Б.Керниган
Д.Ритчи

Язык программирования Си

THE C PROGRAMMING LANGUAGE

Second Edition

Brian W. Kernighan • Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey

PRENTICE HALL, Englewood Cliffs, New Jersey 07632

Б. Керниган
Д. Ритчи

Язык программирования Си

Издание второе,
переработанное и дополненное

Перевод с английского
Вик. С. Штаркмана
Под редакцией и с предисловием
Вс. С. Штаркмана

Москва
"Финансы и статистика"
1992

ББК 24.4.9
И74

Керниган Б., Ритчи Д.

К36 Язык программирования Си: Пер. с англ. / Под ред. и с предисл. Вс. С. Штаркмана. – 2-е изд., перераб. и доп. – М.:Финансы и статистика, 1992. – 272с.: ил.

ISBN 5-279-00473-1.

Новое издание книги широко известных авторов, разработчиков языка Си, переработанное и дополненное с учётом подготовленного стандарта ANSI для языка Си. Включает введение в язык, подробный учебник по языку и справочное руководство для программистов.

Для широкого круга программистов, преподавателей и студентов.

К $\frac{2404090000–003}{010(01)–92}$ КБ 41 – 16 – 90 ББК 24.4.9

ISBN 0-13-110370-9 (США)
ISBN 0-13-110362-8 (pbk.) (США)
ISBN 5-279-00473-1 (СССР)

Предисловие к русскому изданию

Читателю предлагается второе, существенно переработанное издание книги, впервые опубликованной в США в 1978 г. Книга написана авторами языка Си, и прошедшие годы дают все основания отнести её к программистской классике. Перевод первого издания вышел в издательстве «Финансы и статистика» в 1985 г. и стал той книгой, которую многие программисты хотели бы иметь как настольную.

Новое издание, сохранив общую структуру книги и стиль изложения, отражает тот путь, который прошёл в своём развитии язык Си, получивший в 1989 г. статус американского национального стандарта. В стандартизированной версии язык ещё более отшлифован, избавлен от большинства вызвавших критику черт и дополнен рядом полезных возможностей. Интерес к языку Си вполне понятен. Язык занимает едва ли не первое место по распространённости среди инструментальных средств системного программирования во всём мире. Причём благодаря потоку персональных компьютеров, хлынувшему к нам из-за рубежа, он повсеместно используется и в нашей стране.

Можно ещё много говорить о достоинствах книги, её авторов, языка Си и его новой версии, но в этом нет необходимости, поскольку все книги по языку Си и без того являются бестселлерами. Спрос на них многократно превышает предложение. Поэтому перейдём к извечной проблеме переводчиков – проблеме терминологии.

В переводах первого и настоящего издания есть заметные терминологические расхождения. Во-первых, терминология не стоит на месте, во-вторых, переводы выполнялись разными переводчиками и, стало быть, не могла не появиться разница во вкусах.

Перечислим основные термины, по которым имеются расхождения.

В оригинале	В переводе первого издания	В переводе второго издания
character	символ	литера
structure	запись	структура
member	элемент	член
union	смесь	объединение
pipe	канал	трубопровод
line	строчка, строка	строка
string	строка	стринг
statement	оператор	инструкция
operator	операция	оператор
operation	операция	операция
declaration	описание	декларация
tag	тип записи	тег

Новации, как легко заметить, в частности, состоят в следующем. Введён термин *стринг*, чтобы можно было разными английскими терминами *line* и *string* поставить в соответствие разные русские термины. Термин *statement* везде переводится как *инструкция* с тем, чтобы освободить *оператор* для употребления в традиционном смысле – как знак операции.

Из того что не вошло в таблицу отметим термины *инкрементация* и *декрементация*, полученные «калькированием» английских incrementation и decrementation соответственно. Мы отважились внести эти термины, поскольку убеждены, что всё, что широко употребляется в жизни, рано или поздно будет признано в качестве термина, хотя в течение какого-то времени и квалифицируется как жаргон. Нам кажется, что чем скорее перестанут считаться жаргоном «теги», «скипы», «пайпы», «хеширования», «стринги» и т. п., тем лучше. Вспомните, какой горячий протест в своё время вызывали дисплей или файл, и подумайте, как настрадается тот, кто пытается обойтись без этих слов сегодня. Тенденции мировой интеграции и конвергенции очевидны и неодолимы, и мы не скрываем своего желания сближать русские и английские термины.

Наконец, последнее замечание. В оригинале употребляется термин *white space character*, собирательный для литер пробела, табуляции, новой строки, новой страницы и т. п. Нам не хватило смелости (может быть, напрасно) ввести «белый интервал» или что-нибудь подобное, и мы остановились на *пробельной литере*.

В заключение хочется выразить благодарность А. Б. Ходулёву, за неоценимую помощь, оказанную им в овладении техническими и программными средствами, которые были использованы при подготовке на компьютере оригинал-макета книги.

Вс. С. Штаркман

Предисловие

С момента публикации в 1978 г. книги «Язык программирования Си» в мире компьютеров произошла революция. Большие машины стали ещё больше, а возможности персональных ЭВМ теперь сопоставимы с возможностями больших машин десятилетней давности. Язык Си за это время также изменился, хотя и не очень сильно; что же касается сферы применения Си, то она далеко вышла за рамки его начального назначения как инструментального языка операционной системы UNIX.

Рост популярности Си, накапливающиеся с годами изменения, создание компиляторов коллективами разработчиков, ранее не причастных к проектированию языка, – всё это послужило стимулом к более точному и отвечающему времени определению языка по сравнению с первым изданием книги. В 1983 г. Американским институтом национальных стандартов (American National Standards Institute – ANSI) учреждается комитет, перед которым ставится цель выработать «однозначное и машинно-независимое определение языка Си», полностью сохранив при этом его стилистику. Результатом работы этого комитета и явился стандарт ANSI языка Си.

Стандарт формализует средства языка, которые в первом издании были только намечены, но не описаны, как, например, присваивание структурам и перечислимый тип; вводит новый вид описания функций, который позволяет проводить повсеместную проверку согласованности вызовов функции с её определением; специфицирует стандартную библиотеку с широким набором функций ввода-вывода, управления памятью, манипуляций со стрингами (цепочками литер) и другими функциями; уточняет семантику, которая в первоначальном определении была неясной, и явно очерчивает то, что остаётся машинно-зависимым.

Второе издание книги «Язык программирования Си» описывает версию Си, принятую в качестве стандарта ANSI. Мы решили описать язык заново, отметив при этом те места, в которых он претерпел изменения. В большинство разделов это не привнесло существенных изменений, самые заметные различия касаются новой формы описания и определения функции. Следует отметить, что современные компиляторы уже обеспечили под-

держку значительной части свойств стандарта.

Мы попытались сохранить краткость первого издания. Си – небольшой язык, и в качестве описания ему мало подходит большая книга. В новом издании улучшено описание наиболее важных средств, таких, как указатели, которые занимают центральное место в программировании на Си; доработаны старые примеры, а в некоторые главы добавлены новые. Так, для усиления трактовки сложных деклараций в качестве примеров включены программы перевода деклараций в их словесные описания и обратного преобразования. Все примеры, текст которых хранится в машине, протестированы строго в том виде, как они приведены в книге.

Приложение А – справочное руководство, но отнюдь не документ стандарта. В нём мы попытались «предметы первой необходимости языка» уложить на минимуме страниц. По замыслу это приложение должно легко читаться программистом-пользователем; для разработчиков компилятора в качестве определения языка предназначен сам стандарт. В приложении В собраны функции стандартной библиотеки. Оно также представляет собой справочник для программистов, но не для реализаторов. Приложение С содержит краткий перечень отличий от начальной версии языка.

В предисловии к первому изданию мы говорили о том, что «чем больше работаешь с Си, тем он становится удобнее». Это впечатление только усилилось после десяти лет работы с ним. Мы надеемся, что данная книга поможет вам изучить Си и успешно его использовать.

Мы в большом долгу перед друзьями, которые помогали нам в выпуске второго издания. Джон Бентли, Дуг Гуин, Дуг Макилрой, Питер Нельсон и Роб Пайк дали нам чёткие замечания почти по каждой странице первого варианта рукописи. Мы благодарны Алу Ахо, Деннису Аллиссону, Джою Кемпбеллу, Г. Р. Эмлину, Карен Фортганг, Аллену Голубу, Эндрю Хьюму, Дэйву Кристолу, Джону Линдерману, Дэйву Проссеру, Гину Спаффорду и Крис Ван Уику за внимательное прочтение книги. Мы получили полезные советы от Била Чезвика, Марка Кернигана, Эндрю Коэнига, Робина Лейка, Тома Лондона, Джима Ридза, Кловиза Тондо и Питера Вайнбергера. Дэйв Проссер ответил на многочисленные вопросы, касающиеся деталей стандарта ANSI. Мы широко пользовались транслятором с Си++ Бьёрна Строустрапа для локальной проверки наших программ, а Дэйв Кристол предоставил нам ANSI-Си-компилятор для окончательной их проверки. Рич Дрешлер очень помог в наборе книги.

Мы искренне благодарим всех.

Бриан В. Керниган
Деннис М. Ритчи

Предисловие к первому изданию

Си – это универсальный язык программирования с компактным способом записи выражений, современными механизмами управления структурами данных и богатым набором операторов. Си не является ни языком «очень высокого уровня», ни «большим» языком, не специализирован он и на какую-то конкретную область применения. Однако благодаря отсутствию ограничений и универсальности он удобнее и эффективнее для многих задач, чем предположительно более мощные языки.

Первоначально Си был спроектирован Деннисом Ритчи как инструмент написания операционной системы UNIX для машины PDP-11 и реализован в рамках этой операционной системы. И операционная система, и Си-компилятор, и, по существу, все прикладные программы системы UNIX (включая и те, которые использовались для подготовки текста этой книги*) написаны на Си. Фирменные Си-компиляторы существуют и на нескольких машинах других типов, среди которых IBM/370, Honeywell 6000 и Interdata 8/32. Си не привязан к конкретной аппаратуре или системе, однако на нём легко писать программы, которые без каких-либо изменений переносятся на другие машины, где осуществляется его поддержка.

Цель этой книги – помочь читателю научиться программировать на Си. Книга включает введение-учебник, позволяющий новичкам начать программировать как можно скорее; главы, посвящённые основным свойствам языка, и справочное руководство. В основу изложения положены изучение, составление и пересмотр примеров, а не простое перечисление правил. Почти все наши примеры – это законченные реальные программы, а не разобщённые фрагменты. Все они были оттестированы на машине буквально в том виде, как приводятся в книге. Помимо демонстрации эффективного использования языка, там где это было возможно, мы стремились проиллюстрировать полезные алгоритмы и принципы хорошего стиля написания программ и их разумного проектирования.

*Имеется в виду оригинал этой книги на английском языке. – *Примеч. пер.*

Эта книга не является вводным курсом по программированию. Предполагается, что читатель знаком с такими основными понятиями, как переменная, присваивание, цикл, функция. Тем не менее и новичок сможет изучить язык, хотя для него будет очень полезным общение с более знающими специалистами.

Наш опыт показал, что Си – удобный, выразительный и гибкий язык, пригодный для широкого класса задач. Его легко выучить, и чем больше работаешь с Си, тем он становится удобнее. Мы надеемся, что эта книга поможет вам хорошо его освоить.

Вдумчивая критика и предложения многих друзей и коллег были очень полезны для книги и помогали нам её писать. В частности Майк Бианки, Джим Блу, Стью Фелдман, Дуг Макилрой, Бил Рум, Боб Розин и Ларри Рослер с вниманием прочли все многочисленные варианты этой книги. Мы в долгу у Ала Ахо, Стива Бьерна, Дана Дворака, Чука Хейли, Марион Харрис, Рика Холта, Стива Джонсона, Джона Машея, Боба Митца, Ральфа Мухи, Питера Нельсона, Эллиота Пинсона, Била Плейджера, Джерри Спивака, Кена Томпсона и Питера Вайнбергера за полезные советы, полученные от них на различных стадиях подготовки рукописи, а также Майку Леску и Джо Оссанне за помощь при подготовке её издания.

Бриан В. Керниган
Деннис М. Ритчи

Введение

Си – универсальный язык программирования. Он тесно связан с системой UNIX, так как был разработан в этой системе, которая как и большинство программ, работающих в ней, написаны на Си. Однако язык не привязан жёстко к какой-то одной операционной системе или машине. Хотя он и назван «языком системного программирования», поскольку удобен для написания компиляторов и операционных систем, оказалось, что на нём столь же хорошо писать большие программы другого профиля.

Многие важные идеи Си взяты из языка BCPL, автором которого является Мартин Ричардс. Влияние BCPL на Си было косвенным – через язык B, разработанный Кеном Томпсоном в 1970 г. для первой системы UNIX, реализованной на PDP-7.

BCPL и B – «безтиповые» языки. В отличие от них Си обеспечивает разнообразие типов данных. Базовыми типами являются литеры, а также целые и плавающие числа различных размеров. Кроме того, имеется возможность получать целую иерархию выводимых типов данных из указателей, массивов, структур и объединений. Выражения формируются из операторов и операндов. Любое выражение, включая присваивание и вызов функции, может быть инструкцией. Указатели обеспечивают машинонезависимую адресную арифметику.

В Си имеются основные управляющие конструкции, используемые в хорошо структурированных программах: составная инструкция ({...}), ветвление по условию (if-else), выбор одной альтернативы из многих (switch), циклы с проверкой наверху (while, for) и с проверкой внизу (do), а также средство прерывания цикла (break).

В качестве результата функции могут возвращать значения базовых типов, структур, объединений и указателей. Любая функция допускает рекурсивное обращение к себе. Как правило, локальные переменные функции – «автоматические», т.е они создаются заново при каждом обращении к ней. Определения функций нельзя вкладывать друг в друга, но декларации переменных разрешается строить в блочно-структурной манере. Функции программы на Си могут храниться в отдельных исходных файлах и компилироваться независимо. Переменные по отношению к функции могут быть внут-

ренними и внешними. Последние могут быть доступными в пределах одного исходного файла или всей программы.

На этапе препроцессирования выполняется макроподстановка в текст программы, включение других исходных файлов и условная компиляция.

Си – язык сравнительно «низкого уровня». Однако это вовсе не умаляет его достоинств, просто Си имеет дело с теми же объектами, что и большинство компьютеров. т.е. с литерами, числами и адресами. С ними можно оперировать при помощи арифметических и логических операций, существующих в реальной машине.

В Си нет прямых операций над составными объектами, такими, как стринги, множества, списки и массивы. В нём нет операций, которые бы манипулировали с целыми массивами или стрингами, хотя структуры разрешается копировать целиком как единые объекты. В языке нет каких-либо средств распределения памяти, помимо возможности определения статических переменных и стекового механизма при выделении места для локальных переменных функций. Нет в нём «heap-памяти» («памяти-кучи», куда обычно «сваливают» разнородную информацию) и «сборщика мусора». Наконец, в самом Си нет средств ввода-вывода, инструкций READ (читать) и WRITE (писать) и каких-либо методов доступа к файлам. Все это – механизмы высокого уровня, которые в Си должны быть обеспечены исключительно с помощью явно вызываемых функций. Большинство реализованных Си-систем содержат в себе разумный стандартный набор этих функций.

В продолжение сказанного следует отметить. что Си предоставляет средства лишь последовательного управления ходом вычислений: механизм ветвления по условиям, циклы, составные инструкции, подпрограммы – и не содержит средств мультипрограммирования, параллельных процессов, синхронизации и организации сопрограмм.

Отсутствие некоторых из перечисленных средств может показаться серьёзным недостатком («выходит, чтобы сравнить две цепочки литер, нужно обращаться к функции?»). Однако компактность языка имеет реальные выгоды. Поскольку Си относительно мал, то и описание его кратко, и овладеть им можно быстро. Программист может реально рассчитывать на то, что он будет знать, понимать и на практике регулярно пользоваться всеми возможностями языка.

В течение многих лет единственным определением языка Си было первое издание книги «Язык программирования Си». В 1983 г. Институтом американских национальных стандартов (ANSI) учреждается комитет для выработки современного исчерпывающего определения языка Си. Результатом его работы явился стандарт для Си («ANSI-C»), выпущенный в 1988 г. Большинство положений стандарта уже учтено в современных компиляторах.

Стандарт базируется на первоначальном справочном руководстве. По сравнению с последним язык изменился относительно мало. Одной из це-

лей стандарта было обеспечить, чтобы в большинстве случаев существующие программы оставались правильными или вызывали предупреждающие сообщения компиляторов об изменении поведения.

Для большинства программистов самое важное изменение – это новый синтаксис описания и определения функций. Описание функции может теперь включать и описание её аргументов. В соответствии с этим изменился и синтаксис определения функции. Дополнительная информация значительно облегчает компилятору выявление ошибок, связанных с несогласованностью аргументов; по нашему мнению, это очень полезное добавление к языку.

Следует также отметить ряд небольших изменений. Присваивание структур и перечислимый тип, которые уже некоторое время широко используются, в языке узаконены. Вычисления с плавающей точкой теперь допускаются и с одинарной точностью. Уточнены свойства арифметики, особенно для беззнаковых типов. Усовершенствован препроцессор. Большинство программистов эти изменения затронут очень слабо.

Второй значительный вклад стандарта – это определение библиотеки, поставляемой вместе с Си-компилятором, в которой специфицируются функции доступа к возможностям операционной системы (например, чтения-записи файлов), форматного ввода-вывода, динамического запроса памяти, манипуляций со стрингами (цепочками литер) и т. д. Набор стандартных головных файлов обеспечивает единообразный доступ к описаниям функций и типов данных. Гарантируется, что программы, использующие эту библиотеку при взаимодействии с операционной системой, будут работать также и на других машинах. Большинство программ, составляющих библиотеку, созданы по образу и подобию «стандартной библиотеки ввода-вывода» системы UNIX. Эта библиотека описана в первом издании книги и широко используется в других системах. И здесь программисты не заметят существенных различий.

Так как типы данных и управляющих структур языка Си поддерживаются командами большинства существующих машин, административная система обеспечения независимого запуска и счёта программ очень мала. Обращения к библиотечным функциям пишет сам программист (не компилятор), поэтому при желании их можно легко заменить на другие. Почти все программы, написанные на Си, если они не касаются каких-либо деталей, скрытых в операционной системе, переносимы на другие машины.

Си соответствует аппаратным возможностям многих машин, однако он не привязан к архитектуре какой-либо конкретной машины. Проявляя некоторую дисциплину, можно легко писать переносимые программы, т. е. программы, которые без каких-либо изменений могут работать на разных машинах. Стандарт предоставляет возможность для явного описания переносимости с помощью набора констант, отражающих характеристики машины, на которой программа будет работать.

Си не является «строго типизированным» языком, но в процессе его развития контроль за типами был усилен. В первой версии Си разрешался бесконтрольный обмен указателей и целых, что вызывало большие нарекания, но это уже давным-давно запрещено. Согласно стандарту теперь требуется явное описание или явное указание преобразования, что уже и реализовано в хороших компиляторах. Новый вид описания функций – ещё один шаг в этом направлении. Компилятор теперь предупреждает о большей части ошибок в типах и автоматически не выполняет преобразования данных несовместимых типов. Однако главной философией Си остаётся то, что программисты сами знают, что делают; язык лишь требует явного указания об их намерениях.

Си, как и другие языки программирования, не свободен от недостатков. Уровень старшинства некоторых операторов не является общепринятым, некоторые синтаксические конструкции могли бы быть лучше.

Тем не менее, как оказалось, Си – чрезвычайно эффективный и выразительный язык, пригодный для широкого класса задач. Книга имеет следующую структуру. Гл. 1 – это обзор основных средств языка Си. Её назначение побудить читателя по возможности быстрее приступить к программированию, так как мы убеждены, что единственный способ изучить новый язык – это писать на нём программы. Эта часть книги предполагает наличие знаний по основным элементам программирования. Никаких пояснений того, что такое компьютер, компиляция или что означает выражение вида $n = n + 1$ не даётся. Хотя мы и пытались, там, где это возможно, показать полезные приёмы программирования, эта книга не призвана быть справочником ни по работе со структурами данных, ни по алгоритмам; когда оказывалось необходимым выбрать, на что сделать ударение, мы предпочитали сконцентрировать внимание на языке.

В гл. 2–6 различные средства языка обсуждаются более подробно и несколько более формально, чем в гл. 1; при этом по-прежнему упор делается на примеры, являющиеся законченными программами, а не изолированными фрагментами. Гл. 2 знакомит с базовыми типами данных, с операторами и выражениями. В гл. 3 рассматриваются средства управления последовательностью вычислений: if-else, switch, while, for и т. д. В гл. 4 речь идёт о функциях и структуре программы (внешних переменных, правилах видимости, делении программы на несколько исходных файлов и т.д.), а также о препроцессоре. В гл. 5 обсуждаются указатели и адресная арифметика. Гл. 6 посвящена структурам и объединениям.

В гл. 7 описана стандартная библиотека, обеспечивающая общий интерфейс с операционной системой. Эта библиотека узаконена в качестве стандарта ANSI, иначе говоря, она должна быть представлена на всех машинах, где существует Си, благодаря чему программы, использующие вводвывод и другие возможности операционной системы, без каких-либо изменений можно переносить с одной машины на другую.

Гл. 8 содержит описание интерфейса между программами на Си и операционной системой UNIX, в частности описание ввода-вывода, файловой системы и распределения памяти. Хотя некоторые разделы этой главы отражают специфику системы UNIX, программисты, пользующиеся другими системами, всё же найдут в них много полезных сведений, включая определённый взгляд на то, как реализуется одна из версий стандартной библиотеки, и некоторые предложения по переносимости программ.

Приложение A является справочником по языку. Строгое определение синтаксиса и семантики языка Си содержится в официальном документе стандарта ANSI. Последний, однако, более всего подходит разработчикам компилятора. Наш справочник определяет язык более сжато, не прибегая к законодательному стилю, которым пользуется стандарт. Приложение B – сводка по содержимому стандартной библиотеки и предназначена скорее пользователям, чем реализаторам. В приложении C приводится краткий перечень отличий от первой версии языка. В неясных случаях, однако, окончательным судьёй по языку остаётся стандарт и компилятор, которым вы пользуетесь.

Глава 1

Обзор языка

Начнём с быстрого ознакомления с языком Си. Наша цель – показать на реальных программах существенные элементы языка, не вдаваясь в мелкие детали, формальные правила и исключения из них. Поэтому мы не стремимся к полноте и даже точности (заботясь, однако, о корректности примеров). Нам бы хотелось как можно скорее подвести вас к моменту, когда вы сможете писать полезные программы, и, чтобы сделать это, мы должны сконцентрировать внимание на основах: переменных и константах, арифметике, управлении последовательностью вычислений, функциях и простейшем вводе-выводе. В настоящей главе мы умышленно не затрагиваем тех средств языка, которые важны при написании больших программ. Сюда входят указатели, структуры, большая часть богатого набора операторов, несколько управляющих инструкций и стандартная библиотека.

Такой подход имеет свои недостатки. Наиболее заметный из них связан с тем, что мы не даём здесь законченного изложения свойств языка, и эта лаконичность может привести к неправильному восприятию некоторых положений. В силу ограниченного характера подачи материала в примерах не используется вся мощь языка, и потому они не столь кратки и элегантны, как могли бы быть. Мы попытались по возможности смягчить эти эффекты, но считаем необходимым предупредить о них. Другой недостаток в том, что в последующих главах какие-то моменты нам придётся повторить. Мы надеемся, что польза от повторений превысит вызываемое ими раздражение.

В любом случае опытный программист, видимо, сможет материал данной главы проэкстраполировать на свои программистские нужды. Новичкам же рекомендуем её чтение дополнить написанием своих маленьких программ. И те и другие наши читатели могут рассматривать эту главу как «каркас», на который далее, начиная с гл. 2, будут «навешиваться» элементы языка.

1.1. Начнём, пожалуй

Единственный способ выучить новый язык программирования – это писать на нём программы. При изучении любого языка первой, как правило, предлагают написать приблизительно следующую программу:

Напечатайте слова

```
здравствуй, мир
```

Вот первое препятствие, и чтобы его преодолеть, вы должны суметь где-то создать текст программы, успешно его скомпилировать, загрузить, пустить на счёт и разобраться, куда будет отправлен результат. Как только вы овладеете этим, всё остальное окажется относительно простым.

Си-программа, печатающая «здравствуй, мир», выглядит так:

```
#include <stdio.h>
main()
{
    printf ("здравствуй, мир\n");
}
```

Как запустить эту программу, зависит от системы, которую вы используете. Так, в операционной системе UNIX необходимо сформировать исходную программу в файле с именем, заканчивающимся символами «.c», например, в файле `hello.c`, который затем требуется скомпилировать с помощью команды

```
cc hello.c
```

Если вы всё сделали правильно – не пропустили где-либо знака и не допустили орфографических ошибок, то компиляция пройдёт «молча», и вы получите файл, готовый к исполнению и названный `a.out`. Если вы теперь запустите этот файл на счёт командой

```
a.out
```

программа напечатает

```
здравствуй, мир
```

В других системах правила запуска программы на выполнение могут быть иными; чтобы узнать о них, поговорите со специалистами.

Теперь поясним некоторые моменты, касающиеся самой программы. Программа на Си, каких бы размеров она ни была, состоит из *функций* и *переменных*. Функции содержат *инструкции*, описывающие вычисления, которые необходимо выполнить, а переменные хранят значения, используемые в процессе этих вычислений.

Функции в Си похожи на подпрограммы и функции Фортрана и на процедуры и функции Паскаля. Приведённая программа – это функция с

именем main. Обычно вы вольны придумывать любые имена для своих функций, но «main» – особое имя: любая программа начинает свои вычисления с первой инструкции функции main.

Обычно main для выполнения своей работы пользуется услугами других функций; одни из них пишутся самим программистом, а другие берутся им готовыми из имеющихся в его распоряжении библиотек. Первая строка программы:

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

сообщает компилятору, что он должен включить информацию о стандартной библиотеке ввода-вывода. Эта строка встречается в начале многих исходных файлов Си-программ. Стандартная библиотека описана в гл. 7 и приложении B.

Один из способов передачи данных между функциями состоит в том, что функция при обращении к другой функции передаёт ей список значений, называемых *аргументами*. Этот список обрамляется скобками и помещается после имени функции. В нашем примере main определена как функция, которая не ждёт никаких аргументов. что отмечено пустым списком ().

ПЕРВАЯ ПРОГРАММА НА СИ

```
#include <stdio.h>                  Включение информации о стандартной библиотеке.

main()                              Определение функции с именем main, не получающей никаких аргументов.

{                                   Инструкции main заключаются в фигурные скобки.

    printf("здравствуй,мир\n");     Функция main обращается к библиотечной функции printf для печати заданной последовательности литер; \n – литера новая-строка.

}
```

Инструкции функции заключаются в фигурные скобки { }. Функция main содержит только одну инструкцию:

```
printf ("здравствуй, мир\n");
```

Функция вызывается по имени, после которого, в скобках, указывается список аргументов. Таким образом, приведённая строка – это вызов функции printf с аргументом "здравствуй, мир\n". printf – библиотечная функция, которая в данном случае напечатает последовательность литер, заключённую в двойные кавычки.

Цепочка литер в двойных кавычках типа `"здравствуй, мир\n"` называется *стрингом литер или стринговой константой*. Какое-то время в качестве аргументов для `printf` и других функций мы будем использовать только стринги литер.

В Си комбинация `\n` внутри стринга обозначает *литеру новая-строка*, которая при печати вызывает переход к левому краю следующей строки. Если вы удалите `\n` (стоит поэкспериментировать), то обнаружите, что, закончив печать, машина не переходит на новую строку. Литеру новая-строка в текстовый аргумент `printf` следует включать явным образом. Если вы попробуете выполнить, например,

```
printf ("здравствуй, мир
");
```

компилятор выдаст сообщение об ошибке.

Литера новая-строка никогда не вставляется автоматически, так что одну строку можно напечатать по шагам с помощью нескольких обращений к `printf`. Нашу первую программу можно написать и так:

```
#include <stdio.h>
main()
{
    printf ("здравствуй, ");
    printf ("мир");
    printf ("\n");
}
```

В результате её выполнения будет напечатана та же строка, что и раньше.

Заметим, что `\n` обозначает только одну литеру. Такие особые комбинации литер, начинающиеся с обратной наклонной черты, как `\n`, и называемые *эскейп-последовательностями*, широко применяются для обозначения трудно представимых или невидимых литер. Среди прочих в Си имеются литеры `\t`, `\b`, `\"`, `\\`, обозначающие соответственно табуляцию, возврат на одну литеру назад («забой» последней литеры), двойную кавычку, саму наклонную черту. Полный список таких литер представлен в разд. 2.3.

Упражнение 1.1. Выполните программу, печатающую "здравствуй, мир", в вашей системе. Поэкспериментируйте, удаляя некоторые части программы, и посмотрите, какие сообщения об ошибках вы получите.

Упражнение 1.2. Выясните, что произойдёт, если в стринговую константу аргумента `printf` вставить `\`*c*, где *c* – литера, не входящая в перечисленный выше список.

1.2. Переменные и арифметические выражения

Следующая программа выполняет вычисления по формуле $C^\circ = (5/9)(F^\circ - 32)$ и печатает приведённую ниже таблицу соответствия температур по Фаренгейту и по Цельсию:

```
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
```

Как и предыдущая, эта программа состоит из определения одной-единственной функции main. Она длиннее программы, печатающей "здравствуй, мир", но не сложная. На ней мы продемонстрируем несколько новых возможностей, включая комментарий, описания, переменные, арифметические выражения, циклы и форматный вывод.

```
#include <stdio.h>

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

    lower = 0;   /* нижняя граница табл. температур */
    upper = 300; /* верхняя граница */
    step  = 20;  /* шаг */

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

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

Две строки:

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

являются *комментарием*, который в данном случае кратко объясняет, что делает программа. Любые литеры, помещённые между `/*` и `*/`, игнорируются компилятором, и ими можно свободно пользоваться, чтобы сделать программу более понятной. Комментарий можно располагать в любом месте, где могут стоять литеры пробела, табуляции или литера новая-строка.

В Си любая переменная должна быть описана раньше, чем она будет использована; обычно все переменные описываются в начале функции перед первой исполняемой инструкцией. В *декларации (описании)* объявляются свойства переменных. Она состоит из названия типа и списка переменных, как, например

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

Тип `int`, означает, что значения перечисленных переменных есть целые, в отличие от него тип `float` указывает на значения с плавающей точкой, т.е. на числа, которые могут иметь дробную часть. Диапазоны значений обоих типов зависят от машины.

Числа типа `int` бывают как 16-разрядные (они лежат в диапазоне от −32768 до 32767), так и 32-разрядные Числа типа `float` обычно представляются 32-разрядными словами, имеющими по крайней мере 6 десятичных значащих цифр, и лежат приблизительно в диапазоне от 10^{-38} до 10^{+38}.

Помимо `int` и `float` Си допускает ещё несколько базовых типов для данных, это:

`char`	литера – единичный байт
`short`	короткое целое
`long`	длинное целое
`double`	плавающее с двойной точностью

Размеры объектов указанных типов также зависят от машины. Из базовых типов можно создавать: *массивы*, *структуры* и *объединения*, *указатели* на объекты базовых типов и функции, возвращающие в качестве результата значения этих типов. Обо всём этом мы расскажем позже.

Вычисления в программе преобразования температур начинаются с *инструкций присваивания*:

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

которые устанавливают указанным в них переменным их начальные значения. Любая инструкция заканчивается точкой с запятой.

Все строки таблицы вычисляются одним и тем же способом, поэтому мы воспользуемся циклом, повторяющим это вычисление для каждой строки. Необходимые действия выполнит цикл `while`:

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

Он работает следующим образом. Проверяется условие в скобках. Если оно истинно (значение `fahr` меньше или равно значению `upper`), то тело цикла (три инструкции, заключённые в фигурные скобки) выполняется. Затем опять проверяется условие и, если оно истинно, то тело цикла выполняется снова. Когда условие становится ложным (`fahr` превысило `upper`), цикл завершается, и вычисления продолжаются с инструкции, которая следует за циклом. Поскольку никаких инструкций за циклом нет, программа завершает работу.

Телом цикла `while` может быть одна или несколько инструкций, заключённых в фигурные скобки, как в программе преобразования температур, или одна-единственная инструкция без скобок, как в цикле

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

И в том и в другом случае инструкции, находящиеся под управлением `while`, мы будем записывать со сдвигом, равным одной позиции табуляции, которая в программе указывается четырьмя пробелами; благодаря этому будут ясно видны инструкции, расположенные внутри цикла. Отступы подчёркивают логическую структуру программы. Си-компилятор не обращает внимания на внешнее оформление программы, но наличие в нужных местах отступов и пробелов существенно влияют на то, насколько легко она будет восприниматься при её просмотре. Чтобы лучше была видна логическая структура выражения, мы рекомендуем писать только по одной инструкции на каждой строке и обрамлять пробелами знаки операций. Положение скобок менее важно, хотя существуют различные точки зрения на этот счёт. Мы остановились на одном из нескольких распространённых стилей их применения. Выберите тот, который больше всего вам импонирует, и точно ему следуйте.

Бóльшая часть вычислений выполняется в теле цикла. Температура переводится в шкалу Цельсия и присваивается переменной `celsius` посредством инструкции

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

Мы сначала умножаем на 5 и затем делим на 9, а не сразу умножаем на 5/9. Это связано с тем, что в Си, как и во многих других языках, деление целых сопровождается *обрезанием*, т.е. отбрасыванием дробной части. Так как 5 и 9 – целые, обрезание 5/9 дало бы нуль, и на месте температур по Цельсию были бы напечатаны нули.

Этот пример ещё немного прибавил нам знаний о том, как работает printf. printf – универсальная функция форматного ввода-вывода, которая будет подробно описана в гл. 7, Её первый аргумент – стринг, в котором каждый знак % соответствует одному из последующих её аргументов (второму, третьему...), а информация, расположенная за знаком %, указывает на вид, в котором каждый из этих аргументов выводится. Например, %d специфицирует выдачу аргумента в виде целого числа, и инструкция

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

печатает целое fahr, выполняет табуляцию и печатает целое celsius.

В функции printf каждому спецификатору первого аргумента (конструкции, начинающейся с %) соответствует второй аргумент, третий аргумент и т.д. Спецификаторы и соответствующие им аргументы должны быть согласованы по количеству и типам: в противном случае напечатано будет не то, что нужно.

Кстати, printf не является частью языка Си, и вообще нет никаких специальных конструкций языка, определяющих ввод-вывод. Функция printf – лишь полезная функция стандартной библиотеки, которая обычно доступна для Си-программ. Поведение функции printf, однако, оговорено стандартом ANSI и её свойства должны быть одинаковыми во всех Си-системах, удовлетворяющих требованиям стандарта.

Желая сконцентрировать ваше внимание на самом Си, мы не будем много говорить о вводе-выводе до гл. 7. В частности, до этой главы мы отложим разговор о форматном вводе. Если вам потребуется ввести числа, советуем прочитать в разд. 7.4 то, что касается функции scanf. Эта функция аналогична printf с той лишь разницей, что она вводит, а не выводит данные.

Существуют ещё две проблемы связанные с программой преобразования температур. Одна из них (более простая) состоит в том, что выводимый результат выглядит несколько неряшливо, поскольку числа не выровнены по правой позиции колонок. Это легко исправить, добавив в каждый из спецификаторов %d формата указание о ширине поля; при этом программа будет печатать числа, прижимая их к правому краю указанных полей. Например, мы можем написать

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

чтобы в каждой строке первое число печатать в поле из трёх позиций, а второе – из шести. В результате будет напечатано:

```
  0    -17
 20     -6
 40      4
 60     15
 80     28
100     37
...
```

Вторая, более серьёзная проблема связана с тем, что мы пользуемся целочисленной арифметикой и поэтому не совсем точно вычисляем температуры по шкале Цельсия. Например, $0°F$ на самом деле (с точностью до десятой) равно $-17.8°C$, а не -17. Чтобы получить более точные значения температур, нам надо пользоваться не целочисленной, а плавающей арифметикой. Это потребует некоторых изменений в программе.

