеет свой тип данных; базовые типы рассматриваются в разделе A.4.2.

константа:
    целочисленная-константа
    символьная-константа
    константа-с-плавающей-точкой
    константа-перечислимого-типа

A.2.5.1. Целочисленные константы

Целочисленная константа, состоящая из последовательности цифр, воспринимается как восьмеричная, если она начинается с 0 (цифры "нуль"), и как десятичная в противном случае. Восьмеричные константы не содержат цифр 8 и 9. Последовательность цифр, перед которой стоит `0x` или `0X` (также цифра "нуль"), считается шестнадцатеричным целым числом. В число шестнадцатеричных цифр включены буквы от `a` (или `A`) до `f` (или `F`) со значениями соответственно от 10 до 15.

Целочисленная константа может записываться с суффиксом `u` (или `U`); в этом случае она считается константой без знака. Она также может иметь суффикс `l` (или `L`), указывающий на тип `long`.

Тип целочисленной константы зависит от формы ее записи, значения и суффикса. (Типы подробнее освещены в разделе A.4.) Если константа — десятичная и без суффикса, то она будет иметь первый из следующих типов, которого достаточно для представления ее значения: `int`, `long int`, `unsigned long int`. Восьмеричная или шестнадцатеричная константа без суффикса принимает первый подходящий из следующего списка типов: `int`, `unsigned int`, `long int`, `unsigned long int`. Если константа имеет суффикс `u` или `U`, она принимает первый подходящий из типов `unsigned int` и `unsigned long int`. Если константа имеет суффикс `l` или `L`, то список допустимых типов состоит из `long int` и `unsigned long int`.

> Типы целочисленных констант были существенно доработаны и развиты по сравнению с первой редакцией языка, в которой большие целые числа имели просто тип `long`. Суффиксы `U` введены впервые.

A.2.5.2. Символьные константы

Символьная константа — это последовательность из одного или нескольких символов, заключенная в одинарные кавычки (например, `'x'`). Значение символьной константы, состоящей из одного символа, равно числовому коду символа в символьном наборе, принятом в системе в момент выполнения программы. Значение константы из нескольких символов зависит от реализации языка.

Символьные константы не могут содержать одинарную кавычку (`'`) или символ конца строки. Чтобы изобразить их и некоторые другие символы, используются *управляющие последовательности* или *специальные символы* (*escape sequences*).

Конец строки	NL (LF)	\n
Горизонтальная табуляция	HT	\t
Вертикальная табуляция	VT	\v

Возврат на один символ назад	BS	\b
Возврат каретки	CR	\r
Прогон страницы	FF	\f
Звуковой сигнал	BEL	\a
Обратная косая черта	\	\\
Вопросительный знак	?	\?
Одинарная кавычка	'	\'
Двойная кавычка	"	\"
Восьмеричный код символа	*ooo*	*ooo*
Шестнадцатеричный код символа	*hh*	\x*hh*

Управляющий символ *ooo* начинается с обратной косой черты, за которой следуют одна, две или три восьмеричные цифры, воспринимаемые как код символа. Самым распространенным примером такой конструкции является \0 (без дальнейших цифр); она обозначает символ NULL. Управляющая последовательность \x*hh* состоит из обратной косой черты с буквой x, за которыми следуют шестнадцатеричные цифры, воспринимаемые как код символа. Количество цифр формально не ограничено, но результат будет не определен, если код получившегося символа превысит самый большой из допустимых символов. Если в данной реализации тип char считается числом со знаком, то значения и в восьмеричных, и в шестнадцатеричных управляющих последовательностях получаются путем расширения знака, как если бы выполнялась операция приведения к типу char. Если после обратной косой черты не следует ни один из перечисленных выше символов, результат будет не определен.

В некоторых реализациях имеется расширенный набор символов, который не может быть представлен только типом char. Константа для такого набора записывается с буквой L впереди (например, L'x') и называется расширенной символьной константой. Такая константа имеет целочисленный тип wchar_t, определенный в стандартном заголовочном файле <stddef.h>. Как и в случае обычных символьных констант, здесь также возможны восьмеричные и шестнадцатеричные специальные символы; если заданное кодовое значение выходит за пределы типа wchar_t, результат будет не определен.

> Некоторые из приведенных управляющих последовательностей (в частности, шестнадцатеричные) являются новыми для языка. Новым является и расширенный тип символов. Наборы символов, обычно используемые в США и Западной Европе, удобно кодировать типом char, тогда как тип wchar_t был введен главным образом для азиатских языков.

A.2.5.3. Вещественные константы с плавающей точкой

Вещественная константа с плавающей точкой состоит из целой части, десятичной точки, дробной части, символа e или E, целого показателя степени с необязательным знаком и необязательного суффикса типа — одной из букв f, F, l или L. Целая и дробная части состоят из последовательностей цифр. Может отсутствовать либо целая, либо дробная часть (но не обе сразу). Тип определяется суффиксом: F или f обозначают тип float, L или l — тип long double. При отсутствии суффикса принимается тип double.

> Суффиксы вещественных констант введены в новом стандарте языка; первоначально они не существовали.

A.2.5.4. Константы перечислимых типов

Идентификаторы, объявленные в составе перечислимого типа (см. раздел A.8.4), являются константами типа `int`.

A.2.6. Строковые литералы (константы)

Строковый литерал, который также называется строковой константой, — это последовательность символов, заключенная в двойные кавычки (`" . . . "`). Такая строка имеет тип "массив символов" и класс памяти `static` (раздел A.4) и инициализируется заданными символами. Представляются ли одинаковые строковые литералы одним фактическим экземпляром строки в памяти или несколькими, зависит от реализации языка. Результат работы программы, пытающейся изменить строковый литерал, не определен.

Написанные слитно строковые литералы сцепляются (конкатенируются) в одну строку. После любой конкатенации к строке добавляется нулевой байт (`\0`), что позволяет программам, просматривающим строку, находить ее конец. Строковые литералы не могут содержать символ конца строки или двойную кавычку; для представления таких символов нужно использовать те же управляющие последовательности, что и в символьных константах.

Как и в случае символьных констант, строковый литерал с символами из расширенного набора должен начинаться с буквы `L` (`L" . . . "`). Строковый литерал из расширенного набора имеет тип "массив элементов `wchar_t`". Результат конкатенации обычных и расширенных строковых литералов друг с другом не определен.

> То, что строковые литералы не обязательно представляются разными экземплярами в памяти, запрет на их модификацию, а также конкатенация слитно записанных строковых литералов — нововведения стандарта ANSI. Расширенные строковые литералы также введены впервые.

A.3. Система синтаксических обозначений

В системе формальной записи синтаксиса, используемой в этом справочнике, синтаксические категории набираются *курсивом*, а слова и символы, читаемые буквально, — моноширинным шрифтом. Альтернативные конструкции обычно перечисляются в столбик, каждый вариант — в отдельной строке; в ряде случаев длинный набор близкородственных альтернативных вариантов располагается в одной строке, помеченной словами "один из". Необязательный терминальный (основной) или нетерминальный (вспомогательный) символ грамматики помечается индексом "*необ*". Так, следующая запись обозначает выражение, заключенное в фигурные скобки, которое в общем случае может и отсутствовать:

{ *выражение*$_{необ}$ }

Формализованное описание синтаксиса приведено в разделе A.13.

> В отличие от грамматики, приведенной в первом издании этой книги, данная здесь грамматика явно описывает приоритет и ассоциативность операций в выражениях.

A.4. Употребление идентификаторов

Идентификаторы, или имена, обозначают различные компоненты программы: функции; метки структур, объединений и перечислений; элементы структур или объединений; имена новых типов в `typedef`; объекты. Объектом (называемым иногда переменной) является участок памяти, интерпретация которого в программе зависит от двух главных характеристик: ее *класса памяти* и *типа*. Класс памяти определяет время жизни памяти, ассоциированной с обозначаемым объектом, а тип определяет, какой смысл вкладывается в данные, находящихся в объекте. С именем также ассоциируются своя область видимости или действия (т.е. тот участок программы, где это имя известно) и способ связывания, определяющий, обозначает ли это имя тот же самый объект или функцию в другой области действия. Области действия и способы связывания рассматриваются в разделе A.11.

A.4.1. Классы памяти

Существуют два класса памяти: автоматический и статический. Класс памяти объекта задается рядом ключевых слов в совокупности с контекстом объявления этого объекта. Автоматические объекты локальны в блоке (раздел A.9.3) и при выходе из него уничтожаются. Объявления внутри блока создают автоматические объекты, если в них отсутствуют указания на класс памяти или указан модификатор `auto`. Объекты, объявленные с ключевым словом `register`, являются автоматическими и размещаются (по возможности) в быстро доступных регистрах процессора.

Статические объекты могут быть локальными в блоке или внешними по отношению ко всем блокам, но в обоих случаях их значения сохраняются после выхода из блока или функции до повторного входа. Внутри блока, в том числе и в теле функции, статические объекты объявляются с ключевым словом `static`. Объекты, объявляемые вне блоков на одном уровне с определениями функций, всегда являются статическими. С помощью ключевого слова `static` их можно сделать локальными в пределах единицы трансляции; в этом случае для них устанавливается *внутреннее связывание*. Они становятся глобальными для всей программы, если опустить явное указание класса памяти или использовать ключевое слово `extern`; в этом случае для них устанавливается *внешнее связывание*.

A.4.2. Базовые типы

Существует несколько базовых типов. Стандартный заголовочный файл `<limits.h>`, описанный в приложении Б, содержит определения самых больших и самых малых значений для каждого типа в данной конкретной реализации языка. В приложении Б приведены минимально допустимые величины.

Размер объектов, объявляемых как символы (`char`), позволяет хранить любой символ из символьного набора, принятого в системе во время выполнения программы. Если в объекте типа `char` хранится действительно символ из данного набора, то его значение эквивалентно коду этого символа и неотрицательно. Переменные типа `char` могут содержать и другие значения, но тогда диапазон их значений и особенно вопрос о том, имеют ли эти значения знак, определяется конкретной реализацией языка.

Символы без знака, объявленные с помощью ключевых слов `unsigned char`, занимают столько же памяти, сколько и обычные символы, но всегда неотрицательны. Аналогично, с помощью ключевых слов `signed char` можно явно объявить символы со знаком, которые занимают столько же места, сколько и обычные символы.

> Тип `unsigned char` отсутствовал в первой редакции этой книги, хотя давно вошел в широкий обиход. Тип `signed char` — новый.

Помимо `char`, разрешено реализовать еще три целочисленных типа трех различных размеров: `short int`, `int` и `long int`. Обычные объекты типа `int` имеют естественный размер, принятый в архитектуре конкретной системы, другие размеры предназначены для специальных целей. Более длинные целые типы имеют как минимум не меньшую длину в памяти, чем более короткие, однако в некоторых реализациях обычные целые типы могут быть эквивалентны коротким (`short`) или длинным (`long`). Все типы `int` по умолчанию представляют числа со знаком, если не оговорено обратное.

Целые числа без знака объявляются с помощью ключевого слова `unsigned` и подчиняются правилу автоматического взятия остатка от деления 2^n, где n — количество битов в представлении числа. Следовательно, в арифметике целых чисел без знака никогда не бывает переполнения. Множество неотрицательных значений, которые могут храниться в объектах со знаком, является подмножеством значений, которые могут храниться в соответствующих объектах без знака; представления таких чисел со знаком и без знака совпадают.

Любые два из вещественных типов с плавающей точкой — с одинарной (`float`), с двойной (`double`) и с повышенной (`long double`) точностью — могут быть синонимичными (обозначать одно и то же), но каждый следующий тип в этом списке должен обеспечивать точность по крайней мере не хуже предыдущего.

> Тип `long double` введен в новом стандарте. В первой редакции синонимом для `double` был тип `long float`, который теперь изъят из обращения.

Перечисления или *перечислимые типы* (*enumerations*) — единственные в своем роде целочисленные типы; с каждым перечислением связывается набор именованных констант (раздел А.8.4). Перечисления по свойствам аналогичны целочисленным типам, но компилятор обычно выдает предупреждение, если объекту некоторого перечислимого типа присваивается значение, отличное от одной из его констант, или выражение его же типа.

Поскольку объекты этих типов можно воспринимать как числа, такие типы будем называть *арифметическими*. Типы `char` и `int` всех размеров, каждый из которых может иметь или не иметь знака, а также перечислимые типы собирательно называют *целочисленными* (*integral*) типами. Типы `float`, `double` и `long double` называются *вещественными* или *типами с плавающей точкой* (*floating types*).

Тип `void` обозначает пустое множество значений. Он используется как тип "возвращаемого" функцией значения в том случае, когда функция не возвращает ничего.

А.4.3. Производные типы

Наряду с базовыми типами существует практически бесконечный класс производных типов, конструируемых из базовых типов следующими способами:

- как *массивы* объектов определенного типа;

- как *функции*, возвращающие объекты определенного типа;
- как *указатели* на объекты определенного типа;
- как *структуры*, содержащие последовательности объектов различных типов;
- как *объединения*, способные содержать один объект из заданного множества нескольких объектов разных типов.

В общем случае эти методы создания новых объектов могут применяться последовательно, в сочетаниях друг с другом.

A.4.4. Модификаторы типов

Тип объекта может дополняться *модификатором* (*qualifier*). Объявление объекта с модификатором `const` указывает на то, что его значение далее не будет изменяться; объявление объекта как `volatile` указывает на его особые свойства для выполняемой компилятором оптимизации. Ни один из модификаторов не влияет на диапазоны значений и арифметические свойства объектов. Модификаторы рассматриваются более подробно в разделе A.8.2.

A.5. Объекты и именующие выражения

Объект — это именованная область памяти; *именующее выражение* (*lvalue*) — это выражение, обозначающее объект или ссылающееся на него. Очевидным примером именующего выражения является идентификатор с соответствующим типом и классом памяти. Существуют операции, которые выдают именующее выражение в качестве результата. Например, если `E` — выражение типа "указатель", то `*E` — именующее выражение для объекта, на который указывает `E`. Термин *lvalue* образован от *leftside value* ("левостороннее значение") в контексте записи присваивания `E1 = E2`, в которой левый операнд `E1` должен быть как раз именующим выражением, чтобы присваивание было допустимым. При описании тех или иных операций мы сообщаем, требуют ли они именующих выражений в качестве операндов и выдают ли они такое выражение в качестве результата.

A.6. Преобразования типов

Некоторые операции в зависимости от своих операндов могут инициировать преобразование значений из одного типа в другой. В этом разделе рассказывается, каких результатов следует ожидать от таких преобразований. В разделе A.6.5 формулируются правила преобразований для большинства обычных операций; эти правила дополняются и конкретизируются при рассмотрении каждой отдельной операции.

A.6.1. Расширение целочисленных типов

Символ, короткое целое число, целочисленное битовое поле, а также объект перечислимого типа — все они, со знаком или без, могут использоваться в выражениях везде, где разрешено применение целых чисел. Если тип `int` позволяет представить все значения исходного типа операнда, то операнд приводится к `int`, в противном случае — к `unsigned int`. Эта процедура называется *расширением целочисленного типа* (*integral promotion*).

A.6.2. Преобразование целочисленных типов

Чтобы привести целое число к некоторому заданному типу без знака, ищется конгруэнтное (т.е. имеющее то же двоичное представление) наименьшее неотрицательное значение, а затем получается остаток от деления его на `UMAX+1`, где `UMAX` — наибольшее число в данном типе без знака. При использовании двоичного представления в дополнительном коде ("дополнения к двойке") для этого либо отбрасываются лишние левые разряды, если двоичное представление беззнакового типа уступает в длине исходному типу, либо недостающие старшие разряды заполняются нулями (в числах без знака) или значением знака (в числах со знаком), если тип без знака шире исходного.

В результате приведения целого числа к знаковому типу его значение не меняется, если оно представимо в этом новом типе; в противном случае результат зависит от реализации.

A.6.3. Преобразование целых чисел в вещественные и наоборот

При преобразовании из вещественного типа в целочисленный дробная часть числа отбрасывается; если полученное при этом значение нельзя представить величиной заданного целочисленного типа, то результат не определен. В частности, не определен результат преобразования отрицательных чисел с плавающей точкой в целые числа без знака.

Если число преобразуется из целого в вещественное и находится в допустимом диапазоне, но представляется в новом типе недостаточно точно, то результатом будет большее или меньшее ближайшее значение нового типа. Если результат выходит за границы диапазона допустимых значений, результат не определен.

A.6.4. Вещественные типы

При преобразовании из вещественного типа меньшей точности к типу большей точности число не изменяется. Если же преобразование выполняется от большей точности к меньшей и число остается в допустимых пределах нового типа, то результатом будет большее или меньшее ближайшее значение нового типа. Если результат выходит за границы допустимого диапазона, результат не определен.

A.6.5. Арифметические преобразования

Во многих операциях преобразование типов операндов и определение типа результата следуют одной и той же схеме. В результате операнды приводятся к некоторому общему типу, который также является и типом результата. Эта схема включает в себя *обычные арифметические преобразования*.

- Во-первых, если какой-либо из операндов имеет тип `long double`, то и другой приводится к `long double`.
- Иначе, если какой-либо из операндов имеет тип `double`, то и другой приводится к `double`.
- Иначе, если какой-либо из операндов имеет тип `float`, то и другой приводится к `float`.
- Иначе, для обоих операндов выполняется расширение целого типа; затем, если один из операндов имеет тип `unsigned long int`, то и другой преобразуется в `unsigned long int`.
- Иначе, если один из операндов имеет тип `long int`, а другой — `unsigned int`, то результат зависит от того, представляет ли `long int` все значения `unsigned int`; если это так, то операнд типа `unsigned int` приводится к `long int`; если нет, то оба операнда преобразуются в `unsigned long int`.
- Иначе, если один из операндов имеет тип `long int`, то и другой приводится к `long int`.
- Иначе, если один из операндов — `unsigned int`, то и другой приводится к `unsigned int`.
- Иначе, оба операнда имеют тип `int`.

> В эту часть языка внесены два изменения. Во-первых, арифметические операции с операндами типа `float` теперь могут выполняться с одинарной точностью, а не только с двойной; в первой редакции языка вся арифметика с плавающей точкой имела двойную точность. Во-вторых, более короткий тип без знака в комбинации с более длинным знаковым типом не распространяет свойство отсутствия знака на тип результата; в первой редакции типы без знака всегда доминировали. Новые правила немного сложнее, но до некоторой степени уменьшают вероятность неожиданных эффектов в комбинациях знаковых и беззнаковых величин. Однако неожиданный результат все же может получиться при сравнении беззнакового выражения со знаковым того же размера.

A.6.6. Связь указателей и целых чисел

К указателю можно прибавлять (и вычитать из него) выражение целочисленного типа; последнее в этом случае преобразуется так, как описано в разделе A.7.7 при рассмотрении операции сложения.

Можно вычитать два указателя на объекты одного типа, принадлежащие к одному массиву; результат приводится к целому числу так, как описано в разделе A.7.7 при рассмотрении операции вычитания.

Целочисленное константное выражение со значением 0 или такое же выражение, приведенное к типу `void *`, можно преобразовать в указатель любого типа операциями

приведения, присваивания и сравнения. Результатом будет нулевой указатель NULL, который равен любому другому нулевому указателю того же типа, но не равен никакому указателю на функцию или объект.

Допускаются и некоторые другие преобразования с участием указателей, но в них может присутствовать зависимость результата от реализации. Такие преобразования должны указываться явно — с помощью операции приведения типа (см. разделы A.7.5 и A.8.8).

Указатель можно привести к целочисленному типу, достаточно большому для его хранения; требуемый размер зависит от реализации. Функция отображения из множества целых чисел в множество указателей также зависит от реализации.

Объект целочисленного типа можно явно преобразовать в указатель. Если целое число получено из указателя и имеет достаточно большой размер, такое преобразование всегда даст тот же указатель; в противном случае результат зависит от реализации.

Указатель на один тип можно преобразовать в указатель на другой тип. Если исходный указатель не ссылается на объект, должным образом выровненный по границам слов памяти, в результате может возникнуть исключительная ситуация, связанная с адресацией. Если требования к выравниванию у нового типа менее строгие или такие же, как у первоначального типа, то гарантируется, что преобразование указателя к другому типу и обратно не изменит его. Само понятие "выравнивания" зависит от реализации, однако в любой реализации объекты типа `char` предъявляют наименее строгие требования к выравниванию. Как описано в разделе A.6.8, указатель может также преобразовываться в `void *` и обратно без изменения своего значения.

Указатель можно преобразовать в другой указатель того же типа с добавлением или удалением модификаторов (разделы A.4.4, A.8.2) того типа объектов, на которые он указывает. Новый указатель, полученный путем добавления модификатора, имеет то же значение, но с дополнительными ограничениями, внесенными новыми модификаторами. Удаление модификатора у объекта приводит к тому, что восстанавливается действие его первоначальных модификаторов, заданных в объявлении этого объекта.

Наконец, указатель на функцию можно преобразовать в указатель на функцию другого типа. Вызов функции по преобразованному указателю зависит от реализации; но если указатель снова преобразовать к его исходному типу, результат будет идентичен вызову по первоначальному указателю.

A.6.7. Тип `void`

Значение (несуществующее) объекта типа `void` никак нельзя использовать, его также нельзя явно или неявно привести к типу, отличному от `void`. Поскольку выражение типа `void` обозначает отсутствие значения, его можно применять только там, где не требуется значения: например, в качестве выражения-оператора (раздел A.9.2) или левого операнда операции "запятая" (раздел A.7.18).

Выражение можно преобразовать в тип `void` операцией приведения типа. Например, операция приведения к `void` применительно к вызову функции, используемому в роли выражения-оператора, подчеркивает тот факт, что результат функции отбрасывается.

> Тип `void` отсутствовал в первой редакции этой книги, однако за прошедшее время стал общеупотребительным.

A.6.8. Указатели на `void`

Любой указатель на объект можно привести к типу `void *` без потери информации. Если результат подвергнуть обратному преобразованию, получится исходный указатель. В отличие от преобразований "указатель в указатель", рассмотренных в разделе A.6.6, которые требуют явного приведения к типу, указатели типа `void *` можно употреблять совместно с указателями любого типа в операциях присваивания и сравнения каким угодно образом.

> Такая интерпретация указателей `void *` введена в новом стандарте; ранее роль нетипизированного указателя отводилась указателю типа `char *`. Стандарт ANSI подчеркивает допустимость использования указателей `void *` совместно с указателями других типов в операциях присваивания и сравнения; в других сочетаниях указателей стандарт требует явных преобразований типа.

A.7. Выражения

Приоритет знаков операций в языке C совпадает с порядком, в котором следуют подразделы данного раздела: вначале идут подразделы операций с более высоким приоритетом. Например, выражения, являющиеся операндами операции + (раздел A.7.7), определены и описаны в разделах A.7.1–A.7.6. В пределах каждого подраздела операции имеют одинаковый приоритет. В каждом разделе для описываемых операций указывается ассоциативность (левая или правая). Описание грамматики в разделе A.13 также включает информацию о приоритетах и ассоциативности операций.