```
#include <stdio.h>

/* печать таблицы температур по Фаренгейту и Цельсию для
   fahr = 0, 20, ..., 300; вариант с плавающей точкой   */
main()
{
    float fahr, celsius;
    int lower, upper, step;

    lower = 0;   /* нижняя граница табл. температур */
    upper = 300; /* верхняя граница */
    step  = 20;  /* шаг */

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

Программа мало изменилась. Она отличается от предыдущей лишь тем, что `fahr` и `celsius` объявлены как `float`, а формула преобразования написана в более естественном виде. В предыдущем варианте нельзя было писать `5/9`, так как целочисленное деление в результате обрезания дало бы нуль. Десятичная точка в константе указывает, что последняя рассматривается как число с плавающей точкой, и `5.0/9.0`, таким образом, есть частное от деления двух значений с плавающей точкой, которое не предполагает отбрасывания дробной части. В том случае, когда арифметическая операция имеет целые операнды, она выполняется по правилам целочисленной арифметики.

Если же один операнд плавающий, а другой – целый, то перед тем, как операция будет выполнена, последний будет преобразован в плавающий. Если бы мы написали `fahr-32`, то 32 автоматически было бы преобразовано в число с плавающей точкой. Тем не менее при записи констант с плавающей точкой мы всегда используем десятичную точку, причём даже в тех случаях, когда константы на самом деле имеют целые значения. Это делается для того, чтобы обратить внимание читающего программу на их природу.

Более подробно правила, определяющие, в каких случаях целые переводятся в плавающие, рассматриваются в гл. 2. А сейчас заметим, что присваивание

```
fahr = lower;
```

и проверка

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

работают естественным образом, т.е. перед выполнением операции значение `int` приводится к `float`.

Спецификация `%3.0f` в `printf` определяет печать числа с плавающей точкой (в данном случае числа `fahr`) в поле шириной не более трёх позиций без десятичной точки и дробной части. Спецификация `%6.1f` описывает печать другого числа (`celsius`) в поле из шести позиций с одной цифрой после десятичной точки. Напечатано будет следующее:

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

Ширину и точность можно не задавать: `%6f` означает, что число будет занимать не более шести позиций; `%.2f` – число имеет две цифры после десятичной точки, но ширина не ограничена; `%f` просто указывает на печать числа с плавающей точкой.

`%d`	печать десятичного целого
`%6d`	печать десятичного целого в поле из 6 позиций
`%f`	печать числа с плавающей точкой
`%6f`	печать числа с плавающей точкой в поле из 6 позиций
`%.2f`	печать числа с плавающей точкой с 2 цифрами после десятичной точки
`%6.2f`	печать числа с плавающей точкой в поле из 6 позиций и 2 цифрами после десятичной точки

Кроме того, `printf` допускает также следующие спецификаторы: `%o` для восьмеричного числа, `%x` для шестнадцатеричного числа, `%c` для литеры, `%s` для стринга литер и `%%` для самого %.

Упражнение 1.3. Усовершенствуйте программу преобразования температур таким образом, чтобы над таблицей она печатала заголовок.

Упражнение 1.4. Напишите программу, которая будет печатать таблицу соответствия температур по Цельсию температурам по Фаренгейту.

1.3. Инструкция for

Существует много разных способов для написания одной и той же программы. Видоизменим нашу программу преобразования температур:

```
#include <stdio.h>
/* печать 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 печатает тот же результат, но выглядит она, несомненно, по-другому. Главное отличие в отсутствии большинства переменных. Осталась только переменная fahr, которую мы описали как int. Нижняя и верхняя границы и шаг присутствуют в виде констант в инструкции for – новой для нас конструкции, а выражение, вычисляющее температуру по Цельсию, теперь задано третьим аргументом функции printf, а не в отдельной инструкции присваивания.

Это последнее изменение является примером применения общего правила: в любом контексте, где возможно использование значения переменной какого-то типа, можно использовать более сложное выражение того же типа. Так, на месте третьего аргумента функции printf согласно спецификатору %6.1f должно быть значение с плавающей точкой, следовательно, здесь может быть любое выражение этого типа.

Инструкция for описывает цикл, который является обобщением цикла while. Если вы сравните его с ранее написанным while, то вам станет ясно, как он работает. Внутри скобок имеются три выражения, разделяемых точкой с запятой. Первое выражение – инициализация

```
fahr = 0
```

– выполняется один раз перед тем, как войти в цикл. Второе – проверка условия продолжения цикла

```
fahr <= 300
```

Условие вычисляется, и, если оно истинно, тело цикла (в нашем случае это одно обращение к printf) выполняется. Затем осуществляется приращение шага:

```
    fahr = fahr + 20
```

и условие вычисляется снова. Цикл заканчивается, когда условие становится ложным. Как и в случае с `while`, тело `for`-цикла может состоять из одной инструкции или из нескольких, заключённых в фигурные скобки. На месте этих трёх выражений (инициализации, условия и приращения шага) могут стоять произвольные выражения.

Выбор между `while` и `for` определяется соображениями ясности программы. Цикл `for` более удобен в тех случаях, когда инициализация и приращение шага логически связаны друг с другом общей переменной и выражаются единичными инструкциями, поскольку он компактнее цикла `while`, а его управляющие части сосредоточены в одном месте.

Упражнение 1.5. Измените программу преобразования температур так, чтобы она печатала таблицу в обратном порядке, т.е. от 300 до 0.

1.4. Именованные константы

Прежде чем мы закончим рассмотрение программы преобразования температур, выскажем ещё одно соображение. Очень плохо, когда по программе рассеяны «магические числа» типа 300, 20. Для того кто будет читать программу, в них нет и намёка на то, что они собой представляют. Кроме того, их трудно заменить на другие каким-то систематическим способом. Одна из возможностей справиться с такими числами – дать им осмысленные имена. Строка `#define` определяет *символьное имя*, или *именованную константу*, для заданного стринга литер:

```
#define   имя   подставляемый-текст
```

С этого момента при любом появлении имени (если только оно встречается не в тексте, заключённом в кавычки, и не является частью определения другого имени) оно будет заменяться на соответствующий ему *подставляемый-текст*. Имя имеет тот же вид, что и переменная: последовательность букв и цифр, начинающаяся с буквы. *Подставляемый-текст* может быть любой последовательностью литер, среди которых могут встречаться не только цифры.

```
#include <stdio.h>
#define   LOWER  0       /* нижняя граница таблицы */
#define   UPPER  300     /* верхняя граница */
#define   STEP   20      /* размер шага */
/* печать 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 – именованные константы, а не переменные. Поэтому для них нет описаний. По общепринятому соглашению имена именованных констант набираются заглавными буквами, чтобы они отличались от обычных переменных, набираемых строчными. Заметим, что в конце #define-строки точка с запятой не ставится.

1.5. Ввод-вывод литер

Теперь мы намерены рассмотреть семейство программ по текстовой обработке. Вы обнаружите, что многие реальные программы являются просто более развитыми версиями описываемых здесь программ.

Стандартная библиотека поддерживает очень простую модель ввода-вывода. Текстовый ввод-вывод вне зависимости от того, откуда он исходит или куда направляется, имеет дело с потоком литер. *Текстовый поток* – это последовательность литер, разбитая на строки, каждая из которых содержит нуль или более литер и завершается литерой новая-строка. Обязанность следить за тем, чтобы любой поток ввода-вывода отвечал этой модели, возложена на библиотеку: программист, пользуясь библиотекой, не должен заботиться о том, в каком виде строки представляются вне программы.

Стандартная библиотека включает несколько функций для чтения и записи одной литеры. Простейшие из них – getchar и putchar. За одно обращение к getchar читается *следующая литера ввода* из текстового потока, которая и выдаётся в качестве результата. Так, после выполнения

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

переменная c содержит следующую литеру ввода. Обычно литеры поступают с клавиатуры. Ввод из файлов рассматривается в гл. 7.

Обращение к putchar приводит к печати одной литеры. Так,

```
putchar(c);
```

напечатает содержимое целой переменной c в виде литеры (обычно на экране). Вызовы putchar и printf могут произвольным образом перемежаться. Вывод будет формироваться в том же порядке, в каком осуществляются обращения к этим функциям.

1.5.1. Копирование файла

При наличии функций getchar и putchar, ничего больше не зная о вводе-выводе, можно написать удивительно много полезных программ. Простейший пример – это программа, копирующая по одной литере с входного потока в выходной поток

```
чтение литеры
while (литера не есть признак-конца-файла)
    вывод только что прочитанной литеры
    чтение литеры
```

Оформляя её в виде программы на Си, получим

```
#include <stdio.h>
/* копирование ввода на вывод; 1-я версия */
main()
{
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

Оператор отношения != означает «не равно».

Каждая литера, вводимая с клавиатуры или появляющаяся на экране, как и любая другая литера внутри машины, кодируется комбинацией бит. Тип char специально предназначен для хранения литерных данных, однако для этого также годится и любой целый тип. Мы пользуемся типом int и делаем это по одной важной причине, которая требует разъяснений.

Существует проблема: как отличить конец ввода от обычных читаемых данных. Решение в том, чтобы функция getchar по исчерпании входного потока выдавала в качестве результата такое значение, которое нельзя было бы спутать ни с одной реальной литерой. Это значение есть EOF (аббревиатура от end of file – конец файла). Мы должны объявить переменную c такого типа, чтобы его «хватило» для представления всех возможных результатов, выдаваемых функцией getchar. Нам не подходит тип char, так как c должна быть достаточно «ёмкой», чтобы помимо любого значения типа char быть в состоянии хранить и EOF. Вот почему мы используем int, а не char.

EOF – целая константа, определённая в <stdio.h>. Какое значение имеет эта константа – неважно, лишь бы оно отличалось от любого из возможных значений типа char. Использование именованной константы с унифицированным именем гарантирует, что программа не будет зависеть от конкретного числового значения. которое, возможно, в других Си-системах будет иным.

Программу копирования можно написать более сжато. В Си любое присваивание, например,

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

трактуется как выражение со значением, равным значению левой части после присваивания. Это значит, что присваивание может встречаться внутри

более сложного выражения. Если присваивание переменной c расположить в проверке условия цикла while, то программу копирования можно будет записать в следующем виде:

```
#include <stdio.h>
/* копирование ввода на вывод; 2-я версия */
main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

Цикл while, пересылая в c полученное от getchar значение, сразу же проверяет: не является ли оно «концом файла». Если это не так, тело цикла while выполняется и литера печатается. По окончании ввода завершается работа цикла while и тем самым и main.

В данной версии ввод «централизован» – в программе имеется только одно обращение к getchar. В результате она более компактна и легче воспринимается при чтении. Вам часто придётся сталкиваться с такой формой записи, где присваивание делается вместе с проверкой. (Чрезмерное увлечение ею, однако, может запутать программу, поэтому мы постараемся пользоваться указанной формой разумно.)

Скобки внутри условия, обрамляющие присваивание, необходимы. *Приоритет* != выше, чем приоритет =, из чего следует, что при отсутствии скобок проверка != будет выполняться до операции присваивания =. Таким образом, запись

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

эквивалентна записи

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

А это совсем не то, что нам нужно: переменной c будет присваиваться 0 или 1 в зависимости от того, встретит или не встретит getchar признак конца файла. (Более подробно об этом см. в гл. 2.)

Упражнение 1.6. Убедитесь в том, что выражение getchat() != EOF получает значение 0 или 1.

Упражнение 1.7. Напишите программу, печатающую значение EOF.

1.5.2. Подсчёт литер

Следующая программа занимается подсчётом литер; она имеет много сходных черт с программой копирования.

```
#include <stdio.h>
/* подсчёт вводимых литер; 1-я версия */
main()
{
    long nc;
    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

Инструкция

```
++nc
```

демонстрирует новый оператор ++, который означает *увеличить на единицу*. Вместо этого можно было бы написать `nc = nc+1`, но `++nc` намного короче, а часто и эффективнее. Существует аналогичный оператор `--`, означающий *уменьшить на единицу*. Операторы `++` и `--` могут быть и префиксными (`++nc`) и постфиксными (`nc++`). Как будет показано в гл. 2, эти две формы имеют разные значения в выражениях, но и `++nc`, и `nc++` добавляют к `nc` единицу. В данном случае мы остановились на префиксной записи.

Программа подсчёта литер накапливает сумму в переменной типа `long`. Целые типа `long` имеют не менее 32 бит. Хотя на некоторых машинах типы `int` и `long` имеют одинаковый размер, существуют, однако машины, в которых `int` занимает 16 бит с максимально возможным значением 32767, а это – сравнительно маленькое число, и счётчик типа `int` может переполниться. Спецификация преобразования `%ld` в `printf` указывает, что соответствующий аргумент имеет тип `long`.

Возможно охватить ещё больший диапазон значений, если использовать тип `double` (т.е. `float` с двойной точностью). Применим также инструкцию `for` вместо `while`, чтобы продемонстрировать другой способ написания цикла.

```
#include <stdio.h>
/* подсчёт вводимых литер; 2-я версия */
main()
{
    double nc;
    for (nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

В `printf` спецификатор `%f` применяется как для `float`, так и для `double`; спецификатор `%.0f` означает печать без десятичной точки и дробной части (последняя в нашем случае отсутствует).

Тело указанного for-цикла пусто, поскольку кроме проверок и приращений счётчика делать ничего не нужно. Но правила грамматики Си требуют, чтобы for-цикл имел тело. Выполнение этого требования обеспечивает изолированная точка с запятой, называемая *пустой инструкцией*. Мы поставили точку с запятой на отдельной строке для большей наглядности.

Наконец, заметим, что, если ввод не содержит ни одной литеры, то при первом же обращении к getchar условие в while или for не будет выполнено, и программа выдаст нуль, что и будет правильным результатом. Это важно. Одно из привлекательных свойств циклов while и for состоит в том, что условие проверяется до того, как выполняется тело цикла. Если ничего делать не надо, то ничего делаться и не будет, пусть даже тело цикла не будет выполнено ни разу. Программа должна вести себя корректно при нулевом количестве вводимых литер. Само устройство циклов while и for даёт дополнительную уверенность в правильном поведении программы в случае граничных условий.

1.5.3. Подсчёт строк

Следующая программа подсчитывает строки. Как упоминалось выше, стандартная библиотека обеспечивает такую модель ввода-вывода, при которой входной текстовый поток состоит из последовательности строк, каждая из которых заканчивается литерой новая-строка. Следовательно, подсчёт строк сводится к подсчёту числа литер новая-строка.

```
#include <stdio.h>
/* подсчёт строк входного потока */
main()
{
    int c, nl;
    nl=0;
    while ((0 = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

Тело цикла теперь образует инструкция if, под контролем которой находится увеличение счётчика nl на единицу. Инструкция if проверяет условие в скобках и, если оно истинно, выполняет следующую за ним инструкцию (или группу инструкций, заключённую в фигурные скобки). Мы опять делаем отступы в тексте программы, чтобы показать, что чем управляется.

Двойной знак равенства в языке Си обозначает оператор «равно» (он аналогичен оператору = в Паскале и .EQ в Фортране). Удваивание = в операторе проверки на равенство сделано для того, чтобы отличить его от единичного =, используемого в Си для обозначения присваивания. Предупрежда-

ем: начинающие программировать на Си иногда пишут =, а имеют в виду ==. Как мы увидим в гл. 2, в этом случае результатом обычно будет вполне допустимое по форме выражение, на которое компилятор не выдаст никаких предупреждающих сообщений.

Литера, заключённая в одиночные кавычки, представляет целое значение, равное коду этой литеры (в кодировке, принятой на данной машине). Это так называемая *литерная константа*. Существует и другой способ для написания маленьких целых значений. Например, `'A'` есть литерная константа; в наборе литер ASCII её значение равняется 65 – внутреннему представлению буквы A. Конечно, `'A'` в роли константы предпочтительнее, чем 65, поскольку смысл первой записи более очевиден, и она не зависит от конкретного способа кодировки литер.

Эскейп-последовательности, используемые в стринговых константах, допускаются также и в литерных константах. Так, `'\n'` обозначает код литеры новая-строка, который в ASCII равен 10. Следует обратить особое внимание на то, что `'\n'` обозначает одну литеру (код которой в выражении рассматривается как целое значение в то время как `"\n"` – стринговая константа, в которой чисто случайно указана одна литера. Более подробно различие между литерными и стринговыми константами разбирается в гл. 2.

Упражнение 1.8. Напишите программу для подсчёта пробелов, табуляций и новых-строк.

Упражнение 1.9. Напишите программу, копирующую литеры ввода в выходной поток и заменяющую подряд стоящие пробелы на один пробел.

Упражнение 1.10. Напишите программу, копирующую вводимые литеры в выходной поток с заменой литеры табуляции на `\t`, литеры забоя на `\b` и каждой обратной наклонной черты на `\\`. Это сделает видимыми все литеры табуляции и забоя.

1.5.4. Подсчёт слов

Четвёртая из нашей серии полезных программ подсчитывает строки, слова и литеры, причём под словом здесь имеется в виду любой стринг литер, не содержащий в себе пробелов, табуляций и новых-строк. Эта программа является упрощённой версией программы wc системы UNIX.

```
#include <stdio.h>
#define IN  1    /* внутри слова */
#define OUT 0    /* вне слова */
/* подсчёт строк, слов и литер */
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);
}
```

Каждый раз, встречая первую литеру слова, программа изменяет значение счётчика слов на 1. Переменная `state` фиксирует текущее состояние – находимся мы внутри или вне слова. Вначале ей присваивается значение `OUT`, что соответствует состоянию «вне слова». Мы предпочитаем пользоваться именованными константами `IN` и `OUT`, а не собственно значениями 1 и 0, чтобы сделать программу более понятной. В нашей маленькой программе этот приём мало что даёт, но в большой программе увеличение её ясности окупает незначительные дополнительные усилия, потраченные на то, чтобы писать программу в таком стиле с самого начала. Вы обнаружите, что большие изменения гораздо легче вносить в те программы, в которых магические числа встречаются только в виде именованных констант.

Строка

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

устанавливает все три переменные в нуль. Такая запись не является какой-то особой конструкцией и допустима потому, что присваивание есть выражение со своим собственным значением и операции присваивания выполняются справа налево. Указанная строка эквивалентна

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

Оператор `||` означает ИЛИ, так что строка

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

читается как «если `c` есть пробел, *или* `c` есть новая-строка, *или* `c` есть табуляция». (Напомним, что эскейп-последовательность `\t` обозначает литеру табуляции.) Существует также оператор `&&`, означающий И. Его приоритет выше, чем приоритет `||`. Выражения, связанные операторами `&&` или `||`,

вычисляются слева направо; при этом гарантируется, что вычисления сразу прервутся, как только будет установлена истинность или ложность условия. Если c есть пробел, то дальше проверять, является значение c литерой новая-строка или же табуляции, не нужно. В этом частном случае данный способ вычислений не столь важен, но он имеет значение в более сложных ситуациях, которые мы вскоре рассмотрим.

В примере также встречается слово `else`, которое указывает на альтернативные действия, выполняемые в случае, когда условие, указанное в `if`, не является истинным. В общем виде условная инструкция записывается так:

```
if (выражение)
    инструкция1
else
    инструкция2
```

В конструкции `if-else` выполняется одна и только одна из двух инструкций. Если *выражение* истинно, то выполняется *инструкция*$_1$, если нет, то – *инструкция*$_2$. Каждая из этих двух *инструкций* представляет собой либо одну инструкцию, либо несколько, заключённых в фигурные скобки. В нашей программе после `else` стоит инструкция `if`, управляющая двумя такими инструкциями.

Упражнение 1.11. Как протестировать программу подсчёта слов? Какой ввод вероятнее всего не обнаружит ошибок, если они были допущены?

Упражнение 1.12. Напишите программу, которая печатает содержимое своего ввода, помещая по одному слову на каждой строке.

1.6. Массивы

А теперь напишем программу, подсчитывающую по отдельности каждую цифру, пробельные литеры (пробелы, табуляции и новые-строки) и все другие литеры. Это несколько искусственная программа, но она позволит нам в одном примере продемонстрировать ещё несколько возможностей языка Си. Имеется двенадцать категорий вводимых литер. Удобно все десять счётчиков цифр хранить в массиве, а не в виде десяти отдельных переменных. Вот один из вариантов этой программы:

```
#include <stdio.h>
/* подсчёт цифр, пробельных и прочих литер */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];
```

```
    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;
    while ((0 = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;
    printf("цифры =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", пробелы = %d, прочие = %d\n", nwhite, nother);
}
```

Если пропустить текст этой программы в качестве ввода, то будет напечатан следующий результат

```
цифры = 9 3 0 0 0 0 0 0 1, пробелы = 141, прочие = 347
```

Декларация

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

определяет `ndigit` как массив из 10 значений типа `int`. В Си элементы массива всегда нумеруются, начиная с нуля, так что элементами этого массива будут `ndigit[0]`, `ndigit[1]`, ..., `ndigit[9]`, что учитывается в `for`-циклах (при инициализации и печати массива).

Индексом может быть любое целое выражение, образуемое целыми переменными (например, `i`) и целыми константами.

Приведённая программа опирается на определённые свойства кодировки цифр. Например, проверка

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

определяет, является ли находящаяся в `c` литера цифрой. Если это так, то

```
c - '0'
```

есть числовое значение цифры. Сказанное справедливо только в том случае, если для ряда значений `'0'`, `'1'`, ..., `'9'` каждое следующее значение на 1 больше предыдущего. К счастью, это правило соблюдается во всех наборах литер.

По определению значения типа `char` являются всего лишь малыми целыми, так что переменные и константы типа: `char` в арифметических выражениях идентичны значениям типа `int`. Это и естественно, и удобно: например, `c-'0'` есть целое выражение с возможными значениями от 0 до 9, которые соответствуют литерам от `'0'` до `'9'`, хранящимся в переменной `c`. Таким образом, значение данного выражения является правильным индексом для массива `ndigit`.

Следующий фрагмент определяет, является литера цифрой, пробельной литерой или чем-нибудь иным.

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

Конструкция вида

```
if (условие1)
    инструкция1
else if (условие2)
    инструкция2
...
    ...
else
    инструкцияn
```

часто применяется для выбора одного из нескольких альтернативных путей, имеющихся в программе. *Условия* вычисляются по порядку в направлении сверху вниз до тех пор, пока одно из них не будет удовлетворено; в этом случае соответствующая ему *инструкция* будет выполнена, и работа всей конструкции завершится. (Любая из *инструкций* может быть группой инструкций, обрамленных фигурными скобками.) Если ни одно из *условий* не удовлетворено, выполняется последняя *инструкция*, расположенная сразу за `else`, если таковая имеется. Если же `else` и следующей за ней *инструкции* нет (как это было в программе подсчёта слов), то никакие действия вообще не производятся.

Между первым `if` и завершающим `else` может быть сколько угодно комбинаций вида

```
else if (условие)
    инструкция
```

Когда их несколько, программу разумно форматировать так, как мы здесь показали. Если же каждый следующий `if` сдвигать вправо относительно предыдущего `else`, то при длинном каскаде проверок текст окажется слишком близко прижатым к правому краю страницы.

Инструкция `switch` речь о которой пойдёт в гл. 3, обеспечивает другой способ изображения многопутевого ветвления на языке Си. Он более подходит, в частности, тогда, когда условием перехода служит совпадение значения некоторого выражения целочисленного типа с одной из констант, входящих в заданный набор. Вариант нашей программы, реализованной с помощью `switch` приводится в разд. 3.4.

Упражнение 1.13. Напишите программу, печатающую гистограммы длин вводимых слов. Гистограмму легко рисовать горизонтальными полосами. Рисование вертикальными полосами – более трудная задача.

Упражнение 1.14. Напишите программу, печатающую гистограммы частот встречаемости вводимых литер.

1.7. Функции

Функции в Си играют ту же роль, что и подпрограммы и функции в Фортране или процедуры и функции в Паскале. Функция обеспечивает удобный способ отдельно оформить некоторое вычисление и пользоваться им далее, не заботясь о том, как оно реализовано. После того как функции написаны можно забыть, *как* они сделаны, достаточно знать лишь, *что* они умеют делать. Механизм использования функций в Си удобен, лёгок и эффективен. Зачастую вы будете встречать короткие функции, вызываемые только единожды; они оформлены в виде функции с одной-единственной целью – получить более ясную программу.

До сих пор мы пользовались готовыми функциями типа `printf`, `getchar` и `putchar`, теперь настала пора написать несколько наших собственных функций. В Си нет оператора возведения в степень типа оператора `**` в Фортране. Поэтому проиллюстрируем механизм определения функции на примере функции `power(m,n)`, которая возводит целое `m` в целую положительную степень `n`. Так, `power(2,5)` имеет значение 32. На самом деле для практического применения эта функция малопригодна, так как она оперирует лишь малыми целыми степенями, однако она достаточно хороша, чтобы послужить иллюстрацией. (В стандартной библиотеке есть функция `pow(x,y)`, вычисляющая x^y.)

Итак, мы имеем функцию `power` и главную функцию, пользующуюся её услугами, так что вся программа выглядит следующим образом:

```
#include <stdio.h>
int power(int m, int n);
/* тест функции power */
main()
{
    int i;

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

```
/* 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 любой функции имеет следующий вид:

```
тип-результата имя-функции(список параметров, если он есть)
{
    декларации
    инструкции
}
```

Определения функций могут располагаться в любом порядке в одном или в нескольких исходных файлах, но любая функция должна быть целиком расположена в каком-то одном. Если исходный текст программы распределён по нескольким файлам, то, чтобы её скомпилировать и загрузить, вам придётся сказать несколько больше, чем в случае одного файла; но это уже относится к операционной системе, а не к языку. Пока мы предполагаем, что обе функции находятся в одном файле, так что тех знаний, которые вы уже получили относительно запуска программ на Си, будет достаточно.

В следующей строке из функции `main` к `power` обращаются дважды.

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

При каждом вызове функции `power` передаются два аргумента, и каждый раз главная программа в ответ получает целое число, которое затем приводится к должному формату и печатается. Как часть выражения `power(2,i)` представляет собой целое значение точно так же, как `2` или `i`. (Не все функции в качестве результата выдают целые значения; подробно об этом будет сказано позже, в гл. 4.)

В первой строке определения `power`:

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

указываются типы параметров, имя функции и тип результата. Имена параметров локализованы в `power`, и это значит, что они скрыты для любой другой функции, так что остальные подпрограммы могут свободно пользоваться ими для своих целей. Последнее утверждение справедливо также для переменных `i` и `p`: `i` в `power` и `i` в `main` не имеют между собой ничего общего.

Далее *параметром* мы будем называть переменную из списка параметров, заключённого в скобки и заданного в определении функции, а *аргументом* – значение, используемое при обращении к функции. Иногда в том же

смысле мы будем употреблять термины *формальный аргумент* и *фактический аргумент*.

Значение, вычисляемое функцией power, возвращается в main с помощью инструкции return. За словом return может следовать любое выражение:

```
return выражение;
```

Функция не обязательно возвращает какое-нибудь значение. Инструкция return без выражения только передаёт управление в ту программу, которая её вызвала, не передавая ей никакого результирующего значения. То же самое происходит, если в процессе вычислений мы выходим на конец функции, отмеченный в тексте последней закрывающей фигурной скобкой. Возможна ситуация, когда вызывающая функция игнорирует возвращаемый ей результат.

Вы, вероятно, обратили внимание на инструкцию return в конце main. Поскольку main есть функция, как и любая другая она может вернуть результирующее значение тому, кто её вызвал, – фактически в операционную среду, в которой была запущена программа. Обычно возвращается нулевое значение, что говорит о нормальном завершении счёта. Ненулевое значение сигнализирует о необычном или ошибочном завершении. До сих пор ради простоты мы опускали return в main, но с этого момента будем задавать return как напоминание о том, что программы должны сообщать о состоянии своего завершения в операционную систему.

Декларация

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

стоящая непосредственно перед main, сообщает, что функция power ожидает двух целых аргументов и возвращает целый результат. Это описание, называемое *прототипом функции*, должно быть согласовано с определением и всеми вызовами power. Будет ошибкой, если определение функции или вызов не соответствует своему прототипу.

Имена параметров не требуют согласования. Фактически в прототипе они могут быть произвольными или вообще отсутствовать, т.е. прототип можно было бы записать и так:

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

Однако хорошо подобранные имена поясняют программу, и мы будем часто этим пользоваться.

Историческое замечание. Самые большие отличия ANSI-Си от более ранних версий языка как раз и заключаются в способах описания и определения функций. В первой версии Си функцию power требовалось задавать в следующем виде:

```
/* power возводит base в n-ю степень; n >= 0 */
/* (версия в старом стиле языка Си)          */
power(base, n)
int base, n;
{
    int i, p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Здесь имена параметров перечислены в круглых скобках, а их типы заданы перед первой открывающей фигурной скобкой. В случае отсутствия указания о типе параметра, считается, что он имеет тип `int`. (Тело функции не претерпело изменений.)

Описание `power` в начале программы согласно первой версии Си должно было бы выглядеть следующим образом:

```
int power()
```

Нельзя было задавать список параметров, и поэтому компилятор не имел возможности проверить правильность обращений к `power`. Так как при отсутствии описания `power` предполагалось, что функция возвращает целое значение, то в данном случае описание целиком можно было бы опустить.

Новый синтаксис для прототипов функций облегчает компилятору обнаружение ошибок в количестве аргументов и их типах. Старый синтаксис описания и определения функции всё ещё допускается стандартом ANSI, по крайней мере на переходный период, но, если ваш компилятор поддерживает новый синтаксис, мы настоятельно рекомендуем пользоваться только им.

Упражнение 1.15. Перепишите программу преобразования температур, выделив само преобразование в отдельную функцию.

1.8. Аргументы. Вызов по значению

Одно свойство функций в Си, вероятно, будет в новинку для программистов, которые уже пользовались другими языками и Фортраном в частности. В Си все аргументы функции передаются «по значению». Это следует понимать так, что вызываемой функции посылаются значения её аргументов во временных переменных, а не сами аргументы. Такой способ передачи аргументов несколько отличается от «вызова по ссылке» в Фортране и спецификации `var` при параметре в Паскале, которые позволяют подпрограмме иметь доступ к самим аргументам, а не к их локальным копиям.

Главное отличие в том, что в Си вызываемая функция не может непосредственно изменить переменную вызывающей функции: она может изменить только её частную, временную копию.

Однако вызов по значению следует отнести к достоинствам языка, а не к его недостаткам. Благодаря этому свойству обычно удаётся написать более компактную программу, содержащую меньшее число «чужих» переменных, поскольку параметры можно рассматривать как должным образом инициализированные локальные переменные вызванной подпрограммы. В качестве примера приведём ещё одну версию функции power, в которой как раз использовано это свойство.

```
/* power возводит base в n-ю степень; n >= 0; версия 2 */
int power(int base, int n)
{
    int p;
    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

Параметр n выступает здесь в роли временной переменной, в которой циклом for в убывающем порядке ведётся счёт числа шагов до тех пор, пока её значение не станет нулём. При этом отпадает надобность в дополнительной переменной i в качестве счётчика цикла. Что бы мы ни делали с n внутри power, это не окажет никакого влияния на сам аргумент, копия которого была передана функции power в её вызове.

При желании можно сделать так, чтобы функция смогла изменить переменную в вызывающей программе. Последняя для этого должна передать *адрес* подлежащей изменению переменной (указатель на переменную), а в вызываемой функции следует соответствующий параметр описать как указатель и организовать косвенный доступ через него к этой переменной. Все, что касается указателей, мы рассмотрим в гл. 5.

Механизм передачи в качестве аргумента массива – несколько иной. Когда имя массива используется как аргумент, то функции передаётся значение, которое является адресом начала этого массива; никакие элементы массива не копируются. С помощью индексирования относительно полученного значения функция имеет доступ к любому элементу массива. Разговор об этом пойдёт в следующем разделе.

1.9. Массивы литер

Самый распространённый вид массива в Си – массив литер. Чтобы проиллюстрировать использование литерных массивов и работающих с ни-

ми функций, напишем программу, которая читает набор текстовых строк и печатает самую длинную из них.

Её схема достаточно проста:

```
while (есть ли ещё строка?)
    if (данная строка длиннее самой длинной из предыдущих)
        запомнить её
        запомнить её длину
напечатать самую длинную строку
```

Из схемы видно, что программа естественным образом распадается на части. Одна из них получает новую строку, другая проверяет её, третья запоминает, а остальные управляют процессом вычислений.

Поскольку процесс чётко распадается на части, его хорошо бы так и переводить на Си. Поэтому сначала напишем отдельную функцию getline для получения очередной строки. Мы попытаемся сделать эту функцию полезной и для других применений. Как минимум getline должна сигнализировать о возможном конце файла, а ещё лучше, если она будет выдавать длину строки или нуль (в случае исчерпания файла). Нуль годится для признака конца файла, поскольку не бывает строк нулевой длины, – даже строка, содержащая только одну литеру новая-строка имеет длину 1.

Когда мы обнаружим строку более длинную, чем самая длинная из всех предыдущих, то нам надо будет где-то её запомнить. Здесь напрашивается вторая функция, copy, которая умеет копировать новую строку в надёжное место.

Наконец, нам необходима главная программа, которая бы управляла функциями getline и copy. Вот как выглядит наша программа в целом:

```
#include <stdio.h>
#define MAXLINE 1000 /* макс-ный размер вводимой строки */
int getline(char line[], int maxline);
void copy(char to[], char from[]);
/* печать самой длинной строки */
main()
{
    int len;  /* длина текущей строки */
    int max; /* длина максимальной из просмотренных строк */
    char line[MAXLINE];       /* текущая строка */
    char longest[MAXLINE];    /* самая длинная строка */
    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(longest, line);
        }
```

```
    if (max > 0)      /* была ли хоть одна строка? */
    printf("%s", longest);
    return 0;
}

/* getline: читает строку в s, возвращает 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 из 'from' в 'to'; to достаточно большой */
void copy(char to[], char from[])
{
    int i;
    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

Мы предполагаем, что функции getline и copy, описанные в начале программы, находятся в том же файле, что и main.

Функции main и getline взаимодействуют между собой через пару аргументов и возвращаемое значение. В getline аргументы определяются строкой

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

Как мы видим, её первый аргумент, s, есть массив, а второй, lim, имеет целый тип. Задание размера массива в определении имеет целью резервирование памяти. В самой getline задавать длину массива s нет необходимости, так как его размер указан в main. Чтобы вернуть значение вызывающей программе, getline использует return точно так, как это делает функция power. В приведённой строке также сообщается, что getline возвращают значение типа int, но так как при отсутствии указания о типе подразумевается int, то перед getline слово int можно опустить.

Одни функции возвращают результирующее значение, другие (как copy) нужны только для того, чтобы произвести какие-то действия, не вы-

давая никакого значения. На месте типа результата в copy стоит void. Это явное указание на то, что никакого значения данная функция не возвращает.

Функция getline в конец создаваемого ею массива помещает литеру '\0' (*null-литеру*, кодируемую нулевым байтом), чтобы пометить конец стринга литер. То же соглашение относительно окончания null-литерой соблюдается и в случае стринга литер типа

```
"hello\n"
```

В данном случае для него формируется массив из литер этого стринга с '\0' в конце.

h	e	l	l	o	\n	\0

Спецификация %s в формате printf предполагает, что соответствующий ей аргумент – стринг, оформленный указанным выше образом. Функция copy в своей работе также опирается на тот факт, что читаемый ею аргумент заканчивается литерой '\0', который она копирует наряду с остальными литерами. (Все сказанное предполагает, что '\0' не встречается внутри обычного текста.)

Попутно стоит заметить, что даже на такой маленькой программе выявляются некоторые конструктивные трудности. Например, что должна делать main, если встретится строка, превышающая допустимый размер? Функция getline работает надёжно: если массив полон, она прекращает пересылку, даже если литеры новая-строка не обнаружила. Получив от getline длину строки и увидев, что она совпадает с MAXLINE, главная программа могла бы «отловить» этот особый случай и справиться с ним. В интересах краткости мы его здесь опускаем.

Пользователи getline не могут заранее узнать, сколь длинными будут вводимые строки, поэтому getline делает проверки на переполнение. А вот пользователям функции copy размеры копируемых стрингов известны (или они могут их узнать), поэтому дополнительный контроль здесь не нужен.

Упражнение 1.16. Перепишите main предыдущей программы так, чтобы она могла печатать самую длинную строку без каких-либо ограничений на её размер.

Упражнение 1.17. Напишите программу печати всех вводимых строк, содержащих более 80 литер.

Упражнение 1.18. Напишите программу, которая будет в каждой вводимой строке заменять подряд стоящие литеры пробелов и табуляций на один пробел и удалять пустые строки.

Упражнение 1.19. Напишите функцию `reverse(s)`, размещающую литеры в стринге `s` в обратном порядке. Примените её при написании программы, которая «реверсирует» каждую вводимую строку.

1.10. Внешние переменные и область действия

Переменные `line`, `longest` и т.д. принадлежат только функции `main`, или, как говорят, локализованы в ней. Поскольку они определены внутри `main`, никакие другие функции прямо к ним обращаться не могут. То же верно и применительно к переменным других функций; например, `i` в `getline` не имеет никакого отношения к `i` в `copy`. Каждая локальная переменная функции возникает только в момент обращения к этой функции и исчезает после выхода из неё. Вот почему такие переменные, следуя терминологии других языков, называют *автоматическими*. (В гл. 4 обсуждается класс памяти `static`, который позволяет локальным переменным сохранять свои значения в промежутках между вызовами.)

Так как автоматические переменные образуются и исчезают одновременно с входом в функцию и выходом из неё, они не сохраняют своих значений от вызова к вызову и должны устанавливаться заново при каждом новом обращении к функции. Если этого не делать, они будут содержать «мусор».

В качестве альтернативы автоматическим переменным можно определить *внешние* переменные, к которым разрешается обращаться по их именам из любой функции. (Этот механизм аналогичен области `COMMON` в Фортране и определениям переменных в самом внешнем блоке в Паскале.) Так как внешние переменные доступны повсеместно, их можно использовать вместо аргументов для связи между функциями по данным. Кроме того, поскольку внешние переменные существуют постоянно, а не возникают и исчезают на период активизации функции, свои значения они сохраняют и после возврата из функций, их установивших.

Внешняя переменная должна быть *определена*, причём только один раз, вне текста любой функции; в этом случае ей будет выделена память. Она должна быть *описана* (*декларирована*) во всех функциях, которые хотят ею пользоваться. Описание содержит сведения о типе переменной. Описание может быть явным, в виде инструкции `extern`, или неявным, когда нужная информация получается из контекста. Чтобы конкретизировать сказанное перепишем программу печати самой длинной строки с использованием `line`, `longest` и `max` в качестве внешних переменных. Это потребует изменений в вызовах, описаниях и телах всех трёх функций.

```
#include <stdio.h>
#define MAXLINE 1000 /* макс-ный размер вводимой строки */
int max;  /* длина максимальной из просмотренных строк */
char line[MAXLINE];            /* текущая строка */
```

```
char longest[MAXLINE];          /* самая длинная строка */
int getline(void);
void copy(void);
/* печать самой длинной строки */
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 хотя бы одна строка */
        printf("%s", longest);
    return 0;
}

/* getline: специализированная 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: специализированная версия */
void copy(void)
{
    int i;
    extern char line[], longest[];
    i = 0;
    while ((longest[i] = line[i]) != '\0')
```

```
        ++i;
}
```

Внешние переменные для main, getline и copy определяются в начале нашего примера, где им присваивается тип и выделяется память. Определения внешних переменных синтаксически ничем не отличаются от определения локальных переменных, но поскольку они расположены вне функций, эти переменные считаются внешними. Чтобы функция могла пользоваться внешней переменной, ей нужно прежде всего сообщить имя соответствующей переменной. Это можно сделать, например, задав описание extern, которое по виду отличается от определения внешней переменной только тем, что оно начинается с ключевого слова extern.

В некоторых случаях описание extern можно опустить. Если определение внешней переменной в исходном файле расположено выше функции, где она используется, то в описании extern нет необходимости. Описания extern, таким образом, в main, getline и copy избыточны. Обычно определения внешних переменных располагают в начале исходного файла, и все описания extern для них опускают.

Если же программа расположена в нескольких исходных файлах и внешняя переменная определена в *файле1*, а используется в *файле2* и *файле3*, то extern-описания в *файле2* и *файле3* обязательны, поскольку необходимо указать, что во всех трёх файлах функции ссылаются на одну и ту же внешнюю переменную. На практике обычно удобно собрать все описания внешних переменных и функций в отдельный файл, называемый *головным* (*header-файлом*), и помещать его с помощью #include в начало каждого исходного файла. Суффикс .h по общей договорённости используется для имён header-файлов. В головных файлах, в <stdio.h> в частности, описываются также функции стандартной библиотеки. Более подробно о головных файлах говорится в гл. 4, а применительно к стандартной библиотеке – в гл. 7 и приложении B.

Так как специализированные версии getline и copy не имеют аргументов, на первый взгляд кажется, что логично их прототипы задать в виде getline() и copy(). Но из соображений совместимости со старыми Си-программами стандарт рассматривает пустой список как сигнал к тому, чтобы выключить все проверки на соответствие аргументов. Поэтому, когда нужно сохранить контроль и явно указать отсутствие аргументов, следует пользоваться словом void. Мы вернёмся к этой проблеме в гл. 4.

Заметим, что по отношению к внешним переменным в этом разделе мы очень аккуратно используем понятия *определение* и *описание*. «Определение» располагается в месте, где переменная создаётся и ей отводится память; «описание» помещается там, где фиксируется природа переменной, но никакой памяти для неё не отводится.

Следует отметить тенденцию все переменные делать внешними. Дело в том, что, как может показаться на первый взгляд, это приводит к упроще-

нию связей – ведь списки аргументов становятся короче, а переменные везде доступны, где они нужны; правда, они оказываются доступными и там, где они не нужны. Однако чрезмерный акцент на внешние переменные чреват большими опасностями – он приводит к программам, в которых связи по данным не очевидны, поскольку переменные могут неожиданным и даже таинственным способом изменяться. Кроме того, такая программа с трудом поддаётся модификациям. Вторая версия программы поиска самой длинной строки хуже, чем первая, частично по этим причинам, а частично из-за нарушения общности двух полезных функций, вызванного тем, что в них вписаны имена конкретных переменных, с которыми они оперируют.

Итак, мы рассмотрели то, что можно было бы назвать ядром Си. Описанных «кирпичиков» достаточно, чтобы создавать полезные программы значительных размеров, и было бы чудесно, если бы вы, прервав чтение, посвятили этому какое-то время. В следующих упражнениях мы предлагаем вам реализовать несколько более сложные программы, чем те, что рассматривались выше.

Упражнение 1.20. Напишите программу `detab`, заменяющую литеры табуляции вводимого текста нужным числом пробелов (до следующего «стопа» табуляции). Предполагается, что «стопы» табуляции расставлены на фиксированном расстоянии друг от друга, скажем, через `n` позиций. Как лучше задавать `n` – в виде значения переменной или в виде именованной константы?

Упражнение 1.21. Напишите программу `entab`, заменяющую цепочки пробелов минимальным числом литер табуляций и пробелов таким образом, чтобы вид напечатанного текста не изменился. Используйте те же «стопы» табуляции, что и в `detab`. В случае, когда для выхода на очередной «стоп» годится один пробел, что лучше – пробел или литера табуляции?

Упражнение 1.22. Напишите программу, печатающую литеры входного потока так, чтобы строки текста не выходили правее *n*-й позиции. Это значит, что каждая строка, длина которой превышает *n*, должна печататься с переносом на следующие строки. Место переноса следует «искать» после последней непробельной литеры, расположенной левее `n`-й позиции. Позаботьтесь о том, чтобы ваша программа вела себя разумно в случае очень длинных строк, а также, когда до `n`-й позиции не встречается ни одной литеры пробела или табуляции.

Упражнение 1.23. Напишите программу, убирающую все комментарии из любой Си-программы. Не забудьте должным образом обработать стрин-

ги литер и стринговые константы. Комментарии в Си не могут быть вложены друг в друга.

Упражнение 1.24. Напишите программу, проверяющую Си-программы на элементарные синтаксические ошибки типа несбалансированности скобок всех видов. Не забудьте о кавычках (одиночных и двойных), эскейп-последовательностях (\...) и комментариях. (Это – сложная программа, если её писать для общего случая.)

Глава 2

Типы, операторы и выражения

Переменные и константы являются основными объектами, с которыми имеет дело программа. Переменные перечисляются в описаниях, где устанавливаются их типы, а возможно, и начальные значения. Операции специфицируют те действия, которые с ними совершаются. Для получения новых значений выражения могут оперировать с переменными и константами. Тип объекта определяет множество значений, которые этот объект может принимать, и операций, которые над ними могут выполняться. Названные «кирпичики» и будут предметом обсуждения в этой главе.

Стандартом ANSI было утверждено значительное число небольших изменений и добавлений к основным типам и выражениям. Любой целый тип теперь может быть со знаком, `signed`, и без знака, `unsigned`. Предусмотрен способ записи беззнаковых констант и шестнадцатеричных литерных констант. Операции с плавающей точкой допускаются теперь и с одинарной точностью. Введён тип `long double`, обеспечивающий повышенную точность. Стринговые константы конкатенируются («склеиваются») теперь во время компиляции. Перечислимый тип стал частью языка, формализующей установку диапазона значений типа. Объекты разрешено помечать как `const` для защиты их от каких-либо изменений. В связи с введением новых типов расширены правила автоматического преобразования из одного арифметического типа в другой.

2.1. Имена переменных

Хотя мы ничего не говорили об этом в гл. 1, но существуют некоторые ограничения на задание имён переменных и именованных констант. Име-

на составляются из букв и цифр; первой литерой должна быть буква. Знак подчёркивания «_» считается буквой; его иногда удобно использовать, чтобы улучшить восприятие длинных имён переменных. Не начинайте имена переменных с подчёркивания, так как многие переменные библиотечных программ начинаются именно с этого знака. Большие (прописные) и малые (строчные) буквы различаются, так что `x` и `X` – два разных имени. Обычно в программах на Си малыми буквами набирают переменные, а большими – именованные константы.

Для внутренних имён значимыми являются первые 31 литера. Для имён функций и внешних переменных число значимых литер может быть меньше 31, так как эти имена обрабатываются ассемблерами и загрузчиками и языком не контролируются. Уникальность внешних имён гарантируется только в пределах 6 литер, набранных безразлично в каком регистре. Ключевые слова `if`, `else`, `int`, `float` и т.д. зарезервированы, и их нельзя использовать в качестве имён переменных. Все они набираются на нижнем регистре (т.е. малыми буквами).

Разумно переменным давать осмысленные имена в соответствии с их назначением, причём такие, чтобы их было трудно спутать друг с другом. Мы предпочитаем короткие имена для локальных переменных, особенно для счётчиков циклов, и более длинные для внешних переменных.

2.2. Типы и размеры данных

В Си существует всего лишь несколько базовых типов:

`char`	единичный байт, который может содержать одну литеру из допустимого набора литер
`int`	целое, обычно отображаемое на естественное представление целых в машине
`float`	число с плавающей точкой одинарной точности
`double`	число с плавающей точкой двойной точности

Имеется также несколько квалификаторов, которые можно использовать вместе с указанными базовыми типами. Например, квалификаторы `short` (короткий) и `long` (длинный) применяются к целым:

```
short int sh;
long int counter;
```

В таких описаниях слово `int` можно опускать, что обычно и делается.

Если только не возникает противоречий со здравым смыслом, целое `short` и целое `long` должны быть разной длины, а `int` соответствовать естественному размеру целых на данной машине. Чаще всего для представления целого, описанного с квалификатором `short`, отводится 16 бит, с квалификатором `long` – 32 бита, а значению типа `int` – или 16, или 32 бита. Разработчики компилятора вправе сами выбирать подходящие размеры, сообра-

зуясь с характеристиками своего компьютера и соблюдая только следующие ограничения: значения типов short и int представляются по крайней мере 16 битами, типа long – по крайней мере 32 битами, размер short не больше размера int, который в свою очередь не больше размера long.

Квалификаторы signed (со знаком) или unsigned (без знака) можно применять к типу char и любому целому типу. Значения unsigned всегда положительны или равны нулю и подчиняются законам арифметики по модулю 2^n, где n – количество бит в представлении типа. Так, например, если значению char отводится 8 бит, то unsigned char имеет значения в диапазоне от 0 до 255, а signed char – от −128 до 127 (в машине с двоичным дополнительным кодом). Являются ли значения типа просто char знаковыми или беззнаковыми, зависит от машины, но в любом случае коды печатаемых литер положительны.

Тип long double предназначен для арифметики с плавающей точкой повышенной точности. Как и в случае целых, размеры объектов с плавающей точкой зависят от реализации; float, double и long double могут представляться одним размером, а могут – двумя или тремя разными размерами.

Именованные константы для всех размеров вместе с другими характеристиками машины и компилятора содержатся в стандартных головных файлах <limits.h> и <float.h>. (См.приложение B.)

Упражнение 2.1. Напишите программу, которая будет выдавать диапазоны значений типов char, short, int и long, описанных как signed и как unsigned, с помощью печати соответствующих значений из стандартных головных файлов и путём прямого вычисления. Определите диапазоны чисел с плавающей точкой различных типов. Вычислить эти диапазоны сложнее.

2.3. Константы

Целая константа, например, 1234, имеет тип int. Константа типа long завершается буквой l или L, например 123456789L; слишком большое целое, которое невозможно представить как int, будет представлено как long. Беззнаковые константы заканчиваются буквой u или U, а окончание ul или UL говорит о том, что тип константы – unsigned long.

Константы с плавающей точкой имеют десятичную точку (123.4) или экспоненциальную часть (1e-2) или же и то и другое. Если у них нет окончания, считается, что они типа double. Окончание f или F указывает на тип float, а l или L – на тип long double.

Помимо десятичного целое значение может иметь восьмеричное или шестнадцатеричное представление. Если константа начинается с нуля, то она представлена в восьмеричном виде, если с 0x или с 0X, то – в шестнадцатеричном. Например, десятичное целое 31 можно записать как 037 или

как 0X1F. Записи восьмеричной и шестнадцатеричной констант могут завершаться буквой L (для указания на тип long) и U (если нужно показать, что константа беззнаковая). Например, константа 0XFUL имеет значение 15 и тип unsigned long.

Литерная константа есть целое, записанное в виде литеры, обрамленной одиночными кавычками, например 'x'. Значением литерной константы является числовой код литеры из набора литер на данной машине. Например, литерная константа '0' в кодировке ASCII имеет значение 48, которое никакого отношения к числовому значению 0 не имеет. Если мы пишем '0', а не какое-нибудь значение (например, 48), которое следует из способа кодировки, мы тем самым делаем программу независимой от частного значения кода, к тому же она и легче читается. Литерные константы могут участвовать в операциях над числами точно так же, как и любые другие целые, хотя чаще они используются для сравнения с другими литерами.

Некоторые литеры в литерных и стринговых константах записываются с помощью эскейп-последовательностей, например \n (новая-строка); такие последовательности изображаются двумя литерами, но обозначают одну. Кроме того, произвольный восьмеричный код можно задать в виде

```
'\ooo'
```

где *ooo* – одна, две или три восьмеричные цифры (0...7) или

```
'\xhh'
```

где *hh* – одна, две или более шестнадцатеричные цифры (0...9, a...f, A...F). Таким образом, мы могли бы написать

```
#define VTAB '\013'    /* верт. табуляция в ASCII */
#define BELL '\007'    /* звонок в ASCII */
```

или в шестнадцатеричном виде:

```
#define VTAB '\xb'     /* верт. табуляция в ASCII */
#define BELL '\x7'     /* звонок в ASCII */
```

Полный набор эскейп-последовательностей следующий:

\a	сигнал-звонок	\\	обратная-наклонная-черта
\b	возврат-на-шаг	\?	знак-вопроса
\f	перевод-страницы	\'	одиночная-кавычка
\n	новая-строка	\"	двойная-кавычка
\r	возврат-каретки	*ooo*	восьмеричный-код
\t	гор-табуляция	\x*hh*	шестнадцатиричный-код
\v	верт-табуляция		

Литерная константа '\0' – это литера с нулевым значением – так называемая литера null. Вместо просто 0 часто используют запись '\0', чтобы подчеркнуть литерную природу выражения, хотя и в том и другом случае запись обозначает нуль.

Константные выражения – это выражения, оперирующие только с константами. Такие выражения вычисляются во время компиляции, а не во время счёта, и поэтому их можно использовать в любом месте, где допустимы константы, как, например, в

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

или в

```
#define LEAP 1 /* in leap years - в високосные годы */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Стринговая константа, или *стринговый литерал*, – это нуль или более литер, заключённых в двойные кавычки, как, например,

```
"Это стринговая константа"
```

или

```
""   /* пустой стринг */
```

Кавычки не входят в стринг, а служат только её ограничителями. Так же, как и в литерные константы, в стринги можно включать эскейп-последовательности; \", например, представляет собой двойную кавычку. Стринговые константы можно конкатенировать («склеивать») во время компиляции; например, запись двух стрингов

```
"Здравствуй,"  " мир!"
```

эквивалентна записи одного следующего стринга:

```
"Здравствуй, мир!"
```

Указанное свойство позволяет разбивать длинные стринги на части и располагать эти части на отдельных строчках.

Фактически стринговая константа – это массив литер. Во внутреннем представлении стринга в конце обязательно присутствует null-литера `'\0'`, поэтому памяти для стринга требуется на один байт больше, чем число литер, расположенных между двойными кавычками. Это означает, что нет ограничения на длину задаваемого стринга, но чтобы определить его длину, требуется просмотреть весь стринг. Функция `strlen(s)` вычисляет длину стринга `s` без учёта завершающей его литеры `'\0'`. Ниже приводится наша версия этой функции:

```
/* strlen: возвращает длину стринга  */
int strlen(char s[])
{
    int i;
    i = 0;
    while (s[i] != '\0')
        ++i;
    return i;
}
```

Функция strlen и некоторые другие, применяемые к стрингам, описаны в стандартном головном файле <string.h>.

Будьте внимательны и помните, что литерная константа и стринг, содержащий одну литеру, не одно и то же: 'x' не то же самое, что "x". Запись 'x' обозначает целое значение, равное коду буквы x из стандартного набора литер, а запись "x" – массив литер, который содержит одну литеру (букву x) и '\0'.

В Си имеется ещё один вид константы, *константа перечисления*. Перечисление – это список целых констант, как, например, в

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

Первое имя в enum* имеет значение 0, следующее – 1 и т.д. (если не было явных спецификаций значений констант). Если не все значения специфицированы, то они продолжают прогрессию, начиная от последнего специфицированного значения, как в следующих двух примерах:

```
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 есть 2, MAR есть 3 и т.д.  */
```

Имена в различных перечислениях должны отличаться друг от друга. Значения внутри одного перечисления могут совпадать.

Средство enum обеспечивает удобный способ присвоить константам имена, причём в отличие от #define при этом способе значения констант могут генерироваться автоматически. Перечислимый тип разрешено использовать для определения переменных, однако компилятор не обязан контролировать, входят ли присваиваемые этим переменным значения в их тип. Но сама возможность такой проверки часто делает enum лучше, чем #define. Кроме того, отладчик получает возможность печатать значения перечислимых переменных в символьном виде.

2.4. Декларации

Все переменные должны быть декларированы раньше, чем будут использоваться, при этом некоторые декларации могут быть получены неявно – из контекста. Декларация специфицирует тип и содержит список из одной или нескольких переменных этого типа, как, например, в

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

Переменные можно распределять по декларациям произвольным образом, так что указанные выше списки можно записать и в следующем виде:

*От английского слова enumeration – перечисление. – *Примеч. ред.*

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

Эта последняя форма записи занимает больше места, тем не менее она лучше, поскольку позволяет добавлять к каждой декларации комментарий и более удобна для последующих модификаций.

В своей декларации переменная может быть инициализирована, как, например:

```
char  esc = '\\';
int   i = 0;
int   limit = MAXLINE+1;
float eps = 1.0e-5;
```

Инициализация неавтоматической переменной осуществляется только один раз – перед тем, как программа начнёт выполняться, при этом инициализатор должен быть константным выражением. Явно инициализируемая автоматическая переменная получает начальное значение каждый раз при входе в функцию или блок, её инициализатором может быть любое выражение. Внешние и статические переменные по умолчанию получают нулевые значения. Автоматические переменные, явным образом не инициализированные, содержат неопределённые значения («мусор»).

К любой переменной в декларации может быть применён квалификатор const для указания того, что её значение далее не будет изменяться.

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

Применительно к массиву квалификатор const указывает на то, что ни один из его элементов не будет меняться. Указание const можно также применять к аргументу-массиву, чтобы сообщить, что функция не изменяет этот массив:

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

Реакция на попытку изменить переменную, помеченную квалификатором const, оставлена на усмотрение компилятора.

2.5. Арифметические операторы

Бинарными арифметическими операторами являются +, –, *, /, а также оператор взятия модуля %. Деление целых сопровождается отбрасыванием дробной части, какой бы она ни была. Выражение

```
x%y
```

даёт остаток от деления x на y и, следовательно, нуль, если x делится на y нацело. Например, год является високосным, если он делится на 4 (но не на 100). Кроме того, високосным считается год, если он делится на 400. Следовательно,

```
if ((year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
    printf("%d високосный год\n, year);
else
    printf("%d не високосный год\n, year);
```

Оператор % к операндам типов float и double не применяется. В какую сторону (в сторону увеличения или уменьшения числа) будет усечена дробная часть при выполнении / и каким будет знак результата операции % с отрицательными операндами, это зависит от машины.

Бинарные операторы + и – имеют одинаковый приоритет, который ниже приоритета операторов *, / и %, который в свою очередь ниже приоритета унарных операторов + и –. Арифметические операции одного приоритетного уровня выполняются слева направо.

В конце этой главы приводится табл. 2.1, в которой показаны приоритеты всех операторов и порядок их выполнения.

2.6. Операторы отношения и логические операторы

Операторами отношения являются

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

Все они имеют одинаковый приоритет. Ровно на одну ступень ниже приоритета операторов сравнения на равенство:

```
==   !=
```

Операторы отношения имеют более низкий приоритет, чем арифметические, поэтому выражение типа `i < lim-1` будет выполняться так же, как `i < (lim-1)`, т.е. как мы и ожидаем.

Более интересны логические операторы && и ||. Выражения, между которыми стоят операторы && или ||, вычисляются слева направо, и вычисление прекращается, как только становится известна истинность или ложность результата. Многие Си-программы опираются на это свойство, как, например, цикл из функции getline , которую мы приводили в гл. 1:

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

Прежде чем читать очередную литеру, нужно проверить, есть ли место для неё в массиве s, иначе говоря, сначала необходимо проверить условие `i < lim-1`. Если это условие не выполняется, мы не должны продолжать

вычисление, в частности читать следующую литеру. Так же было бы неправильным сравнивать c с EOF до обращения к getchar; следовательно, и вызов getchar и присваивание должны выполняться перед указанной проверкой.

Приоритет оператора && выше, чем оператора ||, однако их приоритеты ниже, чем приоритет операторов отношения и равенства; поэтому выражение вида

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

не нуждается в дополнительных скобках. Но, так как приоритет != выше, чем приоритет присваивания, в

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

скобки необходимы, чтобы сначала выполнить присваивание, а затем сравнение с n.

По определению численным результатом вычисления выражения отношения или логического является 1 в случае, если оно истинно, и 0 в случае, если оно ложно.

Унарный оператор ! преобразует ненулевой операнд в 0, а нуль в 1. Обычно оператор !, используют в конструкциях вида

```
if (!valid)
```

что эквивалентно

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

Трудно сказать, какая из форм записи лучше. Конструкция вида !valid хорошо читается («если не valid»), но в случае более сложных выражений может оказаться, что её не так-то легко понять.

Упражнение 2.2. Напишите цикл, эквивалентный приведённому выше for-циклу, не пользуясь операторами && и ||.

2.7. Преобразования типов

Если операнды оператора принадлежат разным типам, то они приводятся к некоторому общему типу. Приведение выполняется в соответствии с небольшим числом правил. Обычно автоматически производятся лишь те преобразования, которые без какой-либо потери информации превращают операнды с меньшим диапазоном значений в операнды с большим диапазоном значений, как, например, преобразование целого в число с плавающей точкой в выражении вроде f + i. Выражения, не имеющие смысла, например число с плавающей точкой в роли индекса, не допускаются. Выражения, в которых могла бы теряться информация (скажем, при присваивании длинных целых переменным более коротких типов или при присваивании значений

с плавающей точкой целым переменным), могут повлечь предупреждение, но они допустимы.

Значения типа char – это всего лишь малые целые, и их можно свободно использовать в арифметических выражениях, что значительно облегчает всевозможные манипуляции с литерами. В качестве примера приведём простенькую реализацию функции atoi, преобразующей последовательность цифр в её числовой эквивалент.

```
/* atoi: преобразование s в целое */
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 мы уже говорили в гл. 1, выражение

```
s[i] - '0'
```

даёт числовое значение литеры хранящейся в s[i], так как значения '0', '1' и т.д. образуют непрерывную возрастающую последовательность.

Другой пример приведения char к int связан с функцией lower, которая одиночную литеру из набора ASCII, если она является заглавной буквой, превращает в прописную. Если же литера не является заглавной буквой, lower её не изменяет.

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

В случае ASCII эта программа будет правильно работать потому, что между одноимёнными буквами верхнего и нижнего регистров – одинаковое расстояние (если их рассматривать как числовые значения), и, кроме того, латинский алфавит – плотный в том смысле, что между буквами A и Z ничего кроме букв не существует. Для набора EBCDIC последнее условие не выполняется, и поэтому наша программа в этом случае будет преобразовывать не только буквы.

Стандартный головной файл <ctype.h>, описанный в приложении B, определяет семейство функций, которые позволяют проверять и преобразовывать литеры независимо от набора литер. Например, функция tolower(c)

возвращает букву c в коде нижнего регистра, если она была в коде верхнего регистра, поэтому `tolower(c)` – универсальная замена функции `lower(c)`, рассмотренной выше. Аналогично проверку

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

можно заменить на

```
isdigit(c)
```

Далее мы будем пользоваться функциями из `<ctype.h>`.

Существует одна тонкость, касающаяся преобразования литер в целые: язык не определяет, являются ли переменные типа `char` знаковыми или беззнаковыми. При преобразовании `char` в `int` может когда-нибудь получиться отрицательное целое? На машинах с разной архитектурой ответы могут отличаться. На некоторых машинах значение типа `char` с единичным старшим битом будет превращено в отрицательное целое (посредством «размножения знака»). На других – преобразование `char` в `int` осуществляется добавлением нулей слева, и, таким образом, получаемое значение всегда положительно.

Гарантируется, что любая литера из стандартного набора печатаемых литер никогда не будет отрицательным числом, поэтому в выражениях такие литеры всегда являются положительными операндами. Но произвольный восьмибитовый код в переменной типа `char` на одних машинах может быть отрицательным числом, а на других – положительным. Для совместимости переменные типа `char`, в которых хранятся нелитерные данные, следует специфицировать явно как `signed` или `unsigned`.

Отношения типа `i > j` и логические выражения, перемежаемые операторами `&&` и `||`, определяют выражение-условие, которое имеет значение 1, если оно истинно, и 0, если ложно. Так, присваивание

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

установит в `d` значение 1, если `c` есть цифра, и 0 в противном случае. Однако функции, подобные `isdigit`, в случае истины могут выдавать любое ненулевое значение. В местах проверок внутри `if`, `while`, `for` и т.д. «истина» просто означает «не нуль».

Неявные арифметические преобразования, как правило, осуществляются естественным образом. В общем случае, когда оператор типа `+` или `*` с двумя операндами (бинарный оператор) имеет разнотипные операнды, прежде чем операция начнёт выполняться, «младший» тип *подтягивается* к «старшему». Результат будет иметь старший тип. В разд. A.6 приложения A правила преобразования сформулированы точно. Если же в выражении нет беззнаковых операндов, можно удовлетвориться следующим набором неформальных правил:

- Если какой-либо из операндов принадлежит типу `long double`, то другой приводится к `long double`.

- В противном случае, если какой-либо из операндов принадлежит типу double, то другой приводится к double.
- В противном случае, если какой-либо из операндов принадлежит типу float, то другой приводится к float.
- В противном случае операнды типов char и short приводятся к int.
- И наконец, если один из операндов типа long, то другой приводится к long.

Заметим, что операнды типа float не приводятся автоматически к типу double; в этом данная версия языка отличается от первоначальной. Вообще говоря, математические функции, аналогичные собранным в библиотеке <math.h>, базируются на вычислениях с двойной точностью. В основном float используется для экономии памяти на больших массивах и менее часто для убыстрения счёта на тех машинах, где арифметика двойной точности слишком дорога.

Правила преобразования усложняются с появлением unsigned-операндов. Проблема в том, что сравнения знаковых и беззнаковых значений зависят от размеров целых типов, которые на разных машинах могут отличаться. Предположим, что значение типа int занимает 16 бит, а значение типа long – 32 бита. Тогда -1L < 1U, поскольку 1U принадлежит типу int и подтягивается к типу signed long. Но -1L > 1UL, так как -1L подтягивается к типу unsigned long и воспринимается как большое положительное число.

Преобразования имеют место и при присваиваниях: значение правой части присваивания приводится к типу левой части, который и является типом результата.

Литера превращается в целое посредством размножения знака или другим описанным выше способом.

Длинные целые преобразуются в короткие целые или в значения типа char с помощью отбрасывания старших разрядов. Так, в

```
int  i;
char c;
i = c;
c = i;
```

значение c не изменится. Это справедливо независимо от того, размножается знак при переводе char в int или нет. Однако, если изменить порядок присваиваний, возможна потеря информации.

Если x принадлежит типу float, а i типу float, то и x = i, и i = x вызовут преобразования, причём перевод float в int сопровождается отбрасыванием дробной части. Если double переводится в float, то значение либо округляется, либо обрезается; это зависит от реализации.

Так как аргумент в вызове функции есть выражение, при передаче его функции также возможно преобразование типа. При отсутствии прототипа функции аргументы типа char и short переводятся в int, а float – в double. Вот почему мы объявляли аргументы типа int или double даже тогда, когда в вызове функции использовали аргументы типа char или float.

И наконец, для любого выражения можно явно указать преобразование его типа, используя унарный оператор, называемый *приведением*. Конструкция вида

```
(имя-типа) выражение
```

приводит *выражение* к указанному в скобках типу по перечисленным выше правилам. Смысл операции приведения можно представить себе так: *выражение* как бы присваивается некоторой переменной указанного типа, и эта переменная используется вместо всей конструкции. Например, библиотечная программа sqrt рассчитана на аргумент типа double и выдаёт чепуху, если ей подсунуть что-нибудь другое. (sqrt описана в <math.h>.) Поэтому, если n есть целое, мы можем написать

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

и перед тем, как значение n будет передано функции, оно будет переведено в double. Заметим, что операция приведения всего лишь вырабатывает *значение* n указанного типа, но саму переменную n не затрагивает. Приоритет оператора приведения столь же высок, как и любого унарного оператора, что зафиксировано в таблице, показанной в конце этой главы.

В том случае, когда аргументы описаны в прототипе функции, как тому и следует быть, при вызове функции нужное преобразование включается автоматически. Так, при наличии прототипа функции sqrt:

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

перед обращением к sqrt в присваивании

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

целое 2 будет переведено в значение double 2.0 автоматически без явного указания операции приведения.

Операцию приведения проиллюстрируем на переносимой версии генератора псевдослучайных чисел и функции, осуществляющей начальную «затравку», входящих в стандартную библиотеку.

```
unsigned long int next = 1;

/* rand:  получает псевдослучайное целое 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
```

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

Упражнение 2.3. Напишите функцию `htoi(s)`, которая преобразует последовательность шестнадцатиричных цифр, начинающуюся с `0x` или `0X` в соответствующее целое. Шестнадцатиричными цифрами являются литеры `0...9`, `a...f`, `A...F`.