Приоритет операций и их ассоциативность указаны здесь полностью, однако порядок вычисления выражений не определен, за некоторыми исключениями, даже для подвыражений, дающих побочные эффекты. Это значит, что если в определении операции специально не оговаривается та или иная последовательность вычисления ее операндов, то в реализации можно выбирать любой порядок вычислений по желанию и даже чередовать его. Однако любая операция комбинирует значения своих операндов в соответствии с синтаксической структурой выражения, в котором она встречается.

> Это правило отменяет ранее имевшуюся свободу в порядке выполнения операций, которые математически коммутативны и ассоциативны, но которые с вычислительной точки зрения могут и не оказаться ассоциативными. Это изменение затрагивает только вычисления вещественной арифметики, выполняющиеся на пределе точности, и ситуации, когда возможно переполнение.

В языке не определена обработка переполнения, деления на нуль и других исключительных ситуаций, возникающих при вычислении выражений. В большинстве существующих реализаций C переполнение игнорируется при вычислении целочисленных выражений со знаком и присваивании, но в целом такое поведение выполняющей системы стандартом не гарантируется. Обработка деления на нуль и всех исключительных ситуаций вещественной арифметики может отличаться в разных реализациях; иногда для этой цели существует нестандартная библиотечная функция.

A.7.1. Генерирование указателей

Если выражение или подвыражение имеет тип "массив элементов *T*", где *T* — некоторый тип, то значением этого выражения будет указатель на первый элемент массива, и тип такого выражения заменяется на тип "указатель на *T*". Замена типа не выполняется, если выражение является операндом одноместной операции `&`, либо операндом операций `++`, `--`, `sizeof`, либо левым операндом присваивания или операции "точка" (`.`). Аналогично, выражение типа "функция, возвращающая *T*", кроме случая использования в качестве операнда `&`, преобразуется в тип "указатель на функцию, возвращающую *T*".

A.7.2. Первичные выражения

Первичные выражения — это идентификаторы, константы, строки и выражения в скобках.

```
первичное - выражение:
    идентификатор
    константа
    строка
    ( выражение )
```

Идентификатор является первичным выражением, если он был должным образом объявлен; как это делается, рассказано ниже. Тип идентификатора указывается в его объявлении. Идентификатор является именующим выражением (*lvalue*), если он обозначает объект (раздел A.5) и имеет арифметический тип либо тип "структура", "объединение" или "указатель".

Константа — первичное выражение. Ее тип зависит от формы записи, рассмотренной в разделе A.2.5.

Строковый литерал — первичное выражение. Его базовый тип — "массив элементов типа `char`" ("массив элементов типа `wchar_t`" для строк символов расширенного набора), но в соответствии с правилом, приведенным в разделе A.7.1, этот тип обычно преобразуется в "указатель на `char`" (`wchar_t`) и в результате фактически указывает на первый символ строки. Для некоторых инициализаторов такая замена типа не выполняется (см. раздел A.8.7).

Выражение в скобках — первичное выражение, тип и значение которого совпадают с типом и значением этого же выражения без скобок. Наличие или отсутствие скобок не влияет на то, является ли данное выражение именующим (*lvalue*) или нет.

A.7.3. Постфиксные выражения

В постфиксных выражениях операции группируются слева направо.

```
постфиксное-выражение:
    первичное-выражение
    постфиксное-выражение [ выражение ]
    постфиксное-выражение ( список-аргументов-выражений_необ )
    постфиксное-выражение.идентификатор
    постфиксное-выражение->идентификатор
    постфиксное-выражение ++
    постфиксное-выражение --
```

```
список-аргументов-выражений:
    выражение-присваивание
    список-аргументов-выражений , выражение-присваивание
```

A.7.3.1. Обращение к элементам массивов

Постфиксное выражение, за которым следует выражение в квадратных скобках, представляет собой постфиксное выражение, обозначающее обращение к индексируемому массиву. Одно из этих двух выражений должно иметь тип "указатель на *T*", где *T* — некоторый тип, а другое — целочисленный тип; результат обращения по индексу будет иметь тип *T*. Выражение `E1[E2]` по определению идентично выражению `*((E1)+(E2))`. Подробнее об этом написано в разделе A.8.6.2.

A.7.3.2. Вызовы функций

Вызов функции — это постфиксное выражение, которое также называется *именующим обозначением функции* (*function designator*), за которым следуют круглые скобки со списком (возможно, пустым) разделенных запятыми выражений-присваиваний (см. раздел A.7.17), которые представляют собой аргументы этой функции. Если такое постфиксное выражение представляет собой идентификатор, не объявленный в текущей области действия, то считается, что этот идентификатор объявлен неявно — так, как если бы в самом внутреннем блоке, содержащем вызов функции, находилось бы объявление

```
extern int идентификатор();
```

Постфиксное выражение (после возможного неявного объявления и генерирования указателя; см. раздел A.7.1) должно иметь тип "указатель на функцию, возвращающую *T*", где *T* — тип возвращаемого функцией значения.

> В первой версии языка для именующего обозначения функции допускался только тип "функция", и, чтобы вызвать функцию через указатель на нее, требовалось явно писать знак `*`. Стандарт ANSI утвердил практику ряда существующих компиляторов, которые разрешают иметь одинаковый синтаксис для обращения к функции непосредственно и через указатель. Возможность применения старого синтаксиса также остается в силе.

Термин *аргумент* используется для выражения, передаваемого при вызове функции; термин *параметр* — для обозначения получаемого ею объекта (или его идентификатора) в объявлении или определении функции. В том же смысле иногда используются термины "фактический аргумент (аргумент)" и "формальный аргумент (параметр)".

При подготовке вызова функции создаются копии всех ее аргументов; передача аргументов выполняется строго по значениям. Функции разрешается изменять значения своих параметров, которые являются только копиями аргументов-выражений, но эти изменения не могут повлиять на значения самих аргументов. Однако в функцию можно передать указатель, чтобы сознательно позволить ей изменить значение объекта, на который указывает этот указатель.

Имеются два способа (стиля) объявления функций. В новом стиле типы параметров задаются явно и являются частью типа функции; такое объявление еще называется *прототипом* функции. В старом стиле типы параметров не указываются. Способы объявления функций рассматриваются в разделах A.8.6.3 и A.10.1.

Если объявление функции составлено в старом стиле, то при вызове этой функции в области действия объявления каждый аргумент подвергается расширению типа: целочисленные аргументы преобразуются по правилам расширения целых типов

(раздел A.6.1), а аргументы типа `float` — в `double`. Если количество аргументов не соответствует количеству параметров в определении функции или если тип какого-либо аргумента после расширения не согласуется с типом соответствующего параметра, результат вызова будет не определен. Понятие согласованности типов зависит от стиля определения функции (старого или нового). При старом стиле сравнивается расширенный тип аргумента в вызове и расширенный тип соответствующего параметра; при новом стиле расширенный тип аргумента должен совпасть с типом самого параметра без расширения.

Если объявление функции составлено в новом стиле, то при вызове функции в области действия объявления аргументы преобразуются так, как при присваивании — с приведением к типу соответствующих параметров прототипа. Количество аргументов должно совпадать с количеством явно заданных параметров, только если список параметров не заканчивается многоточием (`, ...`). В последнем случае число аргументов должно быть больше числа параметров или равно ему; аргументы на месте многоточия подвергаются автоматическому расширению типа таким образом, как описано в предыдущем абзаце. Если определение функции составлено в старом стиле, то тип каждого параметра в прототипе, в область действия которого входит вызов функции, должен соответствовать типу соответствующего параметра в определении функции после его расширения.

> Эти правила сильно усложнились из-за того, что они призваны обслуживать смешанный стиль объявления и определения функций. По возможности такого стиля следует избегать.

Порядок вычисления аргументов не определяется стандартом; в разных компиляторах он может отличаться. Однако гарантируется, что аргументы и именующее обозначение функции вычисляются полностью (включая и побочные эффекты) до входа в нее. Любая функция допускает рекурсивный вызов.

A.7.3.3. Обращение к структурам

Постфиксное выражение, после которого стоит точка с последующим идентификатором, является постфиксным выражением. Первый операнд (в том числе как результат вычисления выражения) должен быть структурой или объединением, а идентификатор — именем элемента (поля) структуры или объединения. Значение этой конструкции представляет собой именованный элемент структуры или объединения, а ее тип — тип элемента структуры или объединения. Выражение является именующим (*lvalue*), если первое выражение — именующее, а тип второго выражения — не массив.

Постфиксное выражение, после которого стоит стрелка (составленная из знаков - и >) с последующим идентификатором, также является постфиксным выражением. Первый операнд (в том числе как результат вычисления выражения) должен быть указателем на структуру или объединение, а идентификатор — именем элемента структуры или объединения. Результат является именованным элементом структуры (объединения), на которую указывает указатель, а его тип будет типом элемента структуры (объединения). Результат представляет собой именующее выражение (*lvalue*), если тип элемента — не массив.

Таким образом, выражение `E1->MOS` означает то же самое, что и выражение `(*E1).MOS`. Структуры и объединения рассматриваются в разделе A.8.3.

> В первом издании книги уже приводилось правило, по которому имя элемента должно принадлежать структуре или объединению, упомянутому в постфиксном выражении. Правда, там оговаривалось, что это правило не является строго обязательным. Новейшие компиляторы и стандарт ANSI сделали правило обязательным.

A.7.3.4. Постфиксное инкрементирование

Постфиксное выражение, за которым следует знак ++ или --, также представляет собой постфиксное выражение. Значением такого выражения является значение его операнда. После того как его значение было использовано, операнд увеличивается (++) или уменьшается (--) на единицу. Операнд должен быть именующим выражением (*lvalue*); об ограничениях, накладываемых на операнд, и деталях данных операций речь пойдет в разделе A.7.7, посвященном аддитивным операциям, и в разделе A.7.17, где рассматривается присваивание. Сам результат инкрементирования или декрементирования не является именующим выражением.

A.7.4. Одноместные операции

Выражения с одноместными операциями группируются справа налево.

```
одноместное-выражение:
    постфиксное-выражение
    ++ одноместное-выражение
    -- одноместное-выражение
    одноместная-операция выражение-приведения-к-типу
    sizeof одноместное-выражение
    sizeof (имя-типа)

одноместная-операция: один из
    &   *   +   -   ~   !
```

A.7.4.1. Префиксные операции инкрементирования

Одноместное выражение, перед которым стоит знак ++ или --, также является одноместным выражением. Операция увеличивает (++) или уменьшает (--) свой операнд на единицу. Значением выражения служит результат инкрементирования (или декрементирования). Операнд должен быть именующим выражением (*lvalue*); дополнительные ограничения на операнд и подробности выполнения операции см. в разделе A.7.7 об аддитивных операциях и в разделе A.7.17 о присваивании. Сам результат операции не является именующим выражением.

A.7.4.2. Операция взятия адреса

Знак одноместной операции (&) обозначает получение адреса своего операнда. Операнд должен быть либо именующим выражением (*lvalue*), не ссылающимся ни на битовое поле, ни на объект, объявленный как `register`, либо объектом типа "функция". Результат — это указатель на объект или функцию, который обозначается выражением-операндом. Если операнд имеет тип *T*, то типом результата является "указатель на *T*".

A.7.4.3. Операция разыменования (ссылки по указателю)

Знак одноместной операции (*) обозначает обращение по указателю (разыменование), дающее объект (в том числе функцию), на который указывает ее операнд. Результат является именующим выражением (*lvalue*), если операнд — указатель на объект арифметического типа, структуру, объединение или указатель. Если тип выражения — "указатель на *T*", то типом результата будет *T*.

A.7.4.4. Операция "одноместный плюс"

Операнд одноместной операции (+) должен иметь арифметический тип; операция дает в качестве результата значение операнда. Целочисленный операнд преобразуется по правилам расширения целочисленных типов. Типом результата является расширенный тип операнда.

Одноместный плюс был добавлен для симметрии с одноместным минусом.

A.7.4.5. Операция "одноместный минус"

Операнд для одноместного минуса должен иметь арифметический тип; результатом является значение операнда с противоположным знаком. Целочисленный операнд преобразуется по правилам расширения целочисленных типов. Отрицательное значение от числа без знака вычисляется путем вычитания приведенного к расширенному типу операнда из максимального числа этого расширенного типа плюс единица. Единственное исключение: "минус нуль" равен нулю. Типом результата является расширенный тип операнда.

A.7.4.6. Операция вычисления дополнения до единицы (поразрядного отрицания)

Операнд операции (~) должен иметь целочисленный тип; результатом является дополнение операнда до единиц по всем разрядам. Выполняется целочисленное расширение типа операнда. Если операнд не имеет знака, то результат получается путем вычитания его значения из самого большого числа расширенного типа. Если операнд имеет знак, то результат вычисляется путем приведения расширенного операнда к соответствующему типу без знака, применения операции ~ и обратного приведения его к типу со знаком. Тип результата — расширенный тип операнда.

A.7.4.7. Операция логического отрицания

Операнд операции (`!`) должен иметь арифметический тип или быть указателем. Результат операции равен `1`, если значение операнда равно нулю, и `0` в противном случае. Результат имеет тип `int`.

A.7.4.8. Операция вычисления размера `sizeof`

Операция `sizeof` вычисляет число байтов, требуемое для хранения объекта того типа, который имеет ее операнд. Операнд представляет собой либо выражение (которое не вычисляется), либо имя типа, записанное в скобках. Применение операции `sizeof` к типу `char` дает 1; для массива результат равняется общему количеству байтов в массиве. Размер структуры или объединения равен числу байтов в объекте, включая байты-заполнители, которые понадобились бы, если бы из объектов данного типа составлялся массив: дело в том, что размер массива из *n* элементов всегда равняется произведению *n* на размер отдельного его элемента. Данную операцию нельзя применять к операнду типа "функция", операнду неполного (заранее объявленного, но не полностью определенного) типа или битовому полю. Результат является беззнаковой целочисленной константой, а ее конкретный тип зависит от реализации. В стандартном заголовочном файле `<stddef.h>` (см. приложение Б) этот тип объявлен под именем `size_t`.

A.7.5. Приведение типов

Имя типа, записанное перед одноместным выражением в круглых скобках, вызывает приведение значения этого выражения к указанному типу.

```
выражение-приведения-к-типу:
    одноместное-выражение
    ( имя-типа ) выражение-приведения-к-типу
```

Данная конструкция называется *приведением типа* (*type cast*). Имена типов перечислены и описаны в разделе A.8.8. Эффект от приведений типов описан в разделе A.6. Выражение с приведением типа не является именующим (*lvalue*).

A.7.6. Мультипликативные операции

Мультипликативные операции `*`, `/` и `%` группируются слева направо.

```
мультипликативное-выражение:
    выражение-приведения-к-типу
    мультипликативное-выражение * выражение-приведения-к-типу
    мультипликативное-выражение / выражение-приведения-к-типу
    мультипликативное-выражение % выражение-приведения-к-типу
```

Операнды операций `*` и `/` должны быть арифметического типа, а операнды `%` — целочисленного. Над операндами выполняются обычные арифметические преобразования, которые определяют тип результата.

Двухместная операция `*` обозначает умножение.

Двухместная операция `/` дает частное, а `%` — остаток от деления первого операнда на второй; если второй операнд равен 0, то результат не определен. В противном случае выражение `(a / b) * b + a % b` равняется `a`. Если оба операнда неотрицательны, то остаток неотрицателен и меньше делителя; в противном случае гарантируется только то, что абсолютное значение остатка будет меньше абсолютного значения делителя.

A.7.7. Аддитивные операции

Аддитивные операции `+` и `-` группируются слева направо. Если операнды имеют арифметический тип, то выполняются обычные арифметические преобразования. Для каждой операции существует еще несколько дополнительных возможностей сочетания типов.

```
аддитивное-выражение:
    мультипликативное-выражение
    аддитивное-выражение + мультипликативное-выражение
    аддитивное-выражение - мультипликативное-выражение
```

Результатом выполнения операции `+` является сумма его операндов. Указатель на объект в массиве можно складывать со значением любого целочисленного типа. Последнее преобразуется в адресное смещение посредством умножения его на размер объекта, на который ссылается указатель. Сумма является указателем того же типа, что и исходный указатель, но ссылается на другой объект того же массива, отстоящий от исходного на вычисленное смещение. Так, если `P` — указатель на объект в массиве, то `P+1` — указатель на следующий объект того же массива. Если полученный в результате сложения

указатель указывает за пределы массива, то результат будет неопределенным — кроме случая, когда указатель ссылается на участок памяти, находящийся непосредственно за концом массива.

> Возможность указывать на элемент, расположенный сразу за концом массива, является новой. Тем самым узаконена ставшая уже стандартной конструкция для циклического перебора элементов массива.

Результатом выполнения операции - (минус) является разность операндов. Из указателя можно вычитать число или выражение любого целочисленного типа с теми же преобразованиями и при тех же условиях, что и в сложении.

Если к двум указателям на объекты одного и того же типа применить операцию вычитания, то в результате получится целочисленное значение со знаком, представляющее собой расстояние между указываемыми объектами; так, указатели на два соседних объекта отличаются на единицу. Тип результата зависит от реализации; в стандартном заголовочном файле `<stddef.h>` такой тип определен под именем `ptrdiff_t`. Значение не определено, если указатели не указывают на объекты одного и того же массива; однако если `P` указывает на последний элемент массива, то выражение `(P+1)-P` имеет значение 1.

A.7.8. Операции сдвига

Операции сдвига `<<` и `>>` группируются слева направо. Каждый операнд обеих операций должен иметь целочисленный тип, и каждый из них подвергается расширению типа. Результат имеет тот же тип, что и левый операнд после расширения. Результат не определен, если правый операнд отрицателен либо больше или равен количеству битов в типе левого выражения.

выражение-со-сдвигом:
    аддитивное-выражение
    выражение-со-сдвигом **<<** *аддитивное-выражение*
    выражение-со-сдвигом **>>** *аддитивное-выражение*

Значение `E1<<E2` равно значению `E1` (воспринимаемому как набор битов), сдвинутому влево на `E2` битов; при отсутствии переполнения такая операция эквивалентна умножению на 2^{E2}. Значение `E1>>E2` равно значению `E1`, сдвинутому вправо на `E2` битовых позиций. Если `E1` — величина без знака или неотрицательное значение, то сдвиг вправо эквивалентен делению на 2^{E2}; в противном случае результат зависит от реализации.

A.7.9. Операции отношения (сравнения)

Операции отношения группируются слева направо, но это свойство малополезно; согласно грамматике языка выражение `a < b < c` воспринимается как `(a < b) < c`, а результат вычисления `a < b` может быть равен только 0 или 1.

выражение-отношения:
    выражение-со-сдвигом
    выражение-отношения **<** *выражение-со-сдвигом*
    выражение-отношения **>** *выражение-со-сдвигом*
    выражение-отношения **<=** *выражение-со-сдвигом*
    выражение-отношения **>=** *выражение-со-сдвигом*

Операции < (меньше), > (больше), <= (меньше или равно) и >= (больше или равно) дают результат 0, если указанное отношение ложно, и 1, если оно истинно. Результат имеет тип `int`. Над арифметическими операндами выполняются обычные арифметические преобразования. Можно сравнивать указатели на объекты одного и того же типа (без учета модификаторов); результат будет зависеть от относительного расположения указываемых объектов в адресном пространстве. Определено только сравнение указателей на разные части одного и того же объекта:

- если два указателя указывают на один и тот же простой объект, то они равны;
- если они указывают на элементы одной структуры, то указатель на элемент с более поздним объявлением в структуре — больше;
- если указатели указывают на элементы одного и того же объединения, то они равны;
- если указатели указывают на элементы некоторого массива, то сравнение этих указателей эквивалентно сравнению соответствующих индексов.

Если `P` указывает на последний элемент массива, то `P+1` больше, чем `P`, хотя `P+1` указывает за границы массива. В остальных случаях результат сравнения не определен.

> Эти правила несколько ослабили ограничения, установленные в первой редакции языка, поскольку позволяют сравнивать указатели на различные элементы структуры и объединения. Они также узаконили сравнение с участием указателя на место, расположенное непосредственно за концом массива.

A.7.10. Операции проверки равенства

```
выражение-равенства:
    выражение-отношения
    выражение-равенства == выражение-отношения
    выражение-равенства != выражение-отношения
```

Операции == (равно) и != (не равно) аналогичны операциям отношения, но имеют более низкий приоритет. (Таким образом, выражение `a < b == c < d` равно 1 тогда и только тогда, когда отношения `a < b` и `c < d` одновременно истинны или ложны.)

Операции проверки равенства подчиняются тем же правилам, что и операторы отношения, но дают и дополнительные возможности: можно сравнить указатель с целочисленным константным выражением, значение которого равно нулю, или с указателем на `void` (см. раздел A.6.6).

A.7.11. Операция поразрядного И

```
выражение-с-И:
    выражение-равенства
    выражение-с-И & выражение-равенства
```

Выполняются обычные арифметические преобразования; результат представляет собой поразрядное (побитовое) логическое произведение (И) операндов. Операция применяется только к целочисленным операндам.

A.7.12. Операция поразрядного исключающего ИЛИ

выражение-с-исключающим-ИЛИ:
    выражение-с-И
    выражение-с-исключающим-ИЛИ ^ *выражение-с-И*

Выполняются обычные арифметические преобразования; результат получается поразрядным (побитовым) применением операции исключающего ИЛИ к операндам. Операция применяется только к целочисленным операндам.

A.7.13. Операция поразрядного включающего ИЛИ

выражение-с-включающим-ИЛИ:
    выражение-с-исключающим-ИЛИ
    выражение-с-включающим-ИЛИ | *выражение-с-исключающим-ИЛИ*

Выполняются обычные арифметические преобразования; результат получается поразрядным (побитовым) применением к операндам операции включающего ИЛИ. Операция применяется только к целочисленным операндам.

A.7.14. Операция логического И

выражение-с-логическим-И:
    выражение-с-включающим-ИЛИ
    выражение-с-логическим-И **&&** *выражение-с-включающим-ИЛИ*

Операции && группируются слева направо. Операция && дает результат 1, если оба операнда не равны нулю, и 0 в противном случае. В отличие от &, операция && гарантирует вычисление выражения слева направо: вначале вычисляется первый операнд со всеми побочными эффектами; если он равен 0, то значение выражения будет равно 0. В противном случае вычисляется правый операнд, и если он равен 0, то значение выражения будет равно нулю, в противном случае — единице.

Операнды могут иметь разные типы, но каждый из них должен быть либо величиной арифметического типа, либо указателем. Результат имеет тип `int`.

A.7.15. Операция логического ИЛИ

выражение-с-логическим-ИЛИ:
    выражение-с-логическим-И
    выражение-с-логическим-ИЛИ || *выражение-с-логическим-И*

Операции || группируются слева направо. Операция дает результат 1, если хотя бы один из операндов не равен нулю, и 0 в противном случае. В отличие от |, операция || гарантирует порядок вычислений слева направо: вначале вычисляется первый операнд со всеми побочными эффектами; если он не равен 0, то значение выражения будет равно 1.

В противном случае вычисляется правый операнд, и если он не равен 0, то значение выражения будет равно единице, в противном случае — нулю.