2.8. Инкрементные и декрементные операторы

В Си есть два необычных оператора, предназначенных для увеличения и уменьшения переменных. Инкрементный оператор `++` добавляет 1 к своему операнду, а декрементный оператор `--` вычитает 1. Мы уже неоднократно использовали `++` для наращивания значения переменных, как, например, в

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

Необычность `++` и `--` в том, что их можно использовать и как префиксные операторы (помещая перед переменной, например, `++n`), и как постфиксные операторы (помещая после переменной: `n++`). В обоих случаях значение `n` увеличивается на 1. Но выражение `++n` увеличивает `n` *до* того, как его значение будет использовано, а `n++` – *после* того. Предположим, что `n` содержит 5, тогда

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

установит в `x` значение 5, а

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

установит в `x` значение 6. И в том и другом случае значение `n` станет равным 6. Инкрементные и декрементные операторы можно применять только к переменным. Например, запись `(i+j)++` не верна. В контексте, где требуется только увеличить (или уменьшить) значение переменной, как в

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

безразлично, какой выбрать оператор – префиксный или постфиксный. Но существуют ситуации, когда требуется оператор вполне определённого типа. Например, рассмотрим функцию `squeeze(s,c)`, которая удаляет из стринга `s` все литеры, совпадающие с `c`:

```
/* squeeze:  удаляет все c из 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[i] = '\0';
}
```

Каждый раз, когда встречается литера, отличная от c, она копируется в текущую j-ю позицию, и только после этого переменная j продвигается на 1, подготавливаясь таким образом к приёму следующей литеры. Это в точности совпадает со следующими действиями:

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

Другой пример – функция getline, которая нам известна по гл. 1. Приведённую там запись

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

можно переписать более компактно:

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

В качестве третьего примера рассмотрим стандартную функцию strcat(s,t), которая стринг t помещает в конец стринга s. Предполагается, что в s достаточно пространства, чтобы в нём разместить суммарный стринг. Мы написали strcat так, что она не возвращает никакого результата. На самом деле библиотечная strcat возвращает ссылку на результирующий стринг.

```
/* strcat: помещает t в конец s; s достаточно большой */
void strcat(char s[], char t[])
{
    int i,j;

    i = j = 0;
    while (s[i] != '\0')  /* находим конец s */
        i++;
    while ((s[i++] = t[j++]) != '\0' /* копируем t */
        ;
}
```

При копировании очередной литеры из t в s постфиксный оператор ++ применяется и к i, и к j, чтобы на каждом шаге цикла переменные i и j правильно отслеживали позиции перемещаемой литеры.

Упражнение 2.4. Напишите версию функции squeeze(s1,s2), которая удаляет из s1 все литеры, встречающиеся в *стринге* s2.

Упражнение 2.5. Напишите функцию any(s1,s2), которая возвращает либо ту позицию в s1, где стоит первая литера, совпавшая с любой из литер в s2, либо −1 (если ни одна литера s1 не совпадает с литерами из s2). (Стандартная библиотечная функция strpbrk делает то же самое, но выдаёт указатель на литеру, а не номер её позиции.)

2.9. Побитовые операторы

В Си имеются шесть операторов для манипулирования с битами. Их можно применять только к целочисленным операндам, т.е. к операндам типов char, short, int, long, знаковым и беззнаковым.

&	побитовое И
|	побитовое ИЛИ
^	побитовое исключающее ИЛИ
<<	сдвиг влево
>>	сдвиг вправо
~	побитовое отрицание (унарный)

Оператор & (побитовое И) часто используется для обнуления некоторой группы разрядов. Например,

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

очищает в n все разряды, кроме младших семи.

Оператор | (побитовое ИЛИ) применяют для установки разрядов; так,

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

устанавливает единицы в тех разрядах x, которым соответствуют единицы в SET_ON.

Оператор ^ (побитовое исключающее ИЛИ) в каждом разряде установит 1, если соответствующие разряды операндов имеют различные значения, и 0, когда они совпадают.

Поразрядные операторы & и | следует отличать от логических операторов && и ||, которые при вычислении слева направо дают значение истинности. Например, если x есть 1, а y равно 2, то x & y даст нуль, а x && y – единицу.

Операторы << и >> выполняют сдвиг, влево или вправо, своего левого операнда на число битовых позиций, задаваемое правым операндом, которое должно быть положительным. Так, `x << 2` сдвигает значение `x` влево на 2 позиции, заполняя освобождающиеся биты нулями, что эквивалентно умножению `x` на 4. Сдвиг вправо беззнаковой величины всегда сопровождается заполнением освобождающихся разрядов нулями. Сдвиг вправо знаковой величины на одних машинах происходит с размножением знака («арифметический сдвиг»), на других – с заполнением освобождающихся разрядов нулями («логический сдвиг»).

Унарный оператор ~ производит дополнение целого до единиц по всем разрядам, т. е. превращает единичные биты в нулевые и наоборот. Например,

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

обнуляет в `x` последние 6 разрядов. Заметим, что запись `x & ~077` не зависит от длины слова, и, следовательно, она лучше, чем `x & 0177700`, поскольку последняя подразумевает, что `x` занимает 16 бит. Независимая от машины форма записи `~077` не потребует дополнительных затрат при счёте, так как `~077` – константное выражение, которое будет вычислено во время компиляции.

Для иллюстрации некоторых побитовых операций рассмотрим функцию `getbits(x,p,n)`, которая формирует поле в `n` бит, вырезанное из `x`, начиная с позиции `p`, прижимая его к правому краю. Предполагается, что 0-й бит – крайний правый бит, а `n` и `p` – разумные положительные числа. Например, `getbits(x,4,3)` вернёт в качестве результата 4, 3 и 2-й биты значения `x`, прижимая их к правому краю. Вот эта функция:

```
/* getbits:  получает 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 нам поле к правому краю. Константа `~0` состоит только из единиц, и её сдвиг влево на `n` бит `(~0 << n)` приведёт к тому, что правый край этой константы займут `n` нулевых разрядов. Ещё одна операция побитового отрицания, `~(~0 << n)`, позволяет получить справа `n` единиц.

Упражнение 2.6. Напишите функцию `setbits(x,p,n,y)`, возвращающую значение `x`, в котором `n` бит, начиная с `p`-й позиции, заменены на `n` правых разрядов из `y` (остальные биты не изменяются).

Упражнение 2.7. Напишите функцию `invert(x,p,n)`, возвращающую значение `x` с инвертированными `n` битами, начиная с позиции `p` (остальные биты не изменяются).

Упражнение 2.8. Напишите функцию `rightrot(x,n)`, которая циклически сдвигает («вращает») вправо `x` на `n` разрядов.

2.10. Операторы присваивания и выражения

Выражение типа

```
i = i + 2
```

в котором стоящая слева переменная повторяется и справа, можно написать в сжатом виде:

```
i += 2
```

Оператор += называется *оператором присваивания*.

Большинству бинарных операторов (аналогичных + и имеющих левый и правый операнды) соответствуют операторы присваивания *op*=, где *op* – один из операторов

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

Если *выр*$_1$ и *выр*$_2$ – выражения, то запись

выр$_1$ *op*= *выр*$_2$

эквивалентна записи

выр$_1$ = (*выр*$_1$) *op* (*выр*$_2$)

с той лишь разницей, что *выр*$_1$ вычисляется только один раз. Обратите внимание на скобки вокруг *выр*$_2$: запись

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

эквивалентна записи

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

но не

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

В качестве примера приведём функцию `bitcount`, подсчитывающую число единичных битов в своём аргументе целого типа.

```
/* bitcount: подсчёт 1 в x */
int bitcount(unsigned x)
{
    int b;

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

Независимо от машины, на которой будет работать эта программа, описание аргумента x как unsigned гарантирует, что при правом сдвиге освобождающиеся биты будут заполняться нулями, а не знаковым битом.

Помимо краткости операторы присваивания обладают тем преимуществом, что они более соответствуют тому, как человек мыслит. Мы говорим «прибавить 2 к i» или «увеличить i на 2», а не «взять i, добавить 2 и затем вернуть результат в i», так что выражение i += 2 лучше, чем i = i + 2. Кроме того, в сложных выражениях вроде

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

благодаря оператору присваивания запись становится более лёгкой для понимания, так как читателю при такой записи не потребуется старательно сравнивать два длинных выражения или выяснять, почему они не совпадают. Следует иметь в виду и то, что оператор присваивания может помочь компилятору сгенерировать более эффективный код.

Мы уже видели, что присваивание вырабатывает значение и может применяться внутри выражения; во многих рядовых программах мы видим

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

В выражениях встречаются и другие операторы присваивания (+=, -= и т.д.), хотя и реже.

Типом и значением любого выражения присваивания являются тип и значение его левого операнда после завершения присваивания.

Упражнение 2.9. Применительно к числам, в представлении которых использован дополнительный код, выражение x &= (x-1) уничтожает самую правую 1 в x. Объясните, почему. Используйте это наблюдение при написании более быстрого варианта функции bitcount.

2.11. Условные выражения

Инструкции

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

пересылают в z максимальное из двух значений, a и b. *Условное выражение*, написанное с помощью тернарного оператора «?:», представляет собой другой способ записи этой и подобных ей конструкций. В выражении

выр$_1$? *выр*$_2$: *выр*$_3$

первым вычисляется выражение *выр*$_1$. Если его значение не нуль (истина), то вычисляется выражение *выр*$_2$, и значение этого выражения становится значением всего условного выражения. В противном случае вычисляется выражение *выр*$_3$, и его значение становится значением условного выражения. Следует отметить, что из выражений *выр*$_2$ и *выр*$_3$ вычисляется только одно из них. Таким образом, чтобы установить в z наибольшее из a и b, можно написать

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

Следует заметить, что условное выражение и в самом деле является выражением, и его можно использовать в любом месте, где допускается выражение. Если *выр*$_2$ и *выр*$_3$ принадлежат разным типам, то тип результата определяется правилами преобразования, о которых шла речь в этой главе ранее. Например, если f имеет тип float, а n – тип int, то типом выражения

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

будет float, вне зависимости от того, положительно значение n или нет.

Заключать в скобки первое выражение в условном выражении не обязательно, так как приоритет ?: очень низкий (более низкий приоритет имеет только присваивание), однако мы рекомендуем всегда это делать, поскольку благодаря обрамляющим скобкам условие в выражении лучше воспринимается.

Условное выражение часто позволяет сократить программу. В качестве примера приведём цикл, обеспечивающий печать n элементов массива по 10 на каждой строке с одним пробелом между колонками; каждая строка цикла, включая последнюю, заканчивается литерой новая-строка:

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

Литера новая-строка посылается после каждого десятого и после n-го элемента. За всеми другими элементами следует пробел. Эта программа выглядит довольно замысловато, зато она более компактна, чем эквивалентная программа с использованием if-else. Вот ещё один пример:

```
printf("Вы имеете %d элемент%s.\n", n, (n%10==1 && n!=11) ? "" :
          ((n<10 || n>20) && n%10>=2 && n%10<=4) ? "а" : "ов");
```

Упражнение 2.10. Напишите функцию lower, которая переводит большие буквы в малые, используя условное выражение (вместо if-else).

2.12. Приоритет и порядок вычислений

В табл. 2.1 показаны приоритеты и порядок вычислений всех операторов, включая и те, которые мы ещё не рассматривали. Операторы, перечисленные на одной строке, имеют одинаковый приоритет; строки упорядочены по убыванию приоритетов; так, например, *, / и % имеют одинаковый

приоритет, который выше, чем приоритет бинарных + и −. «Оператор» () обозначает вызов функции. Операторы -> и . (точка) обеспечивают доступ к элементам структур; о них пойдёт речь в гл. 6, там же будет рассмотрен и оператор sizeof (размер объекта). Операторы * (адресация по указателю) и & (получение адреса объекта) обсуждаются в гл. 5. Оператор «запятая» будет рассмотрен в гл. 3.

Заметим, что приоритеты побитовых операторов &, ^ и | ниже, чем приоритет == и !=, из-за чего в побитовых проверках типа

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

чтобы получить правильный результат, приходится использовать скобки.

Си подобно многим языкам не фиксирует порядок вычисления операндов оператора (за исключением &&, ||, ?: и «,»). Например, в инструкции вида

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

f может быть вычислена раньше g или наоборот. Из этого следует, что если одна из функций изменяет значение переменной, от которой зависит другая функция, то помещаемый в x результат может зависеть от порядка вычислений. Чтобы обеспечить нужную последовательность вычислений, промежуточные результаты можно запоминать во временных переменных.

Порядок вычисления аргументов функции также не определён, поэтому на разных компиляторах

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

может давать несовпадающие результаты. Результат вызова функции зависит от того, когда компилятор сгенерирует команды увеличения n до или после обращения к power. Чтобы обезопасить себя от возможного побочного эффекта, достаточно написать

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

Обращения к функциям, вложенные присваивания, инкрементные и декрементные операторы дают «побочный эффект», проявляющийся в том, что при вычислении выражения значения некоторых переменных изменяются. В любом выражении с побочным эффектом может быть скрыта трудно просматриваемая зависимость результата выражения от порядка изменения значений переменных, входящих в выражение. В такой, например, типично неприятной ситуации –

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

возникает вопрос: массив a индексируется старым или изменённым значением i? Компиляторы могут по-разному генерировать программу, что проявится в интерпретации данной записи. Стандарт сознательно устроен так,

ТАБЛИЦА 2.1. ПРИОРИТЕТЫ И ПОРЯДОК ВЫЧИСЛЕНИЙ ОПЕРАТОРОВ

ОПЕРАТОРЫ	ВЫПОЛНЯЮТСЯ
`() [] -> .`	слева направо
`! ~ ++ -- + - * & (`*тип*`) sizeof`	справа налево
`* / %`	слева направо
`+ -`	слева направо
`<< >>`	слева направо
`< <= > >=`	слева направо
`== !=`	слева направо
`&`	слева направо
`^`	слева направо
`|`	слева направо
`&&`	слева направо
`||`	слева направо
`?:`	справа налево
`= += -= *= /= %= &= ^= |= <<= >>=`	справа налево
`,`	слева направо

Унарные операторы +, – и * имеют более высокий приоритет, чем те же операторы в бинарном варианте

что большинство подобных вопросов оставлено на усмотрение компиляторов, так как лучший порядок вычислений определяется архитектурой машины. Стандартом только гарантируется, что все побочные эффекты при вычислении аргументов проявятся перед входом в функцию. Правда, в примере с `printf` это нам не поможет.

Мораль такова: писать программы, которые зависят от порядка вычислений, – плохая практика, какой бы язык вы ни использовали. Естественно, надо знать, чего следует избегать, но если вы не знаете, *как* образуются побочные эффекты на вашей машине, то лучше и не рассчитывать на особенности реализации.

Глава 3

Управление

Порядок, в котором выполняются вычисления, определяется инструкциями управления. Мы уже встречались с наиболее распространёнными управляющими конструкциями такого рода в предыдущих примерах; здесь мы завершим их список и более точно определим рассмотренные ранее.

3.1. Инструкции и блоки

Выражение, скажем `x = 0`, или `i++`, или `printf(...)`, становится инструкцией, если в конце его поставить точку с запятой, как, например, в записи

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

В Си точка с запятой является заключающей литерой инструкции, а не разделителем, как в языке Паскаль.

Фигурные скобки { и } используются для объединения описаний и инструкций в *составную инструкцию*, или *блок*, чтобы с точки зрения синтаксиса эта новая конструкция воспринималась как одна инструкция. Фигурные скобки, обрамляющие группу инструкций, образующих тело функции, – это один пример; второй пример – это скобки, объединяющие инструкции, помещённые после `if`, `else`, `while` или `for`. (Переменные могут быть описаны внутри *любого* блока, об этом разговор пойдёт в гл. 4) После правой закрывающей фигурной скобки в конце блока точка с запятой не ставится.

3.2. Конструкция `if-else`

Инструкция `if-else` используется для принятия решения. Вот её синтаксис:

```
if (выражение)
    инструкция1
else
    инструкция2
```

причём `else`-часть может быть, а может и отсутствовать. Сначала вычисляется *выражение*, и, если оно истинно (т.е. отлично от нуля), выполняется *инструкция*$_1$. Если *выражение* ложно (т.е. его значение равно нулю) и существует `else`-часть, то выполняется *инструкция*$_2$.

Так как `if` просто проверяет числовое значение выражения, условие иногда можно записывать в сокращённом виде. Так, запись

```
if (выражение)
```

короче, чем

```
if (выражение != 0)
```

Иногда такие сокращения естественны и ясны, в других случаях, наоборот, затрудняют понимание программы.

Отсутствие в одной из вложенных друг в друга `if`-конструкций `else`-части может привести к неоднозначному толкованию записи. Эту неоднозначность разрешают тем, что `else` связывают с ближайшим `if`, у которого нет своего `else`. Например, в

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

`else` относится к внутреннему `if`, что мы и показали с помощью отступов. Если нам требуется иная интерпретация, необходимо должным образом расставить фигурные скобки:

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

Ниже приводится пример ситуации, когда неоднозначность особенно опасна:

```
if (n >= 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
        }
else        /* НЕВЕРНО */
    printf("ошибка -- отрицательное n\n");
```

С помощью отступов мы недвусмысленно показали, что нам нужно, однако компилятор не воспримет эту информацию и отнесёт `else` к внутреннему `if`. Такого рода ошибки особенно тяжело искать. Здесь уместен следующий совет: вложенные `if` обрамляйте фигурными скобками.

Кстати, обратите внимание на точку с запятой после `z = a` в

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

Здесь она обязательна, поскольку по правилам грамматики за `if` должна следовать инструкция, а выражение-инструкция типа `z = a;` всегда заканчивается точкой с запятой.

3.3. Конструкция `else-if`

Конструкция

```
if (выражение)
    инструкция
else if (выражение)
    инструкция
else if (выражение)
    инструкция
else if (выражение)
    инструкция
else
    инструкция
```

встречается так часто, что о ней стоит поговорить особо. Приведённая последовательность инструкций `if` – самый общий способ описания многоступенчатого принятия решения. *Выражения* вычисляются по порядку; как только встречается *выражение* со значением «истина», выполняется соответствующая ему *инструкция*; на этом последовательность проверок завершается. Здесь под словом *инструкция* имеется в виду либо одна инструкция, либо группа инструкций в фигурных скобках.

Последняя else-часть срабатывает, если все предыдущие условия не выполняются. Иногда в последней части не требуется производить никаких действий, в этом случае фрагмент

```
else
    инструкция
```

можно опустить или использовать для фиксации ошибочной («невозможной») ситуации.

В качестве иллюстрации трёхпутевого ветвления рассмотрим функцию бинарного поиска значения x в массиве v. Предполагается, что элементы v упорядочены по возрастанию. Функция выдаёт положение (число в пределах от 0 до $n-1$) x в v, если оно там встречается, и -1, если его нет.

При бинарном поиске значение x сначала сравнивается с элементом, занимающим серединное положение в массиве v. Если x меньше, чем это значение, то областью поиска становится «верхняя» половина массива v, в противном случае – «нижняя». В любом случае следующий шаг – это сравнение с серединным элементом отобранной половины. Процесс «уполовинивания» диапазона продолжается до тех пор, пока либо будет найдено значение, либо станет пустым диапазон. Запишем функцию бинарного поиска:

```
/* binsearch: найти 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     /* x найден */
            return mid;
    }
    return -1;   /* x не найден */
}
```

Основное действие, выполняемое на каждом шаге поиска, – сравнение значения x (меньше, больше или равно) с элементом v[mid]; это сравнение естественно поручить конструкции else-if.

Упражнение 3.1. В нашей программе бинарного поиска внутри цикла осуществляются две проверки, хотя могла быть только одна (при увеличении числа проверок вне цикла). Напишите программу, предусмотрев в ней одну проверку внутри цикла. Оцените разницу во времени счёта.

3.4. Переключатель

Инструкция switch используется для выбора одного из многих путей. Она проверяет, совпадает ли значение выражения с одним из значений, входящих в некоторое множество целых *констант*, и выполняет соответствующую этому значению ветвь программы:

```
switch (выражение) {
    case конст-выр: инструкции
    case конст-выр: инструкции
    default: инструкции
}
```

Каждая ветвь case помечена одной или несколькими целочисленными константами или же константными выражениями. Вычисления начинаются с той ветви case, в которой константа совпадает со значением выражения. Константы всех ветвей case должны отличаться друг от друга. Если выяснилось, что ни одна из констант не подходит, то выполняется ветвь, помеченная словом default, если таковая имеется, в противном случае ничего не делается. Ветви case и default можно располагать в любом порядке.

В гл. 1 мы написали программу, подсчитывающую число вхождений в текст каждой цифры, пробельных литер (пробелов, табуляций и новых-строк) и всех остальных литер. В ней мы использовали последовательность if...else if...else. Теперь приведём вариант этой программы с переключателем:

```
#include <stdio.h>
main()  /* подсчёт цифр, пробелов и прочих литер */
{
    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("цифры =");
        for (i = 0; i < 10; i++)
            printf(" %d", ndigit[i]);
        printf(", пробелы = %d, прочие = %d\n",
            nwhite, nother);
        return 0;
}
```

Инструкция `break` вызывает немедленный выход из переключателя. Поскольку выбор ветви `case` реализуется как переход на метку, то после выполнения одной ветви `case`, если ничего не предпринять, программа *провалится вниз* на следующую ветвь. Инструкции `break` и `return` – наиболее распространённые средства выхода из переключателя. Инструкция `break` используется также для принудительного выхода из циклов `while`, `for` и `do-while` (мы ещё поговорим об этом чуть позже).

«Сквозное» выполнение ветвей `case` вызывает смешанные чувства. С одной стороны, это хорошо, поскольку позволяет несколько ветвей `case` объединить в одну, как мы и поступили с цифрами в нашем примере. Но с другой – это означает, что в конце почти каждой ветви придётся ставить `break`, чтобы избежать перехода к следующей. Последовательный проход по ветвям – вещь ненадёжная, он чреват ошибками, особенно при изменении программы. За исключением случая с несколькими метками для одного вычисления, старайтесь по возможности редко пользоваться сквозным проходом, но если уж вы его применяете, обязательно комментируйте эти особые места.

Добрый вам совет: даже в конце последней ветви (после `default` в нашем примере) помещайте инструкцию `break`, хотя с точки зрения логики в ней нет необходимости. Но эта маленькая предосторожность спасёт вас, когда однажды вам потребуется добавить в конец ещё одну ветвь `case`.

Упражнение 3.2. Напишите функцию `escape(s,t)`, которая при копировании текста из `t` в `s` преобразует литеры типа новая-строка и табуляция в «видимые последовательности литер» (типа `\n` и `\t`). Используйте инструкцию `switch`. Напишите функцию, выполняющую обратное преобразование эскейп-последовательностей в настоящие литеры.

3.5. Циклы `while` и `for`

Мы уже встречались с циклами `while` и `for`. В цикле

```
while (выражение)
    инструкция
```

вычисляется *выражение*. Если его значение отлично от нуля, то выполняется *инструкция*, и вычисление выражения повторяется. Этот цикл продолжается до тех пор, пока выражение не станет равным нулю, после чего вычисления возобновятся с точки, расположенной сразу за *инструкцией*.

Инструкция for

```
for (выр1; выр2; выр3)
    инструкция
```

эквивалентна конструкции

```
выр1;
while (выр2) {
    инструкция
    выр3;
}
```

если не считать отличий в поведении инструкции continue, речь о которой пойдёт в разд. 3.7.

С точки зрения грамматики три компоненты цикла for представляют собой произвольные выражения, но чаще $выр_1$ и $выр_3$ – это присваивания или вызовы функций, а $выр_2$ – выражение отношения. Любое из этих трёх выражений может отсутствовать, но точку с запятой опускать нельзя. При отсутствии $выр_1$ или $выр_3$ считается, что их просто нет в конструкции цикла; при отсутствии $выр_2$ полагается, что его значение как бы всегда истинно. Например,

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

есть «бесконечный» цикл, выполнение которого, вероятно, прерывается каким-то другим способом, например с помощью инструкций break, или return.

Какой цикл выбрать: while или for – это дело вкуса. Так, в

```
while ((c=getchar()) == ' ' || c == '\n' || c == '\t')
    ;   /* обойти пробельные литеры */
```

нет ни инициализации, ни пересчёта параметра, поэтому здесь больше подходит while.

Там, где есть простая инициализация и пошаговое увеличение значения некоторой переменной, больше подходит цикл for, так как в этом цикле организующая его часть сосредоточена в начале записи. Например, начало цикла, обрабатывающего первые n элементов массива, имеет следующий вид:

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

Это похоже на DO-циклы в Фортране и for-циклы в Паскале. Сходство, однако, не очень точное, так как в Си индекс и его предельное значение могут изменяться внутри цикла, и значение индекса i после выхода из цикла всегда определено.

Поскольку три компоненты цикла могут быть произвольными выражениями, организация for-циклов не ограничивается только случаем арифметической прогрессии. Однако включать в заголовок цикла вычисления, не имеющие отношения к инициализации и инкрементации, считается плохим стилем. Заголовок лучше оставить только для операций управления циклом.

В качестве более внушительного примера приведём другую версию программы atoi, выполняющей преобразование стринга в его числовой эквивалент. Это более общая версия по сравнению с рассмотренной в гл. 2, в том смысле, что она игнорирует левые пробельные литеры (если они есть) и должным образом реагирует на знаки + и –, которые могут стоять перед цепочкой цифр. (В гл. 4 будет рассмотрен вариант atof, который осуществляет подобное преобразование для чисел с плавающей точкой.)

Структура программы отражает вид вводимой информации:

```
проигнорировать пробельные литеры, если они есть
получить знак, если он есть
взять целую часть и преобразовать её
```

На каждом шаге выполняется определённая часть работы и чётко фиксируется её результат, который затем используется на следующем шаге. Обработка данных заканчивается на первой же литере, которая не может быть частью числа.

```
#include <ctype.h>
/* atoi: преобразование s в целое число; версия 2 */
int atoi(char s[])
{
    int i, n, sign;
    for (i = 0; isspace(s[i]); i++)
        ;           /* проигнорировать пробельные литеры */
    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;
}
```

Заметим, что в стандартной библиотеке имеется более совершенная функция преобразования стринга в длинное целое – strtol (см. разд. B.5 приложения B).

Преимущества, которые даёт централизация управления циклом, становятся ещё более очевидными, когда несколько циклов вложены друг в друга. Проиллюстрируем их на примере сортировки массива целых чисел методом Шелла, предложенным им в 1959 г. Основная идея этого алгоритма в том, что на ранних стадиях сравниваются далеко отстоящие друг от друга, а не соседние элементы, как в обычных перестановочных сортировках. Это приводит к быстрому устранению массовой неупорядоченности, благодаря чему на более поздней стадии остаётся меньше работы. Интервал между сравниваемыми элементами постепенно уменьшается до единицы, и в этот момент сортировка сводится к обычным перестановкам соседних элементов. Программа shellsort имеет следующий вид:

```
/* shellsort: сортируются v[0]...v[n-1] в возр.порядке */
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;
            }
}
```

Здесь использованы три вложенных друг в друга цикла. Внешний управляет интервалом gap между сравниваемыми элементами, сокращая его с помощью деления пополам от n/2 до нуля. Средний цикл перебирает элементы. Внутренний – сравнивает каждую пару элементов, отстоящих друг от друга на расстоянии gap, и осуществляет перестановку элементов в неупорядоченных парах. Так как gap обязательно будет сведён к единице, все элементы в конечном счёте будут упорядочены. Обратите внимание на то, что универсальность цикла for позволяет сделать внешний цикл по форме похожим на другие, хотя он и не является циклом типа арифметической прогрессии.

Последний оператор из числа операторов Си «,» чаще всего используют в инструкции for. Пара выражений, разделённых запятой, вычисляется слева направо. Типом и значением результата являются тип и значение правого выражения, что позволяет в инструкции for в каждой из трёх компонент иметь по нескольку выражений, например вести два индекса параллельно. Продемонстрируем это на примере функции reverse(s), которая «переворачивает» стринг s, оставляя результат в том же стринге s:

```
#include <string.h>
/* reverse: переворачивает стринг s (результат в 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;
    }
}
```

Запятые, разделяющие аргументы функции, переменные в описаниях и т.д. *не* являются операторами-запятыми и не обеспечивают вычислений слева направо.

Запятыми как операторами следует пользоваться умеренно. Более всего они уместны в конструкциях, которые тесно связаны друг с другом (как в for-цикле программы reverse), а также в макросах, в которых многоступенчатые вычисления должны быть выражены одним выражением. Запятой-оператором в программе reverse можно было бы воспользоваться и при обмене литерами в проверяемых парах элементов стринга, мысля этот обмен как одну отдельную операцию:

```
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), восстанавливающую сокращённую запись типа a-z в стринге s1 до эквивалентной полной записи abc...xyz в s2. В s1 допускаются буквы (прописные и строчные) и цифры. Следует уметь справляться со случаями типа a-b-c, a-z0-9 и -a-b. Считайте знак - в начале или в конце s1 обычной литерой минус.

3.6. Цикл do-while

Как мы говорили в гл. 1, в циклах while и for проверка условия окончания цикла выполняется наверху. В Си имеется ещё один вид цикла, do-while, в котором эта проверка в отличие от while и for делается внизу *после* каждого прохождения тела цикла, т.е. после того, как тело выполнится по крайней мере один раз.

Цикл do-while имеет следующий синтаксис:

```
do
    инструкция
while (выражение);
```

Сначала выполняется *инструкция*, затем вычисляется *выражение*. Если оно истинно, то *инструкция* выполняется снова и т.д. Когда выражение становится ложным, цикл заканчивает работу. Цикл do-while эквивалентен циклу

repeat-until в Паскале с той лишь разницей, что в первом случае указывается условие продолжения цикла, а во втором – условие его окончания.

Опыт показывает, что цикл do-while гораздо реже используется, чем while и for. Тем не менее потребность в нём время от времени возникает, как, например, в функции itoa (обратной по отношению к atoi), преобразующей число в стринг литер. Выполнить такое преобразование оказалось несколько более сложным делом, чем ожидалось, поскольку простые алгоритмы генерируют цифры в обратном порядке. Мы остановились на варианте, в котором сначала формируется цепочка цифр, а затем она реверсируется.

```
/* itoa: преобразование n в стринг s */
void itoa(int n, char s[])
{
    int i, sign;
    if ((sign = n) < 0) /* сохраняем знак */
        n = -n;         /* делаем n положительным */
    i = 0;
    do {    /* генерируем цифры в обратном порядке */
        s[i++] = n % 10 + '0';  /* следующая цифра */
    } while ((n /= 10) > 0);    /* исключить её */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

Конструкция do-while здесь необходима или по крайней мере удобна, поскольку в s посылается хотя бы одна литера, даже если n равно нулю. В теле цикла одну инструкцию мы выделили фигурными скобками (хотя они и избыточны), чтобы неискушённый читатель не принял по ошибке слово while за *начало* цикла while.

Упражнение 3.4. При условии, что для представления чисел используется дополнительный код, наша версия itoa не справляется с самым большим по модулю отрицательным числом, значение которого равняется -2^{n-1} (где n – размер слова). Объясните, чем это вызвано. Модифицируйте программу таким образом, чтобы она давала правильное значение указанного числа независимо от машины, на которой выполняется.

Упражнение 3.5. Напишите функцию itob(n,s,b), которая переводит целое n в стринг s, представляющий число по основанию b. В частности, itob(n,s,16) помещает в s текст числа n в шестнадцатиричном виде.

Упражнение 3.6. Напишите версию itoa с дополнительным третьим аргументом, задающим минимальную ширину поля. При необходимости преобразованное число слева должно дополняться пробелами.

3.7. Инструкции break и continue

Иногда бывает удобно выйти из цикла не по результату проверки, осуществляемой в начале или в конце цикла, а каким-то другим способом. Такую возможность для циклов for, while и do-while, а также для переключателя switch предоставляет инструкция break. Эта инструкция вызывает немедленный выход из самого внутреннего из объемлющих её циклов или переключателей.

Следующая функция, trim, удаляет из стринга хвостовые литеры пробелов, табуляций и новых-строк; break используется в ней для выхода из цикла по первой обнаруженной справа литере, отличной от названных.

```
/* trim: удаляет хвостовые пробелы, таб. и нов.-строки */
int trim(char[])
{
    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;
}
```

С помощью функции strlen можно получить длину стринга. Цикл for просматривает его в обратном порядке, начиная с конца, до тех пор, пока не встретится литера, отличная от пробела, табуляции и новой-строки. Цикл прерывается, как только такая литера обнаружится или n станет отрицательным (т.е. вся строка будет просмотрена). Убедитесь, что функция ведёт себя правильно и в случаях, когда стринг пуст или состоит только из пробельных литер.

Инструкция continue в чём-то похожа на break, но применяется гораздо реже. Она вынуждает ближайший объемлющий её цикл (for, while или do-while) начать следующий шаг итерации. Для while и do-while это означает немедленный переход к проверке условия, а для for – к приращению шага. Инструкцию continue можно применять только к циклам, но не к switch. Будучи помещённой внутрь switch, расположенного в цикле, она вызовет переход к следующей итерации этого цикла.

Вот фрагмент программы, обрабатывающий только неотрицательные элементы массива a (отрицательные – пропускаются).

```
for (i = 0; i < n; i++) {
    if (a[i] < 0)  /* пропуск отрицательных элементов */
        continue;
    ... /* обработка положительных элементов */
}
```

К инструкции continue часто прибегают в тех случаях, когда оставшаяся часть цикла сложна, а замена условия в нём на противоположное и введение еще одного уровня приводит к слишком большому числу уровней вложенности.

3.8. Инструкция goto и метки

В Си имеются порицаемая многими инструкция goto и метки для перехода на них. Строго говоря, в этой инструкции нет никакой необходимости, и на практике почти всегда можно легко без неё обойтись. До сих пор в нашей книге мы не использовали goto.

Однако существуют случаи, в которых goto может пригодиться. Наиболее типична ситуация, когда необходимо прервать обработку в некоторой глубоко вложенной структуре и выйти сразу из двух или бóльшего числа вложенных циклов. Инструкция break здесь не поможет, так как она осуществляет выход только из самого внутреннего цикла. В качестве примера рассмотрим следующую конструкцию:

```
    for (...)
        for (...) {
            ...
            if (disaster)          /* если бедствие */
                goto error;        /* уйти на ошибку */
        }
    ...
error:                             /* обработка ошибки */
    ликвидировать беспорядок
```

Такая организация программы удобна, если подпрограмма обработки ошибочной ситуации не тривиальна и ошибка может встретиться в нескольких местах.

Метка имеет вид обычного имени переменной, за которым следует двоеточие. На метку можно перейти с помощью goto из любого места данной функции, т.е. метка действительна на протяжении всей функции.