Операнды могут иметь разные типы, но каждый из них должен быть либо величиной арифметического типа, либо указателем. Результат имеет тип `int`.

A.7.16. Операция выбора по условию

условное-выражение:
    выражение-с-логическим-ИЛИ
    выражение-с-логическим-ИЛИ **?** *выражение* **:** *условное-выражение*

Вначале вычисляется первое выражение, включая все побочные эффекты; если оно не равно 0, то результатом будет значение второго выражения, в противном случае — значение третьего выражения. Вычисляется только один из двух последних операндов: второй или третий. Если второй и третий операнды — арифметические, то выполняются обычные арифметические преобразования, приводящие их к некоторому общему типу, который и будет типом результата. Если оба операнда имеют тип `void`, или являются структурами/объединениями одного и того же типа, или представляют собой указатели на объекты одного и того же типа, то результат будет иметь тот же тип, что и операнды. Если один из операндов имеет тип "указатель", а другой является константой 0, то 0 приводится к типу "указатель", и этот же тип будет иметь результат. Если один операнд является указателем на `void`, а второй — указателем другого типа, то последний преобразуется в указатель на `void`, который и будет типом результата.

При сравнении типов указателей модификаторы типов объектов (раздел A.8.2), на которые эти указатели ссылаются, во внимание не принимаются, но тип результата наследует модификаторы из обеих ветвей условного выражения.

A.7.17. Выражения с присваиванием

Существует несколько операций присваивания; все они группируются справа налево.

выражение-присваивания:
    условное-выражение
    одноместное-выражение знак-присваивания выражение-присваивания

знак-присваивания: один из
    = ***=** **/=** **%=** **+=** **-=** **<<=** **>>=** **&=** **^=** **|=**

Все операции присваивания требуют именующего выражения (*lvalue*) в качестве левого операнда, причем это выражение должно быть модифицируемым. Это значит, что оно не может быть массивом, или функцией, или иметь неполный тип. Кроме того, тип левого операнда не может иметь модификатора `const`; если он является структурой или объединением, то они и их вложенные структуры/объединения не должны содержать элементов с модификаторами `const`. Тип выражения присваивания совпадает с типом его левого операнда, а значение равно значению левого операнда после завершения присваивания.

В простом присваивании со знаком = значение выражения в правой части замещает объект, к которому обращается именующее выражение (*lvalue*). При этом должно выполняться одно из следующих условий:

- оба операнда имеют арифметический тип (правый операнд в процессе присваивания приводится к типу левого операнда);
- оба операнда являются структурами или объединениями одного и того же типа;
- один операнд — указатель, а другой — указатель на `void`;
- левый операнд — указатель, а правый — константное выражение со значением `0`;
- оба операнда — указатели на функции или объекты, имеющие одинаковый тип, за исключением возможного отсутствия `const` или `volatile` в типе правого операнда.

Выражение `E1` *`оп`*`= E2` эквивалентно выражению `E1 = E1` *`оп`* `(E2)` с той разницей, что `E1` вычисляется только один раз.

A.7.18. Операция "запятая"

выражение:
выражение-присваивания
выражение `,` *выражение-присваивания*

Два выражения, разделенные запятой, вычисляются слева направо, и значение левого выражения отбрасывается. Тип и значение результата совпадают с типом и значением правого операнда. Вычисление всех побочных эффектов левого операнда завершается перед началом вычисления правого операнда. В контекстах, в которых запятая имеет особое значение, например в списках аргументов функций (раздел A.7.3.2) или в списках инициализаторов (раздел A.8.7), где в качестве синтаксических единиц должны фигурировать выражения присваивания, оператор запятая может стоять только в группирующих круглых скобках. Например, в следующее выражение входят три аргумента, второй из которых имеет значение 5:

```
f(a, (t=3, t+2), c)
```

A.7.19. Константные выражения

С синтаксической точки зрения константное выражение — это выражение с ограниченным подмножеством операций:

константное-выражение:
условное-выражение

В ряде контекстов разрешаются только выражения, эквивалентные константам: после меток `case` в операторе `switch`, при задании границ массивов и длин битовых полей, в качестве значений перечислимых констант, в инициализаторах, а также в некоторых выражениях директив препроцессора.

Константные выражения не могут содержать присваивания, операции инкрементирования и декрементирования, вызовы функций и операции "запятая"; эти ограничения не распространяются на операнды операции `sizeof`. Если требуется целочисленное константное выражение, то его операнды должны состоять из целых, перечислимых, символьных и вещественных констант. Операции приведения должны преобразовывать величины только к целочисленным типам, а любая вещественная константа должна явно приводиться к целому типу. Из этого следует, что в константном выражении не может

быть массивов, операций обращения по указателю, взятия адреса и обращения к полям структуры. (Однако операция `sizeof` может иметь операнды любого вида.)

Для константных выражений в инициализаторах допускается большая свобода. Операндами могут быть константы любого типа, а к внешним или статическим объектам, а также к внешним и статическим массивам, индексируемым константными выражениями, можно применять одноместную операцию `&`. Одноместная операция `&` может также применяться неявно — при использовании массива без индекса или функции без списка аргументов. Вычисление инициализирующего значения должно давать константу либо адрес ранее объявленного внешнего или статического объекта плюс-минус константа.

Меньшая свобода допускается для целочисленных константных выражений в директиве `#if`: не разрешаются выражения с `sizeof`, константы перечислимых типов и операции приведения типа (см. раздел A.12.5).

A.8. Объявления

Объявление задает способ интерпретации идентификатора, но не обязательно резервирует память, ассоциируемую с этим идентификатором. Объявления, которые действительно резервируют память, называются *определениями*. Объявления имеют следующую форму:

объявление:
    спецификаторы-объявления список-иниц-описателей$_{необ}$

Описатели в *список-иниц-описателей* содержат объявляемые идентификаторы; *спецификаторы-объявления* представляют собой последовательности из спецификаторов типа и класса памяти.

спецификаторы-объявления:
    спецификатор-класса-памяти спецификаторы-объявления$_{необ}$
    спецификатор-типа спецификаторы-объявления$_{необ}$
    модификатор-типа спецификаторы-объявления$_{необ}$

список-иниц-описателей:
    иниц-описатель
    список-иниц-описателей **,** *иниц-описатель*

иниц-описатель:
    описатель
    описатель **=** *инициализатор*

Описатели будут рассмотрены ниже, в разделе A.8.5; они содержат объявляемые имена. Объявление должно либо содержать по крайней мере один описатель, либо же его спецификатор типа должен объявлять метку структуры/объединения или задавать элементы перечисления; пустое объявление недопустимо.

A.8.1. Спецификаторы класса памяти

Определение спецификатора класса памяти таково:

спецификатор-класса-памяти:
    auto

```
register
static
extern
typedef
```

Значение этих классов памяти рассматривалось в разделе А.4.

Спецификаторы `auto` и `register` присваивают объявляемым объектам автоматический класс памяти; эти спецификаторы можно применять только внутри функции. Объявления с `auto` и `register` одновременно являются определениями и резервируют память. Спецификатор `register` эквивалентен `auto`, но при этом подсказывает компилятору, что объявленные объекты используются в программе особенно часто. В регистрах можно фактически разместить лишь небольшое количество объектов, причем только определенного типа; конкретные ограничения зависят от реализации. К объекту, объявленному как `register`, нельзя применять одноместную операцию `&` как явно, так и неявно.

> Новым в стандарте является правило, согласно которому вычислять адрес объекта класса `register` нельзя, а класса `auto` — можно.

Спецификатор `static` присваивает объявляемым объектам статический класс памяти; он может использоваться и внутри, и вне функций. Внутри функции этот спецификатор инициирует выделение памяти для объекта и служит определением; его использование вне функций будет рассмотрено в разделе А.11.2.

Объявление со спецификатором `extern` внутри функции заявляет, что для объекта где-то в другом месте программы выделена память; в разделе А.11.2 говорится о том, как этот спецификатор работает вне функций.

Спецификатор `typedef` не резервирует никакой памяти и назван спецификатором класса памяти только для соблюдения стандартного синтаксиса; об этом спецификаторе будет сказано в разделе А.8.9.

Объявление может содержать не больше одного спецификатора класса памяти. Если спецификатор в объявлении отсутствует, то действуют следующие правила:

- объекты, объявленные внутри функций, имеют класс `auto`;
- функции, объявленные внутри функций, имеют класс `extern`;
- объекты и функции, объявляемые вне функций, считаются статическими с внешним связыванием (см. разделы А.10, А.11).

А.8.2. Спецификаторы типа

Спецификаторы типа имеют следующее формальное определение:

спецификатор-типа:

```
void
char
short
int
long
float
double
signed
unsigned
```

спецификатор-структуры-или-объединения
спецификатор-перечисления
типоопределяющее-имя

Вместе с `int` допускается использование еще одного из слов `long` или `short`; слово `int` при этом можно опускать без изменения смысла. Слово `long` может употребляться вместе с `double`. Со словами `char`, `int` и вариантами последнего (`short`, `long`) разрешается употреблять одно из слов `signed` или `unsigned`. Любое из них может использоваться и без `int`, которое в таком случае подразумевается по умолчанию. Спецификатор `signed` бывает полезен, когда требуется гарантировать наличие знака у объектов типа `char`. Его можно применять и к другим целочисленным типам, хотя в этом случае он совершенно лишний.

За исключением описанных выше случаев, объявление не может содержать больше одного спецификатора типа. Если в объявлении нет ни одного спецификатора типа, то подразумевается тип `int`.

Для указания особых свойств объявляемых объектов предназначаются такие модификаторы:

модификатор-типа:
`const`
`volatile`

Модификаторы типа могут употребляться с любым спецификатором типа. Разрешается инициализировать объект с модификатором `const`, однако присваивать ему что-либо впоследствии запрещается. Смысл модификатора `volatile` зависит исключительно от реализации; стандартной универсальной семантики у него нет.

> Свойства `const` и `volatile` введены стандартом ANSI. Модификатор `const` применяется для объявления объектов, размещаемых в памяти, открытой только для чтения, а также для расширения возможностей оптимизации. Назначение модификатора `volatile` — запретить оптимизацию, которая могла бы быть выполнена в противном случае. Например, в системах, где ресурсы устройств ввода-вывода отображены на адресное пространство оперативной памяти, указатель на регистр устройства можно объявить с ключевым словом `volatile`, чтобы запретить компилятору сокращать кажущиеся ему избыточными обращения через указатель. Компилятор может игнорировать указанные модификаторы, однако обязан диагностировать явные попытки изменения значений `const`-объектов.

A.8.3. Объявления структур и объединений

Структура — это объект, состоящий из последовательности именованных элементов различных типов. Объединение — это объект, который в каждый момент времени содержит один из нескольких элементов различных типов. Объявления структур и объединений имеют один и тот же вид.

спецификатор-структуры-или-объединения:
struct-или-union *идентификатор*$_{необ}$ { *список-объявлений-структуры* }
struct-или-union *идентификатор*

struct-или-union:
`struct`
`union`

Конструкция *список-объявлений-структуры* представляет собой последовательность объявлений элементов структуры или объединения:

```
список-объявлений-структуры:
    объявление-структуры
    список-объявлений-структуры объявление-структуры

объявление-структуры:
    список-спецификаторов-модификаторов список-описателей-структуры
;

список-спецификаторов-модификаторов:
    спецификатор-типа список-спецификаторов-модификаторов(необ)
    модификатор-типа список-спецификаторов-модификаторов(необ)

список-описателей-структуры:
    описатель-структуры
    список-описателей-структуры , описатель-структуры
```

Обычно *описатель-структуры* является просто описателем элемента структуры или объединения. Элемент структуры может также состоять из заданного количества битов. Такой элемент называется *битовым полем* или просто *полем*. Его длина отделяется от имени поля двоеточием.

```
описатель-структуры:
    описатель
    описатель(необ) : константное-выражение
```

Спецификатор типа, имеющий приведенную ниже форму, объявляет указанный идентификатор *меткой* (*tag*) структуры или объединения, задаваемой списком элементов:

```
struct-или-union идентификатор { список-объявлений-структуры }
```

Последующее объявление в той же или вложенной области действия может ссылаться на тот же тип, используя в спецификаторе только метку без списка:

```
struct-или-union идентификатор;
```

Если спецификатор с меткой, но без списка фигурирует там, где его метка не объявлена, создается *неполный тип*. Объекты неполного структурного типа могут упоминаться в контексте, где не требуется знать их размер, — например, в объявлениях (но не определениях) для описания указателя или создания нового типа с помощью `typedef`, но никаким иным образом. Тип становится полным при появлении последующего спецификатора с этой меткой, содержащего также список объявлений. Даже в спецификаторах со списком объявляемый тип структуры или объединения является неполным внутри списка и становится завершенным только после появления символа }, заканчивающего спецификатор.

Структура не может содержать элементов неполного типа. Следовательно, невозможно объявить структуру или объединение, которые содержат сами себя. Однако, кроме присвоения имени типу структуры или объединения, метка позволяет еще и определять структуры, обращающиеся сами к себе. Структуры или объединения могут содержать указатели на самих себя, поскольку можно объявлять указатели на неполные типы.

В отношении объявлений следующего вида действует особое правило:

```
struct-или-union идентификатор;
```

Это объявление создает структурный тип, но не содержит списка объявлений и описателя. Даже если *идентификатор* является меткой структуры или объединения, уже объявленной во внешней области действия (раздел A.11.1), такое объявление делает его меткой структуры или объединения нового, неполного типа в текущей области действия.

> Это маловразумительное правило — новое в стандарте ANSI. Оно предназначено для работы со взаимно рекурсивными структурами, объявленными во внутренней области действия, метки которых тем не менее могли быть уже объявлены ранее во внешней области действия.

Спецификатор структуры или объединения со списком, но без метки создает уникальный тип, на который можно ссылаться непосредственно только в объявлении, частью которого он является.

Имена элементов и меток не вступают в конфликт друг с другом или с обычными переменными. Имя элемента не может фигурировать дважды в одной и той же структуре или объединении, однако один и тот же элемент можно использовать в разных структурах или объединениях.

> В первом издании этой книги имена элементов структур и объединений не ассоциировались со своими родительскими объектами. Однако в компиляторах эта ассоциация стала обычной задолго до появления стандарта ANSI.

Элемент структуры или объединения, не являющийся битовым полем, может иметь любой объектный тип. Поле (которое не обязано иметь описатель и, следовательно, может быть безымянным) имеет тип `int`, `unsigned int` или `signed int` и интерпретируется как объект целочисленного типа с длиной, указанной в битах. Считается ли поле типа `int` знаковым или беззнаковым — зависит от реализации языка. Соседние элементы-поля структур упаковываются в системно-зависимые единицы памяти в зависящем от реализации направлении. Если следующее за полем другое поле не помещается в частично заполненную единицу (ячейку) памяти, то оно может оказаться разделенным между двумя ячейками, либо же ячейка может быть дополнена пустыми битами. Безымянное поле нулевой ширины обязательно приводит к такому дополнению, так что следующее поле начинается строго на границе очередной ячейки памяти.

> Стандарт ANSI делает поля еще более зависимыми от реализации, чем в первой редакции книги. Чтобы правильно хранить битовые поля в "зависящем от реализации" виде, желательно ознакомиться с правилами данной реализации языка. Структуры с битовыми полями могут служить системно-переносимым методом для попытки уменьшить размеры памяти под структуру (вероятно, ценой удлинения кода программы и времени обращения к полям) или непереносимым методом для описания распределения памяти на битовом уровне. Во втором случае необходимо понимать правила и соглашения конкретной реализации языка.

Элементы структуры имеют адреса, возрастающие в порядке объявления элементов. Элементы структуры, не являющиеся битовыми полями, выравниваются по границам адресуемых ячеек памяти в зависимости от своего типа; таким образом, в структуре могут быть безымянные дыры. Если указатель на структуру приводится к типу указателя на ее первый элемент, результат указывает на первый элемент.

Объединение можно представить себе как структуру, все элементы которой начинаются со смещения 0 и размеры которой достаточны для хранения любого из элементов. В каждый момент времени в объединении хранится не больше одного элемента. Если указатель на объединение приводится к типу указателя на один из элементов, результат указывает на этот элемент.

Вот простой пример объявления структуры:

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
};
```

Эта структура содержит массив из 20 символов, число типа `int` и два указателя на такую же структуру. Имея это объявление, можно далее объявлять объекты таким образом:

```
struct tnode s, *sp;
```

где `s` объявляется структурой указанного типа, а `sp` — указателем на такую структуру. При наличии приведенных определений следующее выражение будет ссылкой на элемент `count` в структуре, на которую указывает `sp`:

```
sp->count
```

А это выражение представляет собой указатель на левое поддерево в структуре `s`:

```
s.left
```

Наконец, следующее выражение дает первый символ из массива `tword`, элемента правого поддерева структуры `s`:

```
s.right->tword[0]
```

Вообще говоря, можно использовать значение только того элемента объединения, которому последний раз присваивалось значение. Однако есть одно особое правило, облегчающее работу с элементами объединения: если объединение содержит несколько структур, начинающихся с общей для них последовательности данных, и если объединение в текущий момент содержит одну из этих структур, то к общей части данных разрешается обращаться через любую из указанных структур (при этом гарантируется корректность результата). Так, следующий фрагмент кода вполне правомерен:

```
union {
    struct {
        int type;
    } n;
    struct {
        int type;
        int intnode;
    } ni;
    struct {
        int type;
        float floatnode;
    } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
    ... sin(u.nf.floatnode) ...
```

A.8.4. Перечислимые типы (перечисления)

Перечисления — это уникальные типы, значения которых целиком состоят из множества именованных констант (*перечислимых элементов*). Вид спецификатора перечисления заимствован у структур и объединений.

спецификатор-перечисления:
    enum *идентификатор*$_{необ}$ { *список-перечислимых* }
    enum *идентификатор*

список-перечислимых:
    перечислимое
    список-перечислимых , *перечислимое*

перечислимое:
    идентификатор
    идентификатор = *константное-выражение*

Идентификаторы, входящие в список перечислимых, объявляются константами типа `int` и могут употребляться везде, где требуются константы. Если в этом списке нет ни одного элемента со знаком =, то значения констант начинаются с 0 и увеличиваются на 1 по мере чтения объявления слева направо. Перечислимый элемент со знаком = присваивает соответствующему идентификатору указанное значение; последующие идентификаторы продолжают увеличиваться от заданного значения.

Имена перечислимых, используемые в одной области действия, должны отличаться друг от друга и от имен обычных переменных, но их значения не обязаны отличаться.

Роль *идентификатора* в *спецификаторе-перечисления* аналогична роли метки структуры в *спецификаторе-структуры*: он является именем конкретного перечислимого типа. Правила определения *спецификаторов-перечисления* с метками и без, а также для списков совпадают с правилами для спецификаторов структур или объединений, с той разницей, что перечислимые типы не бывают неполными. Метка *спецификатора-перечисления* без списка элементов должна иметь соответствующий спецификатор со списком в пределах области действия.

> В первой версии языка перечислимые типы отсутствовали, но они применяются на практике уже много лет.

A.8.5. Описатели

Описатели имеют следующий формальный синтаксис:

описатель:
    указатель$_{необ}$ *собственно-описатель*

собственно-описатель:
    идентификатор
    (*описатель*)
    собственно-описатель [*константное-выражение*$_{необ}$]
    собственно-описатель (*список-типов-параметров*)
    собственно-описатель (*список-идентификаторов*$_{необ}$)

указатель:

* *список-модификаторов-типа*$_{необ}$
* *список-модификаторов-типа*$_{необ}$ *указатель*

список-модификаторов-типа:
модификатор-типа
список-модификаторов-типа модификатор-типа

Построение описателей имеет много общего с построением обращений по указателям, вызовов функций и обращений к элементам массивов; в частности, группировка выполняется по тем же правилам.

A.8.6. Смысл и содержание описателей

Список описателей располагается сразу после цепочки спецификаторов типа и класса памяти. Главный элемент любого описателя — это объявляемый им уникальный идентификатор, который фигурирует в качестве первого варианта в грамматическом правиле *собственно-описателя*. Спецификаторы класса памяти относятся непосредственно к идентификатору, а его тип зависит от формы описателя. Описатель следует воспринимать как утверждение: если в выражении идентификатор фигурирует в той же форме, что и в описателе, то он обозначает объект указанного типа.

Если рассматривать только те части спецификаторов объявлений и описателей, которые определяют тип (см. раздел A.8.2), то объявление имеет общий вид "`T D`", где `T` — тип, а `D` — описатель. Тип, придаваемый идентификатору в различных формах описателя, можно описать индуктивно через эту запись.

В объявлениях вида `T D`, где `D` — просто идентификатор без дополнительных элементов, типом идентификатора будет `T`.

В объявлениях `T D`, где `D` имеет показанную ниже форму, тип идентификатора в `D1` будет тем же, что и в `D`:

```
( D1 )
```

Скобки не изменяют тип, но могут повлиять на результаты его привязки к идентификаторам в сложных описателях.

A.8.6.1. Описатели указателей

Рассмотрим объявление вида `T D`, где `D` имеет следующую форму:

* *список-модификаторов-типа*$_{необ}$ `D1`

Пусть тип идентификатора в объявлении `T D1` — "*расширитель-типа* `T`". Тогда типом идентификатора `D` будет "*расширитель-типа список-модификаторов-типа* указатель на `T`". Модификаторы типа, стоящие после символа `*`, относятся к самому указателю, а не к объекту, на который он указывает.

Для примера рассмотрим объявление

```
int *ap[];
```

где `ap[]` играет роль `D1`; объявление "`int ap[]`" дает переменной `ap` (см. ниже) тип "массив элементов типа `int`", список модификаторов типа здесь пуст, а расширитель типа — "массив элементов типа ...". Следовательно, на самом деле объявление `ap` означает "массив указателей на `int`".

Вот еще примеры объявлений:

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

В них объявляется целая переменная `i` и указатель на целую переменную `pi`. Значение указателя `cpi` нельзя изменить впоследствии; `cpi` всегда будет указывать на одно и то же место, даже если значение, на которое он указывает, изменится. Целая величина `ci` — константа, ее изменить нельзя (хотя можно инициализировать, как в данном случае). Тип указателя `pci` — "указатель на `const int`"; сам указатель можно изменить, чтобы он указывал на другое место в памяти, но значение, на которое он будет указывать, через обращение к `pci` изменить нельзя.

A.8.6.2. Описатели массивов

Рассмотрим объявление `T D`, где `D` имеет вид

`D1 [`*константное-выражение*$_{необ}$`]`

Пусть тип идентификатора объявления `T D1` — "*расширитель-типа* `T`", тогда типом идентификатора `D` будет "*расширитель-типа* массив элементов типа `T`". Если константное выражение присутствует, оно должно быть целочисленным и больше 0. Если константное выражение, задающее количество элементов в массиве, отсутствует, то массив имеет неполный тип.

Массив можно составлять из объектов арифметического типа, указателей, структур и объединений, а также других массивов (создавая таким образом многомерные массивы). Любой тип, из которого создается массив, должен быть полным; он не может быть структурой или массивом неполного типа. Это значит, что для многомерного массива пропустить можно только первую размерность. Неполный тип массива становится полным либо в другом, полном, объявлении этого массива (раздел A.10.2), либо при его инициализации (раздел A.8.7). Например, следующая запись объявляет массив чисел типа `float` и массив указателей на числа типа `float`:

```
float fa[17], *afp[17];
```

Аналогично, следующее объявление создает статический трехмерный массив целых чисел размером 3×5×7:

```
static int x3d[3][5][7];
```

Точнее говоря, `x3d` является массивом из трех элементов, каждый из которых есть массив из пяти элементов, содержащих по 7 значений типа `int`. Любое из выражений, `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]`, может фигурировать в другом выражении — лишь бы контекст был правильный. Первые три выражения имеют тип "массив", а последнее — тип `int`. Если конкретнее, `x3d[i][j]` — это массив из 7 целых чисел, а `x3d[i]` — массив из пяти массивов, по 7 целых чисел в каждом.

Операция индексирования массива определена так, что выражение `E1[E2]` идентично `*(E1+E2)`. Следовательно, несмотря на асимметричность записи, индексирование — коммутативная операция. Учитывая правила преобразования для операции + и массивов (разделы A.6.6, A.7.1, A.7.7), можно утверждать, что если `E1` — массив, а `E2` — целочисленная величина, то `E1[E2]` обозначает `E2`-й элемент массива `E1`.

Так, в нашем примере `x3d[i][j][k]` означает то же самое, что и `*(x3d[i][j]+k)`. Первое подвыражение, `x3d[i][j]`, согласно разделу A.7.1, приводится к типу "указатель на массив целых чисел"; согласно разделу A.7.7, сложение

включает умножение на размер объекта типа `int`. Из этих же правил следует, что массивы запоминаются построчно (последние индексы пробегаются быстрее) и что первый индекс в объявлении помогает определить количество памяти, занимаемой массивом, но не играет никакой другой роли в вычислении адреса элемента массива.

A.8.6.3. Описатели функций

Рассмотрим новый стиль объявления функций `T D`, где `D` имеет вид

`D1 (`*список-типов-параметров*`)`

Пусть тип идентификатора объявления `T D1` — "*расширитель-типа* `T`", тогда типом идентификатора в `D` будет "*расширитель-типа* функция с аргументами *список-типов-параметров*, возвращающая `T`".

Параметры имеют следующий синтаксис:

список-типов-параметров:
    список-параметров
    список-параметров **, ...**

список-параметров:
    объявление-параметра
    список-параметров **,** *объявление-параметра*

объявление-параметра:
    спецификаторы-объявления описатель
    спецификаторы-объявления абстрактный-описатель$_{необ}$

В новом стиле объявления функций список параметров явно задает их типы. В частном случае, если функция вообще не имеет параметров, в ее описателе на месте списка типов указывается одно ключевое слово `void`. Если список типов параметров заканчивается многоточием `", ..."`, то функция может иметь больше аргументов, чем количество явно описанных параметров (см. раздел A.7.3.2).

Типы параметров, являющихся массивами функций, заменяются на указатели в соответствии с правилами преобразования параметров (раздел A.10.1). Единственный спецификатор класса памяти, который разрешается использовать в объявлении параметра, — это `register`, однако он игнорируется, если описатель функции не является заголовком ее определения. Аналогично, если описатели в объявлениях параметров содержат идентификаторы, а описатель функции не является заголовком определения функции, то эти идентификаторы немедленно устраняются из текущей области действия. Абстрактные описатели, не содержащие идентификаторов, рассматриваются в разделе A.8.8.

В старом стиле вторая форма объявления функции имела вид `T D`, где `D` представляет собой

`D1 (`*список-идентификаторов*$_{необ}$`)`

Пусть тип идентификатора объявления `T D1` — "*расширитель-типа* `T`", тогда типом идентификатора в `D` будет "*расширитель-типа* функция неуказанных аргументов, возвращающая `T`". Параметры, если они есть, имеют следующий синтаксис:

список-идентификаторов:
    идентификатор
    список-идентификаторов **,** *идентификатор*

В старом стиле, если описатель функции не используется в качестве заголовка определения функции (раздел A.10.1), список идентификаторов должен отсутствовать. Никакой информации о типах параметров в объявлениях не содержится.

Для примера рассмотрим объявление

```
int f(), *fpi(), (*pfi)();
```

где объявляется функция `f`, возвращающая число типа `int`, функция `fpi`, возвращающая указатель на `int`, и указатель `pfi` на функцию, возвращающую `int`. Ни для одной функции в объявлении не указаны типы параметров; все функции объявлены в старом стиле.

А вот как выглядит объявление в новом стиле:

```
int strcpy(char *dest, const char *source), rand(void);
```

где `strcpy` — функция с двумя аргументами, возвращающая значение типа `int`; первый аргумент — указатель на значение типа `char`, а второй — указатель на неизменяемую строку символов. Имена параметров играют роль удобных комментариев. Вторая функция, `rand`, аргументов не имеет и возвращает число типа `int`.

> Описатели функций с прототипами параметров — наиболее важное нововведение стандарта ANSI. В сравнении со "старым стилем", принятым в первой редакции языка, они позволяют контролировать на предмет ошибок и приводить к нужным типам аргументы во всех вызовах. За это пришлось заплатить некоторую цену в виде путаницы в процессе введения нового стиля и необходимости согласования обеих форм. Чтобы обеспечить совместимость, потребовалось ввести некоторые неизящные синтаксические конструкции, а именно `void` для явного указания на отсутствие параметров.
>
> Многоточие `", ..."` применительно к функциям с переменным количеством аргументов — также новинка, которая вместе с макросами из стандартного заголовочного файла `<stdarg.h>` наконец обеспечивает официальную поддержку ранее запрещенного, хотя и неофициально допускавшегося механизма.
>
> Указанные синтаксические конструкции заимствованы из языка C++.

A.8.7. Инициализация

Начальное значение для объявляемого объекта можно указать с помощью *инициописателя*. Инициализатору, представляющему собой выражение или список инициализаторов в фигурных скобках, предшествует знак =. Этот список может завершаться запятой для удобства и единообразия формата.

инициализатор:
    выражение-присваивания
    `{` *список-инициализаторов* `}`
    `{` *список-инициализаторов* `,` `}`

список-инициализаторов:
    инициализатор
    список-инициализаторов `,` *инициализатор*

В инициализаторе статического объекта или массива все выражения должны быть константными (см. раздел A.7.19). Если инициализатор объекта или массива, объявленного с модификаторами `auto` или `register`, является списком в фигурных скобках, то входящие в него выражения также должны быть константными. Однако для автоматиче-

ского объекта с одним инициализирующим выражением оно не обязано быть константным, а просто должно иметь соответствующий объекту тип.

> В первой редакции языка не допускалась инициализация автоматических структур, объединений и массивов. Стандарт ANSI разрешает ее, но только с помощью константных конструкций, если инициализатор нельзя представить одним простым выражением.

Статический объект, не инициализированный явно, инициализируется так, как если бы ему (или его элементам) присваивалась константа 0. Начальное значение автоматического объекта, не инициализированного явным образом, не определено.

Инициализатор указателя или объекта арифметического типа — это одно выражение (возможно, заключенное в фигурные скобки), которое присваивается объекту.

Инициализатор структуры — это либо выражение того же структурного типа, либо заключенный в фигурные скобки список инициализаторов ее элементов, заданных по порядку. Безымянные битовые поля игнорируются и не инициализируются. Если инициализаторов в списке меньше, чем элементов в структуре, то оставшиеся элементы инициализируются нулями. Инициализаторов не должно быть больше, чем элементов.

Инициализатор массива — это список инициализаторов его элементов, заключенный в фигурные скобки. Если размер массива не известен, то он считается равным числу инициализаторов, при этом его тип становится полным. Если размер массива фиксирован, то число инициализаторов не должно превышать числа его элементов; если инициализаторов меньше, чем элементов, то оставшиеся элементы инициализируются нулями.

Особым случаем является инициализация массива символов. Ее можно выполнять с помощью строкового литерала; символы инициализируют элементы массива в том порядке, в каком они заданы в строковом литерале. Точно так же с помощью литерала из расширенного набора символов (см. раздел A.2.6) можно инициализировать массив типа `wchar_t`. Если размер массива не известен, то он определяется числом символов в строке, включая и завершающий нулевой символ; если размер массива известен, то число символов в строке, не считая завершающего нулевого символа, не должно превышать его размера.

Инициализатором объединения может быть либо выражение того же типа, либо заключенный в фигурные скобки инициализатор его первого элемента.

> В первой редакции языка не позволялось инициализировать объединения. Правило "первого элемента" несколько неуклюже, но зато практически не поддается обобщенной трактовке без введения новых синтаксических конструкций. Кроме введения явного способа инициализации объединений, пусть даже примитивным способом, этим правилом стандарт ANSI проясняет еще и семантику статических объединений, не инициализируемых явно.

Структуры или массивы обобщенно называются *составными объектами* (*aggregates*). Если составной объект содержит элементы составного типа, то правила инициализации применяются рекурсивно. Фигурные скобки в некоторых случаях инициализации можно опускать. Так, если инициализатор элемента составного объекта, который сам является составным объектом, начинается с левой фигурной скобки, то этот подобъект инициализируется последующим списком разделенных запятыми инициализаторов; считается ошибкой, если количество инициализаторов подобъекта превышает количество его элементов. Если, однако, инициализатор подобъекта не начинается с левой фигурной скобки, то чтобы его инициализировать, отсчитывается соответствующее число элементов из списка; следующие элементы инициализируются оставшимися инициализаторами объекта, частью которого является данный подобъект.

Рассмотрим пример:

```
int x[] = { 1, 3, 5 };
```

Здесь объявляется и инициализируется одномерный массив x из трех элементов, поскольку размер не указан, а список состоит из трех инициализаторов. Еще пример:

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

Эта конструкция представляет собой инициализацию с полным набором фигурных скобок: числа 1, 3 и 5 инициализируют первую строку в массиве `y[0]`, т.е. `y[0][0]`, `y[0][1]` и `y[0][2]`. Аналогично инициализируются следующие две строки: `y[1]` и `y[2]`. Инициализаторов не хватило на весь массив, поэтому элементы строки `y[3]` будут нулевыми. Точно такой же результат был бы получен с помощью следующего объявления:

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

Инициализатор для `y` начинается с левой фигурной скобки, но инициализатор для `y[0]` скобки не содержит, поэтому из списка будут взяты три элемента. Аналогично будут взяты по три элемента для `y[1]`, а затем для `y[2]`. И еще один пример:

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

Здесь инициализируется первый столбец матрицы `y` (рассматриваемой как двумерный массив), а все остальные элементы остаются нулевыми.

Наконец, вот пример инициализации массива символов:

```
char msg[] = "Syntax error on line %s\n";
```

Этот массив символов инициализируется с помощью строки; в его размере учитывается и завершающий нулевой символ `\0`.

A.8.8. Имена типов

В ряде контекстов (например, при явном приведении к типу, при указании типов параметров в объявлениях функций, в аргументе операции `sizeof`) возникает потребность в применении имени типа данных. Эта потребность реализуется с помощью *имени типа*, определение которого синтаксически почти совпадает с объявлением объекта того же типа, в котором только опущено имя объекта.

имя-типа:
    список-спецификаторов-модификаторов абстрактный-описатель$_{необ}$

абстрактный-описатель:
    указатель
    указатель$_{необ}$ *собственно-абстрактный-описатель*

собственно-абстрактный-описатель:

(абстрактный-описатель)
собственно-абстрактный-описатель$_{необ}$ *[константное-выражение*$_{необ}$ *]*
собственно-абстрактный-описатель$_{необ}$ *(список-типов-параметров*$_{необ}$ *)*

Можно указать ровно одно место в абстрактном описателе, где мог бы находиться идентификатор, если бы данная конструкция была описателем в полноценном объявлении объекта. Именованный тип совпадает с типом этого предполагаемого идентификатора. Вот примеры:

```
int
int *
int *[3]
int (*)[]
int *()
int (*[])(void)
```

Здесь объявляются соответственно типы "целое число" (`int`), "указатель на `int`", "массив из трех указателей на `int`", "указатель на массив из неизвестного количества элементов типа `int`", "функция неизвестного количества параметров, возвращающая указатель на `int`", "массив неизвестного размера, состоящий из указателей на функции без параметров, каждая из которых возвращает `int`".

A.8.9. Объявление `typedef`

Объявления, в которых спецификатором класса памяти является `typedef`, не объявляют объектов — они определяют идентификаторы, представляющие собой имена типов. Эти идентификаторы называются *типоопределяющими именами*.

типоопределяющее-имя:
идентификатор

Объявление с ключевым словом `typedef` ассоциирует тип с каждым именем из своих описателей (см. раздел A.8.6). С этого момента типоопределяющее имя становится синтаксически эквивалентным ключевому слову спецификатора типа и обозначает связанный с ним тип.

Рассмотрим следующие объявления:

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

После них становятся допустимыми следующие объявления:

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

Переменная `b` имеет тип `long`, `bp` — тип "указатель на `long`"; `z` —структура заданного вида; `zp` — указатель на такую структуру.

Объявление `typedef` не вводит новых типов, оно только дает имена типам, которые могли бы быть заданы и другим способом. Например, `b` имеет тот же тип, что и любой другой объект типа `long`.

Типоопределяющие имена могут быть переопределены в более внутренней области действия, но при условии, что в них присутствует непустой набор спецификаторов типа. Возьмем, например, конструкцию

```
extern Blockno;
```

Такое объявление не переопределяет `Blockno`. Переопределить его можно, например, следующим образом:

```
extern int Blockno;
```

A.8.10. Эквивалентность типов

Два списка спецификаторов типа эквивалентны, если они содержат одинаковый набор спецификаторов типа с учетом умолчаний (например, `long` подразумевает `long int`). Структуры, объединения и перечисления с разными метками считаются разными, а каждое объединение, структура или перечисление без метки представляет собой уникальный тип.

Два типа считаются совпадающими, если их абстрактные описатели (см. раздел A.8.8) окажутся одинаковыми с точностью до эквивалентности типов, после замены всех типоопределяющих имен соответствующими типами и отбрасывания идентификаторов параметров функций. При сравнении учитываются размеры массивов и типы параметров функции.

A.9. Операторы

За исключением оговоренных случаев, операторы выполняются в том порядке, в каком они записаны. Операторы не имеют вычисляемых значений и выполняются, чтобы выполнить определенные действия. Все операторы можно разбить на несколько групп:

оператор:
    оператор-с-меткой
    оператор-выражение
    составной-оператор
    оператор-выбора
    оператор-цикла
    оператор-перехода

A.9.1. Операторы с метками

Операторы могут иметь метки-префиксы.

оператор-с-меткой:
    идентификатор **:** *оператор*
    case *константное-выражение* **:** *оператор*
    default **:** *оператор*

Метка, состоящая из идентификатора, одновременно служит объявлением этого идентификатора. Единственное назначение идентификатора-метки — указать место перехода для оператора `goto`. Областью действия идентификатора-метки является текущая функция. Так как метки имеют свое собственное пространство имен, они не конфликтуют с другими идентификаторами и не могут быть переопределены (раздел A.11.1).

Метки блоков `case` и `default` используются в операторе `switch` (раздел A.9.4). Константное выражение в `case` должно быть целочисленным.

Сами по себе метки никак не изменяют порядка выполнения программы.

A.9.2. Операторы-выражения

Наиболее употребительный вид оператора — это оператор-выражение.

оператор-выражение:
    выражение$_{необ}$;

Чаще всего оператор-выражение — это присваивание или вызов функции. Все действия, реализующие побочный эффект выражения, завершаются до начала выполнения следующего оператора. Если выражение в операторе опущено, он называется *пустым* (*null*); пустой оператор часто используется для обозначения пустого тела оператора цикла или в качестве места перехода для метки.

A.9.3. Составные операторы

Чтобы в местах, где по синтаксису полагается один оператор, можно было выполнить несколько, предусматривается возможность задания составного оператора (который также называют *блоком*). Тело определения функции — это также составной оператор.

составной-оператор:
    { *список-объявлений*$_{необ}$ *список-операторов*$_{необ}$ }

список-объявлений:
    объявление
    список-объявлений объявление

список-операторов:
    оператор
    список-операторов оператор

Если идентификатор из *списка-объявлений* находился в более широкой области действия, чем блок, то действие внешнего объявления при входе в данный блока приостанавливается (раздел A.11.1), а после выхода из него — возобновляется. Внутри блока идентификатор можно объявить только один раз. Для каждого пространства имен эти правила действуют независимо (раздел A.11); идентификаторы из разных пространств имен всегда различны.

Инициализация автоматических объектов выполняется при каждом входе в блок и продолжается по мере перебора описателей. При переходе по метке внутрь блока никакие инициализации не выполняются. Инициализация статических объектов выполняется только один раз перед запуском программы.

A.9.4. Операторы выбора

Операторы выбора направляют выполнение программы по одному из нескольких альтернативных путей.

оператор-выбора:
    if (*выражение*) *оператор*
    if (*выражение*) *оператор* **else** *оператор*
    switch (*выражение*) *оператор*

Оба вида операторов `if` содержат выражение, которое должно иметь арифметический тип или тип указателя. Сначала вычисляется выражение со всеми его побочными эффектами, и результат сравнивается с 0. Если он не равен нулю, выполняется первый подоператор. В случае совпадения с 0 для второго вида `if` выполняется второй подоператор. Связанная со словом `else` неоднозначность разрешается тем, что `else` относят к последнему не имеющему своего `else` оператору `if`, расположенному на одном с этим `else` уровне вложенности блоков.

Оператор `switch` вызывает передачу управления одному из нескольких операторов в зависимости от значения выражения, которое должно иметь целочисленный тип. Входящий в состав `switch` подоператор обычно составной. Любой оператор внутри него может быть помечен одной или несколькими метками `case` (см. раздел A.9.1). Управляющее выражение подвергается расширению целочисленного типа (см. раздел A.6.1), а константы в `case` приводятся к расширенному типу. После такого преобразования никакие две `case`-константы в одном операторе `switch` не должны иметь одинаковых значений. С оператором `switch` может быть связано не больше одной метки `default`. Операторы `switch` можно вкладывать друг в друга; метки `case` и `default` относятся к самому внутреннему оператору `switch` из всех, которые их содержат.

Оператор `switch` выполняется следующим образом. Вычисляется выражение со всеми побочными эффектами, и результат сравнивается с каждой константой из блоков `case`. Если одна из `case`-констант равна значению выражения, управление передается оператору с соответствующей меткой. Если не обнаружено совпадения ни с одной из констант в блоках `case`, управление передается оператору с меткой `default`, если таковая имеется; в противном случае ни один из подоператоров `switch` не выполняется.

> В первой редакции языка требовалось, чтобы и управляющее выражение `switch`, и константы в блоках `case` имели тип `int`.

A.9.5. Операторы цикла

Операторы цикла служат для организации циклов — повторяющихся последовательностей операторов.

оператор-цикла:

while (*выражение*) *оператор*

do *оператор* **while** (*выражение*);

for ($выражение_{необ}$; $выражение_{необ}$; $выражение_{необ}$) *оператор*

В операторах `while` и `do` выполнение вложенного *оператора* (тела цикла) повторяется до тех пор, пока значение *выражения* не станет равно 0; *выражение* должно иметь арифметический тип или быть указателем. В операторе `while` вычисление выражения со всеми побочными эффектами и проверка условия выполняется перед каждым выполнением тела, а в операторе `do` эта проверка выполняется после.

В операторе `for` первое выражение вычисляется один раз, тем самым осуществляя инициализацию цикла. На тип этого выражения никакие ограничения не накладываются. Второе выражение должно иметь арифметический тип или тип указателя; оно вычисляется перед каждой итерацией (проходом) цикла. Как только его значение становится равным `0`, цикл `for` прекращает свою работу. Третье выражение вычисляется после каждой итерации и таким образом выполняет повторную инициализацию цикла. Никаких ограничений на его тип нет. Побочные эффекты всех трех выражений заканчиваются

сразу по завершении их вычисления. Если тело цикла `for` не содержит `continue`, то следующие две конструкции эквивалентны:

```
for ( выражение1 ; выражение2 ; выражение3 ) оператор
```

```
выражение1;
while ( выражение2 ) {
    оператор
    выражение3 ;
}
```

Любое из трех *выражений* цикла `for` можно опустить. Считается, что отсутствие второго выражения равносильно неявному сравнению ненулевой константы с нулем.

A.9.6. Операторы перехода

Операторы перехода выполняют безусловную передачу управления.

оператор-перехода:
goto *идентификатор* **;**
continue ;
break ;
return *выражение*$_{необ}$ **;**

В операторе `goto` указанный *идентификатор* должен быть меткой (см. раздел A.9.1), расположенной в текущей функции. Управление передается оператору с меткой.

Оператор `continue` может находиться только внутри цикла. Он вызывает переход к следующей итерации самого внутреннего цикла, содержащего его. Говоря более точно, в каждой из следующих конструкций оператор `continue`, не вложенный в более внутренний блок, эквивалентен `goto contin`:

```
while (...){
  ...
contin: ;
}
```

```
do {
  ...
contin: ;
} while (...);
```

```
for (...){
  ...
contin: ;
}
```

Оператор `break` можно использовать только в операторах цикла или в `switch`. Он завершает работу самого внутреннего цикла или `switch`, содержащего данный оператор `break`, после чего управление переходит к следующему оператору после только что завершенного тела.

С помощью оператора `return` функция возвращает управление в программу, откуда она была вызвана. Если после `return` стоит выражение, то его значение возвращается вызвавшей программе. Значение выражения приводится к типу, возвращаемому функцией, по тем же правилам, что и в процессе присваивания.

Ситуация, когда нормальный ход операций приводит в конец функции (т.е. к последней закрывающей фигурной скобке), равносильна выполнению оператора `return` без выражения. В этом случае возвращаемое значение не определено.

A.10. Внешние объявления

Материал, подготовленный в качестве входных данных для компилятора С, называется *единицей трансляции*. Такая единица состоит из последовательности внешних объявлений, каждое из которых представляет собой либо объявление, либо определение функции.

единица-трансляции:
    внешнее-объявление
    единица-трансляции внешнее-объявление

внешнее-объявление:
    определение-функции
    объявление

Область действия (видимости) внешних объявлений простирается до конца единицы трансляции, в которой они объявлены, точно так же, как область действия объявлений в блоке распространяется до конца этого блока. Синтаксис внешнего объявления не отличается от синтаксиса любого другого объявления с одним исключением: исходный код функции можно определять только на этом внешнем уровне.

A.10.1. Определения функций

Определение функции имеет следующий вид:

определение-функции:
    спецификаторы-объявления$_{необ}$ *описатель список-объявлений*$_{необ}$
    составной-оператор

Из спецификаторов класса памяти в *спецификаторах-объявления* разрешаются только `extern` и `static`; различия между ними рассматриваются в разделе A.11.2.

Функция может возвращать значения арифметического типа, структуры, объединения, указатели и `void`, но не "функции" и не "массивы". Описатель в объявлении функции должен явно указывать на то, что описываемый идентификатор имеет тип "функция", т.е. иметь одну из следующих двух форм (см. раздел A.8.6.3):

собственно-описатель (*список-типов-параметров*)
собственно-описатель (*список-идентификаторов*$_{необ}$)

Здесь *собственно-описатель* есть просто идентификатор либо идентификатор, заключенный в круглые скобки. Заметим, что тип "функция" для описателя нельзя получить посредством `typedef`.

Первая форма соответствует определению функции в новом стиле, для которого характерно объявление параметров в *списке-типов-параметров* вместе с их типами: после описателя не должно быть *списка-объявлений*. Если *список-типов-параметров* не состоит из единственного слова `void`, показывающего отсутствие параметров у функции, то в каждом описателе в *списке-типов-параметров* обязан присутствовать идентификатор. Если *список-типов-параметров* заканчивается символами ", ...", то в вызове функции может быть больше аргументов, чем параметров в ее определении. В таком случае, чтобы обращаться к дополнительным аргументам, следует пользоваться механизмом макроопределения `va_arg` из заголовочного файла `<stdarg.h>`, описанного в приложении Б. Функции с переменным количеством аргументов должны иметь по крайней мере один именованный параметр.

Вторая форма записи — это определение функции в старом стиле. *Список-идентификаторов* содержит имена параметров, а *список-объявлений* приписывает им типы. В *списке-объявлений* разрешено объявлять только именованные параметры, инициализация запрещается, и из спецификаторов класса памяти допускается только `register`.

В обоих стилях определения функций параметры считаются как бы объявленными в самом начале составного оператора, образующего тело функции, и совпадающие с ними имена здесь объявляться не должны (хотя, как и любые идентификаторы, их можно переобъявить в более внутренних блоках). Объявление параметра "массив типа..." трактуется как "указатель на тип...". Аналогично, объявление параметра с типом "функция, возвращающая тип..." трактуется как "указатель на функцию, возвращающую тип...". В момент вызова функции ее аргументы соответствующим образом преобразуются и присваиваются параметрам (см. раздел A.7.3.2).

> Новый стиль определения функций введен стандартом ANSI. Внесены также небольшие изменения в операции расширения типа; в первой редакции языка параметры типа `float` следовало воспринимать как `double`. Различие между `float` и `double` становилось заметным лишь тогда, когда внутри функции генерировался указатель на параметр.

Ниже приведен законченный пример определения функции в новом стиле:

```
int max(int a, int b, int c)
{
    int m;
    m = (a > b) ? a: b;
    return (m > c) ? m : c;
}
```

где `int` — спецификатор объявления; `max(int a, int b, int c)` — описатель функции; `{ ... }` — блок, задающий ее код. Определение той же функции в старом стиле выглядит следующим образом:

```
int max(a, b, c)
int a, b, c;
{
    /* ... */
}
```

Здесь уже `max(a, b, c)` — описатель, а `int a, b, c` — список объявлений параметров.

A.10.2. Внешние объявления

Внешние объявления задают характеристики объектов, функций и других идентификаторов. Термин "внешний" здесь используется, чтобы подчеркнуть расположение объявлений вне функций; напрямую с ключевым словом `extern` ("внешний") он не связан. Класс памяти для объекта с внешним объявлением либо вообще не указывается, либо указывается в виде `extern` или `static`.

В одной единице трансляции для одного идентификатора может содержаться несколько внешних объявлений, если они согласуются друг с другом по типу и способу связывания и если для этого идентификатора существует не более одного определения.

Два объявления объекта или функции считаются согласованными по типу в соответствии с правилами, рассмотренными в разделе A.8.10. Кроме того, если объявления от-

личаются лишь тем, что в одном из них тип структуры, объединения или перечисления неполон (см. раздел А.8.3), а в другом соответствующий ему тип с той же меткой полон, то такие типы считаются согласованными. Если два типа массива отличаются тем, что один — полный (см. раздел А.8.6.2), а другой — неполный, то такие типы также считаются согласованными при условии их совпадения во всем остальном. Наконец, если один тип определяет функцию в старом стиле, а другой — ту же функцию в новом стиле (с объявлениями параметров), то такие типы также считаются согласованными.

Если первое внешнее объявление функции или объекта содержит спецификацию `static`, то объявленный идентификатор имеет *внутреннее связывание*; в противном случае — *внешнее связывание*. Способы связывания рассматриваются в разделе А.11.2.

Внешнее объявление объекта считается определением, если оно имеет инициализатор. Внешнее объявление, в котором нет инициализатора и спецификатора `extern`, называется *предварительным определением*. Если в единице трансляции появится определение объекта, то все его предварительные определения станут считаться просто лишними объявлениями. Если никакого определения для этого объекта в единице трансляции не будет найдено, все его предварительные определения будут считаться одним определением с инициализатором `0`.

Каждый объект должен иметь ровно одно определение. Для объекта с внутренним связыванием это правило относится к каждой отдельной единице трансляции, поскольку объекты с внутренним связыванием в каждой единице уникальны. В случае объектов с внешним связыванием указанное правило действует в отношении всей программы.

> Хотя правило "одного определения" формулируется несколько иначе, чем в первой редакции языка, по существу оно совпадает с прежним. Некоторые реализации его ослабляют, более широко трактуя понятие предварительного определения. В другом варианте указанного правила, распространенном в системах Unix и признанном в стандарте как общепринятое расширение, все предварительные определения объектов с внешним связыванием из всех единиц трансляции программы рассматриваются вместе, а не отдельно в каждой единице. Если где-то в программе обнаруживается полное определение, то предварительные определения становятся просто объявлениями, но если никакого полного определения нет, то все предварительные определения становятся одним определением с инициализатором 0.

А.11. Область действия и связывание

Каждый раз компилировать всю программу нет необходимости: ее исходный текст можно хранить в нескольких файлах, представляющих собой единицы трансляции, а ранее скомпилированные функции можно загружать из библиотек. Связи между функциями программы могут осуществляться через вызовы и обращение к внешним данным.

Следовательно, существуют два вида областей действия (видимости) объектов: во-первых, *лексическая область действия* идентификатора, т.е. область в тексте программы, где компилятором воспринимаются все его характеристики; во-вторых, область действия, ассоциируемая с объектами и функциями, имеющими внешнее связывание. Внешнее связывание определяет связи, устанавливаемые между идентификаторами из раздельно компилируемых единиц трансляции.

A.11.1. Лексическая область действия

Идентификаторы относятся к нескольким пространствам имен, никак не связанным друг с другом. Один и тот же идентификатор может использоваться в разных смыслах даже в одной области действия, если в разных употреблениях он принадлежит к разным пространствам имен. Ниже перечислены классы объектов, имена которых представляют собой отдельные независимые пространства.

- Объекты, функции, типоопределяющие (`typedef`) имена и константы перечислимых типов.
- Метки операторов для перехода.
- Метки структур, объединений и перечислимых типов.
- Элементы каждой отдельной структуры или объединения.

> Сформулированные правила в нескольких аспектах отличаются от прежних, описанных в первом издании этого справочного руководства. Метки операторов не имели раньше собственного пространства; метки структур и объединений (а в некоторых реализациях и перечислимых типов) имели отдельные пространства. Помещение меток структур, объединений и перечислений в одном общем пространстве — это дополнительное ограничение, которого раньше не существовало. Наиболее существенное отклонение от первой редакции состоит в том, что каждая отдельная структура или объединение создает свое собственное пространство имен для своих элементов. Таким образом, одно и то же имя может использоваться в нескольких различных структурах. Это правило широко применяется уже в течение многих лет.

Лексическая область действия идентификатора объекта (или функции), объявленного во внешнем объявлении, начинается с места, где заканчивается его описатель, и распространяется до конца единицы трансляции, в которой он объявлен. Область действия параметра в определении функции начинается с начала блока, представляющего собой тело функции, и распространяется на всю функцию; область действия параметра в объявлении функции заканчивается в конце этого объявления. Область действия идентификатора, объявленного в начале блока, начинается от места, где заканчивается его описатель, и продолжается до конца этого блока. Областью действия для метки перехода является вся функция, где эта метка встречается. Область действия метки структуры, объединения или перечисления, а также константы перечислимого типа начинается от ее появления в спецификаторе типа и продолжается до конца единицы трансляции (для объявления на внешнем уровне) или до конца блока (для объявления внутри функции).

Если идентификатор явно объявлен в начале блока, в частности тела функции, то любое объявление того же идентификатора, находящееся снаружи этого блока, временно перестает действовать при входе в блок вплоть до его конца.

A.11.2. Связывание

В пределах одной единицы трансляции все объявления одного и того же идентификатора с *внутренним связыванием* относятся к одному и тому же объекту или функции, причем этот объект уникален для данной единицы трансляции. В случае *внешнего связывания* все объявления одного и того же идентификатора относятся к одному и тому же объекту, но уже в пределах всей программы.

Как говорилось в разделе A.10.2, если первое внешнее объявление идентификатора имеет спецификатор `static`, оно дает идентификатору внутреннее связывание, а если такого спецификатора нет, то внешнее. Если объявление находится внутри блока и не содержит `extern`, то соответствующий идентификатор не имеет связывания и уникален для данной функции. Если объявление содержит `extern` и блок находится в области действия внешнего объявления этого идентификатора, то последний имеет тот же способ связывания, что и во внешнем объявлении, и относится к тому же объекту (функции). Однако если ни одного внешнего объявления для этого идентификатора нет, то его способ связывания — внешний.

A.12. Препроцессор

Препроцессор языка C выполняет макроподстановку, условную компиляцию и включение именованных файлов. Строки, начинающиеся со знака # (перед которым разрешены символы пустого пространства), задают препроцессору инструкции-директивы. Их синтаксис не зависит от остальной части языка; они могут фигурировать где угодно и оказывать влияние (независимо от области действия) вплоть до конца единицы трансляции. Границы строк принимаются во внимание: каждая строка анализируется отдельно (но есть и возможность сцеплять строки; подробнее об этом в разделе A.12.2). Лексемами для препроцессора являются все лексемы языка и последовательности символов, задающие имена файлов, как, например, в директиве `#include` (раздел A.12.4). Кроме того, любой символ, не определенный каким-либо другим способом, также воспринимается как лексема. Влияние символов пустого пространства, отличающихся от пробелов и горизонтальных табуляций, внутри строк препроцессора не определено.

Сама препроцессорная обработка выполняется в несколько логически последовательных этапов. В отдельных реализациях некоторые этапы объединены вместе.

1. Прежде всего, комбинации из трех символов, описанные в разделе A.12.1, заменяются их эквивалентами. Между строками исходного кода вставляются символы конца строки, если этого требует операционная система.
2. Пары символов, состоящие из обратной косой черты с последующим символом конца строки, удаляются из текста; тем самым строки "склеиваются" (раздел A.12.2).
3. Программа разбивается на лексемы, разделенные символами пустого пространства. Комментарии заменяются единичными пробелами. Затем выполняются директивы препроцессора и макроподстановки (разделы A.12.3–A.12.10).
4. Управляющие последовательности в символьных константах и строковых литералах (разделы A.2.5.2, A.2.6) заменяются символами, которые они обозначают. Строковые литералы, записанные вплотную, сцепляются (подвергаются конкатенации).
5. Результат транслируется. Затем компонуются внешние связи с другими программами и библиотеками посредством сбора всех необходимых программ и данных воедино, а также подключения ссылок на внешние функции и объекты к их определениям.

A.12.1. Комбинации из трех символов

Множество символов, которыми набираются исходные тексты программ на C, содержится в семибитовом наборе ASCII, но является надмножеством инвариантного кодового набора ISO 646-1983 (ISO 646-1983 Invariant Code Set). Чтобы дать возможность программам пользоваться сокращенным набором символов, все перечисленные ниже комбинации трех символов при препроцессорной обработке заменяются на соответствующие им единичные символы. Замена выполняется до начала любой другой обработки.

??=	#	??(	[	??<	{
??/	\	??)	]	??>	}
??'	^	??!	|	??-	~

Никакие другие замены, кроме указанных, не выполняются.

Комбинации из трех символов (*trigraphs*) введены в язык стандартом ANSI.

A.12.2. Слияние строк

Каждая строка, заканчивающаяся обратной косой чертой, сцепляется в единое целое со следующей, поскольку символ \ и следующий за ним символ конца строки удаляются. Это делается перед разбивкой текста на лексемы.

A.12.3. Макроопределения и их раскрытие

Управляющая строка следующего вида заставляет препроцессор заменять *идентификатор* на *последовательность-лексем* везде далее по тексту программы:

`#define` *идентификатор последовательность-лексем*

При этом символы пустого пространства в начале и в конце последовательности лексем выбрасываются. Повторная строка `#define` с тем же идентификатором считается ошибкой, если последовательности лексем не идентичны (несовпадения в символах пустого пространства не играют роли).

Строка следующего вида, где между первым *идентификатором* и открывающей круглой скобкой не должно быть символов пустого пространства, представляет собой макроопределение с параметрами, задаваемыми *списком-идентификаторов*:

`#define` *идентификатор*`(`*список-идентификаторов*`)`
последовательность-лексем

Как и в первой форме, символы пустого пространства в начале и в конце последовательности лексем выбрасываются, и макрос может быть повторно определен только с идентичным по количеству и именам списком параметров и с той же последовательностью лексем.

Управляющая строка следующего вида приказывает препроцессору "забыть" определение, данное *идентификатору*:

`#undef` *идентификатор*

Применение директивы `#undef` к не определенному ранее идентификатору не считается ошибкой.

Если макроопределение было задано во второй форме, то любая следующая далее в тексте программы цепочка символов, состоящая из идентификатора макроса (возможно, с последующими символами пустого пространства), открывающей скобки, списка лексем, разделенных запятыми, и закрывающей скобки, представляет собой вызов макроса. Аргументами вызова макроса являются лексемы, разделенные запятыми, причем запятые, взятые в кавычки или вложенные скобки, в разделении аргументов не участвуют. Во время группировки аргументов раскрытие макросов в них не выполняется. Количество аргументов в вызове макроса должно соответствовать количеству параметров макроопределения. После выделения аргументов из текста символы пустого пространства, окружающие их, отбрасываются. Затем в замещающей последовательности лексем макроса каждый идентификатор-параметр, не взятый в кавычки, заменяется на соответствующий ему фактический аргумент из текста. Если в замещающей последовательности перед параметром не стоит знак #, если и ни перед ним, ни после него нет знака ##, то лексемы аргумента проверяются на наличие в них макровызовов; если таковые есть, то до подстановки аргумента в нем выполняется раскрытие соответствующих макросов.

На процесс подстановки влияют два специальных знака операций. Во-первых, если перед параметром в замещающей строке лексем вплотную стоит знак #, то вокруг соответствующего аргумента ставятся строковые кавычки ("), а потом идентификатор параметра вместе со знаком # заменяется получившимся строковым литералом. Перед каждым символом " или \, встречающимся вокруг или внутри строковой или символьной константы, автоматически вставляется обратная косая черта.

Во-вторых, если последовательность лексем в макроопределении любого вида содержит знак ##, то сразу после подстановки параметров он вместе с окружающими его символами пустого пространства отбрасывается, благодаря чему сцепляются соседние лексемы, образуя тем самым новую лексему. Результат не определен при генерировании таким образом недопустимых лексем языка или в случае, когда получающийся текст зависит от порядка применения операции ##. Кроме того, знак ## не может стоять ни в начале, ни в конце замещающей последовательности лексем.

В макросах обоих видов замещающая последовательность лексем повторно просматривается в поиске новых `define`-идентификаторов. Однако если какой-либо идентификатор уже был заменен в текущем процессе раскрытия, повторное появление такого идентификатора не вызовет его замены; он останется нетронутым.

Даже если развернутая строка макровызова начинается со знака #, она не будет воспринята как директива препроцессора.

> В стандарте ANSI процесс раскрытия макросов определен более точно, чем в первом издании книги. Наиболее важные изменения — это введение операций # и ##, которые позволяют брать аргументы в кавычки и выполнять конкатенацию. Некоторые из новых правил, особенно касающиеся конкатенации, могут показаться странноватыми. (См. приведенные ниже примеры.)

Описанные возможности можно, например, применять для введения "буквальных констант":

```
#define TABSIZE 100
int table[TABSIZE];
```

Следующее определение задает макрос, возвращающий абсолютное значение разности его аргументов:

```
#define ABSDIFF(a, b)   ((a)>(b) ? (a)-(b) : (b)-(a))
```

В отличие от функции, делающей то же самое, аргументы и возвращаемое значение здесь могут иметь любой арифметический тип и даже быть указателями. Кроме того, аргументы, каждый из которых может оказывать побочный эффект, вычисляются дважды: один раз при проверке условия и второй раз при вычислении результата.

Пусть имеется такое макроопределение:

```
#define tempfile(dir)   #dir "/%s"
```

Макровызов `tempfile(/usr/tmp)` даст при раскрытии следующее:

```
"/usr/tmp" "/%s"
```

Далее эти две строки будут сцеплены в одну. Вот еще один пример макроопределения:

```
#define cat(x,y) x ## y
```

Вызов `cat(var, 123)` сгенерирует результат `var123`. Однако вызов `cat(cat(1,2),3)` не даст желаемого результата, поскольку операция `##` помешает правильному раскрытию аргументов внешнего вызова `cat`. В результате получится следующая цепочка лексем:

```
cat ( 1 , 2 )3
```

где `)3` — результат сцепления последней лексемы первого аргумента с первой лексемой второго аргумента — сам не является допустимой лексемой. Но можно задать второй уровень макроопределения в таком виде:

```
#define xcat(x,y)   cat(x,y)
```

Теперь все пройдет благополучно, и вызов `xcat(xcat(1, 2), 3)` действительно даст результат `123`, потому что в раскрытии самого макроса `xcat` не участвует знак `##`.

Аналогично, макровызов `ABSDIFF(ABSDIFF(a, b), c)` при раскрытии даст ожидаемый, полностью развернутый результат.

A.12.4. Включение файлов

Следующая управляющая строка заменяется препроцессором на все содержимое файла с именем *имя-файла*:

```
#include <имя-файла>
```

Среди символов, составляющих *имя-файла*, не должно быть знака `>` и конца строки; результат не определен, если *имя-файла* содержит любой из символов `"`, `'`, `\` или `/*`. Порядок поиска указанного файла по его имени в ресурсах системы зависит от реализации.

Аналогично выполняется и управляющая строка

```
#include "имя-файла"
```

Вначале поиск файла выполняется по тем же правилам, по каким компилятор ищет сам файл исходного кода (это определение нестрого и зависит от реализации), а в случае неудачи — так же, как в `#include` первого типа. Результат остается неопределенным, если имя файла содержит `"`, `\` или `/*`, а вот использование знака `>` разрешается.

Третья директива не совпадает ни с одной из предыдущих форм:

```
#include последовательность-лексем
```

Она интерпретируется путем раскрытия *последовательности-лексем* как нормального текста, который в результате всех макроподстановок должен дать

#include <...> или #include "...". Полученная таким образом директива далее будет обрабатываться в соответствии с полученной формой.

Допускается вложение данных директив, т.е. файлы, включаемые с помощью #include, также могут содержать директивы #include.

A.12.5. Условная компиляция

Части программы могут компилироваться условно, если они оформлены в соответствии со следующей синтаксической схемой:

условная-конструкция-препроцессора:
    строка-if текст блоки-elif блок-else$_{необ}$ **#endif**

строка-if:
    #if *константное-выражение*
    #ifdef *идентификатор*
    #ifndef *идентификатор*

блоки-elif:
    строка-elif текст
    блоки-elif$_{необ}$

строка-elif:
    #elif *константное-выражение*

блок-else:
    строка-else текст

строка-else:
    #else

Каждая из директив (*строка-if*, *строка-elif*, *строка-else* и #endif) записывается в отдельной строке. Константные выражения в директиве #if и последующих строках #elif вычисляются по порядку, пока не будет обнаружено выражение с ненулевым значением; текст, следующий за строкой с нулевым значением, отбрасывается. Текст, расположенный за директивой с ненулевым значением, обрабатывается обычным образом. Под словом "текст" здесь имеется в виду любой текстовый материал, включая строки препроцессора, не входящие в условную структуру; текст может быть и пустым. Если строка #if или #elif с ненулевым значением выражения найдена и ее текст обработан, то последующие строки #elif и #else вместе со своими текстами отбрасываются. Если все выражения имеют нулевые значения и присутствует строка #else, то следующий за ней текст обрабатывается обычным образом. Тексты неактивных ветвей условных конструкций игнорируются; выполняется только проверка вложенности условных конструкций.

Константные выражения в #if и #elif подвергаются обычной макроподстановке. Более того, перед раскрытием макросов выражения следующего вида всегда заменяются константами 1L, если указанный *идентификатор* определен, и 0L, если не определен:

defined *идентификатор*
defined (*идентификатор*)

Все идентификаторы, оставшиеся после макроподстановки, заменяются на `0L`. Наконец, предполагается, что любая целая константа всегда имеет суффикс `L`, т.е. все арифметические операции работают только с операндами типа `long` или `unsigned long`.

Получившееся константное выражение (см. раздел A.7.19) должно подчиняться некоторым ограничениям: быть целочисленным, не содержать перечислимых констант, преобразований типов и операций `sizeof`.

Имеются также следующие управляющие директивы:

```
#ifdef идентификатор
#ifndef идентификатор
```

Они эквивалентны соответственно строкам

```
#if defined идентификатор
#if !defined идентификатор
```

> Директивы `#elif` не было в первой версии языка, хотя она и использовалась в некоторых препроцессорах. Конструкция `defined` — также новая.

A.12.6. Нумерация строк

Для удобства работы других препроцессоров, генерирующих программы на C, введены следующие директивы:

```
#line константа "имя-файла"
#line константа
```

Эти директивы предписывают компилятору считать, что указанные в них десятичное целое число и идентификатор являются соответственно номером следующей строки и именем текущего файла. Если имя файла отсутствует, то ранее запомненное имя не изменяется. Раскрытие макровызовов в директиве `#line` выполняется до ее интерпретации.

A.12.7. Генерирование сообщений об ошибках

Директива препроцессора следующего вида приказывает ему выдать диагностическое сообщение, включающее заданную последовательность лексем:

```
#error последовательность-лексем_необ
```

A.12.8. Директива `#pragma`

Управляющая строка следующего вида предписывает препроцессору выполнить некоторую операцию, зависящую от реализации:

```
#pragma последовательность-лексем_необ
```

Неопознанная директива этого вида просто игнорируется.

A.12.9. Пустая директива

Строка препроцессора следующего вида не вызывает никаких операций:

```
#
```

A.12.10. Заранее определенные имена

Существует несколько заранее определенных препроцессором идентификаторов, которые он заменяет специальной информацией. Эти идентификаторы (и выражения с операцией препроцессора `defined`) нельзя переопределять или отменять директивой `#undef`.

`__LINE__`	Номер текущей строки исходного текста (десятичная константа)
`__FILE__`	Имя компилируемого файла (символьная строка)
`__DATE__`	Дата компиляции в виде `"Ммм дд гггг"` (символьная строка)
`__TIME__`	Время компиляции в виде `"чч:мм:сс"` (символьная строка)
`__STDC__`	Константа `1`. Предполагается, что этот идентификатор определен со значением `1` только в тех реализациях, которые следуют стандарту

Директивы `#error` и `#pragma` впервые введены стандартом ANSI. Заранее определенные макросы препроцессора также до сих пор не были стандартизированы, хотя и использовались в некоторых реализациях.

A.13. Грамматика

Ниже приведена сводка грамматических правил, которые уже рассматривались в данном приложении. Они имеют то же содержание, но даны в другом порядке.

В этой грамматике не приводятся определения таких терминальных (фундаментальных, базовых) символов, как *целочисленная-константа*, *символьная-константа*, *вещественная-константа*, *идентификатор*, *строка* и *константа-перечислимого-типа*. Слова и символы, набранные **полужирным моноширинным** латинским шрифтом, являются терминальными символами и используются точно в таком виде, как записаны. Данную грамматику можно механически преобразовать во входные данные для автоматического генератора синтаксических анализаторов. Кроме добавления синтаксической разметки, предназначенной для указания альтернатив в правилах подстановки, для этого потребуется раскрыть конструкции "один из", а также (в зависимости от правил генератора анализаторов) ввести дубликаты каждого правила с индексом *необ*: записать один вариант правила с символом, а второй без него. Если внести еще одно изменение, а именно удалить правило *типоопределяющее-имя: идентификатор* и объявить *типоопределяющее-имя* терминальным символом, данная грамматика сможет служить входными данными для генератора синтаксических анализаторов YACC. Ей присуще лишь одно противоречие, вызванное неоднозначностью конструкции `if-else`.

единица-трансляции:
    внешнее-объявление
    единица-трансляции внешнее-объявление

внешнее-объявление:
    определение-функции
    объявление

определение-функции:
    спецификаторы-объявления$_{необ}$ *описатель список-объявлений*$_{необ}$ *составной-оператор*

объявление:
    спецификаторы-объявления список-иниц-описателей$_{необ}$

список-объявлений:
    объявление
    список-объявлений объявление

спецификаторы-объявления:
    спецификатор-класса-памяти спецификаторы-объявления$_{необ}$
    спецификатор-типа спецификаторы-объявления$_{необ}$
    модификатор-типа спецификаторы-объявления$_{необ}$

спецификатор-класса-памяти: один из
    auto **register** **static** **extern** **typedef**

спецификатор-типа: один из
    void char short int long float double signed
    unsigned *спецификатор-структуры-или-объединения*
    спецификатор-перечисления типоопределяющее-имя

модификатор-типа: один из
    const **volatile**

спецификатор-структуры-или-объединения:
    struct-или-union идентификатор$_{необ}$ **{** *список-объявлений-структуры* **}**
    struct-или-union идентификатор

struct-или-union: один из
    struct **union**

список-объявлений-структуры:
    объявление-структуры
    список-объявлений-структуры объявление-структуры

список-иниц-описателей:
    иниц-описатель
    список-иниц-описателей **,** *иниц-описатель*

иниц-описатель:
    описатель
    описатель **=** *инициализатор*

объявление-структуры:
    список-спецификаторов-модификаторов список-описателей-структуры **;**

список-спецификаторов-модификаторов:
    спецификатор-типа список-спецификаторов-модификаторов$_{необ}$
    модификатор-типа список-спецификаторов-модификаторов$_{необ}$

список-описателей-структуры:
    описатель-структуры
    список-описателей-структуры **,** *описатель-структуры*

описатель-структуры:
    описатель
    описатель$_{необ}$ **:** *константное-выражение*

спецификатор-перечисления:
    enum *идентификатор*$_{необ}$ **{** *список-перечислимых* **}**
    enum *идентификатор*

список-перечислимых:
    перечислимое
    список-перечислимых **,** *перечислимое*

перечислимое:
    идентификатор
    идентификатор **=** *константное-выражение*

описатель:
    указатель$_{необ}$ *собственно-описатель*

собственно-описатель:
    идентификатор
    (*описатель* **)**
    собственно-описатель **[** *константное-выражение*$_{необ}$ **]**
    собственно-описатель **(** *список-типов-параметров* **)**
    собственно-описатель **(** *список-идентификаторов*$_{необ}$ **)**

указатель:
    ***** *список-модификаторов-типа*$_{необ}$
    ***** *список-модификаторов-типа*$_{необ}$ *указатель*

список-модификаторов-типа:
    модификатор-типа
    список-модификаторов-типа модификатор-типа

список-типов-параметров:
    список-параметров
    список-параметров **, ...**

список-параметров:
    объявление-параметра
    список-параметров **,** *объявление-параметра*

объявление-параметра:
    спецификаторы-объявления описатель
    спецификаторы-объявления абстрактный-описатель$_{необ}$

список-идентификаторов:
    идентификатор
    список-идентификаторов **,** *идентификатор*

инициализатор:
    выражение-присваивания
    { *список-инициализаторов* **}**
    { *список-инициализаторов* **, }**

список-инициализаторов:
    инициализатор
    список-инициализаторов **,** *инициализатор*

имя-типа:
    список-спецификаторов-модификаторов абстрактный-описатель$_{необ}$

абстрактный-описатель:
    указатель
    указатель$_{необ}$ *собственно-абстрактный-описатель*

собственно-абстрактный-описатель:
    (*абстрактный-описатель* **)**
    собственно-абстрактный-описатель$_{необ}$ **[** *константное-выражение*$_{необ}$ **]**
    собственно-абстрактный-описатель$_{необ}$ **(** *список-типов-параметров*$_{необ}$ **)**

типоопределяющее-имя:
    идентификатор

оператор:
    оператор-с-меткой
    оператор-выражение
    составной-оператор
    оператор-выбора
    оператор-цикла
    оператор-перехода

оператор-с-меткой:
    идентификатор **:** *оператор*
    case *константное-выражение* **:** *оператор*
    default : *оператор*

оператор-выражение:
    выражение$_{необ}$**;**

составной-оператор:
    { *список-объявлений*$_{необ}$ *список-операторов*$_{необ}$ **}**

список-операторов:
    оператор
    список-операторов оператор

оператор-выбора:
    if (*выражение* **)** *оператор*
    if (*выражение* **)** *оператор* **else** *оператор*
    switch (*выражение* **)** *оператор*

оператор-цикла:
    while (*выражение* **)** *оператор*
    do *оператор* **while (** *выражение* **);**
    for (*выражение*$_{необ}$ **;** *выражение*$_{необ}$ **;** *выражение*$_{необ}$ **)** *оператор*

оператор-перехода:
    goto *идентификатор* **;**
    continue ;
    break ;
    return *выражение*$_{необ}$ **;**

выражение:
    выражение-присваивания
    выражение **,** *выражение-присваивания*

выражение-присваивания:
    условное-выражение
    одноместное-выражение знак-присваивания выражение-присваивания

знак-присваивания: один из
    = *= /= %= += -= <<= >>= &= ^= |=

условное-выражение:
    выражение-с-логическим-ИЛИ
    выражение-с-логическим-ИЛИ **?** *выражение* **:** *условное-выражение*

константное-выражение:
    условное-выражение

выражение-с-логическим-ИЛИ:
    выражение-с-логическим-И
    выражение-с-логическим-ИЛИ **||** *выражение-с-логическим-И*

выражение-с-логическим-И:
    выражение-с-включающим-ИЛИ
    выражение-с-логическим-И **&&** *выражение-с-включающим-ИЛИ*

выражение-с-включающим-ИЛИ:
    выражение-с-исключающим-ИЛИ
    выражение-с-включающим-ИЛИ **|** *выражение-с-исключающим-ИЛИ*

выражение-с-исключающим-ИЛИ:
    выражение-с-И
    выражение-с-исключающим-ИЛИ **^** *выражение-с-И*

выражение-с-И:
    выражение-равенства
    выражение-с-И **&** *выражение-равенства*

выражение-равенства:
    выражение-отношения
    выражение-равенства **==** *выражение-отношения*
    выражение-равенства **!=** *выражение-отношения*

выражение-отношения:
    выражение-со-сдвигом
    выражение-отношения **<** *выражение-со-сдвигом*
    выражение-отношения **>** *выражение-со-сдвигом*
    выражение-отношения **<=** *выражение-со-сдвигом*
    выражение-отношения **>=** *выражение-со-сдвигом*

выражение-со-сдвигом:
    аддитивное-выражение
    выражение-со-сдвигом **>>** *аддитивное-выражение*
    выражение-со-сдвигом **<<** *аддитивное-выражение*

аддитивное-выражение:
    мультипликативное-выражение
    аддитивное-выражение **+** *мультипликативное-выражение*
    аддитивное-выражение **-** *мультипликативное-выражение*

мультипликативное-выражение:
    выражение-приведения-к-типу

```
    мультипликативное-выражение * выражение-приведения-к-типу
    мультипликативное-выражение / выражение-приведения-к-типу
    мультипликативное-выражение % выражение-приведения-к-типу

выражение-приведения-к-типу:
    одноместное-выражение
    ( имя-типа ) выражение-приведения-к-типу

одноместное-выражение:
    постфиксное-выражение
    ++ одноместное-выражение
    -- одноместное-выражение
    знак-одноместной-операции выражение-приведения-к-типу
    sizeof одноместное-выражение
    sizeof ( имя-типа )

знак-одноместной-операции: один из
    &   *   +   -   ~   !

постфиксное-выражение:
    первичное-выражение
    постфиксное-выражение [ выражение ]
    постфиксное-выражение ( список-аргументов-выражений_необ )
    постфиксное-выражение . идентификатор
    постфиксное-выражение -> идентификатор
    постфиксное-выражение ++
    постфиксное-выражение --

первичное-выражение:
    идентификатор
    константа
    строка
    ( выражение )

список-аргументов-выражений:
    выражение-присваивания
    список-аргументов-выражений , выражение-присваивания

константа:
    целочисленная-константа
    символьная-константа
    вещественная-константа
    константа-перечислимого-типа
```

Ниже приведено формальное описание грамматики языка препроцессора, определяющее структуру управляющих строк (директив). Для автоматического генерирования синтаксического анализатора это описание непригодно. Грамматика включает символ *текст*, который обозначает обычный текст программы, безусловные управляющие строки препроцессора или его законченные условные конструкции.

```
управляющая-строка:
    #define идентификатор последовательность-лексем
    #define идентификатор( идентификатор, ..., идентификатор ) по-
следовательность-лексем
    #undef идентификатор
    #include <имя-файла>
```

```
    #include "имя-файла"
    #include последовательность-лексем
    #line константа "имя-файла"
    #line константа
    #error последовательность-лексем_необ
    #pragma последовательность-лексем_необ
    #
    условная-конструкция-препроцессора

условная-конструкция-препроцессора:
    строка-if текст блоки-elif блок-else_необ #endif

строка-if:
    #if константное-выражение
    #ifdef идентификатор
    #ifndef идентификатор

блоки-elif:
    строка-elif текст
    блоки-elif_необ

строка-elif:
    #elif константное-выражение

блок-else:
    строка-else текст

строка-else:
    #else
```

Приложение Б

Стандартная библиотека

В этом приложении кратко описана библиотека, определенная стандартом ANSI. Стандартная библиотека С не является частью собственно языка, однако всякая среда, поддерживающая язык С, обязана предоставить программисту объявления функций, типов и макросов, содержащиеся в этой библиотеке. Здесь опущены описания некоторых функций, имеющих слишком ограниченное применение или легко синтезируемых на основе других; не рассматриваются многобайтные символы, а также вопросы локализации, т.е. свойства библиотеки, зависящие от конкретного языка, страны или культурной среды.

Функции, типы и макросы стандартной библиотеки объявлены в стандартных *заголовочных файлах*:

`<assert.h>`	`<float.h>`	`<math.h>`	`<stdarg.h>`	`<stdlib.h>`
`<ctype.h>`	`<limits.h>`	`<setjmp.h>`	`<stddef.h>`	`<string.h>`
`<errno.h>`	`<locale.h>`	`<signal.h>`	`<stdio.h>`	`<time.h>`

Обращение к заголовочному файлу (включение) выполняется следующим образом:

`#include <`*`заголовочный_файл`*`>`

Заголовочные файлы можно включать в любом порядке и любое количество раз. Они должны включаться за пределами всех внешних объявлений или определений и до обращения к любым объектам, объявленным в них. Заголовочный файл не обязан быть файлом исходного кода.

Внешние идентификаторы, начинающиеся со знака подчеркивания, зарезервированы для использования библиотекой, как и все прочие идентификаторы, которые начинаются с этого знака плюс буква в верхнем регистре или еще один знак подчеркивания.

Б.1. Ввод-вывод: `<stdio.h>`

Функции ввода-вывода, типы и макросы, определенные в файле `<stdio.h>`, составляют почти треть библиотеки.

Потоком (*stream*) называется источник или получатель данных, который можно ассоциировать с диском или другим периферийным устройством. Библиотека поддерживает текстовые и двоичные потоки, хотя в некоторых системах (в частности, в Unix) они не различаются. Текстовый поток — это последовательность строк; каждая строка содержит нуль или больше символов и заканчивается символом `'\n'`. Операционная среда часто преобразует текстовые потоки в другие представления и обратно (например, символ `'\n'` может представляться комбинацией возврата каретки и перевода строки). Двоичный поток — это последовательность необработанных байтов, представляющих некие внутренние данные. Основное свойство двоичного потока состоит в том, что если его записать, а затем прочитать в одной и той же системе, поток от этого не изменится.

Поток подсоединяется к файлу или устройству путем *открытия*; соединение разрывается путем *закрытия* потока. При открытии файла возвращается указатель на объект

типа FILE, хранящий всю необходимую информацию для управления потоком. Во всех случаях, где это не вызывает путаницы, выражения "поток" и "файловый указатель" будут употребляться как взаимозаменяемые.

В начале выполнения любой программы автоматически открываются и предоставляются в ее пользование три потока: stdin, stdout и stderr.

Б.1.1. Файловые операции

Для работы с файлами используются перечисленные ниже функции. Тип size_t представляет собой целочисленный тип без знака; к данным именно этого типа относятся результаты операции sizeof.

FILE *fopen(const char *filename, const char *mode)

Функция fopen открывает именованный файл и возвращает указатель потока или NULL, если попытка открытия оказалась неудачной. Параметр режима, mode, может принимать следующие значения.

"r"	Открытие текстового файла для чтения
"w"	Создание текстового файла для записи; старое содержимое, если оно было, стирается
"a"	Открытие или создание текстового файла для записи в конец (дописывания)
"r+"	Открытие текстового файла для модифицирования (чтения и записи)
"w+"	Создание текстового файла для модифицирования; старое содержимое, если оно было, стирается
"a+"	Открытие или создание текстового файла для модифицирования, записи в конец

Режим модифицирования позволяет выполнять чтение и запись в один и тот же файл; перед переходом от чтения к записи и обратно необходимо вызвать функцию fflush или функцию позиционирования в файле. Если включить в параметр режима букву b после первой буквы (например, "rb" или "w+b"), это будет означать, что файл — двоичный. Имя файла должно быть не длиннее FILENAME_MAX символов. Одновременно могут оставаться открытыми не более FOPEN_MAX файлов.

FILE *freopen(const char *filename, const char *mode, FILE *stream)

Функция freopen открывает файл в заданном режиме и ассоциирует с ним поток. Она возвращает указатель на поток или NULL в случае ошибки. Как правило, freopen используется для замены файлов, ассоциированных с потоками stdin, stdout и stderr.

int fflush(FILE *stream)

Применительно к потоку вывода функция fflush выполняет запись всех буферизованных, но еще не записанных данных; результат применения к потоку ввода не определен. В случае ошибки записи функция возвращает EOF, в противном случае — нуль. Вызов fflush(NULL) выполняет указанные операции для всех потоков вывода.

int fclose(FILE *stream)

Функция fclose выполняет запись буферизованных, но еще не записанных данных, уничтожает непрочитанные буферизованные входные данные, освобождает все автома-

тически выделенные буфера, после чего закрывает поток. Возвращает EOF в случае ошибки и нуль в противном случае.

int remove(const char *filename)

Функция remove удаляет файл с указанным именем; последующая попытка открыть файл с этим именем приведет к ошибке. Возвращает ненулевое значение в случае неудачной попытки.

int rename(const char *oldname, const char *newname)

Функция rename заменяет старое имя файла (oldname) на новое (newname); возвращает ненулевое значение, если попытка изменить имя оказалась неудачной.

FILE *tmpfile(void)

Функция tmpfile создает временный файл с режимом доступа "wb+", который автоматически удаляется при его закрытии или нормальном завершении программы. Эта функция возвращает указатель потока или NULL, если не смогла создать файл.

char *tmpnam(char s[L_tmpnam])

Вызов tmpnam(NULL) создает строку, не совпадающую ни с одним из имен существующих файлов, и возвращает указатель на внутренний статический массив. Вызов tmpnam(s) помещает строку в s и возвращает ее в качестве значения функции; длина s должна быть не менее L_tmpnam символов. При каждом вызове tmpnam генерируется новое имя; при этом гарантируется не более TMP_MAX различных имен за один сеанс работы программы. Отметим, что tmpnam создает имя, а не файл.

int setvbuf(FILE *stream, char *buf, int mode, size_t size)

Функция setvbuf управляет буферизацией потока; ее нужно вызвать до того, как будет выполняться чтение, запись или какая-либо другая операция. Параметр mode со значением _IOFBF задает полную буферизацию, _IOLBF — построчную буферизацию текстового файла, а _IONBF отменяет буферизацию вообще. Если параметр buf не равен NULL, то его значение будет использоваться как указатель на буфер; в противном случае для буфера будет выделена память. Параметр size задает размер буфера. Функция setvbuf возвращает ненулевое значение в случае ошибки.

void setbuf(FILE *stream, char *buf)

Если параметр buf равен NULL, то для потока stream буферизация отменяется. В противном случае вызов setbuf эквивалентен вызову (void) setvbuf(stream, buf, _IOFBF, BUFSIZ).

Б.1.2. Форматированный вывод

Функции семейства printf выполняют вывод данных с преобразованием в заданный формат.

`int fprintf(FILE *stream, const char *format, ...)`

Функция `fprintf` преобразует и выводит данные в поток `stream` под управлением строки формата `format`. Возвращается количество записанных символов или, в случае ошибки, отрицательное число.

Строка формата содержит два вида объектов: обычные символы, копируемые в поток вывода, и спецификации формата, которые задают преобразование и вывод следующих далее аргументов `fprintf` в том порядке, в каком они перечислены. Каждая спецификация формата начинается с символа `%` и заканчивается символом-спецификацией вывода. Между `%` и символом-спецификацией могут находиться следующие дополнительные элементы в том порядке, в котором они перечислены ниже.

- Флаги (в любом порядке), модифицирующие спецификацию формата:

  `-` задает выравнивание форматированного аргумента по левому краю его поля;

  `+` требует обязательного вывода числа со знаком;

  пробел — при отсутствии знака перед числом должен стоять пробел;

  `0` задает дополнение поля слева до заданной ширины нулями при выводе числа;

  `#` задает альтернативную форму вывода: для `o` первой цифрой должен быть `0`; для `x` или `X` ненулевому результату должны предшествовать символы `0x` или `0X`; для `e`, `E`, `f`, `g` и `G` результат должен обязательно содержать десятичную точку; для `g` и `G` запрещается удалять нули в конце числа.

- Число, определяющее минимальную ширину поля. Преобразованный аргумент выводится в поле, размер которого не меньше указанной ширины, а если потребуется — то и в поле большего размера. Если количество символов преобразованного аргумента меньше ширины поля, то поле будет дополнено слева (или справа, если задано выравнивание по левому краю) до заданной ширины. Обычно поле дополняется пробелами, но в присутствии соответствующей спецификации — нулями.
- Точка, отделяющая параметр ширины поля от точности.
- Число, задающее точность, которое определяет максимальное количество символов, выводимых из строки, или количество цифр после десятичной точки в спецификациях `e`, `E` или `f`, или же количество значащих цифр для `g` или `G`, или минимальное количество цифр при выводе целого числа (до необходимой ширины поля число дополняется нулями).
- Модификатор длины `h`, `l` или `L`. Символ `h` определяет, что соответствующий аргумент должен выводиться как `short` или `unsigned short`; `l` сообщает, что аргумент имеет тип `long` или `unsigned long`; `L` указывает, что аргумент принадлежит к типу `long double`.

Ширина, точность или оба эти параметра одновременно могут быть заданы в виде звездочки (`*`); в этом случае необходимое число берется из следующего аргумента, который должен иметь тип `int`. При наличии двух звездочек используются два аргумента.

Символы-спецификации формата и их значения приведены в табл. Б.1. Если после `%` нет одного из стандартных символов-спецификаций, результат операции не определен.

Таблица Б.1. Спецификации вывода функции printf

Символ	Тип аргумента и форма вывода
d, i	int; десятичное число со знаком
o	unsigned int; восьмеричное число без знака (без 0 в начале)
x, X	unsigned int; шестнадцатеричное число без знака (без 0x или 0X в начале); в качестве цифр от 10 до 15 используются abcdef для x и ABCDEF для X
u	unsigned int; десятичное целое число без знака
c	int; одиночный символ после преобразования в unsigned char
s	char *; символы строки выводятся, пока не встретится '\0' или пока не будет выведено количество символов, заданное спецификацией точности
f	double; десятичное число вида [-]*mmm.ddd*, где количество цифр d задается точностью. По умолчанию точность равна 6; нулевая точность запрещает вывод десятичной точки
e, E	double; десятичное число вида [-]*m.ddddde±xx* или [-]*m.ddddddE±xx*, где количество цифр d задается точностью. По умолчанию точность равна 6; нулевая точность запрещает вывод десятичной точки
g, G	double; используется %e или %E, если показатель степени меньше -4, или больше, или равен точности; в противном случае используется %f. Завершающие нули и десятичная точка в конце не выводятся
p	void *; выводится в виде указателя (представление зависит от реализации)
n	int *; число символов, выведенных к текущему моменту текущим вызовом printf, *записывается в аргумент*. Никакого преобразования формата не происходит
%	никакие аргументы не преобразуются; выводится символ %

```
int printf(const char *format, ...)
```

Функция printf(...) эквивалентна fprintf(stdout, ...).

```
int sprintf(char *s, const char *format, ...)
```

Функция sprintf работает так же, как и printf, только вывод выполняет в строку s, завершая ее символом '\0'. Строка s должна быть достаточно большой, чтобы вмещать результат вывода. Возвращает количество записанных символов без учета '\0'.