В качестве ещё одного примера рассмотрим такую задачу: определить, есть ли в массивах a и b совпадающие элементы. Один из возможных вариантов её реализации имеет следующий вид:

```
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
```

```
            if (a[i] == b[j])
                goto found;
    /* нет одинаковых элементов */
    ...
found:
    /* обнаружено совпадение: a[i] == b[j] */
    ...
```

Программу нахождения совпадающих элементов можно написать и без goto, правда, заплатив за это дополнительными проверками и ещё одной переменной:

```
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)
    /* обнаружено совпадение: a[i-1] == b[j-1] */
    ...
else
    /* нет одинаковых элементов */
    ...
```

За исключением редких случаев, подобных только что приведённым, программы с привлечением goto, как правило, труднее для понимания и сопровождения, чем программы решающие те же задачи без goto. Хотя мы и не догматики, всё же думается, что к goto следует прибегать крайне редко, если использовать вообще.

Глава 4

Функции и структура программы

Функции подразделяют большие вычислительные задачи на более мелкие и позволяют воспользоваться тем, что уже сделано другими разработчиками, а не начинать создание программы каждый раз «с нуля». В выбранных должным образом функциях «упрятаны» несущественные для других частей программы детали их функционирования, что делает программу в целом более ясной и облегчает внесение в неё изменений.

Язык проектировался так, чтобы функции были эффективными и простыми в использовании. Обычно программы на Си состоят из большого числа небольших функций, а не из немногих больших. Программу можно располагать в одном или нескольких исходных файлах. Эти файлы можно компилировать отдельно, а загружать вместе, в том числе и с ранее откомпилированными библиотечными функциями. Процесс загрузки здесь не рассматривается поскольку он различен в разных системах.

Описание и определение функции – это та область, где стандартом ANSI в язык внесены самые существенные изменения. Как мы видели в гл. 1, в описании функции теперь разрешено задавать типы аргументов. Синтаксис определения функции также изменён, так что теперь описания и определения функций соответствуют друг другу. Это позволяет компилятору обнаруживать много больше ошибок, чем раньше. Кроме того, если типы аргументов соответствующим образом описаны, то необходимые преобразования аргументов выполняются автоматически.

Стандарт вносит ясность в правила, определяющие области действия имён; в частности, он требует, чтобы для каждого внешнего объекта было только одно определение. В нём обобщены средства инициализации: теперь можно инициализировать автоматические массивы и структуры.

Улучшен также препроцессор Си. Он включает более широкий набор директив условной компиляции, предоставляет возможность из макроаргументов генерировать стринги в кавычках, а кроме того, содержит более совершенный механизм управления процессом макрорасширения.

4.1. Основные сведения о функциях

Начнём с того, что сконструируем программу, печатающую те строки вводимого текста, в которых содержится некоторый «образец», заданный в виде стринга литер. (Эта программа представляет собой частный случай функции grep системы UNIX.) Рассмотрим пример: в результате поиска образца "ould" в строках текста

```
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!
```

мы получим

```
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!
```

Работа по поиску образца чётко распадается на три этапа:

```
while (существует ещё строка)
    if (строка содержит образец)
        напечатать её
```

Хотя все три составляющие процесса поиска можно поместить в функцию main, всё же лучше сохранить приведённую структуру и каждую её часть реализовать в виде отдельной функции. Легче иметь дело с тремя небольшими частями, чем с одной большой, поскольку, если несущественные особенности реализации скрыты в функциях, вероятность их нежелательного воздействия друг на друга минимальна. Кроме того, оформленные в виде функций соответствующие части могут оказаться полезными и в других программах.

Конструкция «while (*существует ещё строка*)» реализована в getline, см. гл. 1, а фразу «*напечатать её*» можно записать с помощью готовой функции printf. Таким образом, нам остаётся перевести на Си только то, что определяет, входит ли заданный образец в строку.

Чтобы решить эту задачу, мы напишем функцию strindex(s,t), которая указывает место (индекс) в строке s, где начинается строка t, или –1, если s не содержит t. Так как в Си нумерация элементов в массивах начинается с нуля, отрицательное число –1 подходит в качестве сигнала неудачного поиска. Если далее нам потребуется более сложное отождествление по образцу,

мы просто заменим strindex на другую функцию, оставив при этом остальную часть программы без изменений. (Библиотечная функция strstr аналогична функции strindex и отличается от последней только тем, что выдаёт указатель, а не индекс.)

После такого проектирования программы её «деталировка» оказывается очевидной. Мы имеем представление о программе в целом и знаем, как взаимодействуют её части. В нашей программе образец для поиска задаётся стрингом-литералом, что снижает её универсальность. В гл. 5 мы ещё вернёмся к проблеме инициализации литерных массивов и покажем, как образец сделать параметром, устанавливаемым при запуске программы. Здесь приведена несколько изменённая версия функции getline, и было бы поучительно сравнить её с версией, рассмотренной в гл. 1.

```
#include <stdio.h>
#define MAXLINE 1000 /* макс-ный размер вводимой строки */
int getline(char line[], int max);
int strindex(char sourse[], char searchfor[]);
char pattern[] = "ould"; /* образец для поиска */

/* найти все строки, содержащие 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: читает строку в s, возвращает 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: вычисляет место t в s
   или выдаёт -1, если  t  нет в s */
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;
}
```

Определение любой функции имеет следующий вид:

```
тип-результата имя-функции(декларации-аргументов)
{
    декларации и инструкции
}
```

Отдельные части определения могут отсутствовать, как, например, в определении «минимальной» функции

```
dummy() { }
```

которая ничего не вычисляет и ничего не выдаёт. Такая ничего не делающая функция в процессе разработки программы бывает полезна в качестве «хранителя места». Если тип результата опущен, то предполагается, что функция возвращает значение типа int.

Любая программа – это просто совокупность определений переменных и функций. Связи между функциями осуществляются через аргументы, возвращаемые значения и внешние переменные. В исходном файле функции разрешается располагать в любом порядке; исходную программу можно разбивать на любое число файлов, но так, чтобы ни одна из функций не оказалась разрезанной.

Инструкция return реализует механизм возврата результата от вызываемой к вызывающей функции. За словом return может следовать любое выражение:

```
return выражение;
```

Если потребуется, *выражение* будет приведено к типу *тип-результата*. Часто *выражение* заключают в скобки, но они не обязательны.

Вызывающая функция в праве проигнорировать возвращаемое значение. Более того, *выражение* в return может отсутствовать, и тогда вообще никакое значение не будет возвращено в вызывающую функцию. Управление

возвращается в вызывающую функцию без результирующего значения также и в том случае, когда вычисления достигли «конца» (т.е. последней закрывающей фигурной скобки функции). Не запрещена (но должна вызывать настороженность) ситуация, когда в одной и той же функции одни return имеют при себе выражения, а другие – не имеют. Во всех случаях, когда функция «забыла» передать результат в return, её значение есть «мусор».

Функция main в программе поиска по образцу выдаёт в качестве результата количество найденных строк. Это число доступно в той среде, из которой данная программа была вызвана.

Механизмы компиляции и загрузки Си-программ, расположенных в нескольких исходных файлах, могут различаться разных системах. В системе UNIX, например, эти работы выполняет упомянутая в гл. 1 команда cc. Предположим, что три функции нашего последнего примера расположены в трёх разных файлах: main.c, getline.c и strindex.c. Тогда команда

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

скомпилирует указанные файлы, поместив результат компиляции в файлы объектных модулей main.o, getline.o и strindex.o, и затем загрузит их в исполняемый файл a.out. Если обнаружилась ошибка, например, в main.c, то его можно скомпилировать снова и результат загрузить с ранее полученными объектными файлами, выполнив следующую команду:

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

Команда cc использует стандартные окончания файлов «.c» и «.o», чтобы отличать исходные файлы от объектных.

Упражнение 4.1. Напишите функцию strrindex(s,t), которая выдаёт позицию *самого правого* вхождения t в s или –1, если вхождения не обнаружено.

4.2. Функции, возвращающие нецелые значения

В предыдущих примерах функции либо вообще не возвращали результирующих значений (void), либо возвращали целые значения (int). А как быть, когда результат функции должен иметь другой тип? Многие вычислительные функции, как, например, sqrt, sin и cos, возвращают значения типа double; другие специальные функции могут выдавать значения ещё каких-то типов. Чтобы проиллюстрировать, каким образом сделать так, чтобы функция возвращала нецелое значение, напишем функцию atof(s), которая переводит стринг s в соответствующее число с плавающей точкой двойной точности. Функция atof представляет собой расширение функции atoi, две версии которой были рассмотрены в гл. 2 и 3. Она имеет дело со знаком (которого может и не быть), с десятичной точкой, а также с целой и дробной частями, одна из которых может отсутствовать. Наша версия *не* является высококачественной программой преобразования вводимых чисел; такая

программа потребовала бы заметно больше памяти. Функция atof входит в стандартную библиотеку программ; её описание содержится в головном файле <stdlib.h>.

Прежде всего отметим, что декларировать тип возвращаемого значения должна сама atof, так как этот тип не есть int. Указатель типа задаётся перед именем функции.

```
#include <ctype.h>
/* atof: преобразование стринга s в double */
double atof(char s[])
{
    double val, power;
    int i, sign;
    for (i = 0; isspace(s[i]); i++)
        ;        /* игнорирование левых пробельных литер */
    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 *= 10.0;
    }
    return sign * val / power;
}
```

Кроме того, важно, чтобы вызывающая программа знала, что atof возвращает нецелое значение. Один из способов обеспечить это – явно описать atof в вызывающей программе. Подобное описание демонстрируется ниже в программе простенького калькулятора (достаточного для проверки баланса чековой книжки), который каждую вводимую строку воспринимает как число, прибавляет его к текущей сумме и печатает её новое значение.

```
#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 декларации

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

говорится, что sum – переменная типа double, а atof – функция, которая принимает аргумент (один) типа char[] и возвращает результат типа double.

Описание и определение функции atof должны соответствовать друг другу. Если в одном исходном файле сама функция atof и обращение к ней в main имеют разные типы, то это несоответствие будет зафиксировано компилятором как ошибка. Но если функция atof была скомпилирована отдельно (что более вероятно), то несоответствие типов не будет обнаружено, и atof возвратит значение типа double, которое функция main воспримет как int, что приведёт к бессмысленному результату.

Это последнее утверждение, вероятно, вызовет у вас удивление, поскольку ранее говорилось о необходимости соответствия описаний и определений. Причина несоответствия, возможно, будет следствием того, что вообще отсутствует прототип функции, и функция неявно декларируется при первом своём появлении в выражении, как, например, в

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

Если в выражении встретилось имя, нигде ранее не описанное, за которым следует открывающая скобка, то такое имя по контексту считается именем функции, возвращающей результат типа int; при этом относительно её аргументов ничего не предполагается. Если в декларации функции аргументы не указаны, как в

```
double atof();
```

то и в этом случае считается, что ничего об аргументах atof не известно, и все проверки на соответствие её параметров будут выключены. Предполагается, что такая специальная интерпретация пустого списка позволит новым компиляторам транслировать старые Си-программы. Если у функции есть аргументы, опишите их, если их нет, используйте слово void.

Располагая соответствующим образом описанной функцией atof, мы можем написать функцию atoi, преобразующую стринг литер в целое значение, следующим образом:

```
/* atoi: преобразование стринга s в int с помощью atof */
int atoi(char s[])
{
    double atof(char s[]);
    return (int) atof(s);
}
```

Обратите внимание на вид декларации и инструкции return. Значение выражения в

```
return выражение;
```

приводится к типу функции перед тем, как оно будет передано в качестве результата. Следовательно, поскольку функция atoi возвращает значение int, результат вычисления atof типа double в инструкции return автоматически преобразуется в тип int. При преобразовании возможна потеря информации, и некоторые компиляторы предупреждают об этом. Оператор приведения явно указывает на необходимость преобразования типа и подавляет любое предупреждающее сообщение.

Упражнение 4.2. Дополните функцию atof таким образом, чтобы она справлялась с числами вида

```
123.45e-6
```

в которых в конце может стоять e (или E) с последующей экспонентой (быть может, со знаком).

4.3. Внешние переменные

Программа на Си обычно оперирует с множеством внешних объектов: переменных и функций. Прилагательное «внешний» («external») противоположно прилагательному «внутренний» («internal»), которое относится к аргументам и переменным, определяемым внутри функций. Внешние переменные специфицируются вне функций и потенциально доступны для многих функций. Сами функции всегда являются внешними объектами, поскольку в Си запрещено определять функции внутри других функций. По умолчанию одинаковые внешние имена, используемые в разных файлах, ссылаются на один и тот же внешний объект (функцию). (В стандарте это называется *соединением внешних связей* (*external linkage*).) В этом смысле внешние переменные похожи на области COMMON в Фортране и на переменные самого внешнего блока в Паскале. Позже мы покажем, как внешние функции и переменные сделать видимыми только внутри одного исходного файла.

Поскольку внешние переменные доступны всюду, их можно использовать в качестве связующих данных между функциями как альтернативу связей через аргументы и возвращаемые значения. Для любой функции внешняя переменная доступна по её имени, если это имя было должным образом описано.

Если число переменных, совместно используемых функциями, велико, связи между последними через внешние переменные могут оказаться более удобными и эффективными, чем длинные списки аргументов. Но, как отмечалось в гл. 1, к этому заявлению следует относиться критически, поскольку

такая практика ухудшает структуру программы и приводит к слишком большому числу связей между функциями по данным.

Внешние переменные полезны, так как они имеют бóльшую область действия и время жизни. Автоматические переменные действительны только внутри функции, они возникают в момент входа в функцию и исчезают при выходе из неё. Внешние переменные, напротив, существуют постоянно, так что их значения сохраняются в интервалах между обращениями к функциям. Таким образом, если двум функциям приходится пользоваться одними и теми же данными и ни одна из них не вызывает другую, то часто бывает удобно оформить эти общие данные в виде внешних переменных, а не передавать их в функцию и обратно через аргументы.

В связи с приведёнными рассуждениями разберём пример. Поставим себе задачу написать программу-калькулятор, понимающую операторы +, –, * и /. Такой калькулятор легче будет написать, если ориентироваться на польскую, а не инфиксную запись выражений. (Обратная польская запись применяется в некоторых карманных калькуляторах и в языках типа Forth и Postscript.)

В обратной польской записи каждый оператор следует за своими операндами. Выражение в инфиксной записи, скажем

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

в польской записи представляется как

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

Скобки не нужны, неоднозначности в вычислениях не бывает, поскольку известно, сколько операндов требуется для каждого оператора.

Реализовать нашу программу весьма просто. Каждый операнд посылается в стек; если встречается оператор, то соответствующее число операндов (два в случае бинарных операторов) берётся из стека и выполняется операция, после чего результат посылается в стек. В нашем примере числа 1 и 2 посылаются в стек, затем они замещаются на их разность -1. Далее в стек посылаются числа 4 и 5, которые затем заменяются на их сумму, 9. Числа -1 и 9 заменяются в стеке на их произведение (т.е. на -9). Встретив литеру новая-строка, программа выталкивает значение из стека и печатает его.

Программа, следовательно, состоит из цикла, обрабатывающего на каждом своём шаге очередной встречаемый оператор или операнд:

```
while (следующий элемент не конец-файла)
    if (число)
        послать его в стек
    else if (оператор)
        взять из стека операнды
        выполнить операцию
        результат послать в стек
    else if (новая-строка)
```

```
        взять с вершины стека число и напечатать
    else
        ошибка
```

Операции «послать в стек» и «взять из стека» сами по себе тривиальны, однако необходимость добавления механизмов обнаружения и нейтрализации ошибок делает их достаточно длинными. Поэтому эти операции стоит оформить в виде функций. Также разумно иметь отдельную функцию, получающую очередной оператор или операнд.

Главный вопрос, который мы ещё не рассмотрели, – это вопрос о том, где расположить стек и каким функциям разрешить к нему прямой доступ. Стек можно расположить в функции main и передавать сам стек и текущую позицию в нём в качестве аргументов функциям push («послать в стек») и pop («взять из стека»). Но функции main нет дела до переменных, относящихся к стеку, – ей нужны только операции push и pop. Поэтому мы решили стек и связанную с ним информацию хранить во внешних переменных, доступных для push и pop, но не для main.

Переход от эскиза к конкретной программе достаточно лёгок. Если теперь программу представить как текст, расположенный в некотором исходном файле, она будет иметь следующий вид:

```
#include...  /* могут быть в любом количестве */
#define...   /* могут быть в любом количестве */
описания функций для main
main() { ... }
внешние переменные для push и pop
void push(double f) { ... }
double pop(void) { ... }
int getop(char s[]) { ... }
подпрограммы, вызываемые функцией getop
```

Позже мы обсудим, как текст этой программы можно разбить на два или большее число файлов.

Функция main – это цикл, содержащий большой переключатель, передающий управление на ту или иную ветвь в зависимости от типа оператора или операнда. Здесь представлен более типичный случай применения переключателя по сравнению с рассмотренным в разд. 3.4.

```
#include <stdio.h>
#include <stdlib.h>    /* для atof() */
#define MAXOP 100      /* макс. размер опер-да или опер-ра */
#define NUMBER '0'     /* признак числа */
int getop(char []);
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("ошибка: деление на нуль\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("ошибка: неизвестная операция %s\n", s);
            break;
        }
    }
    return 0;
}
```

Так как операторы + и * коммутативны, порядок, в котором операнды берутся из стека, не важен, однако в случае операторов - и / левый и правый операнды должны различаться. Так, в

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

порядок, в котором выполняются обращения к pop, не определён. Чтобы

гарантировать правильный порядок, необходимо первое значение из стека присвоить временной переменной, как это и сделано в main.

```
#define MAXVAL  100   /* максимальная глубина стека */
int sp = 0;           /* позиция след. своб. эл-та стека */
double val[MAXVAL];   /* стек */

/* push: положить значение f в стек */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("ошибка: стек полон, %g не помещается\n", f);
}

/* pop: взять с вершины стека и выдать в качестве рез-та */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("ошибка: стек пуст\n");
        return 0.0;
    }
}
```

Переменная считается внешней, если она определена вне функций. Таким образом, стек и индекс стека, которые должны быть доступны и для push, и для pop, определяются вне этих функций. Но main не использует ни стек, ни позицию в стеке, и поэтому их представление может быть скрыто от main.

Займёмся реализацией getop – функции, получающей следующий оператор или операнд. Нам предстоит решить довольно простую задачу. Более точно: требуется пропустить пробелы и табуляции; если следующая литера – не цифра и не десятичная точка, то выдать её; в противном случае накопить цепочку цифр с десятичной точкой, если она есть, и выдать признак числа NUMBER в качестве результата.

```
#include <ctype.h>
int getch(void);
void ungetch(int);

/* getop: получает следующий оператор или операнд */
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))        /* накапливаем целую часть */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')          /* накапливаем дробную часть */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

Как работают функции getch и ungetch? Во многих случаях программа не может «сообразить», прочла ли она всё, что требуется, пока не прочтёт лишнего. Так, накопление числа производится до тех пор, пока не встретится литера, отличная от цифры. Но это означает, что программа прочла на одну литеру больше, чем нужно, а именно литеру, которую нельзя включать в число.

Эту проблему можно было бы решить при наличии обратной чтению операции «положить-назад», с помощью которой можно было бы вернуть преждевременно прочитанную литеру. Тогда каждый раз, когда программа считает на одну литеру больше, чем требуется, эта операция возвращала бы её вводу, и поэтому остальная часть программы могла бы вести себя так, как будто эта литера и вовсе не читалась. К счастью, описанный механизм обратной посылки литеры легко моделируется с помощью пары согласованных друг с другом функций, из которых getch поставляет очередную литеру из ввода, а ungetch отправляет назад литеру во входной поток, так что при следующем обращении к getch мы вновь её получим.

Нетрудно догадаться, как они работают вместе. Функция ungetch запоминает посылаемую назад литеру в некотором буфере, представляющем собой массив литер, доступный для обеих этих функций; getch читает из буфера, если там что-то есть, или обращаются к getchar, если буфер пустой. Следует предусмотреть индекс, указывающий на положение текущей литеры в буфере.

Так как функции getch и ungetch совместно используют буфер и индекс, значения последних должны сохраняться между вызовами. Поэтому

буфер и индекс должны быть внешними по отношению к этим программам, и мы можем записать getch, ungetch и общие для них переменные в следующем виде:

```
#define BUFSIZE 100
char buf[BUFSIZE];  /* буфер для ungetch */
int bufp = 0;       /* след. свободная позиция в буфере */

int getch(void)     /* дай (возможно возвращённую) литеру */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}
void ungetch(int c) /* верни литеру на ввод */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: слишком много литер\n");
    else
        buf[bufp++] = c;
}
```

Стандартная библиотека включает функцию ungetc, обеспечивающую возврат одной литеры (см. гл. 7). Мы же, чтобы проиллюстрировать более общий подход, для запоминания возвращаемых литер использовали массив.

Упражнение 4.3. Исходя из предложенной нами схемы, дополните программу-калькулятор таким образом, чтобы она «понимала» оператор взятия модуля (%) и отрицательные числа.

Упражнение 4.4. Добавьте команды, с помощью которых можно было бы печатать верхний элемент стека (с сохранением его на стеке), дублировать его на стеке, менять местами два верхних элемента стека. Введите команду очистки стека.

Упражнение 4.5. Предусмотрите возможность использования в программе библиотечных функций sin, exp и pow. См. библиотеку <math.h> в приложении B (разд. B.4).

Упражнение 4.6. Введите команды для работы с переменными (легко обеспечить до 26 переменных, каждая из которых имеет имя, представленное одной буквой латинского алфавита). Добавьте переменную, предназначенную для хранения самого последнего из напечатанных значений.

Упражнение 4.7. Напишите программу `ungets(s)`, возвращающую стринг `s` во входной поток. Должна ли `ungets` «знать» что-либо о переменных `buf` и `bufp` или ей достаточно пользоваться только функцией `ungetch`?

Упражнение 4.8. Предположим, что число литер, возвращаемых назад, не превышает 1. Модифицируйте с учётом этого факта функции `getch` и `ungetch`.

Упражнение 4.9. В наших функциях не предусмотрена возможность возврата `EOF`. Подумайте, что надо сделать, чтобы можно было возвращать `EOF`, и скорректируйте соответственно программу.

Упражнение 4.10. В основу программы калькулятора можно положить применение функции `getline`, которая читает целиком строку; при этом отпадает необходимость в `getch` и `ungetch`. Напишите программу, реализующую этот подход.

4.4. Правила областей действия

Функции и внешние переменные, из которых состоит Си-программа, каждый раз компилировать все вместе нет никакой необходимости. Исходный текст можно хранить в нескольких файлах. Ранее скомпилированные программы можно загружать из библиотек. В связи с этим возникают следующие вопросы:

- Как писать декларации, чтобы на протяжении компиляции используемые переменные были должным образом описаны?
- В каком порядке располагать декларации, чтобы во время загрузки все части программы оказались связаны нужным образом?
- Как организовать декларации, чтобы для нескольких одноимённых деклараций была только одна копия?
- Как инициализировать внешние переменные?

Начнём с того, что разобьём программу-калькулятор на несколько файлов. Конечно, эта программа слишком мала, чтобы её стоило разбивать на файлы, однако разбиение нашей программы позволит продемонстрировать проблемы, возникающие в больших программах.

Областью действия имени считается часть программы, в которой это имя можно использовать. Для автоматических переменных, описанных в начале функции, областью действия является функция, в которой они описаны. Локальные переменные разных функций, имеющие, однако, одинаковые

имена, никак не связаны друг с другом. То же утверждение справедливо и в отношении параметров функции, которые фактически являются локальными переменными.

Область действия внешней переменной или функции простирается от точки программы, где она декларирована, до конца файла, подлежащего компиляции. Например, если main, sp, val, push и pop определены в одном файле в указанном порядке, т.е.

```
main() { ... }
int sp = 0;
double val[MAXVAL];
void push(double f) { ... }
double pop(void) { ... }
```

то к переменным sp и val можно адресоваться из push и pop просто по их именам; никаких дополнительных деклараций для них не требуется. Заметим, что в main эти имена не видимы так же, как и сами push и pop.

С другой стороны, если на внешнюю переменную нужно сослаться до того, как она определена, или если она определена в другом файле, то её декларация должна быть помечена словом extern.

Важно отличать *декларацию (описание)* внешней переменной от её *определения*. Декларация объявляет свойства переменной (прежде всего её тип), а определение, кроме того, приводит к выделению для неё памяти. Если строки

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

расположены вне всех функций, то они *определяют* внешние переменные sp и val, т.е. отводят для них память, и, кроме того, служат декларациями для остальной части исходного файла. А вот строки

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

декларируют для остальной части файла, что sp – переменная типа int и что val – массив типа double (размер которого определён где-то в другом месте); при этом ни переменная, ни массив не создаются, и память им не отводится.

На всю совокупность файлов, из которых состоит исходная программа, для каждой внешней переменной должно быть одно-единственное *определение*; другие файлы, чтобы получить доступ к внешней переменной, должны иметь в себе декларацию extern. (Впрочем, декларацию extern можно поместить и в файл, в котором содержится определение.) В определениях массивов необходимо указывать их размеры, что же касается деклараций extern, то здесь они не обязательны.

Инициализировать внешнюю переменную можно только в определении.

Хотя маловероятно, что нашу программу кто-нибудь станет организовывать таким способом, но мы определим push и pop в одном файле, а val и sp – в другом, где их и инициализируем. При этом для установления связей понадобятся такие определения и декларации:

В файле 1:

```
extern int sp;
extern double val[];
void push(double f) { ... }
double pop(void) { ... }
```

В файле 2:

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

Поскольку декларации extern находятся в начале *файла 1* и вне определений функций, их действие распространяется на все функции, причём одного набора деклараций достаточно для всего *файла 1*. Та же организация extern-деклараций необходима и в случае, когда программа состоит из одного файла, но определения sp и val расположены после их использования.

4.5. Головные файлы

Теперь представим себе, что компоненты программы-калькулятора существенно бóльших размеров, и зададимся вопросом, как в этом случае распределить их по нескольким файлам. Программу main поместим в файл, который мы назовём main.c; push, pop и их переменные расположим во втором файле, stack.c; а getop – в третьем, getop.c. Наконец, getch и ungetch разместим в четвёртом файле getch.c; мы отделили их от остальных функций, поскольку в реальной программе они будут получены из заранее скомпилированной библиотеки.

Существует ещё один момент, о котором следует предупредить читателя, – определения и декларации совместно используются несколькими файлами. Насколько это возможно, мы бы хотели централизовать описания так, чтобы для них существовала только одна копия. Тогда программу в процессе её развития будет легче и исправлять, и поддерживать в правильном состоянии. Для этого общую информацию расположим в *головном файле* calc.h, который будем по мере необходимости включать в другие файлы. (Строка #include описывается в разд. 4.11.) В результате получим программу, файловая структура которой показана на следующей странице.

Неизбежен компромисс между стремлением, чтобы каждый файл владел только той информацией, которая ему необходима для работы, и тем, что на практике иметь дело с большим количеством головных файлов довольно

трудно. Для программ, не превышающих некоторого среднего размера, вероятно, лучше всего иметь один головной файл, в котором собраны вместе все объекты, каждый из которых используется в двух различных файлах; так мы здесь и поступили. Для программ бóльших размеров потребуется более сложная организация с бóльшим числом головных файлов.

4.6. Статические переменные

Переменные sp и val в файле stack.c, а также buf и bufp в getch.c находятся в личном пользовании функций этих файлов, и нет смысла открывать к ним доступ кому-либо ещё. Указание static, применённое к внешней переменной или функции, ограничивает область действия соответствующего объекта концом файла. Это способ скрыть имена. Так, переменные buf и bufp должны быть внешними, поскольку их совместно используют функции getch и ungetch, но их следует сделать невидимыми для «пользователей» функций getch и ungetch.

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) {
      ...
}
```

Статическая память специфицируется словом static, которое помещается перед обычным описанием. Если рассматриваемые нами две функ-

ции и две переменные компилируются в одном файле, как в показанном ниже примере:

```
static char buf[BUFSIZE];    /* буфер для ungetch */
static int  bufp = 0;        /* след. своб. позиция в buf */
int getch(void) { ... }
void ungetch(int c) { ... }
```

то никакая другая программа не будет иметь доступ ни к buf, ни bufp, и этими именами можно свободно пользоваться в других файлах для совсем иных целей. Точно так же, помещая указание static перед декларациями переменных sp и val, с которыми работают только push и pop, мы можем скрыть их от остальных функций.

Указание static чаще всего используется для переменных, но с равным успехом его можно применять и к функциям. Обычно имена функций глобальны и видимы из любого места программы. Если же функция помечена словом static, то её имя становится невидимым вне файла, в котором она определена.

Декларацию static можно использовать и для внутренних переменных. Как и автоматические переменные, внутренние статические переменные локальны в функциях, но в отличие от автоматических они не возникают только на период активации функции, а существуют постоянно. Это значит, что для внутренних статических переменных отводится постоянная память (так же, как и для команд самой функции).

Упражнение 4.11. Модифицируйте функцию getop так, чтобы отпала необходимость в функции ungetch. Совет: используйте внутреннюю статическую переменную.

4.7. Регистровые переменные

Спецификация register в декларации сообщает компилятору, что данная переменная будет использоваться интенсивно. Идея состоит в том, чтобы переменные со спецификацией register разместить на регистрах машины, благодаря чему программа, возможно, станет более короткой и быстрой. Однако компилятор имеет право проигнорировать это указание.

Вот два примера деклараций с register:

```
register int x;
register char c;
```

Спецификация register может применяться только к автоматическим переменным и к формальным параметрам функции. Для последних спецификации register задаются в списке параметров, как, например, в

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

На практике существуют ограничения на register-переменные, связанные с возможностями аппаратуры. Только небольшое число переменных каждой функции, и причём только определённых типов, могут располагаться в регистрах. Избыточные указания register ни на что не влияют, так как эти указания игнорируются в отношении переменных, которым не хватило регистров, или которым они по каким-то причинам не были выделены. Кроме того, применительно к регистровой переменной независимо от того, выделен на самом деле регистр для неё или нет, не определено понятие адреса (см. гл. 5). Конкретные ограничения на количество и типы регистровых переменных зависят от машины.

4.8. Блочная структура

Поскольку функции в Си нельзя определять внутри других функций, он не является языком, допускающим блочную структуру программы в том смысле, как они разрешены в Паскале и подобных ему языках. Но переменные внутри функций можно определять в блочно-структурной манере. Декларации переменных (вместе с инициализацией) разрешено помещать не только в начале функции, но и после *любой* левой фигурной скобки, открывающей составную инструкцию. Переменная, описанная таким способом, «затеняет» переменные с тем же именем, расположенные в объемлющих блоках, и существует вплоть до соответствующей правой фигурной скобки. Например, в

```
if (n > 0) {
    int i;   /* описание новой переменной i */
    for (i = 0; i < n; i++)
        ...
}
```

областью действия переменной i является ветвь if, выполняемая при n > 0; и эта переменная никакого отношения к любым i, расположенным вне данного блока, не имеет. Автоматические переменные, декларируемые и инициализируемые в блоке, инициализируются каждый раз при входе в блок. Переменные static инициализируются только один раз при первом входе в блок.

Автоматические переменные и формальные параметры также «затеняют» внешние переменные и функции с теми же именами. Например, в

```
int x;
int y;
f(double x)
{
    double y;
    ...
}
```

ссылки на x внутри функции f рассматриваются как параметр типа double, в то время как вне f они отсылают нас к внешней переменной типа int. То же самое можно сказать и о переменной y.

Конечно, это вопрос стиля программирования, использовать ли одни и те же имена для разных переменных. Однако потенциальная опасность получить ошибку при «перекрытии» имён столь велика, что лучше этого избегать.

4.9. Инициализация

Мы уже много раз упоминали об инициализации, но всегда по случаю, в ходе обсуждения других вопросов. В этом разделе мы суммируем все правила, определяющие инициализацию памяти различных классов.

При отсутствии явной инициализации для внешних и статических переменных гарантируется их обнуление; автоматические и регистровые переменные имеют неопределённые начальные значения («мусор»).

Скалярные переменные можно инициализировать в их определениях, помещая после имени знак = и соответствующее выражение.

```
int x = 1;
char squote = '\'';
long day = 1000L * 60L * 60L * 24L;/* день в миллисекундах */
```

Для внешних и статических переменных инициализирующие выражения должны быть константными, при этом инициализация осуществляется только один раз до начала выполнения программы. Инициализация автоматических и регистровых переменных выполняется каждый раз при входе в функцию или блок. Для таких переменных инициализирующее выражение – не обязательно константное. Это может быть любое выражение, использующее ранее определённые значения, включая даже и вызовы функций. Например, в программе бинарного поиска, описанной в разд. 3.3, инициализации можно записать в виде

```
int binsearch(int x, int v[], int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

а не как

```
int low, high, mid;
low = 0;
high = n - 1;
```

В сущности, инициализация автоматической переменной – это более короткая запись инструкции присваивания. Какая запись предпочтительнее – в основном дело вкуса. До сих пор мы пользовались главным образом явными присваиваниями, поскольку инициализация в декларациях менее заметна и дальше отстоит от места использования переменной.

Массив можно инициализировать в его определении с помощью заключённого в фигурные скобки списка инициализаторов, разделённых запятыми. Например, чтобы инициализировать массив days, элементы которого суть количества дней в каждом месяце, можно написать

```
int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Если размер массива не указан, то длину массива компилятор вычисляет по числу заданных инициализаторов; в нашем случае их количество равно 12.

Если количество инициализаторов меньше числа, указанного в определении длины массива, то для внешних, статических и автоматических переменных оставшиеся элементы будут нулевыми. Задание слишком большого числа инициализаторов считается ошибкой. В языке нет возможности ни задавать кратность для инициализатора, ни инициализировать средние элементы массива без задания всех предшествующих значений.

Инициализация литерных массивов – особый случай: вместо конструкции с фигурными скобками и запятыми можно использовать стринг литер. Например, возможна такая запись:

```
int pattern[] = "ould";
```

что представляет собой более короткий эквивалент записи

```
int pattern[] = {'o', 'u', 'l', 'd', '\0'};
```

В данном случае размер массива равен пяти (четыре обычные литеры и завершающая литера '\0').

4.10. Рекурсия

В Си допускается рекурсивное обращение к функциям, т.е. функция может обращаться сама к себе, прямо или косвенно. Рассмотрим печать числа в виде стринга литер. Как мы упоминали ранее, цифры генерируются в обратном порядке – младшие цифры получаются раньше старших, а печататься они должны в правильной последовательности.

Проблему можно решить двумя способами. Первый – запомнить цифры в некотором массиве в том порядке, как они получались, а затем напечатать их в обратном порядке; так это и было сделано в itoa, приведённой в разд. 3.6. Второй способ – воспользоваться рекурсией, при которой printd сначала вызывает себя, чтобы напечатать все старшие цифры, и затем печатает последнюю младшую цифру. Эта программа, как и предыдущий её вариант, работает неправильно в случае самого большого по модулю отрицательного числа.