```
int vprintf (const char *format, va_list arg)
int vfprintf (FILE *stream, const char *format, va_list
arg)
int vsprintf (char *s, const char *format, va_list arg)
```

Функции vprintf, vfprintf и vsprintf эквивалентны соответствующим функциям printf с той разницей, что переменный список аргументов в них представлен параметром arg, инициализированным с помощью макроса va_start и, возможно, вызовами va_arg (см. описание файла <stdarg.h> в разделе Б.7,).

Б.1.3. Форматированный ввод

Для ввода данных с преобразованием их в заданный формат используются функции семейства scanf.

`int fscanf(FILE *stream, const char *format, ...)`

Функция `fscanf` считывает данные из потока `stream` под управлением строки формата `format` и присваивает преобразованные значения последующим аргументам, *каждый из которых должен быть указателем*. Функция завершает работу, когда исчерпывается строка формата `format`. Она возвращает `EOF`, если до преобразования формата ей встречается конец файла или появилась ошибка. В противном случае функция возвращает количество введенных и помещенных по назначению элементов данных.

Строка формата обычно содержит спецификации ввода, которые используются для правильной интерпретации вводимых данных. В эту строку могут входить следующие элементы:

- пробелы или табуляции, игнорируемые функцией;
- обычные символы (кроме `%`), которые должны совпасть с соответствующими символами в потоке ввода, отличными от символов пустого пространства;
- спецификации ввода, состоящие из знака `%`; необязательного символа `*`, запрещающего присваивание; необязательной ширины поля; необязательного символа `h`, `l` или `L`, уточняющего размер заполняемой переменной; и самого символа формата.

Спецификация ввода определяет преобразование следующего поля ввода. Обычно результат помещается в переменную, на которую указывает соответствующий аргумент. Но если присваивание запрещено с помощью знака `*`, как, например, в спецификации `%*s`, то поле ввода пропускается и никакого присваивания не происходит. Поле ввода — это строка символов, отличных от символов пустого пространства; ввод прекращается, как только встретится символ пустого пространства или как только исчерпается заданная ширина поля (если она задана). Из этого следует, что `scanf` может переходить через границы строк, поскольку символ новой строки является символом пустого пространства. (В число которых также входят символы пробела, табуляции, конца строки, возврата каретки, вертикальной табуляции и прогона страницы.)

Символ формата задает способ интерпретации поля ввода. Соответствующий аргумент должен быть указателем. Список допустимых символов формата приводится в табл. Б.2.

Перед символами формата `d`, `i`, `n`, `o`, `u` и `x` может стоять `h`, если аргумент — указатель на `short`, а не `int`, или `l`, если аргумент является указателем на `long`. Символам формата `e`, `f` и `g` может предшествовать `l`, если аргумент — указатель на `double`, а не `float`, или `L`, если аргумент — указатель на `long double`.

Таблица Б.2. Спецификации ввода функции `scanf`

Символ	Входные данные и тип аргумента
`d`	Десятичное целое число; `int *`
`i`	Целое число; `int *`. Число может быть восьмеричным (с нулем в начале) или шестнадцатеричным (с `0x` или `0X` в начале)
`o`	Восьмеричное целое число (с нулем в начале или без него); `int *`
`u`	Десятичное целое число без знака; `unsigned int *`
`x`	Шестнадцатеричное целое число (с `0x` или `0X` в начале или без них); `int *`
`c`	Символ; `char *`. Очередные символы из входного потока помещаются в указанный массив в количестве, заданном шириной поля; по умолчанию это количество равно 1. Символ `'\0'` не добавляется. Символы пустого пространства здесь рассматриваются как обычные и поэтому не пропускаются. Для ввода следующего символа *после* пустого пространства используется `%1s`

Символ	Входные данные и тип аргумента
`s`	Строка символов, отличных от пустого пространства (без кавычек); параметр типа `char *` должен указывать на массив достаточного размера, чтобы вместить строку и добавляемый к ней символ `'\0'`
`e, f, g`	Вещественное число; `float *`. Формат ввода для `float` состоит из необязательного знака, строки цифр (возможно, с десятичной точкой) и необязательного показателя степени, состоящего из `E` или `e` и целого числа (возможно, со знаком)
`p`	Значение указателя в виде, в котором его выводит `printf ("%p"); void *`
`n`	Записывает в аргумент количество символов, введенных к текущему моменту текущим вызовом `scanf`; `int *`. Ввод из потока не выполняется. Счетчик количества введенных элементов не увеличивается
`[...]`	Выбирает из потока ввода самую длинную непустую строку из символов, заданных в квадратных скобках; `char *`. В конец строки добавляется `'\0'`. Спецификация вида `[]...]` включает `]` в задаваемое множество символов
`[^...]`	Выбирает из потока ввода самую длинную непустую строку из символов, не входящих в заданное в скобках множество. В конец строки добавляется `'\0'`. Спецификация вида `[^]...]` включает `]` в заданное множество
`%`	Символ `%`; присваивание не выполняется

`int scanf (const char *format, ...)`

Функция `scanf(...)` эквивалентна `fscanf(stdin, ...)`.

`int sscanf (const char *s, const char *format, ...)`

Функция `sscanf(s, ...)` эквивалентна `scanf(...)`, только ввод выполняется из строки `s`.

Б.1.4. Функции ввода-вывода символов

`int fgetc(FILE *stream)`

Функция `fgetc` возвращает следующий символ из потока `stream` в виде `unsigned char` (преобразованным в `int`) или `EOF` в случае достижения конца файла или обнаружения ошибки.

`char *fgets(char *s, int n, FILE *stream)`

Функция `fgets` вводит не более `n-1` следующих символов в массив `s`, прекращая ввод, если встретится символ конца строки, который включается в массив; кроме того, в массив записывается `'\0'`. Функция возвращает `s` или `NULL` в случае достижения конца файла или обнаружения ошибки.

`int fputc(int c, FILE *stream)`

Функция `fputc` записывает символ `c` (преобразованный в `unsigned char`) в поток `stream`. Возвращает записанный символ или `EOF`, если обнаружена ошибка.

int fputs(const char *s, FILE *stream)

Функция fputs записывает строку s (которая может не содержать '\n') в поток stream; возвращает неотрицательное целое число или EOF, если обнаружена ошибка.

int getc(FILE *stream)

Функция getc эквивалентна fgetc с тем отличием, что если она определена как макрос, то может обращаться к потоку stream несколько раз.

int getchar(void)

Функция getchar() эквивалентна getc(stdin).

char *gets(char *s)

Функция gets считывает следующую строку из потока ввода в массив s, заменяя символ конца строки на '\0'. Возвращает s при нормальном завершении или NULL, если достигнут конец файла или обнаружена ошибка.

int putc(int c, FILE *stream)

Функция putc эквивалентна fputc с тем отличием, что если она определена как макрос, то может обращаться к потоку stream несколько раз.

int putchar(int c)

Функция putchar(c) эквивалентна putc(c, stdout).

int puts(const char *s)

Функция puts записывает строку s и символ конца строки в stdout. Возвращает EOF в случае ошибки или неотрицательное число при нормальном завершении.

int ungetc(int c, FILE *stream)

Функция ungetc возвращает символ c (преобразованный в unsigned char) в поток stream; при следующем чтении из stream он будет получен повторно. Для каждого потока гарантированно вернуть можно не больше одного символа. Нельзя возвращать EOF. Функция возвращает отправленный назад в поток символ или, если обнаружена ошибка, EOF.

Б.1.5. Функции прямого ввода-вывода

size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)

Функция fread считывает из потока stream в массив ptr не больше nobj объектов размера size. Она возвращает количество считанных объектов, которое может оказаться меньше запрашиваемого. Для определения успешного завершения или ошибки при чтении следует использовать feof и ferror.

size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)

Функция fwrite записывает nobj объектов размера size из массива ptr в поток stream; она возвращает число записанных объектов, которое в случае ошибки будет меньше nobj.

Б.1.6. Функции позиционирования в файлах

int fseek(FILE *stream, long offset, int origin)

Функция fseek устанавливает файловый указатель в заданную позицию в потоке stream; последующее чтение или запись будет производиться с этой позиции. Для двоичного файла позиция устанавливается со смещением offset символов относительно точки отсчета, задаваемой параметром origin: начала, если origin равен SEEK_SET; текущей позиции, если origin равен SEEK_CUR; и конца файла, если origin равен SEEK_END. Для текстового файла параметр offset должен быть нулем или значением, полученным с помощью вызова функции ftell (соответственно origin должен быть равен SEEK_SET). Функция возвращает ненулевое число в случае ошибки.

long ftell(FILE *stream)

Функция ftell возвращает текущую позицию указателя в потоке stream, или -1L, если обнаружена ошибка.

void rewind(FILE *stream)

Функция rewind(fp) эквивалентна fseek(fp, 0L, SEEK_SET); clearerr(fp).

int fgetpos(FILE *stream, fpos_t *ptr)

Функция fgetpos записывает текущую позицию в потоке stream в *ptr для последующего использования ее в функции fsetpos. Тип fpos_t позволяет хранить значения такого рода. В случае ошибки функция возвращает ненулевое число.

int fsetpos(FILE *stream, const fpos_t *ptr)

Функция fsetpos устанавливает позицию в потоке stream, читая ее из аргумента *ptr, куда она была записана ранее с помощью функции fgetpos. В случае ошибки функция возвращает ненулевое число.

Б.1.7. Функции обработки ошибок

Многие функции библиотеки в случае обнаружения ошибки или достижения конца файла устанавливают индикаторы состояния, которые можно изменять и проверять. Кроме того, целочисленное выражение errno (объявленное в <errno.h>) может содержать номер ошибки, который дает дополнительную информацию о последней из случившихся ошибок.

`void clearerr(FILE *stream)`

Функция `clearerr` сбрасывает индикаторы конца файла и ошибок потока `stream`.

`int feof(FILE *stream)`

Функция `feof` возвращает ненулевое значение, если для потока `stream` установлен (включен) индикатор конца файла.

`int ferror(FILE *stream)`

Функция `ferror` возвращает ненулевое значение, если для потока `stream` установлен (включен) индикатор ошибки.

`void perror(const char *s)`

Функция `perror(s)` выводит `s` и сообщение об ошибке (зависящее от реализации языка), которое соответствует целому числу в `errno`, т.е. делает то же, что и оператор

`fprintf(stderr, "%s: %s\n", s, "`*сообщение об ошибке*`")`

Функция `strerror` описывается в разделе Б.3.

Б.2. Анализ и классификация символов: `<ctype.h>`

В заголовочном файле `<ctype.h>` объявляются функции, предназначенные для анализа символов. Аргумент каждой из них имеет тип `int` и должен представлять собой либо `EOF`, либо `unsigned char`, приведенный к `int`; возвращаемое значение тоже имеет тип `int`. Функции возвращают ненулевое значение (*истина*), если аргумент `c` удовлетворяет соответствующему условию или принадлежит указанному классу символов, и нуль (*ложь*) в противном случае.

`isalnum(c)`	Истинно `isalpha(c)` или `isdigit(c)`
`isalpha(c)`	Истинно `isupper(c)` или `islower(c)`
`iscntrl(c)`	Управляющий символ
`isdigit(c)`	Десятичная цифра
`isgraph(c)`	Отображаемый символ, за исключением пробела
`islower(c)`	Буква нижнего регистра
`isprint(c)`	Отображаемый символ, в том числе пробел
`ispunct(c)`	Отображаемый символ, за исключением пробела, буквы или цифры
`isspace(c)`	Пробел, прогон страницы, конец строки, возврат каретки, табуляция, вертикальная табуляция
`isupper(c)`	Буква верхнего регистра
`isxdigit(c)`	Шестнадцатеричная цифра

В семибитовом символьном наборе ASCII отображаемые символы занимают диапазон от `0x20` (`' '`) до `0x7E` (`'~'`); управляющие символы — от `0` (`NUL`) до `0x1F` (`US`), а также `0x7F` (`DEL`).

Помимо перечисленных, есть еще две функции, изменяющие регистр букв:

- `int tolower(int c)` переводит `c` в нижний регистр;
- `int toupper(int c)` переводит `c` в верхний регистр.

Если `c` — буква в верхнем регистре, функция `tolower(c)` возвратит эту букву в нижнем регистре; в противном случае она вернет `c` без изменений. Если `c` — буква в нижнем регистре, функция `toupper(c)` возвратит эту букву в верхнем регистре; в противном случае она вернет `c` без изменений.

Б.3. Функции для работы со строками: <string.h>

Имеются две группы функций для работы со строками, определенных в заголовочном файле `<string.h>`. Имена функций первой группы начинаются с `str`, а второй — с `mem`. За исключением `memmove`, результат работы функций не определен, если выполняется копирование объектов, перекрывающихся в памяти. Функции сравнения воспринимают аргументы как массивы элементов типа `unsigned char`.

В следующей таблице переменные `s` и `t` принадлежат к типу `char *`, `cs` и `ct` — к типу `const char *`, `n` — к типу `size_t`, а `c` является числом типа `int`, приведенным к типу `char`.

`Char *strcpy(s,ct)`	Копирует строку `ct` в строку `s`, включая `'\0'`; возвращает `s`
`char *strncpy(s,ct,n)`	Копирует не более `n` символов строки `ct` в `s`; возвращает `s`. Дополняет результат символами `'\0'`, если в `ct` меньше `n` символов
`char *strcat(s,ct)`	Присоединяет `ct` в конец строки `s`; возвращает `s`
`char *strncat(s,ct,n)`	Присоединяет не более `n` символов строки `ct` к `s`, завершая `s` символом `'\0'`; возвращает `s`
`int strcmp(cs,ct)`	Сравнивает строки `cs` и `ct`; возвращает `<0`, если `cs<ct`; `0`, если `cs==ct`; и `>0`, если `cs>ct`
`int strncmp(cs,ct,n)`	Сравнивает не более `n` символов строк `cs` и `ct`; возвращает `<0`, если `cs<ct`; `0`, если `cs==ct`; и `>0`, если `cs>ct`
`char *strchr(cs,c)`	Возвращает указатель на первое вхождение `c` в `cs` или `NULL` при отсутствии такового
`char *strrchr(cs,c)`	Возвращает указатель на последнее вхождение `c` в `cs` или `NULL` при отсутствии такового
`size_t strspn(cs,ct)`	Возвращает длину начального участка `cs`, состоящего из символов, которые входят в строку `ct`
`size_t strcspn(cs,ct)`	Возвращает длину начального участка `cs`, состоящего из символов, которые *не входят* в строку `ct`
`char *strpbrk(cs,ct)`	Возвращает указатель на первый символ в строке `cs`, который совпадает с одним из символов, входящих в строку `ct`, или `NULL`, если такого не оказалось
`char *strstr(cs,ct)`	Возвращает указатель на первое вхождение строки `ct` в строку `cs` или `NULL`, если не найдено ни одного
`size_t strlen(cs)`	Возвращает длину строки `cs`

`char *strerror(n)`	Возвращает указатель на строку, соответствующую номеру ошибки `n` (зависит от реализации)
`char *strtok(s,ct)`	Ищет в строке `s` лексемы, ограниченные символами из строки `ct`; более подробное описание см. ниже

Последовательность вызовов `strtok(s,ct)` разбивает строку `s` на лексемы, ограничителем которых служит любой символ из строки `ct`. При первом вызове в этой последовательности указатель `s` не равен NULL. Функция находит в строке `s` первую лексему, состоящую из символов, которые не входят в `ct`; она заканчивает работу тем, что записывает поверх следующего символа `s` символ `'\0'` и возвращает указатель на лексему. Каждый последующий вызов, в котором указатель `s` равен NULL, возвращает указатель на следующую такую лексему, которую функция будет искать сразу после конца предыдущей. Функция `strtok` возвращает NULL, если далее не обнаруживает никаких лексем. Параметр `ct` от вызова к вызову может меняться.

Функции семейства `mem...` предназначены для манипулирования объектами как массивами символов; они образуют интерфейс к быстродействующим системным функциям. В приведенной ниже таблице параметры `s` и `t` относятся к типу `void *`; `cs` и `ct` — к типу `const void *`; `n` — к типу `size_t`; а параметр `c` представляет собой число типа `int`, приведенное к типу `char`.

`void *memcpy(s,ct,n)`	Копирует `n` символов из `ct` в `s` и возвращает `s`
`void *memmove(s,ct,n)`	Делает то же самое, что и `memcpy`, но работает корректно также в случае перекрывающихся в памяти объектов
`int memcmp(cs,ct,n)`	Сравнивает первые `n` символов `cs` и `ct`; возвращает те же результаты, что и функция `strcmp`
`void *memchr(cs,c,n)`	Возвращает указатель на первое вхождение символа `c` в `cs` или NULL, если он отсутствует среди первых `n` символов
`void *memset(s,c,n)`	Помещает символ `c` в первые `n` позиций массива `s` и возвращает `s`

Б.4. Математические функции: <math.h>

В заголовочном файле <math.h> объявляются математические функции и макросы.

Макросы EDOM и ERANGE (определенные в <errno.h>) — это отличные от нуля целочисленные константы, используемые для индикации ошибки области определения и ошибки выхода за диапазон; HUGE_VAL представляет собой положительное число типа `double`. *Ошибка области определения* возникает, если аргумент выходит за пределы числовой области, в которой определена функция. При возникновении такой ошибки переменной `errno` присваивается значение EDOM; возвращаемое значение зависит от реализации. *Ошибка выхода за диапазон* возникает тогда, когда результат функции нельзя представить в виде `double`. В случае переполнения функция возвращает HUGE_VAL с правильным знаком, и в `errno` помещается значение ERANGE. Если происходит потеря значимости (результат оказывается меньше, чем можно представить данным типом), функция возвращает нуль; получает ли в этом случае переменная `errno` значение ERANGE — зависит от реализации.

В таблице, приведенной ниже, параметры x и y имеют тип double, n — тип int, и все функции возвращают значения типа double. Углы для тригонометрических функций задаются в радианах.

`sin(x)`	Синус x
`cos(x)`	Косинус x
`tan(x)`	Тангенс x
`asin(x)`	Арксинус x в диапазоне $[-\pi/2, \pi/2]$, $x \in [-1,1]$
`acos(x)`	Арккосинус x в диапазоне $[0, \pi]$, $x \in [-1,1]$
`atan(x)`	Арктангенс x в диапазоне $[-\pi/2, \pi/2]$
`atan2(y,x)`	Арктангенс y/x в диапазоне $[-\pi, \pi]$
`sinh(x)`	Гиперболический синус x
`cosh(x)`	Гиперболический косинус x
`tanh(x)`	Гиперболический тангенс x
`exp(x)`	Экспоненциальная функция e^x
`log(x)`	Натуральный логарифм $\ln(x)$, $x > 0$
`log10(x)`	Десятичный логарифм $\lg(x)$, $x > 0$
`pow(x,y)`	x^y; возникает ошибка области определения, если x = 0 и y ≤ 0 или если $x < 0$ и y – не целое
`sqrt(x)`	Квадратный корень x, $x \geq 0$
`ceil(x)`	Наименьшее целое число в формате `double`, не меньшее x
`floor(x)`	Наибольшее целое число в формате `double`, не превосходящее x
`fabs(x)`	Абсолютное значение $\lvert x \rvert$
`ldexp(x, n)`	$x \cdot 2^n$
`frexp(x, int *exp)`	Разбивает x на два сомножителя, первый из которых — нормализованная дробь в интервале [1/2, 1), возвращаемая функцией, а второй — степень двойки; это число помещается в `*exp`. Если x равен нулю, то обе части результата равны нулю
`modf(x, double *ip)`	Разбивает x на целую и дробную части, обе с тем же знаком, что и x. Целая часть помещается в `*ip`, а дробная часть возвращается из функции
`fmod(x, y)`	Остаток от деления x на y в виде вещественного числа. Знак результата совпадает со знаком x. Если y равен нулю, результат зависит от реализации языка

Б.5. Вспомогательные функции: <stdlib.h>

В заголовочном файле <stdlib.h> объявлены функции, предназначенные для преобразования чисел, распределения памяти и других подобных задач.

double atof(const char *s)

Функция atof преобразует строку s в double; эквивалентна strtod(s, (char**) NULL).

int atoi(const char *s)

Функция atoi преобразует строку s в int; эквивалентна (int)strtol(s, (char**)NULL, 10).

int atol(const char *s)

Функция atol преобразует строку s в long; эквивалентна strtol(s, (char**) NULL, 10).

double strtod(const char *s, char **endp)

Функция strtod преобразует первые символы строки s в double, игнорируя пустое пространство в начале; помещает указатель на не преобразованную часть в *endp (если endp не NULL). При переполнении она возвращает HUGE_VAL с соответствующим знаком; в случае потери значимости (результат слишком мал для его представления данным типом) возвращается 0; в обоих случаях переменная errno устанавливается равной ERANGE.

long strtol(const char *s, char **endp, int base)

Функция strtol преобразует первые символы строки s в long, игнорируя пустое пространство в начале; она помещает указатель на не преобразованную часть в *endp (если endp не NULL). Если base находится в диапазоне от 2 до 36, то преобразование выполняется в предположении, что число в строке записано по основанию base. Если base равно нулю, то основанием числа считается 8, 10 или 16. Число, начинающееся с цифры 0, предполагается восьмеричным, а с 0x или 0X — шестнадцатеричным. Цифры от 10 до base-1 записываются первыми буквами латинского алфавита в любом регистре. При основании 16 в начале числа разрешается помещать 0x или 0X. В случае переполнения функция возвращает LONG_MAX или LONG_MIN в зависимости от знака результата, а в errno помещается код ошибки ERANGE.

unsigned long strtoul(const char *s, char **endp, int base)

Функция strtoul работает так же, как и strtol, с той разницей, что она возвращает результат типа unsigned long, а в случае ошибки — ULONG_MAX.

int rand(void)

Функция rand возвращает псевдослучайное число в диапазоне от 0 до RAND_MAX; значение RAND_MAX — не меньше 32767.

void srand(unsigned int seed)

Функция srand использует параметр seed как инициализирующее значение для новой последовательности псевдослучайных чисел. Вначале параметр seed равен 1.

void *calloc(size_t nobj, size_t size)

Функция calloc возвращает указатель на место в памяти, отведенное для массива nobj объектов, каждый из которых имеет размер size или NULL, если запрос на память выполнить нельзя. Выделенная область памяти обнуляется.

`void *malloc(size_t size)`

Функция `malloc` возвращает указатель на место в памяти для объекта размера `size`, или `NULL`, если запрос невыполним. Выделенная область памяти не инициализируется.

`void *realloc(void *p, size_t size)`

Функция `realloc` изменяет размер объекта, на который указывает `p`, на заданный `size`. Для участка объекта, длина которого равна меньшему из старого и нового размеров, содержимое не изменяется. Если новый размер больше старого, дополнительное пространство не инициализируется. Функция возвращает указатель на новый участок памяти или `NULL`, если запрос невыполним (в этом случае `*p` не изменяется).

`void free(void *p)`

Функция `free` освобождает область памяти, на которую указывает `p`; она не делает ничего, если `p` равен `NULL`. Переменная `p` должна указывать на область памяти, ранее выделенную одной из функций `calloc`, `malloc` или `realloc`.

`void abort(void)`

Функция `abort` вызывает аварийное завершение программы, в точности как по вызову `raise(SIGABRT)`.

`void exit(int status)`

Функция `exit` вызывает нормальное завершение программы. При этом функции, зарегистрированные с помощью `atexit`, выполняются в порядке, обратном регистрации. Записываются и очищаются буферы открытых файлов, закрываются открытые потоки, и управление возвращается в операционную среду. Какие значения аргумента `status` передавать в среду — зависит от реализации, однако нуль общепринят как сигнал успешного завершения программы. Можно также использовать значения `EXIT_SUCCESS` (успешное завершение) и `EXIT_FAILURE` (ошибочное завершение).

`int atexit(void (*fcn)(void))`

Функция `atexit` регистрирует функцию `fcn` для вызова при нормальном завершении программы; возвращает ненулевое значение, если регистрация невыполнима.

`int system(const char *s)`

Функция `system` передает строку `s` в операционную среду для выполнения. Если `s` равна `NULL` и при этом командный процессор среды существует, то `system` возвращает ненулевое значение. Если `s` — не `NULL`, то возвращаемое значение зависит от реализации.

`char *getenv(const char *name)`

Функция `getenv` возвращает строку среды, ассоциированную с `name`, или `NULL`, если такой строки не существует. Детали зависят от реализации.