```
#include <stdio.h>
/* printd: печатает n как целое десятичное число */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

Когда функция рекурсивно обращаются сама к себе, каждое следующее обращение сопровождается получением ею нового полного комплекта автоматических переменных, независимых от предыдущих комплектов. Так, в обращении printd(123) при первом вызове аргумент n = 123, при втором – printd получает аргумент 12, при третьем вызове – значение 1. Функция printd на третьем уровне вызова печатает 1 и возвращается на второй уровень, после чего она печатает цифру 2 и возвращается на первый уровень. Здесь она печатает 3 и заканчивает работу.

Следующий хороший пример рекурсии – это быстрая сортировка, предложенная Ч.А.Р. Хоаром в 1962 г. Для заданного массива выбирается один элемент, который разбивает остальные элементы на два подмножества – те, что меньше, и те, что не меньше него. Та же процедура рекурсивно применяется и к двум полученным подмножествам. Если в подмножестве менее двух элементов, то сортировать нечего, и рекурсия завершается.

Наша версия быстрой сортировки, разумеется, не самая быстрая среди всех возможных, но зато одна из самых простых. В качестве делящего эле-

мента мы используем серединный элемент.

```
/* qsort: сортирует v[left]...v[right] по возрастанию */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);
    if (left >= right) /* ничего не делается, если       */
        return;        /* в массиве менее двух элементов */
    swap(v, left, (left + right)/2); /* делящий элемент    */
    last = left;                     /* переносится в v[0] */
    for (i = left+1; i <= right; i++) /* деление на части */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);   /* перезапоминаем делящий эл-т */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

В нашей программе операция перестановки оформлена в виде отдельной функции (`swap`), поскольку трижды встречается в `qsort`.

```
/* swap: переставить v[i] и v[j] между собой */
void swap(int v[], int i, int j)
{
    int temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Стандартная библиотека имеет функцию `qsort`, позволяющую сортировать объекты любого типа.

Рекурсивная программа не обеспечивает ни экономного расходования памяти, поскольку требуется где-то поддерживать стек значений, подлежащих обработке, ни быстродействия; но по сравнению со своим нерекурсивным эквивалентом она часто короче, а часто намного легче для написания и понимания. Такого рода программы особенно удобны для обработки рекурсивно определяемых структур данных типа деревьев; с хорошим примером на эту тему вы познакомитесь в разд. 6.5.

Упражнение 4.12. Примените идеи, которые мы использовали в `printd` для написания рекурсивной версии функции `itoa`; иначе говоря, преобразуйте целое число в цепочку цифр с помощью рекурсивной программы.

Упражнение 4.13. Напишите рекурсивную версию функции `reverse(s)`, оборачивающую стринг на его же месте.

4.11. Си-препроцессор

Некоторые возможности языка Си обеспечиваются препроцессором, который работает на первом шаге компиляции. Наиболее часто используются две возможности: #include, включающая файл во время компиляции, и #define, заменяющая одни текстовые цепочки на другие. В этом разделе обсуждаются условная компиляция и макроподстановка с аргументами.

4.11.1. Включение файла

Средство #include позволяет, в частности, легко манипулировать наборами #define и деклараций. Любая строка вида

```
#include "имя-файла"
```

или

```
#include <имя-файла>
```

заменяется содержимым файла с именем *имя-файла*. Если *имя-файла* заключено в двойные кавычки, то, как правило, файл ищется среди исходных файлов программы; если такового не оказалось или *имя-файла* заключено в угловые скобки < и >, то поиск осуществляется в системных библиотеках. Включаемый файл сам может содержать в себе строки #include.

Часто исходные файлы начинаются с нескольких строк #include, ссылающихся на общие инструкции #define и extern-декларации или прототипы нужных библиотечных функций из головных файлов типа <stdio.h>. (Строго говоря, эти включения не обязательно являются файлами; технические детали того, как осуществляется доступ к головной информации, зависят от конкретной реализации.)

Средство #include – хороший способ собрать вместе декларации большой программы. Он гарантирует, что все исходные файлы будут пользоваться одними и теми же определениями и описаниями переменных, благодаря чему предотвращаются особенно неприятные ошибки. Естественно, при внесении изменений во включаемый файл все зависимые от него файлы должны перекомпилироваться.

4.11.2. Макроподстановка

Определение макроподстановки имеет вид:

```
#define   имя   замещающий-текст
```

Макроподстановка используется для простейшей замены: во всех местах, где встречается лексема *имя*, вместо неё будет помещён *замещающий-текст*. Имена в #define задаются по тем же правилам, что и имена обычных переменных. Замещающий текст может быть произвольным. Обычно замещающий текст завершает строку, в которой расположено слово #define, но в

длинных определениях его можно продолжить на следующих строках, поставив в конце каждой продолжаемой строки обратную наклонную черту \. Область действия имени в #define простирается от данного определения до конца файла. В определении макроподстановки могут фигурировать более ранние #define-определения. Подстановка осуществляется только для тех имён, которые расположены вне текстов, заключённых в кавычки. Например, если YES определено с помощью #define, то никакой подстановки в printf("YES") или в YESMAN выполнено не будет.

Любое имя можно определить с произвольным замещающим текстом. Например,

```
#define forever for(;;)    /* бесконечный цикл */
```

определяет новое слово forever для бесконечного цикла.

Макроподстановку можно определить с аргументами, вследствие чего замещающий текст будет варьироваться в зависимости от задаваемых параметров. Например, определим max следующим образом:

```
#define max(A, B)   ((A) > (B) ? (A) : (B))
```

Хотя обращения к max выглядят как обычные обращения к функции, они будут вызывать только текстовую замену. Каждый формальный параметр (в данном случае A и B) будет заменяться соответствующим ему аргументом. Так, строка

```
x = max(p+q, r+s);
```

будет заменена на строку

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Поскольку аргументы допускают любой вид замены, указанное определение max подходит для данных любого типа, так что не нужно писать разные max для данных разных типов, как это было бы в случае задания с помощью функций.

Если вы внимательно проанализируете текстовое расширение max, то обнаружите некоторые подводные камни. Выражения вычисляются дважды и, если они вызывают побочный эффект (из-за инкрементных операций или функций ввода-вывода), это может привести к нежелательным последствиям. Например,

```
max(i++, j++)    /* НЕВЕРНО */
```

вызовет увеличение i и j дважды. Кроме того, следует позаботиться о скобках, чтобы обеспечить нужный порядок вычислений. Посмотрите, что случится, если при определении

```
#define   square(x)   x * x    /* НЕВЕРНО */
```

воспользоваться записью типа square(z+1).

Тем не менее макросредства имеют свои преимущества. Практическим примером их использования являются часто применяемые getchar и

putchar из <stdio.h>, которые реализованы с помощью макросов, чтобы избежать накладных расходов от вызова функции на каждую обрабатываемую литеру. Функции в <ctype.h> обычно также реализуются с помощью макросов.

Имена можно «скрыть» от препроцессора с помощью #undef.

```
#undef getchar
int getchar(void) { ... }
```

Как правило, это делается, чтобы макроопределение «перекрыть» настоящей функцией с тем же именем.

Имена формальных параметров не заменяются, если они находятся в стрингах, заключённых в кавычки. А вот комбинация в замещающем тексте: формальный параметр с предшествующим знаком # – в макрорасширении будет заменена на аргумент, заключённый в кавычки. Такую конструкцию можно использовать для конкатенации с другими литерными стрингами, как, например, в следующем макроопределении, предназначенном для отладочной выдачи:

```
#define   dprint(expr)   printf(#expr " = %g\n", expr)
```

Обращение к макроопределению

```
dprint(x/y)
```

даст расширение

```
printf("x/y" " = %g\n", x/y);
```

а в результате конкатенации двух соседних стрингов получим

```
printf("x/y = %g\n", x/y);
```

Внутри фактического аргумента каждый знак " заменяется на \", а каждая \ на \\ так, чтобы результат подстановки приводил к правильной литерной константе.

Оператор ## препроцессором понимается как операция конкатенации; он позволяет в макрорасширениях конкатенировать аргументы. Если в замещающем тексте параметр соседствует с ##, то он заменяется на соответствующий ему аргумент, а оператор ## и окружающие его пробельные литеры выбрасываются. Например, в макроопределении paste конкатенируются два аргумента

```
#define   paste(front, back)   front ## back
```

так что paste(name, 1) сгенерирует имя name1.

Правила вложенных использований оператора ## не определены; другие подробности, относящиеся к ##, можно найти в приложении А.

Упражнение 4.14. Определите swap(t,x,y) в виде макроса, который осуществляет обмен значениями указанного типа t между аргументами x и y. (Примените блочную структуру.)

4.11.3. Условная компиляция

Самим ходом препроцессирования можно управлять с помощью условных инструкций. Они представляют собой средство для выборочного включения того или иного текста программы в зависимости от значения условия, вычисляемого во время компиляции.

Вычисляется константное целое выражение, заданное в строке `#if`. Это выражение не должно содержать ни одного оператора `sizeof` или приведения к типу и ни одной `enum`-константы. Если оно имеет ненулевое значение, то будут включены все последующие строки вплоть до `#endif`, или `#elif`, или `#else`. (Инструкция препроцессора `#elif` похожа на `else if`.) Выражение `defined(`*имя*`)` в `#if` есть 1, если *имя* было определено, и 0 в противном случае.

Например, чтобы застраховаться от повторного включения головного файла `hdr.h`, его можно оформить следующим образом:

```
#if !defined(HDR)
#define HDR

/* здесь содержимое hdr.h */

#endif
```

При первом включении файла `hdr.h` будет определено имя `HDR`, а при последующих включениях препроцессор обнаружит, что имя `HDR` уже определено и перепрыгнет сразу на `#endif`. Этот приём может оказаться полезным, когда нужно избежать многократного включения одного и того же файла. Если им пользоваться систематически, то в результате каждый головной файл будет сам включать головные файлы, от которых он зависит, освободив от этого занятия пользователя.

Вот пример цепочки проверок имени `SYSTEM`, позволяющей выбрать нужный файл для включения:

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BCD
    #define HDR "bcd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

Инструкции `#ifdef` и `#ifndef` специально предназначены для проверки того, определено или не определено заданное в них имя. И следовательно, пер-

вый пример, приведённый выше для иллюстрации `#if`, можно записать и в таком виде:

```
#ifndef HDR
#define HDR

/* здесь содержимое hdr.h */

#endif
```

Глава 5

Указатели и массивы

Указатель – это переменная, содержащая адрес переменной. Указатели широко применяются в Си – отчасти потому, что в некоторых случаях без них просто не обойтись, а отчасти потому, что программы с ними обычно короче и эффективнее. Указатели и массивы тесно связаны друг с другом; в этой главе мы рассмотрим эту зависимость и покажем, как ею пользоваться.

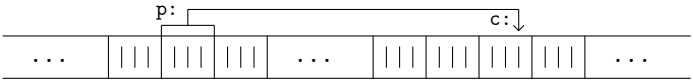
Наряду с `goto` указатели когда-то были объявлены лучшим средством для написания малопонятных программ. Так оно и есть, если ими пользоваться бездумно. Ведь очень легко получить указатель, ссылающийся на что-нибудь совсем нежелательное. При соблюдении же определённой дисциплины с помощью указателей можно достичь ясности и простоты. Мы попытаемся убедить вас в этом.

Изменения, внесённые стандартом ANSI, связаны в основном с формулированием точных правил работы с указателями. Он узаконил накопленный положительный опыт программистов и удачные нововведения разработчиков компиляторов. Кроме того, взамен `char *` в качестве типа обобщённого указателя предлагается тип `void *` (указатель на `void`).

5.1. Указатели и адреса

Начнём с того, что рассмотрим упрощённую схему организации памяти. Память типичной машины представляет собой массив последовательно пронумерованных и проадресованных ячеек, с которыми можно работать по отдельности или связными кусками. Применительно к любой машине верны следующие утверждения: один байт может хранить значение типа `char`, двухбайтовые ячейки могут рассматриваться как целое типа `short`, а четырёхбайтовые – как целые типа `long`. Указатель – это группа ячеек (как правило, две или четыре), в которых может храниться адрес. Так, если с име-

ет тип char, а p – указатель, ссылающийся на c, то ситуация выглядит следующим образом:



Унарный оператор & выдаёт адрес объекта, так что инструкция

```
p = &c;
```

присваивает адрес ячейки c переменной p (говорят, что p указывает на c или, что то же, p ссылается на c). Оператор & применяется только к объектам, расположенным в памяти: к переменным и элементам массивов. Его операндом не может быть ни выражение, ни константа, ни регистровая переменная.

Унарный оператор * есть оператор *раскрытия ссылки*. Применённый к указателю он выдаёт объект, на который данный указатель ссылается. Предположим, что x и y – целые, а ip – указатель на int. Следующие несколько строк придуманы специально для того, чтобы показать, каким образом декларируются указатели и используются операторы & и *.

```
int x = 1, y = 2, z[10];
int *ip;            /* ip - указатель на int */

ip = &x;            /* теперь ip  указывает на x */
y = *ip;            /* y теперь равен 1 */
*ip = 0;            /* x теперь равен 0 */
ip = &z[0];         /* ip теперь указывает на z[0] */
```

Декларации x, y и z нам уже знакомы. Декларацию указателя ip

```
int *ip;
```

мы стремились сделать мнемоничной – она гласит: «выражение *ip есть нечто типа int». Синтаксис декларации переменной «подстраивается» под синтаксис выражений, в которых эта переменная может встретиться. Указанный принцип применим и в отношении описаний функций. Например, запись

```
double *dp, atof(char *);
```

означает, что выражение *dp и atof(s) имеют тип double, а аргумент функции atof есть указатель на char.

Вы, наверное, заметили, что указателю разрешено ссылаться только на объекты заданного типа. (Существует одно исключение: «указатель на void» может ссылаться на объекты любого типа, но к такому указателю нельзя применять оператор раскрытия ссылки. Мы вернёмся к этому в разд. 5.11.)

Если ip ссылается на x целого типа, то *ip можно использовать в любом месте, где допустимо применение x; например,

```
*ip = *ip + 10;
```

увеличивает *ip на 10.

Унарные операторы * и & имеют более высокий приоритет, чем арифметические операторы, так что присваивание

```
y = *ip + 1
```

берет то, на что указывает ip, и добавляет к нему 1, а результат присваивает переменной y. Аналогично

```
*ip += 1
```

увеличивает на единицу то, на что ссылается ip; те же действия выполняют

```
++*ip
```

и

```
(*ip)++
```

В последней записи скобки необходимы, поскольку, если их не будет, увеличится значение самого указателя, а не то, на что он ссылается. Это обусловлено тем, что унарные операторы * и ++ имеют одинаковые приоритет и порядок выполнения – справа налево.

И наконец, так как указатели сами являются переменными, в тексте они могут встречаться и без оператора раскрытия ссылки. Например, если iq есть другой указатель на int, то

```
iq = ip
```

копирует содержимое ip в iq, чтобы ip и iq ссылались на один и тот же объект.

5.2. Указатели и аргументы функций

Поскольку функции в Си в качестве своих аргументов получают значения параметров, прямой возможности, находясь в вызванной функции, изменить переменную вызывающей функции нет. В программе сортировки нам понадобилась функция swap, переставляющая местами два неупорядоченных элемента. Однако недостаточно написать

```
swap(a, b);
```

где функция swap определена следующим образом:

```
void swap(int x, int y)          /* НЕВЕРНО */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Поскольку swap получает лишь *копии* значений переменных a и b, она не может повлиять на переменные a и b той программы, которая к ней обратилась.

Чтобы получить желаемый эффект, надо вызывающей программе передать *указатели* на те значения, которые должны быть изменены:

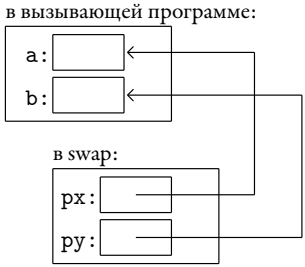
```
swap(&a, &b);
```

Так как оператор & получает адрес переменной, &a есть указатель на a. В самой функции swap параметры должны быть описаны как указатели, при этом доступ к значениям параметров будет осуществляться через них косвенно.

```
void swap(int *px, int *py)  /* перестановка *px и *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

Графически это можно изобразить следующим образом:



Аргументы-указатели позволяют функции осуществлять доступ к объектам вызвавшей её программы и дают ей возможность изменить эти объекты. Рассмотрим, например, функцию getint, которая осуществляет ввод в свободном формате одного целого числа и его перевод из текстового представления в значение типа int. Функция getint должна возвращать значение полученного числа или сигнализировать значением EOF о конце файла, если входной поток исчерпан. Эти значения должны возвращаться по разным каналам, так как нельзя рассчитывать на то, что полученное в результате перевода число никогда не совпадёт с EOF.

Одно из решений состоит в том, чтобы getint выдавала характеристику состояния файла (исчерпан или не исчерпан) в качестве результата, а значение самого числа помещала согласно указателю, переданному ей в виде аргумента. Похожая схема действует в программе scanf, которую мы рассмотрим в разд. 7.4.

Показанный ниже цикл заполняет некоторый массив целыми числами, полученными с помощью getint.

```
int n, array[SIZE], getint(int *);
for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
    ;
```

Результат каждого очередного обращения к getint посылается в array[n], и n увеличивается на единицу. Заметим, что существенным здесь является то, что функции getint передаётся адрес элемента array[n]. Если этого не сделать, у getint не будет способа вернуть в вызывающую программу переведённое целое число.

В предлагаемом нами варианте функция getint выдаёт EOF по концу файла; нуль, если следующие вводимые литеры не представляют собою числа; и положительное значение, если введённые литеры есть правильное число.

```
#include <ctype.h>
int getch(void);
void ungetch(int);
/* getint: читает следующее целое из ввода в *pn */
int getint(int *pn)
{
    int c, sign;
    while (isspace(c = getch()))
        ;           /* пропуск пробельных литер */
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c);   /* не число */
        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;
}
```

Везде в getint под комбинацией *pn подразумевается обычная переменная типа int. Функция ungetch вместе с getch (разд. 4.3) включена в программу, чтобы обеспечить возможность отослать назад лишнюю прочитанную литеру.

Упражнение 5.1. Функция `getint` написана так, что знаки – или +, за которыми не следует цифра, она понимает как «правильное» представление нуля. Скорректируйте программу таким образом, чтобы она в подобных случаях возвращала прочитанный знак назад во ввод.

Упражнение 5.2. Напишите функцию `getfloat` – аналог `getint` для чисел с плавающей точкой. Какой тип будет иметь результирующее значение, выдаваемое функцией `getfloat`?

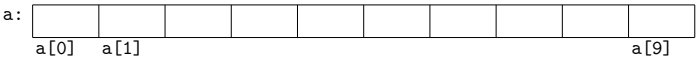
5.3. Указатели и массивы

В Си существует связь между указателями и массивами, и связь эта настолько тесная, что эти средства лучше рассматривать вместе. Любой доступ к элементу массива, осуществляемый операцией индексирования, может быть выполнен при помощи указателя. Вариант с указателями в общем случае работает быстрее, но разобраться в нём, особенно непосвящённому, довольно трудно.

Декларация

```
int a[10];
```

определяет массив `a` размера 10, т.е. блок из 10 последовательных объектов с именами `a[0]`, `a[1]`, ..., `a[9]`.



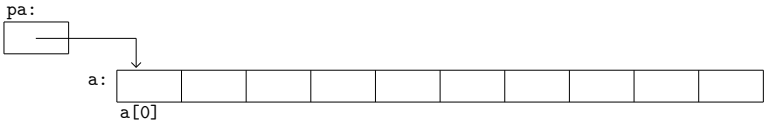
Запись `a[i]` отсылает нас к `i`-му элементу массива. Если `pa` есть указатель на `int`, т.е. определён как

```
int *pa;
```

то в результате присваивания

```
pa = &a[0];
```

`pa` будет указывать на нулевой элемент `a`; иначе говоря, `pa` будет содержать адрес элемента `a[0]`.



Теперь присваивание

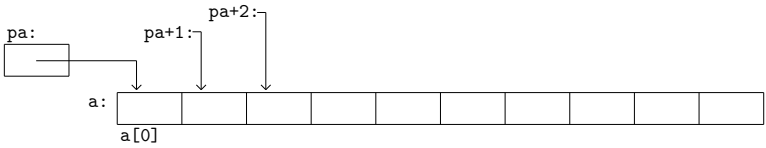
```
x = *pa;
```

будет копировать содержимое a[0] в x.

Если pa указывает на некоторый элемент массива, то pa+1 по определению указывает на следующий элемент, pa+i – на i-й элемент после pa, а pa-i – на i-й элемент перед pa. Таким образом, если pa указывает на a[0], то

```
*(pa+1)
```

есть содержимое a[1], pa+i – адрес a[i], а *(pa+i) – содержимое a[i].



Сделанные замечания верны безотносительно к типу и размеру элементов массива a. Смысл слов «добавить 1 к указателю», как и смысл любой арифметики с указателями, в том, чтобы pa+1 указывал на следующий объект, а pa+i – на i-й после pa.

Между индексированием и арифметикой с указателями существует очень тесная связь. По определению значение переменной или выражения типа массив есть адрес нулевого элемента массива. После присваивания

```
pa = &a[0];
```

pa и a имеют одно и то же значение. Поскольку имя массива есть не что иное, как адрес его начального элемента, присваивание pa=&a[0]; можно также записать в следующем виде:

```
pa = a;
```

Ещё более удивительно (по крайней мере на первый взгляд) то, что a[i] можно записать как *(a+i). Встречая запись a[i], компилятор сразу преобразует её в *(a+i); указанные две формы записи эквивалентны. Из этого следует, что полученные в результате применения оператора & записи &a[i] и a+i также будут эквивалентными, т.е. и в том и в другом случае это адрес i-го элемента после a. С другой стороны, если pa – указатель, то в выражениях его можно использовать с индексом, т.е. запись pa[i] эквивалентна записи *(pa+i). Элемент массива одинаково разрешается изображать и в виде указателя со смещением и в виде имени массива с индексом.

Между именем массива и указателем, выступающим в роли имени массива, существует одно различие. Указатель – это переменная, поэтому можно написать pa = a или pa++. Но имя массива не является переменной, и записи типа a = pa или a++ не допускаются.

Если имя массива передаётся функции, то последняя получает в качестве аргумента адрес его начального элемента. Внутри вызываемой функции этот аргумент является локальной переменной, содержащей адрес. Мы можем воспользоваться отмеченным фактом и написать ещё одну версию функции strlen, вычисляющей длину стринга.

```
/* strlen: возвращает длину стринга */
int strlen(char *s)
{
    int n;
    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

Так как переменная s – указатель, к нему применима операция ++; s++ никакого влияния на стринг литер функции, которая обратилась к strlen, не оказывает. Просто увеличивается на 1 некоторая копия указателя, находящаяся в личном пользовании функции strlen. Это значит, что все вызовы типа:

```
strlen("Здравствуй, мир");    /* стринговая константа */
strlen(array);                 /* char array[100];  */
strlen(ptr);                   /* char *ptr;  */
```

законны.

Записи

```
char s[];
```

и

```
char *s;
```

в определении функции в качестве формальных параметров эквивалентны. Мы отдаём предпочтение последней, поскольку она более явно сообщает, что s есть указатель. Если функции в качестве аргумента передаётся имя массива, то она может рассматривать его так, как ей удобно – либо как имя массива, либо как указатель, и поступать с ним соответственно. Она может даже использовать оба вида записи, если это покажется ей уместным.

Функции можно передать часть массива, для этого аргумент должен указывать на начало подмассива. Например, если a – массив, то в записях

```
f(&a[2])
```

или

```
f(a+2)
```

функции f передаётся адрес подмассива, начинающегося с элемента a[2]. Внутри функции f описание параметров может выглядеть как

```
f(int arr[]) { ... }
```

или

```
f(int *arr) { ... }
```

Следовательно, для f тот факт, что параметр ссылается на часть массива, а не на весь массив, не имеет значения.

Если есть уверенность, что элементы массива существуют, то возможно индексирование и в «обратную» сторону по отношению к нулевому элементу; выражения p[-1], p[-2] и т.д. не противоречат синтаксису языка и ссылаются на элементы, стоящие непосредственно перед p[0]. Разумеется, нельзя «выходить» за границы массива и тем самым ссылаться на несуществующие «объекты».

5.4. Адресная арифметика

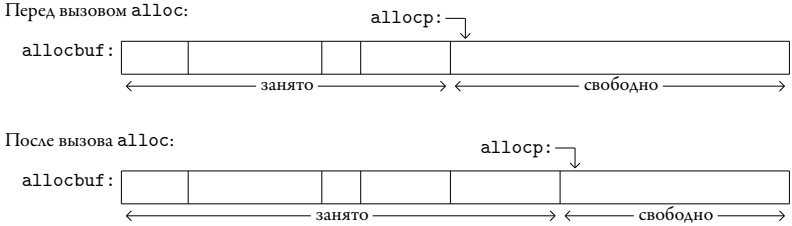
Если p есть указатель на некоторый элемент массива, то p++ продвигает p так, чтобы он указывал на следующий элемент, а p += i увеличивает его, чтобы он указывал на i-й элемент после того, на который он указывал ранее. Эти и подобные конструкции – самые простые примеры арифметики над указателями, называемой также адресной арифметикой.

Си последователен и единообразен в своём подходе к адресной арифметике. Это соединение в одном языке указателей, массивов и адресной арифметики – одна из сильных его сторон. Проиллюстрируем сказанное построением простого распределителя памяти, состоящего из двух программ. Первая, alloc(n), возвращает указатель p на n последовательно расположенных ячеек типа char; программой, обращающейся к alloc, эти ячейки могут быть использованы для запоминания литер. Вторая, afree(p), освобождает память для, возможно, повторной её утилизации. Простота алгоритма обусловлена предположением, что обращения к afree делаются в обратном порядке по отношению к соответствующим обращениям к alloc. Таким образом, память, с которой работают alloc и afree, является стеком (списком, в основе которого лежит принцип «последним вошёл, первым ушёл»). В стандартной библиотеке имеются функции malloc и free, которые делают то же самое, только без упомянутых ограничений; в разд. 8.7 мы покажем, как они выглядят.

Функцию alloc легче всего реализовать, если условиться, что она будет выдавать куски некоторого большого массива типа char, который мы назовём allocbuf. Этот массив отдадим в личное пользование функциям alloc и afree. Так как они имеют дело с указателями, а не с индексами массива, то другим программам знать его имя не нужно. Кроме того, этот массив можно определить в том же исходном файле, что и alloc и afree, объявив его с классификатором static, благодаря чему он станет невидимым вне этого файла. На практике такой массив может и вовсе не иметь имени, поскольку его можно запросить с помощью malloc у операционной системы и получить указатель на некоторый безымянный блок памяти.

Естественно, нам нужно знать, сколько элементов массива allocbuf

уже занято. Мы введём указатель allocp, который будет указывать на первый свободный элемент. Если запрашивается память для n литер, то alloc возвращает текущее значение allocp (т.е. адрес начала свободного блока) и затем увеличивает его на n, чтобы указатель allocp ссылался на следующую свободную область. Если же пространства нет, то alloc выдаёт нуль. Функция afree(p) просто устанавливает в allocp значение p, если оно не выходит за пределы массива allocbuf.




```
#define ALLOCSIZE 10000 /* размер доступного пространства */
static char allocbuf[ALLOCSIZE];   /* память для alloc */
static char *allocp = allocbuf;    /* ук-ль на своб. место */

char *alloc(int n)  /* возвращает указатель на n литер */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) {
        allocp += n;        /* пространство есть */
        return allocp - n;  /* старое p */
    } else                  /* пространства нет */
        return 0;
}

void afree(char *p)  /* освобождается память по ук-лю p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

Как и любую другую переменную, указатель можно инициализировать, но только такими осмысленными для него значениями, как нуль или выражение, приводящее к некоторому адресу ранее определённых данных соответствующего типа. Декларация

```
static char *allocp = allocbuf;
```

определяет allocp как указатель на char и инициализирует его адресом массива allocbuf, поскольку перед началом работы программы массив allocbuf пуст. Указанная декларация могла бы иметь и такой вид:

```
static char *allocp = &allocbuf[0];
```

поскольку имя массива и есть адрес его нулевого элемента.

Проверка

```
if (allocbuf + ALLOCSIZE - allocp >= n) {
```

контролирует, достаточно ли пространства, чтобы удовлетворить запрос на `n` литер. Если памяти достаточно, то новое значение для `allocp` должно указывать не далее чем на следующую позицию за последним элементом `allocbuf`. При выполнении этого требования `alloc` выдаёт указатель на начало выделенного блока литер (обратите внимание на описание типа самой функции). Если требование не выполняется, функция `alloc` должна выдать какой-то сигнал о том, что памяти не хватает. Си гарантирует, что нуль никогда не будет правильной ссылкой на данные, поэтому мы будем использовать его в качестве признака аварийного события, в нашем случае нехватки памяти.

Указатели и целые не являются взаимозаменяемыми объектами. Константа нуль – единственное исключение из этого правила: её можно присвоить указателю, и указатель можно сравнить с нулевой константой. Чтобы показать, что нуль – это специальное значение для указателя, вместо цифры нуль, как правило, записывают `NULL` – константу, определённую в файле `<stdio.h>`. С этого момента и мы будем ею пользоваться.

Проверки

```
if (allocbuf + ALLOCSIZE - allocp >= n) {
```

и

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

демонстрируют несколько важных свойств арифметики с указателями. Во-первых, при соблюдении некоторых правил указатели можно сравнивать. Если `p` и `q` указывают на элементы одного массива, то к ним можно применять операторы отношения `==`, `!=`, `<`, `>=` и т.д. Например, отношение вида

```
p < q
```

истинно, если `p` указывает на более ранний элемент массива, чем `q`. Любой указатель всегда можно сравнить на равенство и неравенство с нулём. А вот для указателей, ссылающихся на элементы разных массивов, результат арифметических операций или сравнений не определён. (Существует одно исключение: в арифметике с указателями можно использовать адрес несуществующего «следующего за массивом» элемента, т.е. адрес того «элемента», который станет последним, если в массив добавить ещё один элемент.)

Во-вторых, как вы уже, наверное, заметили, указатели и целые можно складывать и вычитать. Запись вида

```
p + n
```

означает адрес объекта, занимающего n-е место после объекта, на который указывает p. Это справедливо безотносительно к типу объекта, на который ссылается p; n автоматически домножается на коэффициент, соответствующий размеру объекта. Информация о размере неявно присутствует в описании p. Если, к примеру, int занимает четыре байта, то коэффициент умножения будет равен четырём.

Допускается также вычитание указателей. Например, если p и q ссылаются на элементы одного массива и p < q, то q - p + 1 есть число элементов от p до q включительно. Этим фактом можно воспользоваться при написании ещё одной версии strlen:

```
/* strlen: возвращает длину стринга s */
int strlen(char *s)
{
    char *p = s;
    while (*p != '\0')
        p++;
    return p - s;
}
```

В своём определении p инициализируется значением s, т.е. вначале p указывает на первую литеру стринга. На каждом шаге цикла while проверяется очередная литера; цикл продолжается до тех пор, пока не встретится '\0'. Каждое продвижение указателя p на следующую литеру выполняется инструкцией p++, и разность p-s даёт число пройденных литер, т.е. длину стринга. (Число литер в стринге может быть слишком большим, чтобы хранить его в переменной типа int. Тип ptrdiff_t, достаточный для хранения разности (со знаком) двух указателей, определён в головном файле <stddef.h>. Однако, если быть очень осторожными, нам следовало бы для возвращаемого результата использовать тип size_t, в этом случае наша программа соответствовала бы стандартной библиотечной версии. Тип size_t есть тип беззнакового целого, возвращаемого оператором sizeof.)

Арифметика с указателями учитывает тип: если она имеет дело со значениями float, занимающими больше памяти, чем char, и p – указатель на float, то p++ продвинет p на следующее значение float. Это значит, что другую версию alloc, которая имеет дело с элементами типа float, а не char, можно получить простой заменой в alloc и afree всех char на float. Все операции с указателями будут автоматически откорректированы в соответствии с размером объектов, на которые ссылаются указатели.

Допускаются следующие операции с указателями: присваивание значения указателя другому указателю того же типа, сложение и вычитание указателя и целого, вычитание и сравнение двух указателей, ссылающихся на элементы одного и того же массива, а также присваивание указателю нуля и сравнение указателя с нулём. Все другие операции с указателями не допускаются. Нельзя складывать два указателя, перемножать их, делить, сдвигать,

выделять разряды; указатель нельзя складывать со значением типа `float` или `double`; указателю одного типа нельзя даже присвоить указатель другого типа, не выполнив предварительно операции приведения (исключение составляют лишь указатели типа `void *`).

5.5. Литерные указатели и функции

Стринговая константа, написанная в виде

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

есть массив литер. Во внутреннем представлении этот массив заканчивается «пустой» литерой `'\0'`, по которой программа может найти конец стринга. Число занятых ячеек памяти на одну больше, чем количество литер, помещённых между двойными кавычками.

Чаще всего стринговые константы используются в качестве аргументов функций, как, например, в

```
printf("здравствуй, мир\n");
```

Когда такой литерный стринг появляется в программе, доступ к нему осуществляется через литерный указатель; `printf` получает указатель на начало массива литер. Точнее, доступ к стринговой константе осуществляется через указатель на её первый элемент.

Стринговые константы нужны не только в качестве аргументов функций. Если, например, переменную `pmessage` описать как

```
char  *pmessage
```

то присваивание

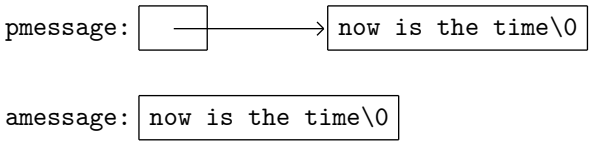
```
pmessage = "now is the time";
```

поместит в неё указатель на литерный массив, при этом сам стринг *не* копируется, копируется лишь указатель на него. Операции для работы со стрингом как с единым целым в Си не предусмотрены.

Существует важное различие между следующими определениями:

```
char amessage[] = "now is the time";  /* массив */
char *pmessage = "now is the time";   /* указатель */
```

`amessage` – это массив, имеющий такой объем, что в нём как раз помещается указанная последовательность литер и `'\0'`. Отдельные литеры внутри массива могут изменяться, но `amessage` всегда ссылается на одно и то же место памяти. В противоположность ему `pmessage` есть указатель, инициализированный ссылкой на стринговую константу. А значение указателя можно изменить, и тогда последний будет ссылаться на что-либо другое. Кроме того, результат будет неопределён, если вы попытаетесь изменить содержимое константы.



Дополнительные моменты, связанные с указателями и массивами, проиллюстрируем на несколько видоизменённых вариантах двух полезных программ, взятых нами из стандартной библиотеки. Первая из них, функция strcpy(s,t), копирует стринг t в стринг s. Хотелось бы написать прямо s=t, но такой оператор копирует указатель, а не литеры. Чтобы копировать литеры, нам нужно организовать цикл. Первый вариант strcpy, с использованием массива, имеет следующий вид:

```
/* strcpy: копирует t в s; вариант с индексируемым массивом */
void strcpy(char *s, char *t)
{
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

Для сравнения приведём версию strcpy с указателями:

```
/* strcpy: копирует t в s; версия 1 (с указателями) */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Поскольку передаются лишь копии значений аргументов, strcpy может свободно пользоваться параметрами s и t как своими локальными переменными. Они должным образом инициализированы указателями, которые продвигаются каждый раз на следующую литеру в каждом из массивов до тех пор, пока в копируемом стринге t не встретится '\0'.

На практике strcpy так не пишут. Опытный программист предпочтёт более короткую запись:

```
/* strcpy: копирует t в s; версия 2 (с указателями) */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

Продвижение s и t здесь осуществляется в управляющей части цикла. Значением *t++ является литера, на которую указывает переменная t перед тем, как её значение будет продвинуто; постфиксный оператор ++ не изменяет указатель t, пока не будет взята литера, на которую он указывает. То же в отношении s, сначала литера запомнится в позиции, на которую указывает старое значение s, и лишь после этого значение переменной s увеличится. Пересылаемая литера является одновременно и значением, которое сравнивается с '\0'. В итоге копируются все литеры, включая и заключительную литеру '\0'.

Заметив, что сравнение с '\0' здесь лишнее (поскольку в Си ненулевое значение выражения в условии трактуется и как его истинность), мы можем сделать ещё одно и последнее сокращение текста программы:

```
/* strcpy: копирует t в s; версия 3 (с указателями) */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Хотя на первый взгляд то, что мы получили, выглядит как криптограмма, всё же такая запись значительно удобнее, и следует освоить её, поскольку в Си-программах вы будете с ней часто встречаться.

Что касается функции strcpy из стандартной библиотеки <string.h>, то она возвращает в качестве своего результата ещё и ссылку на новую копию стринга.

Вторая программа, которую мы здесь рассмотрим, это strcmp(s,t). Она сравнивает литеры стрингов s и t и возвращает отрицательное, нулевое или положительное значение, если стринг s соответственно лексикографически меньше, равен или больше, чем стринг t. Результат получается вычитанием первых несовпадающих литер из s и t.

```
/* strcmp: выдаёт <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];
}
```

Та же программа с использованием указателей записывается так:

```
/* strcmp: выдаёт <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 операторы ++ и -- могут быть и префиксными, и постфиксными, возможны (хотя встречаются и не так часто) другие их сочетания с оператором *. Например,

```
*--p
```

уменьшит p прежде, чем по этому указателю будет получена литера. Например, следующие два выражения:

```
*p++ = val;   /* поместить val в стек */
val = *--p;   /* взять из стека значение и поместить в val */
```

являются стандартными записями для посылки в стек и взятия из стека. (См. разд. 4.3)

Описания функций, упомянутых в этом разделе, а также ряда других стандартных функций, работающих со стрингами, содержатся в головном файле <string.h>.

Упражнение 5.3. Используя указатели, напишите функцию strcat, которую мы рассматривали в гл. 2 (функция strcat(s,t) копирует стринг t в конец стринга s).

Упражнение 5.4. Напишите функцию strend(s,t), которая выдаёт 1, если стринг t расположен в конце стринга s, и нуль в противном случае.

Упражнение 5.5. Напишите варианты библиотечных функций strncpy, strncat и strncmp, которые оперируют с первыми литерами своих аргументов, число которых не превышает n. Например, strncpy(t,s,n) копирует не более n литер t в s. Полные описания этих функций содержатся в приложении B.

Упражнение 5.6. Отберите подходящие программы из предыдущих глав и упражнений и перепишите их, используя вместо индексирования указатели. Подойдут, в частности, программы getline (гл. 1 и 4), atoi, itoa и их варианты (гл. 2, 3 и 4), reverse (гл. 3), а также strindex и getop (гл. 4).

5.6. Массивы указателей, указатели на указатели

Как и любые другие переменные, указатели можно группировать в массивы. Для иллюстрации этого напишем программу, сортирующую в ал-

фавитном порядке текстовые строки; это будет упрощённый вариант программы sort системы UNIX.

В гл. 3 мы привели функцию сортировки по Шеллу, которая упорядочивает массив целых, а в гл. 4 улучшили её, повысив быстродействие.

Те же алгоритмы используются и здесь, однако теперь они будут обрабатывать текстовые строки, которые могут иметь разную длину и сравнение или перемещение которых невозможно выполнить за одну операцию. Нам необходимо выбрать некоторое представление данных, которое бы позволило удобно и эффективно работать с текстовыми строками произвольной длины. Для этого воспользуемся массивом указателей на начала строк. Поскольку строки в памяти расположены вплотную друг к другу, к каждой отдельной строке доступ просто осуществлять через указатель на её первую литеру. Сами указатели можно организовать в виде массива. Одна из возможностей сравнить две строки – передать указатели на них функции strcmp. Чтобы поменять местами строки, достаточно будет поменять местами в массиве их указатели (а не сами строки).

Здесь снимаются сразу две проблемы: одна – связанная со сложностью управления памятью, а вторая – с большими накладными расходами при перестановках самих строк.

Процесс сортировки распадается на три этапа:

```
чтение всех строк из ввода
сортировка введённых строк
печать их по порядку
```

Как обычно, выделим функции, соответствующие естественному делению задачи, и напишем главную программу, управляющую этими функциями. Отложим на время реализацию этапа сортировки и сосредоточимся на структуре данных и вводе-выводе.

Программа ввода должна прочитать и запомнить литеры всех строк, а также построить массив указателей на строки. Она, кроме того, должна подсчитать число введённых строк – эта информация понадобится для сортировки и печати. Так как функция ввода может работать только с конечным числом строк, то, если их введено слишком много, она будет выдавать некоторое значение, которое никогда не совпадёт ни с каким количеством строк, например, –1.

Программа вывода занимается только тем, что печатает строки, причём в том порядке, в котором в массиве указателей на них расположены ссылки.

```
#include <stdio.h>
#include <string.h>
#define MAXLINES 5000       /* максимальное число 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 */
main()
{
    int nlines;     /* количество прочитанных строк */
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("ошибка: слишком много строк\n");
        return 1;
    }
}

#define MAXLEN 1000    /* максимальная длина строки */
int getline(char *, int);
char *alloc(int);

/* readlines: чтение строк */
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'; /* убираем литеру \n */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
```

```
    return nlines;
}

/* writelines: печать строк */
void writelines(char *lineptr[], int nlines)
{
    int i;
    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

Функция getline взята из разд. 1.9.

Основное новшество здесь – декларация lineptr:

```
char *lineptr[MAXLINES];
```

в которой сообщается, что lineptr есть массив из MAXFILES элементов, каждый из которых представляет собой указатель на char. Иначе говоря, lineptr[i] – указатель на литеру, а *lineptr[i] – литера, на которую он указывает (первая литера i-й строки текста).

Так как lineptr – имя массива, его можно трактовать как указатель, т.е. так же, как мы это делали в предыдущих примерах, и writelines переписать следующим образом:

```
/* writelines: печать строк */
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}
```

Вначале *lineptr ссылается на первую строку; каждое приращение указателя приводит к тому, что *lineptr ссылается на следующую строку, и делается это до тех пор, пока nlines не станет нулём.

Теперь, когда мы разобрались с вводом и выводом, можно приступить к сортировке. Быструю сортировку, описанную в гл. 4, надо несколько модифицировать: нужно изменить декларации, а операцию сравнения заменить обращением к strcmp. Алгоритм остался тем же, и это даёт нам определённую уверенность в его правильности.

```
/* qsort: сортирует v[left]...v[right] по возрастанию */
void qsort(char *v[], int left, int right)
{
    int i, last;
    void swap(char *v[], int i, int j);
    if (left >= right) /* ничего не делается, если       */
        return;        /* в массиве менее 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);
}
```

Небольшие поправки требуются и в программе перестановки.

```
/* swap: переставить v[i] и v[j] между собой */
void swap(char *v[], int i, int j)
{
    char *temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Так как каждый элемент массива v (т.е. lineptr) является указателем на литеру, temp должен иметь тот же тип, что и v – тогда можно будет осуществлять пересылки между temp и элементами v.

Упражнение 5.7. Напишите новую версию readlines, которая запоминала бы строки в массиве, определённом в main, а не запрашивала память посредством программы alloc. Насколько быстрее эта программа?

5.7. Многомерные массивы

В Си имеется возможность задавать прямоугольные многомерные массивы, правда, на практике по сравнению с массивами указателей они используются значительно реже. В этом разделе мы продемонстрируем некоторые их свойства.

Рассмотрим задачу перевода даты «день-месяц» в «день года» и обратно. Например, 1 марта – это 60-й день невисокосного или 61-й день високосного года. Определим две функции для этих преобразований: функция day_of_year будет преобразовывать месяц-день в день года, а month_day – день года в месяц-день. Поскольку последняя функция вычисляет два значения, аргументы месяц и день будут указателями. Так,

```
month_day(1988, 60, &m, &d)
```

установит в m число 2, а в d – значение 29 (29 февраля).

Нашим функциям нужна одна и та же информация, а именно таблица, содержащая числа дней каждого месяца. Так как для високосного и невисокосного годов эти таблицы будут различаться, проще иметь две отдельные строки в двумерном массиве, чем во время вычислений отслеживать особый случай с февралём. Массив и функции, выполняющие преобразования, имеют следующий вид:

```
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: определяет день 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: определяет месяц и день 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;
}
```

Напоминаем, что арифметическое значение логического выражения (например, выражения, с помощью которого вычислялось leap) равно нулю (ложь) или единице (истина), так что мы можем использовать его как индекс в массиве daytab.

Массив daytab должен быть внешним по отношению к обеим функциям day_of_year и month_day, так как он нужен и той и другой. Мы сделали его типа char, чтобы проиллюстрировать законность применения типа char для малых целых.

Массив daytab – это первый массив из числа двумерных, с которыми мы ещё не имели дела. Строго говоря, в Си двумерный массив рассматривается как одномерный массив, каждый элемент которого – также массив. Поэтому индексирование изображается как

```
daytab[i][j]          /* [строка][столбец] */
```

а не как

```
daytab[i,j]           /* НЕВЕРНО */
```

Особенность двумерного массива в Си заключается лишь в форме записи, в остальном его можно трактовать почти так же, как в других языках. Элементы запоминаются строками, следовательно, при переборе их в том порядке, как они расположены в памяти, чаще будет изменяться самый правый индекс.

Массив инициализируется списком начальных значений, заключённым в фигурные скобки; каждая строка двумерного массива инициализируется соответствующим подсписком. Нулевой столбец добавлен в начало daytab лишь для того, чтобы индексы, которыми мы будем пользоваться, совпадали с естественными номерами месяцев от 1 до 12. Экономить пару ячеек памяти здесь нет никакого смысла, а программа, в которой уже не надо корректировать индекс, выглядит более ясной.

Если двумерный массив передаётся функции в качестве аргумента, то декларация соответствующего ему параметра должна содержать количество столбцов; количество строк в данном случае несущественно, поскольку, как и прежде, функции будет передана ссылка на массив строк, каждая из которых есть массив из 13 значений типа char. В нашем частном случае имеем указатель на объекты, являющиеся массивами из 13 значений типа char. Таким образом, если массив daytab передаётся некоторой функции f, то эту функцию можно было бы определить следующим образом:

```
f(char daytab[2][13]) { ... }
```

Вместо этого можно записать

```
f(char daytab[][13]) { ... }
```

поскольку число строк здесь не имеет значения, или

```
f(char (*daytab)[13]) { ... }
```

последняя запись декларирует, что параметр есть указатель на массив из 13 значений типа char. Скобки здесь необходимы, так как квадратные скобки [] имеют более высокий приоритет, чем *. Без скобок декларация

```
char *daytab[13]
```

определяет массив из 13 указателей на char. В более общем случае только первое измерение (соответствующее первому индексу) можно не задавать, все другие специфицировать необходимо.

В разд. 5.12 мы продолжим рассмотрение сложных деклараций.

Упражнение 5.8. В функциях day_of_year и month_day нет никаких проверок правильности вводимых дат. Устраните этот недостаток.

5.8. Инициализация массивов указателей

Напишем функцию month_name(n), которая возвращает ссылку на стринг литер, содержащий название n-го месяца. Эта функция идеальна для демонстрации использования статического массива. Функция month_name имеет в своём личном распоряжении массив стрингов, на один из которых она и возвращает ссылку. Ниже покажем, как инициализируется этот массив имён.

Синтаксис задания начальных значений аналогичен синтаксису предыдущих инициализаций:

```
/* month_name: возвращает имя n-го месяца */
char *month_name(int n)
{
    static char *name[] = {
        "Неверный месяц",
        "Январь", "Февраль", "Март",
        "Апрель", "Май", "Июнь",
        "Июль", "Август", "Сентябрь",
        "Октябрь", "Ноябрь", "Декабрь"
    };
    return (n < 1 || n > 12) ? name[0] : name [n];
}
```

Декларация, определяющая name как массив указателей на литеры, такая же, как и декларация lineptr в программе сортировки. Инициализатором служит список стрингов, каждому из которых соответствует определённое место в массиве. Литеры i-го стринга где-то размещены, и указатель на них запоминается в name[i]. Так как размер массива name не специфицирован, компилятор вычислит его по количеству заданных начальных значений.

5.9. Указатели вместо многомерных массивов

Начинающие программировать на Си иногда не понимают, в чём разница между двумерным массивом и массивом указателей типа name из приведённого примера. Для двух следующих определений:

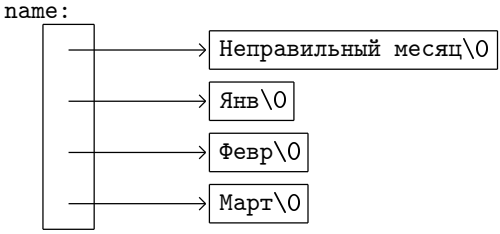
```
int a[10][20];
int *b[10];
```

записи a[3][4] и b[3][4] будут синтаксически правильными ссылками на некоторое значение типа int. Однако только a является истинно двумерным массивом: для двухсот элементов типа int будет выделена память, а вычисление смещения элемента a[*строка*,*столбец*] от начала массива будет вестись

по формуле $20 \times$ *строка* $+$ *столбец*, учитывающей его прямоугольную природу. Для b же определяются только 10 указателей, причём без инициализации. Инициализация должна задаваться явно – либо статически, либо в процессе счёта. Предположим, что каждый элемент b ссылается на двадцатиэлементный массив, в результате где-то будут выделены пространство, в котором разместятся 200 значений типа int, и ещё 10 ячеек для указателей. Важное преимущество массива указателей в том, что строки такого массива могут иметь разные длины. Таким образом, каждый элемент b не обязательно ссылается на двадцатиэлементный вектор; один может ссылаться на два элемента, другой – на пятьдесят, а некоторые и вовсе могут ни на что не ссылаться.

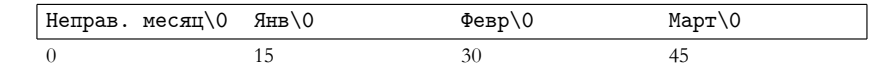
Наши рассуждения здесь велись в отношении целых значений, однако чаще массивы указателей используются для работы со стрингами литер, различающимися по длине, как это было в функции month_name. Сравните определение массива указателей и соответствующий ему рисунок:

```
char *name[] = {"Неправильный месяц", "Янв", "Февр", "Март"};
```



с определением и рисунком для двумерного массива:

```
char aname[][15] = {"Неправ. месяц", "Янв", "Февр", "Март"};
```



Упражнение 5.9. Перепишите программы day_of_year и month_of_year, используя вместо индексов указатели.

5.10. Аргументы в командной строке

В операционной среде, обеспечивающей поддержку Си, имеется возможность передать аргументы или параметры запускаемой программе при помощи командной строки. В момент вызова main получает два аргумента. В первом, обычно называемом argc (сокращение от argument count), стоит

количество аргументов, задаваемых в командной строке. Второй, argv (от argument vector), является указателем на массив литерных стрингов, содержащих сами аргументы. Для работы с этими стрингами обычно используются указатели нескольких уровней.

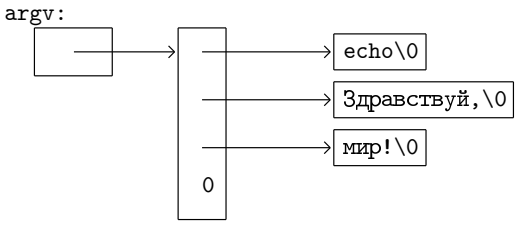
Простейший пример – программа echo («эхо»), которая печатает аргументы своей командной строки в одной строчке, отделяя их друг от друга пробелами. Так, команда

```
echo Здравствуй, мир!
```

напечатает

```
Здравствуй, мир!
```

По соглашению argv[0] есть имя вызываемой программы, так что значение argc никогда не бывает меньше 1. Если argc равен 1, то в командной строке после имени программы никаких аргументов нет. В нашем примере argc равен 3, и соответственно argv[0], argv[1] и argv[2] есть стринги "echo", "Здравствуй," и "мир!". Первый необязательный аргумент – это argv[1], последний – argv[argc-1]. Кроме того, стандарт требует, чтобы argv[argc] всегда был пустым указателем.



Первая версия программы echo трактует argv как массив литерных указателей.

```
#include <stdio.h>
/* эхо аргументов командной строки; версия 1 */
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;
}
```

Так как argv есть указатель на массив указателей, мы можем работать с ним как с указателем, а не как с индексируемым массивом. Следующая программа основана на продвижении argv, он продвигается так, что его значение в каждый отдельный момент ссылается на очередной указатель на char; перебор указателей заканчивается, когда исчерпан argc.

```
#include <stdio.h>
/* эхо аргументов командной строки; версия 2 */
main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

Аргумент argv – указатель на начало массива аргументных стрингов. Использование в ++argv префиксного оператора ++ приведёт к тому, что первым будет напечатан argv[1], а не argv[0]. Каждое очередное продвижение указателя даёт нам следующий аргумент, на который ссылается *argv. В это же время значение argc уменьшается на 1, и, когда оно станет нулём, все аргументы будут напечатаны.

Инструкцию printf можно было бы написать и так:

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

Как видим, формат в printf может быть выражением.

В качестве второго примера возьмём программу поиска образца, рассмотренную в разд. 4.1, и несколько усовершенствуем её. Если вы помните, образец для поиска мы «вмонтировали» глубоко в программу, а это, очевидно, не лучшее решение. Построим нашу программу по аналогии с grep из UNIXa, т.е. так, чтобы образец для поиска задавался первым аргументом в командной строке.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000
int getline(char *line, int max);
/* find: печать строк с образцом заданным 1-м аргументом */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;
    if (argc != 2)
        printf("Используйте в find образец\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
            }
    return found;
}
```

Стандартная функция `strstr(s,t)` выдаёт указатель на первый найденный стринг `t` в стринге `s` или `NULL`, если такого в `s` не оказалось. Её описание хранится в головном файле `<string.h>`.

Эту модель можно развивать и дальше, чтобы проиллюстрировать другие конструкции с указателями. Предположим, что мы вводим ещё два необязательных аргумента. Один из них предписывает печатать все строки, *кроме* тех, в которых встречается образец; второй – перед каждой выводимой строкой печатать её порядковый номер.

По общему соглашению для Си-программ в системе UNIX знак минус перед аргументом может иногда играть роль необязательного признака или дополнительного параметра. Так, `-x` служит признаком слова «кроме», которое изменяет задание на противоположное, а `-n` указывает на потребность в нумерации строк. Тогда, например, команда

```
find -x -n образец
```

напечатает все строки, в которых не найден указанный образец, и, кроме того, перед каждой строкой укажет её номер.

Необязательные аргументы разрешается располагать в любом порядке, при этом лучше, чтобы остальная часть программы не зависела от числа представленных аргументов. Кроме того, пользователю было бы удобно, если бы он мог комбинировать необязательные аргументы, например, так:

```
find -nx образец
```

А теперь запишем нашу программу.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000
int getline(char *line, int max);

/* find: печать строк по образцу из 1-го 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: неверный парам. %c\n", c);
                    argc = 0;
                    found = -1;
                    break;
                }
        if (argc != 1)
            printf("Используйте: find -x -n образец\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;
    }
```

Перед получением очередного аргумента `argc` уменьшается на 1, а `argv` «продвигается» на следующий аргумент. После завершения цикла при отсутствии ошибок `argc` содержит количество ещё не обработанных аргументов, а `argv` указывает на первый из них. Таким образом, `argc` должен быть равен 1, а `*argv` указывать на образец. Заметим, что `*++argv` является указателем на аргумент-стринг, а `(*++argv)[0]` – его первой литерой, на которую можно сослаться и другим способом: `**++argv`. Поскольку оператор индексации `[]` имеет более высокий приоритет, чем `*` и `++`, круглые скобки здесь обязательны, без них запись трактовалась бы так же, как `*++(argv[0])`. Именно эту запись мы применим во внутреннем цикле, где просматриваются литеры конкретного аргумента. Во внутреннем цикле выражение `*++argv[0]` продвигает указатель `argv[0]`.

Потребность в более сложных указательных выражениях возникает не так уж часто. Но если такое случится, то, разбивая процесс вычисления указателя на два или три шага, вы облегчите восприятие этого выражения

Упражнение 5.10. Напишите программу `expr`, интерпретирующую обратную польскую запись выражения, задаваемую командной строкой, в которой каждый оператор и операнд представлен отдельным аргументом. Например,

```
expr 2 3 4 + *
```

вычисляется так же, как выражение $2 \times (3 + 4)$.

Упражнение 5.11. Усовершенствуйте программы `entab` и `detab` (см. упражнения 1.20 и 1.21) таким образом, чтобы через аргументы можно было задавать список «стопов» табуляции.

Упражнение 5.12. Расширьте возможности `entab` и `detab` таким образом, чтобы при обращении вида

```
entab -m +n
```

«стопы» табуляции начинались *m*-й позиции и выполнялись через каждые *n* позиций. Разработайте удобный для пользователя вариант поведения программы по умолчанию (когда нет никаких аргументов).

Упражнение 5.13. Напишите программу `tail`, печатающую *n* последних введённых строк. По умолчанию значение *n* равно 10, но при желании *n* можно задать с помощью аргумента. Обращение вида

```
tail -n
```

печатает *n* последних строк. Программа должна вести себя осмысленно при любых входных данных и любом значении *n*. Напишите программу так, чтобы наилучшим образом использовать память; запоминание строк организуйте, как в программе сортировки, описанной в разд. 5.6, а не на основе двумерного массива с фиксированным размером строки.

5.11. Указатели на функции

В Си сама функция не является переменной, но можно определить указатель на функцию и работать с ним, как с обычной переменной: присваивать, размещать в массиве, передавать в качестве параметра функции, возвращать как результат из функции и т.д. Для иллюстрации этих возможностей воспользуемся программой сортировки, которая уже встречалась в настоящей главе. Изменим её так, чтобы при задании необязательного аргумента `-n` вводимые строки упорядочивались по их числовому значению, а не в лексикографическом порядке.

Сортировка, как правило, распадается на три части: на сравнение, определяющее упорядоченность пары объектов; перестановку, меняющую порядок пары объектов на обратный, и сортирующий алгоритм, который осуществляет сравнения и перестановки до тех пор, пока все объекты не будут упорядочены. Алгоритм сортировки не зависит от операций сравнения и перестановки, так что, передавая ему различные функции сравнения и перестановки в качестве параметров, его можно настроить на различные критерии сортировки.

Лексикографическое сравнение двух строк выполняется функцией `strcmp` (мы уже использовали эту функцию в ранее рассмотренной программе сортировки); нам также потребуется программа `numcmp`, сравнивающая

две строки как числовые значения и возвращающая результат сравнения в том же виде, в каком его выдаёт strcmp. Эти функции описываются перед main, а указатель на одну из них передаётся функции qsort. Чтобы сосредоточиться на главном, мы упростили себе задачу, отказавшись от анализа возможных ошибок при задании аргументов.

```
#include <stdio.h>
#include <string.h>
#define MAXLINES 5000       /* максимальное число строк */
char *lineptr[MAXLINES];    /* указатели на строки текста */
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 */
main(int argc, char *argv[])
{
    int nlines;        /* количество прочитанных строк */
    int numeric = 0;   /* 1, если сорт. по числ. знач. */

    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("введено слишком много строк\n");
        return 1;
    }
}
```

В обращениях к функциям qsort, strcmp и numcmp их имена трактуются как адреса этих функций. Поэтому оператор & перед ними не нужен, как он был не нужен и перед именем массива.

Мы написали qsort так, чтобы она могла обрабатывать данные любого типа, а не только стринги литер. Как видно из прототипа, функция qsort в качестве своих аргументов ожидает массив ссылок, два целых значения и функцию с двумя аргументами-указателями. В качестве указателей-аргументов заданы указатели обобщённого типа void *. Любой указатель можно привести к типу void * и обратно без потери информации. Поэтому мы можем обратиться к qsort, предварительно преобразовав аргумен-

ты в void *. Внутри функции сравнения её аргументы будут приведены к нужному ей типу. На самом деле эти преобразования никакого влияния на представления аргументов не оказывают, они лишь обеспечивают согласованность типов для компилятора.

```
/* qsort: сортирует v[left]...v[right] по возрастанию */
void qsort(void *v[], int left, int right,
           int (*comp)(void *, void *))
{
    int i, last;
    void swap(void *v[], int, int);

    if (left >= right) /* ничего не делается, если       */
        return;        /* в массиве менее 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);
}
```

Повнимательней приглядимся к декларациям. Четвёртый параметр функции qsort:

```
int (*comp)(void *, void *)
```

сообщает, что comp есть указатель на функцию, которая имеет два аргумента-указателя и выдаёт результат типа int.

Использование comp в строке

```
if ((*comp)(v[i], v[left]) < 0)
```

согласуется с декларацией «comp – это указатель на функцию», и, следовательно, *comp есть функция, а

```
(*comp)(v[i], v[left])
```

обращение к ней. Скобки здесь нужны, чтобы обеспечить правильную трактовку декларации; без них декларация

```
int *comp(void *, void *)        /* НЕВЕРНО */
```

описывала бы comp, как функцию, возвращающую ссылку на int, а это совсем не то, что требуется.

Мы уже рассматривали функцию strcmp, сравнивающую два стринга. Ниже приведена функция numcmp, которая сравнивает два стринга, рассматривая их как числа; предварительно они переводятся в числовые значения функцией atof.

```
#include <stdlib.h>

/* numcmp: сравнивает s1 и s2 как числа */
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;
}
```

Программу сортировки можно пополнить и другими возможностями; реализовать некоторые из них предлагается в качестве упражнений.

Упражнение 5.14. Модифицируйте программу сортировки, чтобы она реагировала на параметр `-r`, указывающий, что объекты нужно сортировать в обратном порядке, т.е. в порядке убывания. Обеспечьте, чтобы `-r` работал и вместе с `-n`.

Упражнение 5.15. Введите в программу необязательный параметр `-f`, задание которого делало бы неразличимыми литеры нижнего и верхнего регистров (например, `a` и `A` должны оказаться при сравнении равными).

Упражнение 5.16. Предусмотрите в программе необязательный параметр `-d`, который заставит программу при сравнении учитывать только буквы, цифры и пробелы. Организуйте программу таким образом, чтобы этот параметр мог работать вместе с параметром `-f`.

Упражнение 5.17. Реализуйте в программе возможность работы с полями: возможность сортировки по полям внутри строк. Для каждого поля предусмотрите свой набор параметров. Предметный указатель этой книги* упорядочивался с параметрами: `-df` для терминов и `-n` для номеров страниц.

*Здесь имеется в виду оригинал книги на английском языке. – *Примеч. пер.*

5.12. Сложные декларации

Иногда Си ругают за синтаксис деклараций, особенно тех, которые содержат в себе указатели на функции. Таким синтаксис получился в результате нашей попытки сделать похожими записи объектов при их описании и использовании. В простых случаях этот синтаксис хорош, однако в сложных ситуациях он вызывает затруднения, поскольку декларации перенасыщены скобками и их невозможно читать слева направо. Проблему иллюстрирует различие следующих двух деклараций:

```
int *f();       /* f: фу-ция, возвращающая ук-ль на int */
int (*pf)();    /* pf: ук-ль на ф-цию, возвращающую int */
```

Приоритет префиксного оператора * ниже, чем приоритет (), поэтому во втором случае скобки необходимы.

Хотя на практике по-настоящему сложные декларации встречаются редко, всё же важно знать, как их понимать, а если потребуется, и как их конструировать.

Укажем хороший способ: декларации можно синтезировать, двигаясь небольшими шагами с помощью typedef; этот способ рассмотрен в разд. 6.7. В настоящем разделе на примере двух программ, осуществляющих преобразования правильных Си-деклараций в соответствующие им словесные описания и обратно, мы демонстрируем иной способ конструирования деклараций. Словесное описание читается слева направо.

Первая программа, dcl, – более сложная. Она преобразует Си-декларации в словесные описания так, как показано в следующих примерах:

```
char **argv
    argv: указ. на указ. на char
int (*daytab)[13]
    daytab: указ. на массив[13] из int
void *comp()
    comp: функц. возвр. указ. на void
void (*comp)()
    comp: указ. на функц. возвр. void
char (*(*x())[])()
    x: функц. возвр. указ. на массив[] из указ. на функц.
       возвр. char
char (*(*x[3])())[5]
    x: массив[3] из указ. на функц. возвр. указ.
       на массив[5] из char
```

Функция dcl в своей работе использует грамматику, специфицирующую декларатор. Эта грамматика строго изложена в разд. А.8.5 приложения А, а в упрощённом виде записывается так:

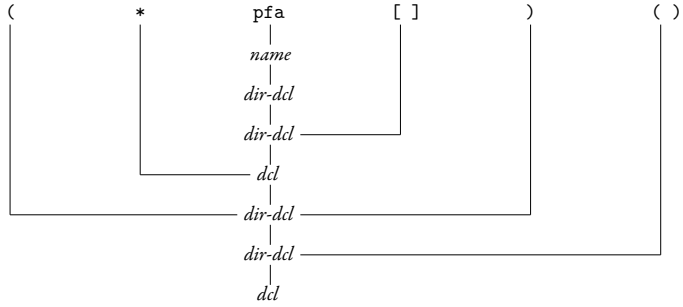
```
dcl:            optional *'s direct-dcl
direct-dcl:     name
                (dcl)
                direct-dcl()
                direct-dcl[optional size]
```

Говоря простым языком, *dcl* есть *direct-dcl*, перед которым может стоять *'s* (т.е. одна или несколько звёздочек), где *direct-dcl* есть *name* (имя), или *dcl* в скобках, или *direct-dcl* с последующей парой скобок, или *direct-dcl* с последующей парой квадратных скобок, внутри которых может быть помещён размер (*size*).

Эту грамматику можно использовать для грамматического разбора деклараций. Рассмотрим, например, такой декларатор:

```
(*pfa[])()
```

Имя `pfa` будет классифицировано как *name* и, следовательно, как *direct-dcl*. Затем `pfa[]` будет распознано как *direct-dcl*, а `*pfa[]` – как *dcl* и, следовательно, `(*pfa[])` есть *direct-dcl*. Далее `(*pfa[])()` есть *direct-dcl* и, таким образом, *dcl*. Этот грамматический разбор можно проиллюстрировать деревом разбора (здесь *direct-dcl* обозначено более коротко, а именно *dir-dcl*):



Сердцевиной программы обработки декларатора является пара функций `dcl` и `dirdcl`, осуществляющих грамматический разбор декларации согласно приведённой грамматике. Поскольку грамматика определена рекурсивно, эти функции обращаются друг к другу рекурсивно, по мере распознавания отдельных частей декларации. Метод, применённый в обсуждаемой программе для грамматического разбора, называется рекурсивным спуском.

```
/* dcl: разбор декларатора */
void dcl(void)
{
    int ns;
    for (ns = 0; gettoken() == '*'; ) /* подсчёт звёздочек */
        ns++;
```

```
    dirdcl();
    while (ns-- > 0)
        strcat(out, " указ. на");
}
/* dirdcl: разбор непосредственного декларатора */
void dirdcl(void)
{
    int type;
    if (tokentype == '(') {      /* ( dcl ) */
        dcl();
        if (tokentype != ')')
            printf("ошибка: пропущена )\n");
    } else if (tokentype == NAME)     /* имя переменной */
        strcpy(name, token);
    else
        printf("ошибка: должно быть name или (dcl)\n");
    while ((type=gettoken()) == PARENS || type == BRACKETS)
        if (type == PARENS)
            strcat(out, " функц. возвр.");
        else {
            strcat(out, " массив");
            strcat(out, token);
            strcat(out, " из");
        }
}
```

Приведённые программы служат только иллюстративным целям и не вполне надёжны. Что касается dcl, то её возможности существенно ограничены. Она может работать только с простыми типами вроде char и int и не справляется с типами аргументов в функциях и с квалификаторами подобными const. Лишние пробелы для неё опасны. Она не предпринимает никаких мер по выходу из ошибочной ситуации, и поэтому неправильные декларации также противопоказаны ей. Устранение этих недостатков мы оставляем до упражнений.

Ниже 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];       /* текст последней лексемы */
char name[MAXTOKEN];        /* имя */
char datatype[MAXTOKEN];    /* тип = char, int и т.д. */
char out[1000];             /* выдаваемый текст */

main()   /* преобразование декларации в словесное описание */
{
    while (gettoken() != EOF) {    /* 1-я лексема в строке */
        strcpy(datatype, token);   /* это тип данных */
        out[0] = '\0';
        dcl();         /* разбор остальной части строки */
        if (tokentype != '\n')
            printf("синтаксическая ошибка\n");
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}
```

Функция gettoken пропускает пробелы и табуляции и затем получает следующую лексему из ввода; «лексема» – это имя, или пара круглых скобок, или пара квадратных скобок (быть может, с помещённым в них числом), или любая другая единичная литера.

```
int gettoken(void)     /* возвращает следующую 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 были рассмотрены в гл. 4.

Обратное преобразование реализуется легче, особенно если не придавать значения тому, что будут генерироваться лишние скобки. Программа undcl превращает фразу типа «x есть функция, возвращающая указатель на массив указателей на функции, возвращающие char», которую мы будем представлять в виде

```
x () * [] () char
```

в декларацию

```
char (*(*x())[])()
```

Сокращённая запись словесного описания позволяет воспользоваться функцией gettoken. Функция undcl использует те же самые внешние переменные, что и dcl.

```
/* undcl: преобразует словесное описание в декларацию */
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("неверный элемент %s в фразе\n",token);
        printf("%s\n", out);
    }
    return 0;
}
```

Упражнение 5.18. Видоизмените `dcl` таким образом, чтобы она нормально обрабатывала ошибки во входной информации.

Упражнение 5.19. Модифицируйте `undcl` так, чтобы она не генерировала лишних скобок.

Упражнение 5.20. Расширьте возможности `dcl`: функция должна справляться с описателями типов её аргументов, квалификаторами `const` и т.д.