```
void *bsearch(const void *key, const void *base,
      size_t n, size_t size,
      int (*cmp)(const void *keyval, const void *datum))
```

Функция bsearch ищет среди base[0]...base[n-1] элемент, соответствующий ключу поиска *key. Функция сравнения cmp должна возвращать отрицательное число, если ее первый аргумент (ключ поиска) меньше второго (записи в таблице), нуль в случае их равенства и положительное число, если ключ поиска больше. Элементы массива base должны быть упорядочены по возрастанию. Функция возвращает указатель на элемент с совпавшим ключом или NULL, если ключ не найден.

```
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *, const void *))
```

Функция qsort сортирует массив base[0]...base[n-1] объектов размера size в порядке возрастания. Функция сравнения cmp должна иметь те же свойства, что и в описании bsearch.

```
int abs(int n)
```

Функция abs возвращает абсолютное значение своего аргумента типа int.

```
long labs(long n)
```

Функция labs возвращает абсолютное значение своего аргумента типа long.

```
div_t div(int num, int denom)
```

Функция div_t вычисляет частное и остаток от деления числителя num на знаменатель denom. Результаты запоминаются в элементах quot и rem типа int структуры типа div_t.

```
ldiv_t ldiv(long num, long denom)
```

Функция ldiv вычисляет частное и остаток от деления num на denom. Результаты запоминаются в элементах quot и rem типа long структуры ldiv_t.

Б.6. Диагностика: <assert.h>

Макрос assert используется для включения в программу средств диагностики.

```
void assert (int выражение)
```

Если заданное *выражение* имеет значение 0 во время выполнения оператора assert(*выражение*), то в поток stderr будет выведено сообщение примерно следующего вида:

Assertion failed: *выражение*, file *имя_файла*, line *nnn*

После этого будет вызвана функция abort для завершения работы. Имя исходного файла и номер строки берутся из макросов препроцессора __FILE__ и __LINE__.

Если в момент включения файла <assert.h> имя NDEBUG является определенным, то макрос assert игнорируется.

Б.7. Переменные списки аргументов: <stdarg.h>

Заголовочный файл <stdarg.h> предоставляет программисту средства для перебора аргументов функции, количество и типы которых заранее не известны.

Пусть *посл_арг* — последний именованный параметр функции f с переменным количеством аргументов. Внутри f необходимо объявить переменную ap типа va_list, содержащую указатель на очередной аргумент:

```
va_list ap;
```

Необходимо один раз инициализировать ap, обратившись к макросу va_start, чтобы получить доступ к безымянным аргументам:

```
va_start(va_list ap, посл_арг);
```

С этого момента каждое обращение к макросу va_arg будет выдавать значение очередного безымянного аргумента с указанным типом, и всякий раз указатель ap будет получать приращение, чтобы при следующем вызове va_arg давать следующий аргумент:

```
тип va_arg(va_list ap, тип);
```

После перебора всех аргументов, но до выхода из функции f необходимо один раз выполнить следующий макрос:

```
void va_end(va_list ap);
```

Б.8. Нелокальные переходы: <setjmp.h>

Объявления в файле <setjmp.h> дают возможность отклониться от обычной последовательности вызовов и возвратов из функций — как правило, чтобы сразу вернуться из глубоко вложенного вызова функции на верхний уровень.

int setjmp(jmp_buf env)

Макрос setjmp сохраняет информацию о текущем состоянии вызовов в переменной env для последующего ее использования в функции longjmp. Возвращает нуль при прямом вызове setjmp и не нуль при всех последующих вызовах longjmp. Обращение к setjmp возможно только в определенном контексте; в основном это проверки условий в операторах if, switch и циклах, причем только в простых сравнениях.

```
if (setjmp() == 0)
    /* после возврата при прямом вызове */
else
    /* после возврата из longjmp */
```

void longjmp(jmp_buf env, int val)

Функция longjmp восстанавливает состояние, информация о котором была сохранена при последнем вызове setjmp в переменной env; выполнение программы продолжается так, как если бы функция setjmp только что закончила работу и вернула ненулевое значение val. Функция, содержащая вызов setjmp, не должна завершиться к моменту обращения к longjmp. Доступные ей объекты имеют те значения, которые они имели в момент вызова longjmp. Исключение составляют автоматические переменные без модификатора volatile в функции, вызвавшей setjmp: они становятся неопределенными, если подвергались модификации после вызова setjmp.

Б.9. Сигналы: <signal.h>

Заголовочный файл <signal.h> содержит средства для обработки исключительных ситуаций, таких как сигнал прерывания от внешнего источника или ошибка выполнения программы.

void (*signal(int sig, void (*handler)(int)))(int)

Функция signal устанавливает, как должны обрабатываться последующие сигналы. Если параметр handler имеет значение SIG_DFL, то используется обработка по умолчанию (детали зависят от реализации); если значение равно SIG_IGN, то сигнал игнорируется; в остальных случаях вызывается функция, на которую указывает handler, с типом сигнала в качестве аргумента. Имеются следующие типы сигналов.

SIGABRT	Аварийное завершение, например от функции abort
SIGFPE	Арифметическая ошибка, например деление на 0 или переполнение
SIGILL	Ошибка в теле функции, например недопустимая инструкция
SIGINT	Интерактивное системное обращение, например прерывание
SIGSEGV	Неразрешенное обращение к памяти, например выход за ее пределы
SIGTERM	Запрос на завершение, посланный в программу

Функция signal возвращает предыдущее значение handler для соответствующего сигнала или SIGERR в случае возникновения ошибки.

Когда в дальнейшем поступает сигнал sig, вначале восстанавливается его обработка по умолчанию. После этого вызывается функция-обработчик, заданная в параметре handler, т.е. как бы выполняется вызов (*handler)(sig). После возврата из функции handler работа программы возобновляется с того места, при выполнении которого поступил сигнал.

Начальное состояние сигналов зависит от реализации.

int raise(int sig)

Функция raise посылает в программу сигнал sig. В случае неудачи возвращает ненулевое значение.

Б.10. Функции даты и времени: <time.h>

В заголовочном файле <time.h> объявляются типы и функции для работы с датой и временем. Некоторые функции работают с *местным временем*, которое может отличаться от календарного, например в связи с часовыми поясами. Определены арифметические типы clock_t и time_t для представления времени, а структура struct tm содержит компоненты календарного времени.

int tm_sec;	Секунды от начала минуты (0,61)
int tm_min;	Минуты от начала часа (0,59)
int tm_hour;	Часы от полуночи (0,23)
int tm_mday;	Число месяца (1,31)
int tm_mon;	Месяцы *после* января (0,11)
int tm_year;	Годы с 1900
int tm_wday;	Дни с воскресенья (0,6)
int tm_yday;	Дни с 1 января (0,365)
int tm_isdst;	Признак летнего времени

Поле tm_isdst имеет положительное значение, если активен режим летнего времени, нуль в противном случае и отрицательное значение, если информация о сезоне времени недоступна.

clock_t clock(void)

Функция clock возвращает время, измеряемое процессором в тактах от начала выполнения программы, или -1, если оно не известно. Пересчет этого времени в секунды выполняется по формуле clock() / CLOCKS_PER_SEC.

time_t time(time_t *tp)

Функция time возвращает текущее календарное время или -1, если время не известно. Если указатель tp не равен NULL, возвращаемое значение записывается также и в *tp.

double difftime(time_t time2, time_t time1)

Функция difftime возвращает разность time2-time1, выраженную в секундах.

time_t mktime(struct tm *tp)

Функция mktime преобразует местное время, заданное структурой *tp, в календарное и возвращает его в том же виде, что и функция time. Компоненты структуры будут иметь значения в указанных выше диапазонах. Функция возвращает календарное время или -1, если оно не представимо.

Следующие четыре функции возвращают указатели на статические объекты, каждый из которых может быть модифицирован другими вызовами.

char *asctime(const struct tm *tp)

Функция asctime преобразует время из структуры *tp в строку вида

```
Sun Jan  3 15:14:13 1988\n\0
```

char *ctime(const time_t *tp)

Функция ctime преобразует календарное время в местное, что эквивалентно вызову функции asctime(localtime(tp)).

struct tm *gmtime(const time_t *tp)

Функция gmtime преобразует календарное время в так называемое "скоординированное универсальное время" (*Coordinated Universal Time — UTC*). Она возвращает NULL, если UTC не известно. Имя этой функции сложилось исторически и означает *Greenwich Mean Time* (среднее гринвичское время).

struct tm *localtime(const time_t *tp)

Функция localtime преобразует календарное время *tp в местное.

size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)

Функция strftime форматирует информацию о дате и времени из *tp и помещает ее в строку s согласно строке формата fmt, аналогичной той, которая используется в функции printf. Обычные символы (включая завершающий символ '\0') копируются в s. Каждая пара, состоящая из % и буквы, заменяется, как описано ниже, с использованием значений по форме, соответствующей конкретной культурной среде. В строку s помещается не более smax символов. Функция возвращает количество символов без учета '\0' или нуль, если число сгенерированных символов больше smax.

- %a Сокращенное название дня недели
- %A Полное название дня недели
- %b Сокращенное название месяца
- %B Полное название месяца
- %c Местное представление даты и времени
- %d День месяца (01-31)
- %H Час (по 24-часовому времени) (00-23)
- %I Час (по 12-часовому времени) (01-12)
- %j День от начала года (001-366)
- %m Месяц (01-12)
- %M Минута (00-59)
- %p Местное представление времени до и после полудня (AM и PM)
- %S Секунда (00-61)
- %U Неделя от начала года (считая первым днем недели воскресенье) (от 00 до 53)
- %w День недели (0-6, номер воскресенья равен 0)
- %W Номер недели от начала года (считая первым днем недели понедельник) (от 00 до 53)
- %x Местное представление даты

%X	Местное представление времени
%y	Год без указания века (00-99)
%Y	Год с указанием века
%Z	Название часового пояса, если есть
%%	Символ %

Б.11. Системно-зависимые константы: <limits.h> и <float.h>

В заголовочном файле <limits.h> определяются константы, описывающие размеры целочисленных типов. Ниже приведены минимально допустимые величины, а в конкретных реализациях возможны значения, большие указанных.

CHAR_BIT	8	Битов в char
SCHAR_MAX	UCHAR_MAX *или* SCHAR_MAX	Максимальное значение char
CHAR_MIN	0 *или* CHAR_MIN	Минимальное значение char
INT_MAX	+32767	Максимальное значение int
INT_MIN	-32767	Минимальное значение int
LONG_MAX	+2147483647	Максимальное значение long
LONG_MIN	-2147483647	Минимальное значение long
SCHAR_MAX	+127	Максимальное значение signed char
SCHAR_MIN	-127	Минимальное значение signed char
SHRT_MAX	+32767	Максимальное значение short
SHRT_MIN	-32767	Минимальное значение short
UCHAR_MAX	255	Максимальное значение unsigned char
UINT_MAX	65535	Максимальное значение unsigned int
ULONG_MAX	4294967295	Максимальное значение unsigned long
USHRT_MAX	65535	Максимальное значение unsigned short

Имена, приведенные в следующей таблице, взяты из файла <float.h> и являются константами для использования в вещественной арифметике с плавающей точкой[1]. В тех случаях, когда приводятся значения констант, они представляют собой минимально допустимые величины соответствующих параметров. Различные реализации библиотеки устанавливают свои значения.

FLT_RADIX	2	Основание для экспоненциальной формы представления, например 2, 16
FLT_ROUNDS		Режим округления при сложении чисел с плавающей точкой
FLT_DIG	6	Точность — количество десятичных цифр

[1] Имена с префиксом FLT относятся к величинам типа float, а с префиксом DBL — к величинам типа double. — *Примеч. ред.*

FLT_EPSILON	1E-5	Наименьшее число x такое, что $1.0 + x \neq 1.0$
FLT_MANT_DIG		Количество цифр по основанию FLT_RADIX в мантиссе
FLT_MAX	1E+37	Наибольшее число с плавающей точкой
FLT_MAX_EXP		Наибольшее n такое, что $FLT\_RADIX^n - 1$ представимо
FLT_MIN	1E-37	Наименьшее нормализованное число с плавающей точкой
FLT_MIN_EXP		Наименьшее n, такое, что 10^n — нормализованное число
DBL_DIG	10	Количество значащих десятичных цифр
DBL_EPSILON	1E-9	Наименьшее x, такое, что $1.0 + x \neq 1.0$
DBL_MANT_DIG		Количество цифр по основанию FLT_RADIX в мантиссе
DBL_MAX	1E+37	Наибольшее число с плавающей точкой
DBL_MAX_EXP		Наибольшее n, такое, что $FLT\_RADIX^n - 1$ представимо
DBL_MIN	1E-37	Наименьшее нормализованное число с плавающей точкой
DBL_MIN_EXP		Наименьшее n, такое, что 10^n — нормализованное число

Приложение B

Сводка изменений

Со времени выхода в свет первого издания этой книги определение языка C подверглось изменениям. Практически все они представляли собой расширение исходного языка и проектировались как можно тщательнее, чтобы сохранить совместимость с существующей практикой; одни касались устранения двусмысленностей в исходном описании, а другие привели к изменению самой практики программирования на C. Многие из новых средств были заявлены в документации к компиляторам от AT&T, а затем приняты другими разработчиками компиляторов C. Не так давно комитет ANSI по стандартизации языка принял большинство этих изменений и ввел еще ряд существенных модификаций. Их отчет был частично предвосхищен некоторыми коммерческими компиляторами еще до выпуска формального стандарта C.

В этом приложении дается сводка различий между языком, описанным в первом издании этой книги, и той его версией, которая, по-видимому, будет принята окончательным стандартом. Здесь рассматривается только сам язык, а не его библиотека и системное окружение; хотя это и существенная часть стандарта, сравнивать тут особо нечего, поскольку в первом издании не было сделано никаких попыток предписать языку определенную среду и библиотеку.

- В стандарте более тщательно определена, а также расширена по сравнению с первым изданием препроцессорная обработка. Теперь она явно основана на системе лексем; введены новые операции для сцепления лексем (`##`) и создания строк (`#`); появились новые директивы `#elif` и `#pragma`; разрешено переопределение макросов теми же строками лексем; параметры в символьных строках не подвергаются замене. Разбиение директив с помощью обратной косой черты разрешено в любых случаях, а не только в символьных строках и макроопределениях. См. п. A.12.
- Минимальная длина значащей части всех внутренних идентификаторов увеличена до 31 символа; минимальная обязательная длина значащей части идентификаторов с внешним связыванием остается равной 6 символам (без различия регистра). Во многих реализациях языка разрешено больше.
- Введены управляющие комбинации из трех символов (*триграфы*), начинающиеся с `??`, позволяющие теперь представить отдельные символы, которых нет в некоторых символьных наборах. Определены управляющие комбинации для символов `#\^[]{}|~` см. п. A.12.1. Обратите внимание, что введение комбинаций из трех символов может изменить смысл строк, содержащих сочетание `??`.
- Введены новые ключевые слова (`void`, `const`, `volatile`, `signed`, `enum`). "Мертворожденное" ключевое слово `entry` удалено из языка.
- Определены новые управляющие последовательности для использования в символьных константах и строковых литералах. Действие символа `\` в сочетании с буквой, не входящей в число заданных стандартом, не определено. См. п. A.2.5.2.
- А это изменение пришлось по душе всем: `8` и `9` больше не являются восьмеричными цифрами.

- В стандарте введен более широкий набор суффиксов для явного определения типов констант: `U` и `L` для целых, `F` и `L` для вещественных. Также уточнены правила определения типа для констант без суффиксов (п. А.2.5).
- Записанные вплотную строковые литералы автоматически сцепляются в один.
- Введена система записи строковых литералов и символьных констант с расширенными символами; см. п. А.2.6.
- Символы, как и данные других типов, можно объявлять имеющими или не имеющими знак с помощью ключевых слов `signed` и `unsigned`. Конструкция `long float` как синоним `double` удалена из языка, но для работы с вещественными числами очень большой точности можно использовать тип `long double`.
- Тип `unsigned char` существовал и раньше, но теперь в стандарте появилось ключевое слово `signed` для явного обозначения наличия знака у переменной типа `char` или другого целочисленного объекта.
- Уже много лет в ряде реализаций был в наличии тип `void`. Стандарт предлагает тип `void *` в качестве основного нетипизированного указателя; раньше эту роль играл `char *`. В то же время введены правила, запрещающие смешивать в выражениях и конструкциях указатели и целые числа, а также указатели различных типов, без явного приведения к одному типу.
- Стандарт вводит требования к минимальному диапазону числовых типов и регламентирует обязательное наличие заголовочных файлов (`<limits.h>` и `<float.h>`) с характеристиками конкретных реализаций.
- Со времени первого издания книги в языке С появились перечислимые типы (`enum`).
- Стандарт заимствовал из языка C++ понятие модификатора типа — например, `const` (п. А.8.2).
- Строки теперь нельзя модифицировать, поэтому их можно хранить в памяти, доступной только для чтения.
- Изменены правила "обычных арифметических преобразований" (приведений типов). Ранее при приведении целых типов всегда доминировал `unsigned`, а вещественных — `double`. Теперь правило таково: результат приводится к наименьшему из достаточно вместительных типов (п. А.6.5).
- Старые операции присваивания наподобие `=+` исчезли из языка. Знаки составных операций присваивания теперь представляют собой единые неразрывные лексемы; в первом издании они считались парами символов и могли разделяться пробелами.
- У компилятора отобрано право обрабатывать математически ассоциативные операции как ассоциативные также и в вычислительном плане.
- Для симметрии с одноместной операцией "минус" (`-`) введена аналогичная операция "плюс" (`+`).
- Указатель на функцию может использоваться непосредственно как имя для ее вызова без знака `*`. См. п. А.7.3.2.
- Структуры можно присваивать, передавать в функции и возвращать из функций.
- Разрешается применение операции взятия адреса к массиву; результатом является указатель на массив.

- Операция `sizeof` в первом издании давала результат типа `int`; впоследствии во многих реализациях языка тип сменился на `unsigned`. В стандарте написано, что этот тип может зависеть от реализации, но при этом регламентируется, что в заголовочном файле `<stddef.h>` должен быть определен соответствующий тип `size_t`. Аналогичное изменение сделано в отношении типа `ptrdiff_t`, выражающего разницу между указателями. См. пп. А.7.4.8 и А.7.7.
- Операция взятия адреса `&` неприменима к объекту, объявленному с модификатором `register`, даже если компилятор решает не хранить объект в регистре.
- Тип результата операции сдвига совпадает с типом левого операнда; правый операнд не может расширить его тип. См. п. А.7.8.
- Стандарт разрешает создавать указатель сразу за концом массива и выполнять над ним арифметические операции и сравнение; см. п. А.7.7.
- Стандарт вводит понятие (заимствованное из C++) объявления прототипа функции, в котором содержатся типы аргументов и есть возможность различать функции с переменным количеством аргументов; вводится стандарт работы с такими функциями. См. пп. А.7.3.2, А.8.6.3, Б.7. Старый стиль также разрешен, но с некоторыми ограничениями.
- Пустые объявления, в которых нет идентификаторов переменных и не объявляется как минимум структура, объединение или перечисление, не разрешены стандартом. С другой стороны, объявление, содержащее только метку структуры или объединения, переопределяет эту метку, даже если она уже была объявлена в более широкой области действия.
- Запрещены внешние объявления переменных без спецификаций типа и модификаторов (т.е. объявления из одних идентификаторов переменных).
- В некоторых реализациях объявление `extern`, встречающееся во внутреннем блоке, распространяется на всю оставшуюся часть файла. В стандарте четко определено, что областью действия такого объявления является только блок.
- Область действия параметров вкладывается в составной оператор тела функции, так что объявление переменных на верхнем уровне функции не позволяет скрыть ее параметры.
- Пространства имен для идентификаторов несколько изменились. В стандарте все структурные метки отнесены к одному пространству имен, а метки переходов — к другому, отдельному; см. п. А.11.1. Имена элементов (членов, полей) ассоциированы с соответствующими структурами или объединениями. (Это уже давно общепринятая практика.)
- Объединения можно инициализировать; инициализирующее значение присваивается первому элементу.
- Автоматические структуры, объединения и массивы можно инициализировать, хотя и с ограничениями.
- Массивы символов с явно указанной длиной можно инициализировать строковыми литералами в точности той же длины (символ `'\0'` автоматически удаляется).
- Управляющее выражение и метки блоков `case` в операторе `switch` могут иметь любой целый тип.

Предметный указатель

Символы

A

B

C

D

E

F

G

H

I

L

M

O

P

Q

R

S

T

U

V

W

А

Б

В

Г

Д

Е

З

И

К

Л

М

О

П

P

C

T

У

Ф

X

Ц

Ч

Э

Научно-популярное издание

Брайан У. Керниган, Деннис М. Ритчи

Язык программирования C
2-е издание

Литературный редактор	*И.А. Попова*
Верстка	*А.Н. Полинчик*
Художественный редактор	*Е.П. Дынник*
Корректоры	*Л.А. Гордиенко*, *Л.В. Чернокозинская*

ООО "И.Д. Вильямс"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 14.07.2009. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 19,0. Уч.-изд. л. 16,7.
Доп. тираж 1000 экз. Заказ № 17376.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

ЯЗЫК ПРОГРАММИРОВАНИЯ С

второе издание

БРАЙАН КЕРНИГАН И ДЕННИС РИТЧИ

Из предисловия:

Мы постарались сохранить краткость первого издания. Язык С невелик по объему, и нет большого смысла писать о нем толстые книги. Мы улучшили изложение ключевых вопросов — таких как указатели, являющиеся центральным моментом в программировании на С. Мы доработали первоначальные примеры, а также добавили новые в некоторые из глав. Например, рассказ о сложных объявлениях дополнен программой преобразования деклараций в текстовые описания и наоборот.

Как и раньше, все примеры протестированы и отлажены непосредственно из текста, который подготовлен в электронном виде.

Как говорилось в предисловии к первому изданию, язык С "становится все удобнее по мере того, как растет опыт работы с ним". После нескольких десятилетий работы мы не изменили своего мнения. Надеемся, что эта книга поможет вам изучить С и пользоваться им как можно эффективнее.

ISBN 978-5-8459-0891-9

05161

9 785845 908919

ВИЛЬЯМС

Издательский дом "Вильямс"
www.williamspublishing.com

PH PTR

Prentice Hall P T R,
Englewood Cliffs, New Jersey 07632
www.prenhall.com