Глава 6

Структуры

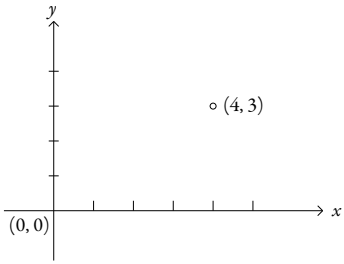
Структура – это одна или несколько переменных (возможно, различных типов), которые для удобства работы с ними сгруппированы под одним именем. (В некоторых языках, в частности в Паскале, структуры называются записями.) Структуры помогают в организации сложных данных (особенно в больших программах), поскольку позволяют группу связанных между собой переменных трактовать не как множество отдельных элементов, а как единое целое.

Распространённый пример структуры – строка платёжной ведомости. Она содержит такие сведения о служащем, как его полное имя, адрес, номер карточки социального страхования, зарплата и т.д. Некоторые из этих характеристик сами могут быть структурами: например, полное имя состоит из нескольких компонент (фамилии, имени и отчества); аналогично адрес, и даже зарплата. Другой пример (более типичный для Си) – из области графики: точка есть пара координат, прямоугольник есть пара точек и т.д.

Главные изменения, внесённые стандартом ANSI в отношении структур, – это введение для них операции присваивания. Структуры могут копироваться, над ними могут выполняться операции присваивания, их можно передавать функциям в качестве аргументов, а функции могут возвращать их в качестве результатов. В большинстве компиляторов уже давно реализованы эти возможности, но теперь они точно оговорены стандартом. Для автоматических структур и массивов теперь также допускается инициализация.

6.1. Основные сведения о структурах

Сконструируем несколько графических структур. В качестве основного объекта выступает точка с координатами *x* и *y*, и пусть они имеют тип `int`.



Указанные две компоненты можно поместить в структуру, описанную, например, следующим образом:

```
struct point {
    int x;
    int y;
};
```

Описание структуры начинается с ключевого слова `struct` и содержит список деклараций, заключённый в фигурные скобки. За словом `struct` может следовать имя, называемое *тегом*[*] *структуры* (`point` в нашем случае). Тег даёт название структуре данного вида и далее может служить кратким обозначением той части декларации, которая заключена в фигурные скобки.

Перечисленные в структуре переменные называются *членами*. Имена членов и тегов без каких-либо коллизий могут совпадать с именами обычных переменных (т.е. не членов), так как они всегда различимы по контексту. Более того, одни и те же имена членов могут встречаться в разных структурах, хотя, если следовать хорошему стилю программирования, лучше одинаковые имена давать только близким по смыслу объектам.

Декларация структуры – это тип. За правой фигурной скобкой, закрывающей список членов, могут следовать переменные точно так же, как они могут быть указаны после названия любого базового типа. Таким образом, запись

```
struct { ... } x, y, z;
```

с точки зрения синтаксиса аналогична записи

```
int x, y, z;
```

в том смысле, что каждая декларирует `x`, `y` и `z` как переменные указанного типа. Обе записи приведут к тому, что где-то будет выделена память соответствующего размера.

Декларация структуры, не содержащей списка переменных, не резервирует памяти: она просто описывает шаблон, или образец структуры. Однако если структура имеет тег, то этим тегом далее можно пользоваться при определении структурных объектов. Например, с помощью заданной выше декларации структуры `point` строка

[*] От английского слова tag – ярлык, этикетка. – *Примеч. пер.*

```
struct point pt;
```

определяет структурную переменную pt типа struct point. Структурную переменную при её определении можно инициализировать, формируя список инициализаторов её членов в виде константных выражений:

```
struct point maxpt = { 320, 200 };
```

Инициализировать автоматические структуры можно также присваиванием или обращением к функции, возвращающей результат в виде структуры соответствующего типа.

Доступ к отдельному члену структуры осуществляется посредством конструкции вида:

имя-структуры . член

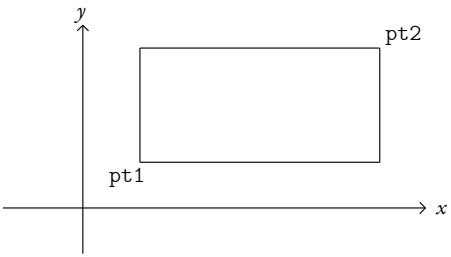
Оператор доступа к члену структуры « . » соединяет имя структуры и имя члена. Чтобы напечатать, например, координаты точки pt, годится следующее обращение к printf:

```
printf("%d,%d", pt.x, pt.y);
```

Другой пример: чтобы вычислить расстояние от начала координат $(0, 0)$ до pt, можно написать

```
double dist, sqrt(double);
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Структуры могут быть вложены друг в друга. Одно из возможных представлений прямоугольника – это пара точек на углах одной из его диагоналей:




```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

Структура rect содержит две структуры point. Если мы декларируем screen как

```
struct rect screen;
```

то

```
screen.pt1.x
```

ссылается на координату *x* точки pt1 из screen.

6.2. Структуры и функции

Единственно возможные операции над структурами – это их копирование, присваивание, взятие адреса с помощью & и осуществление доступа к её членам. Передача структур функциям в качестве аргументов и возврат их от функций в виде результата также относятся к операциям копирования и присваивания. Структуры нельзя сравнивать. Инициализировать структуру можно списком константных значений её членов; автоматическую структуру можно инициализировать также присваиванием.

Чтобы лучше познакомиться со структурами, напишем несколько функций, манипулирующих точками и прямоугольниками. Возникает вопрос: а как передавать функциям названные объекты? Существует по крайней мере три подхода: передавать компоненты по отдельности, передавать всю структуру целиком и передавать указатель на структуру. Каждый подход имеет свои плюсы и минусы.

Первая функция, makepoint, получает два целых значения и возвращает структуру point.

```
/* makepoint: формирует точку по компонентам x и y */
struct point makepoint(int x, int y)
{
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
}
```

Заметим: никакой путаницы из-за того, что имя аргумента совпадает с именем члена структуры не возникает; более того, одно и то же имя подчёркивает родство обозначаемых им объектов.

Теперь с помощью makepoint можно выполнять динамическую инициализацию любой структуры или формировать структурные аргументы для той или иной 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);
```

Нам может понадобиться ряд функций, реализующих различные операции над точками. В качестве примера рассмотрим следующую функцию:

```
/* addpoint: сложение двух точек */
struct point addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Здесь оба аргумента и возвращаемое значение – структуры. Мы увеличиваем компоненты прямо в p1 и не используем для этого временной переменной, чтобы подчеркнуть, что структурные параметры передаются по значению так же, как и любые другие.

В качестве другого примера приведём функцию ptinrect, которая проверяет: находится ли точка внутри прямоугольника, относительно которого мы принимаем соглашение, что в него входят его левая и нижняя стороны, но не входят верхняя и правая.

```
/* ptinrect: возвращает 1, если p в r, и 0 в прот. 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 стандартном виде, т.е. координаты точки pt1 меньше соответствующих координат точки pt2. Следующая функция гарантирует получение прямоугольника в каноническом виде.

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
/* canonrect: канонизация координат прямоугольника */
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 функции передаётся большая структура, то, чем копировать её целиком, эффективнее передать указатель на неё. Указатели на структуры ничем не отличаются от указателей на обычные переменные. Декларация

```
struct point *pp;
```

сообщает, что pp есть указатель на структуру типа struct point. Если pp ссылается на структуру point, то *pp есть сама структура, а (*pp).x и (*pp).y – её члены. Используя указатель pp, мы могли бы написать

```
struct point origin, *pp;
pp = &origin;
printf("origin: (%d,%d)\n", (*pp).x, (*pp).y);
```

Скобки в (*pp).x необходимы поскольку приоритет оператора . выше, чем приоритет *. Выражение *pp.x будет проинтерпретировано как *(pp.x), что неверно, поскольку pp.x не является указателем.

Указатели на структуры используются весьма часто, поэтому для доступа к её членам была придумана ещё одна, более короткая форма записи. Если p – указатель на структуру, то

```
p->член-структуры
```

есть её отдельный член. (Оператор -> состоит из знака -, за которым сразу следует знак >.) Поэтому printf можно переписать в виде

```
printf("origin: (%d,%d)\n", pp->x, pp->y);
```

Оба оператора . и -> выполняются слева направо. Таким образом, при наличии декларации

```
struct rect r, *rp = r;
```

следующие четыре выражения будут эквивалентны:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Операторы доступа к членам структуры . и -> вместе с операторами вызова функции () и индексации массива [] занимают самое высокое положение в иерархии приоритетов и выполняются раньше любых других операторов. Например, если задана декларация

```
struct {
    int len;
    char *str;
} *p;
```

то

```
++p->len
```

увеличит на 1 значение члена структуры len, а не указатель p, поскольку в этом выражении как бы неявно присутствуют скобки: ++(p->len). Чтобы изменить порядок выполнения операций, нужны явные скобки. Так, в (++p)->len, прежде чем взять значение len, программа продвинет указатель p. В (p++)->len указатель p увеличится после того, как будет взято значение len (в последнем случае скобки не обязательны).

По тем же правилам `*p->str` обозначает содержимое объекта, на который ссылается `str`; `*p->str++` продвинет указатель `str` после получения значения объекта, на который он указывал (как и в выражении вида `*s++`); `(*p->str)++` увеличит значение объекта, на который ссылается `str`; `*p++->str` продвинет `p` после того, как будет получено то, на что указывает `str`.

6.3. Массивы структур

Рассмотрим программу, определяющую число вхождений каждого ключевого слова в текст Си-программы. Нам нужно уметь хранить ключевые слова в виде массива стрингов и счётчики ключевых слов в виде массива целых. Один из возможных вариантов – это иметь два параллельных массива:

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

Однако именно тот факт, что они параллельны, подсказывает нам другую организацию хранения – через массив структур. Каждое ключевое слово можно описать парой характеристик

```
char *word;
int count;
```

Такие пары составляют массив. Декларация

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

описывает структуру типа `key` и определяет массив `keytab`, каждый элемент которого есть структура этого типа и которому где-то будет выделена память. Это же можно записать и по-другому:

```
struct key {
    char *word;
    int count;
};
struct key keytab[NKEYS];
```

Так как `keytab` содержит постоянный набор имён, его легче всего сделать внешним массивом и инициализировать один раз в момент определения. Инициализация структур аналогична ранее демонстрировавшимся инициализациям – за определением следует список инициализаторов, заключённый в фигурные скобки:

```
struct key {
    char *word;
    int count;
} keytab[] = {
    "auto", 0,
    "break", 0,
    /* ... */
    "while", 0
};
```

Инициализаторы задаются парами, чтобы соответствовать конфигурации структуры. Строго говоря, пару инициализаторов для каждой отдельной структуры следовало бы заключить в фигурные скобки, как, например, в

```
{ "auto", 0 },
{ "break", 0 },
...
```

Однако, когда инициализаторы – простые константы или цепочки литер, и все они имеются в наличии, во внутренних скобках нет необходимости. Число элементов массива keytab будет вычислено по количеству инициализаторов, поскольку они представлены полностью, а внутри квадратных скобок [] ничего не задано.

Программа подсчёта ключевых слов начинается с определения keytab. Программа main читает ввод, многократно обращаясь к функции getword и получая на каждом её вызове очередное слово. Каждое слово ищется в keytab. Для этого используется функция бинарного поиска, которую мы написали в гл. 3. Список ключевых слов должен быть упорядочен в алфавитном порядке.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
int getword(char *, int);
int binsearch(char *, struct key *, int);
/* подсчёт ключевых слов Си */
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: найти слово в 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;
}
```

Чуть позже мы рассмотрим функцию getword, а сейчас нам достаточно знать, что при каждом её вызове получается очередное слово, которое запоминается в массиве, заданном первым аргументом.

NKEYS – количество ключевых слов в keytab. Хотя мы могли бы подсчитать число таких слов вручную, гораздо легче и безопасней сделать это с помощью машины, особенно если список ключевых слов может быть изменён. Одно из возможных решений – поместить в конец списка инициализаторов пустой указатель (NULL) и затем перебирать в цикле элементы keytab, пока не встретится концевой элемент.

Но возможно и более простое решение. Поскольку размер массива полностью определён во время компиляции и равен произведению количества элементов массива на размер его отдельного элемента, число элементов массива можно вычислить по формуле

размер keytab / *размер* struct key

В Си имеется унарный оператор sizeof, который работает во время компиляции. Его можно применять для вычисления размера любого объекта. Выражения

sizeof *объект*

и

```
sizeof(имя типа)
```

выдают целые значения, равные размеру указанного объекта или типа в байтах. (Строго говоря, sizeof выдаёт беззнаковое целое, тип которого size_t определён в головном файле <stddef.h>.) Что касается объекта, то это может быть переменная, массив или структура. В качестве имени типа может выступать имя базового типа (int, double, ...) или имя производного типа, например, структуры или указателя.

В нашем случае, чтобы вычислить количество ключевых слов, размер массива надо поделить на размер одного элемента. Указанное вычисление используется в инструкции #define для установки значения NKEYS:

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

Этот же результат можно получить другим способом – поделить размер массива на размер какого-то его конкретного элемента:

```
#define NKEYS (sizeof keytab / sizeof keytab[0])
```

Преимущество такого рода записей в том, что их не надо корректировать при изменении типа.

Поскольку препроцессор не обращает внимания на имена типов, оператор sizeof нельзя применять в #if. Но в #define выражение препроцессором не вычисляется, так что предложенная нами запись допустима.

Теперь поговорим о функции getword. Мы написали getword в несколько более общем виде, чем требуется для нашей программы, но она от этого не стала заметно сложнее. Функция getword берет из входного потока следующее «слово». Под словом понимается цепочка букв-цифр, начинающаяся с буквы, или отдельная непробельная литера. По концу файла функция выдаёт EOF, в остальных случаях её значением является код первой литеры слова или код отдельной литеры, если она не буква.

```
/* getword: принимает следующее слово или литеру из 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 обращается к getch и ungetch, которые мы написали в гл. 4. При завершении набора букв-цифр оказывается, что getword взяла лишнюю литеру. Обращение к ungetch позволяет вернуть её назад во входной поток. В getword используются также isspace – для пропуска пробельных литер, isalpha – для идентификации букв и isalnum – для распознавания букв-цифр. Все они описаны в стандартном головном файле <ctype.h>.

Упражнение 6.1. Наша версия getword не обрабатывает должным образом знак подчёркивания, стринговые константы, комментарии и управляющие строки препроцессора. Напишите более совершенный вариант программы.

6.4. Указатели на структуры

Для иллюстрации некоторых моментов, касающихся указателей на структуры и массивов структур, перепишем программу подсчёта ключевых слов, пользуясь для получения элементов массива вместо индексов указателями.

Внешняя декларация массива keytab остаётся без изменения, а main и binsearch нужно модифицировать.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* подсчёт ключевых слов Си; версия с указателями */
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: найти слово в 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;
}
```

Некоторые детали этой программы требуют пояснений. Первое, описание функции binsearch должно отражать тот факт, что она возвращает указатель на struct key, а не целое; соответствующие изменения коснулись как прототипа функции, так и её заголовка. Если binsearch находит слово, то она выдаёт указатель на него, в противном случае она возвращает NULL. Второе, к элементам keytab доступ осуществляется в нашей программе через указатели. Это потребовало значительных изменений в binsearch. Инициализаторами для low и high теперь служат указатели на начало и на место сразу после конца массива. Вычисление положения среднего элемента с помощью формулы

```
mid = (low+high) / 2        /* НЕВЕРНО */
```

не годится, поскольку указатели нельзя складывать. Однако к ним можно применить операцию вычитания, и так как high-low есть число элементов, присваивание

```
mid = low + (high-low) / 2
```

установит в mid указатель на элемент, лежащий посередине между low и high.

Самое важное при переходе на новый вариант программы – сделать так, чтобы не генерировались неправильные указатели и не было попыток

обращений за пределы массива. Проблема в том, что и `&tab[-1]`, и `&tab[n]` находятся вне границ массива. Первый адрес определённо неверен, нельзя также осуществить доступ и по второму адресу. По правилам языка, однако, гарантируется, что адрес ячейки памяти, следующей сразу за концом массива (т.е. `&tab[n]`), в арифметике с указателями воспринимается правильно.

В главной программе мы написали

```
for (p = keytab; p < keytab + NKEYS; p++)
```

Если `p` – указатель на структуру, то при выполнении операций с `p` учитывается размер структуры. Поэтому `p++` увеличит `p` на такую величину, чтобы выйти на следующий структурный элемент массива, а проверка условия вовремя остановит цикл.

Не следует, однако, полагать, что размер структуры равен сумме размеров её членов. Вследствие выравнивания объектов разной длины в структуре могут появляться безымянные «дыры». Так, например, если переменная типа `char` занимает один байт, а `int` – четыре байта, то для структуры

```
struct {
    char c;
    int i;
};
```

может потребоваться восемь байт, а не пять. Оператор `sizeof` возвращает правильное значение.

Наконец, несколько слов относительно формата программы. Если функция возвращает значение сложного типа, как, например, в нашем случае указатель на структуру:

```
struct key *binsearch(char *word, struct key *tab, int n)
```

то имя функции «высмотреть» оказывается совсем не просто. В таких случаях иногда пользуются записью вида:

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

Какой форме отдать предпочтение – дело вкуса. Выберите ту, которая больше всего вам нравится.

6.5. Структуры со ссылками на себя

Предположим, что мы хотим решить более общую задачу – написать программу, подсчитывающую частоту встречаемости для *любых* слов входного потока. Так как список слов заранее не известен, мы не можем предварительно упорядочить его и применить бинарный поиск. Было бы неразумно пользоваться и линейным поиском каждого полученного слова, чтобы определять, встречалось оно ранее или нет – в этом случае программа работала бы слишком медленно. (Более точная оценка: время работы такой

программы пропорционально квадрату количества слов.) Как можно организовать данные, чтобы эффективно справиться со списком произвольных слов?

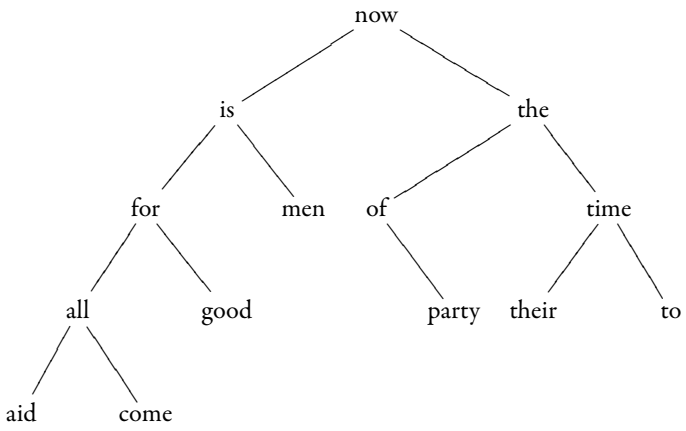
Один из способов – постоянно поддерживать упорядоченность уже полученных слов, помещая каждое новое слово в такое место, чтобы не нарушалась имеющаяся упорядоченность. Делать это передвижкой слов в линейном массиве не следует, – хотя бы потому, что указанная процедура тоже слишком долгая. Вместо этого мы воспользуемся структурой данных, называемой *бинарным деревом*.

В дереве на каждое отдельное слово предусмотрен «узел», который содержит:

```
указатель на текст слова
счётчик числа встречаемости
указатель на левый сыновний узел
указатель на правый сыновний узел
```

У каждого узла может быть один или два сына, или узел вообще может не иметь сыновей.

Узлы в дереве располагаются так, что по отношению к любому узлу левое поддерево содержит только те слова, которые лексикографически меньше, чем слово данного узла, а правое – слова, которые больше него. Вот как выглядит дерево, построенное для фразы «now is the time for all good men to come to the aid of their party» («настало время всем добрым людям помочь своей партии»), по завершении процесса, в котором для каждого нового слова в него добавлялся новый узел:



Чтобы определить, помещено ли уже в дерево вновь поступившее слово, начинают с корня, сравнивая это слово со словом из корневого узла. Если они

совпали, то ответ на вопрос – положительный. Если новое слово меньше слова из дерева, то поиск продолжается в левом поддереве, если больше, то – в правом. Если же в выбранном направлении поддерева не оказалось, то этого слова в дереве нет, а пустующая ссылка, говорящая об отсутствии поддерева, как раз то место, куда нужно «подвесить» узел с новым словом. Описанный процесс по сути рекурсивен, так как поиск в любом узле использует результат поиска в одном из своих сыновних узлов. В соответствии с этим для добавления узла и печати дерева здесь наиболее естественно применить рекурсивные функции.

Вернёмся к описанию узла, которое удобно представить в виде структуры с четырьмя компонентами:

```
struct tnode {      /* узел дерева */
    char *word;             /* указатель на текст */
    int count;              /* число вхождений */
    struct tnode *left;     /* левый сын */
    struct tnode *right;    /* правый сын */
};
```

Приведённое рекурсивное определение узла может показаться рискованным, но оно правильное. Структура не может включать саму себя, но ведь

```
struct tnode *left;
```

определяет left как указатель на tnode, а не сам tnode.

Иногда возникает потребность во взаимоссылающихся структурах: двух структурах, ссылающихся друг на друга. Приём, позволяющий справиться с этой задачей, демонстрирует следующий фрагмент:

```
struct t {
    ...
    struct s *p;     /* p указывает на s */
};
struct s {
    ...
    struct t *q;     /* q указывает на t */
};
```

Вся программа удивительно мала – правда, она использует вспомогательные программы типа getword, уже написанные нами. Главная программа читает слова с помощью getword и вставляет их в дерево посредством 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);
/* подсчёт частоты встречаемости 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 рекурсивна. Первое слово функция main помещает на верхний уровень дерева (корень дерева). Каждое вновь поступившее слово сравнивается со словом узла и «погружается» или в левое, или в правое поддерево с помощью рекурсивного обращения к addtree. Через некоторое время это слово обязательно либо совпадёт с каким-нибудь из имеющихся в дереве слов (в этом случае к счётчику будет добавлена 1), либо программа встретит пустую ссылку, что послужит сигналом для заведения нового узла и добавления его к дереву. Создание нового узла сопровождается тем, что addtree возвращает на него указатель, который вставляется в узел родителя.

```
struct tnode *talloc(void);
char *strdup(char *);
/* addtree: добавляет узел со словом w в p или ниже него */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;
    if (p == NULL) {        /* слово встречается впервые */
        p = talloc();       /* создаётся новый узел */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;         /* это слово уже встречалось */
    else if (cond < 0)      /* < корня левого поддерева */
        p->left = addtree(p->left, w);
    else                    /* > корня правого поддерева */
        p->right = addtree(p->right, w);
    return p;
}
```

Память для нового узла запрашивается с помощью программы `talloc`, которая возвращает указатель на свободное пространство, достаточное для хранения одного узла дерева, а копирование нового слова в отдельное место памяти осуществляется с помощью `strdup`. (Мы рассмотрим эти программы чуть позже.) В тот (и только в тот) момент, когда к дереву подвешивается новый узел, происходит инициализация счётчика и заполнение пустыми ссылками указателей на сыновей. Мы опустили (что неразумно) контроль ошибок, который должен выполняться при получении значений от `strdup` и `talloc`.

Функция `treeprint` печатает дерево в лексикографическом порядке; для каждого узла она печатает его левое поддерево (все слова, которые меньше слова данного узла), затем само слово и, наконец, правое поддерево (слова, которые больше слова данного узла).

```
/* treeprint: упорядоченная печать дерева p */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

Если вы не уверены, что досконально разобрались в том, как работает рекурсия, «проиграйте» действия `treeprint` на дереве, приведённом выше.

Практическое замечание: если дерево «несбалансировано» (что бывает, когда слова поступают не в случайном порядке), то время работы программы может сильно возрасти. Худший вариант, когда слова уже упорядочены; в этом случае затраты на вычисления будут такими же, как при линейном поиске. Существуют обобщения бинарного дерева, которые не страдают этим недостатком, но здесь мы их не описываем.

Прежде чем окончательно оставить этот пример, стоит сделать краткое отступление от темы и поговорить о механизме запроса памяти. Очевидно, хотелось бы иметь всего лишь одну функцию, выделяющую память, даже если эта память предназначается для разного рода объектов. Но если одна и та же функция обеспечивает память, скажем, и для указателей на `char`, и для указателей на `struct tnode`, то возникают два вопроса. Первый, как справиться с требованием большинства машин, в которых объекты определённого типа должны быть выровнены (например, целые часто должны размещаться, начиная с чётных адресов)? И второе, как описать функцию, которая вынуждена в качестве результата выдавать указатели разных типов?

Вообще говоря, требования, касающиеся выравнивания, можно легко выполнить за счёт некоторой потери памяти. Однако для этого возвращаемый указатель должен быть таким, чтобы удовлетворялись любые огра-

ничения, связанные с выравниванием. Функция alloc, описанная в гл. 5, не гарантирует нам любое конкретное выравнивание, поэтому мы будем пользоваться функцией malloc из стандартной библиотеки, которая это делает. В гл. 8 мы покажем один из способов её реализации.

Вопрос об описании типа таких функций, как malloc, является камнем преткновения в любом языке с жёсткой проверкой типов. В Си вопрос решается естественным образом: malloc объявляется как функция, которая возвращает указатель на void. Полученный указатель затем явно приводится к желаемому типу. Описания malloc и связанных с ней функций находятся в стандартном головном файле <stdlib.h>. Таким образом, функцию talloc можно записать следующим образом:

```
#include <stdlib.h>
/* talloc: создаёт tnode */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

Функция strdup просто копирует стринг, указанный в аргументе, в место, полученное с помощью malloc:

```
char *strdup(char *s)  /* дублирует 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 свободного пространства нет; strdup передаёт это значение, оставляя заботу о выходе из ошибочной ситуации той программе, которая к ней обратилась.

Память, полученную с помощью malloc, можно освободить для повторного использования, обратившись к функции free (см. гл. 7 и 8).

Упражнение 6.2. Напишите программу, которая читает текст Си-программы и печатает в алфавитном порядке все группы имён переменных, в которых совпадают первые 6 литер, но последующие в чём-то различаются. Не обрабатывайте внутренности стрингов и комментариев. Число 6 сделайте параметром, задаваемым в командной строке.

Упражнение 6.3. Напишите программу печати таблицы «перекрёстных ссылок», которая будет печатать все слова документа и указывать для каждого из них номера строк, где оно встретилось. Программа должна игнорировать «шумовые» слова типа «и», «или» и т.д.

Упражнение 6.4. Напишите программу, которая печатает весь набор различных слов, образующих входной поток, в порядке возрастания частоты их встречаемости. Перед каждым словом должно быть указано число вхождений.

6.6. Просмотр таблиц

В этом разделе, чтобы проиллюстрировать новые аспекты применения структур, мы напишем ядро пакета программ, осуществляющих вставку элементов в таблицы и их поиск внутри таблиц. Этот пакет – типичный набор программ, с помощью которых работают с таблицами имён в любом макропроцессоре или компиляторе. Рассмотрим, например, инструкцию `#define`. Когда встречается строка вида

```
#define  IN  1
```

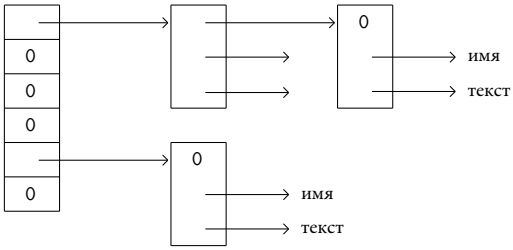
имя `IN` и замещающий его текст `1` должны запоминаться в таблице. Если затем это имя `IN` встретится в инструкции, например, в

```
state = IN;
```

оно должно быть заменено на `1`.

Существуют две программы, манипулирующие с именами и замещающими их текстами. Это `install(s, t)`, которая записывает имя `s` и замещающий его текст `t` в таблицу, где `s` и `t` – стринги, и `lookup(s)`, осуществляющая поиск `s` в таблице и возвращающая указатель на место, где имя `s` было найдено, или `NULL`, если `s` в таблице не оказалось.

Алгоритм основан на «хэшировании» (функции расстановки): поступающее имя свёртывается в неотрицательное число (хэш-код), которое затем используется в качестве индекса в массиве указателей. Каждый элемент этого массива является указателем на начало связанного ссылками списка блоков, описывающих имена с данным хэш-кодом. Если элемент массива содержит `NULL`, это значит, что среди имён не встретилось ни одного с соответствующим хэш-кодом.



Блок в списке – это структура, содержащая указатели на имя, на замещающий текст и на следующий блок в списке; значение `NULL` в указателе на следующий блок означает конец списка.

```
struct nlist {      /* элемент таблицы */
    struct nlist *next;     /* ук-ль на следующий элемент */
    char *name;             /* определяемое имя */
    char *defn;             /* замещающий текст */
};
```

А вот как записывается определение массива указателей:

```
#define HASHSIZE 101
static struct nlist *hashtab[HASHSIZE]; /* таблица ук-лей */
```

Хэш-функция, используемая в lookup и install, суммирует коды литер имени, тем самым «замешивая» их, и в качестве результата выдаёт остаток от деления полученной суммы на размер массива указателей. Это не самый лучший способ получения хэш-кода, но достаточно лаконичный и эффективный.

```
/* hash: получает хэш-код по стрингу s */
unsigned hash(char *s)
{
    unsigned hashval;
    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}
```

Беззнаковая арифметика гарантирует, что хэш-код будет неотрицательным.

Хэширование порождает стартовый индекс для массива hashtab; если соответствующий стринг в таблице есть, он может быть обнаружен только в списке блоков, на начало которого указывает элемент массива hashtab с этим индексом. Поиск осуществляется с помощью lookup. Если lookup находит элемент с заданным стрингом, то он возвращает указатель на него, если не находит, то возвращает NULL.

```
/* lookup: ищет s */
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;      /* нашли */
    return NULL;            /* не нашли */
}
```

В for-цикле функции lookup для просмотра списка используется стандартная конструкция

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
    ...
```

Функция install обращается к lookup, чтобы определить, имеется ли в наличии вставляемый стринг. Если это так, то старое определение будет заменено новым. В противном случае будет образован новый элемент. Если запрос памяти для нового элемента не может быть удовлетворён, функция install выдаёт NULL.

```
struct nlist *lookup(char *);
char *strdup(char *);

/* install: заносит (name, defn) в таблицу */
struct nlist *install(char *name, char *defn)
{
    struct nlist *np;
    unsigned hashval;
    if ((np = (lookup(name))) == NULL) {  /* не найден */
        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 имеется */
        free((void *) np->defn); /* освобождаем прежн.defn */
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}
```

Упражнение 6.5. Напишите функцию undef, удаляющую имя и определение из таблицы, организация которой поддерживается функциями lookup и install.

Упражнение 6.6. Реализуйте простую версию #define-процессора (без аргументов), которая использовала бы программы этого раздела и годилась бы для Си-программ. Вам могут помочь программы getch и ungetch.

6.7. Средство typedef

Язык Си предоставляет средство, называемое typedef, позволяющее давать новые имена типам данных. Например, декларация

```
typedef int Length
```

делает имя Length синонимом int. С этого момента тип Length можно применять в декларациях, в операторе приведения и т.д. точно так же, как тип int:

```
Length len, maxlen;
Length *lengths[];
```

Аналогично декларация

```
typedef char *String
```

делает String синонимом char *, т.е. указателем на char, и правомерным будет, например, следующее его использование:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Заметим, что объявляемый в typedef тип стоит на месте имени переменной в обычной декларации, а не сразу за словом typedef. С точки зрения синтаксиса слово typedef занимает место, где обычно располагается спецификатор класса памяти – extern, static и т.д. Имена типов записаны с заглавных букв для того, чтобы они выделялись.

Для демонстрации более сложных примеров применения typedef воспользуемся этим средством при задании узлов деревьев, с которыми мы уже встречались в данной главе.

```
typedef struct tnode *Treeptr;
typedef struct node { /* узел дерева: */
    char *word;             /* указатель на текст */
    int count;              /* число вхождений */
    Treeptr left;           /* левый сын */
    Treeptr right;          /* правый сын */
} Treenode;
```

В результате создаются два новых названия типов: Treenode (структура) и Treeptr (указатель на структуру). Теперь программу talloc можно записать в следующем виде:

```
Treeptr talloc(void)
{
    return (Treeptr) malloc(sizeof(Treenode));
}
```

Следует подчеркнуть, что декларация typedef не создаёт новый тип, она лишь сообщает новое имя уже существующего типа. Никакого нового смысла эти новые имена не несут, они декларируют переменные в точности с теми же свойствами, как если бы они были объявлены напрямую без переименования типа. Фактически typedef аналогичен #define с тем лишь отличием, что, будучи интерпретируемым компилятором, он может справиться с такой текстовой подстановкой, которая не может быть обработана препроцессором. Например,

```
typedef int (*PHI)(char *, char *);
```

определяет тип PHI как «указатель на функцию (двух аргументов типа char *), возвращающую int», который, например, в программе сортировки, описанной в гл. 5, можно использовать в таком контексте:

```
PHI strcmp, numcmp;
```

Помимо просто эстетических соображений, для привлечения typedef существуют две важные причины. Первая – параметризация программы, связанная с проблемой переносимости. Если с помощью typedef объявить типы данных, которые, возможно, являются машинно-зависимыми, то при переносе программы на другую машину потребуется внести изменения только в определения typedef. Одна из распространённых ситуаций – использование typedef-имён для варьирования целыми величинами. Для каждой конкретной машины это предполагает соответствующие установки short, int или long, которые делаются аналогично установкам стандартных типов, например, size_t и ptrdiff_t.

Вторая причина, побуждающая к применению typedef, – желание сделать более ясным текст программы. Тип, названный Treeptr (от английских слов tree – дерево и pointer – указатель) более понятен, чем тот же тип, записанный как указатель на некоторую сложную структуру.

6.8. Объединения

Объединение – это переменная, которая может содержать (в разные моменты времени) объекты различных типов и размеров. Все требования относительно размеров и выравнивания выполняет компилятор. Объединения позволяют хранить разнородные данные в одной и той же области памяти без включения в программу машинно-зависимой информации. Эти средства аналогичны вариантным записям в Паскале.

Примером использования объединений мог бы послужить сам компилятор, заведующий таблицей символов, если предположить, что константы могут иметь тип int, float или являться указателем на стринговый литерал и иметь тип char *. Значение каждой конкретной константы должно храниться в переменной соответствующего этой константе типа. Работать с таблицей символов всегда удобнее, если значения занимают одинаковую по объёму память и запоминаются в одном и том же месте независимо от своего типа. Цель введения в программу объединения – иметь переменную, которая бы на законных основаниях хранила в себе значения нескольких типов. Синтаксис объединений аналогичен синтаксису структур. Приведём пример объединения.

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

Переменная u будет достаточно большой, чтобы в ней поместилась любая переменная из указанных трёх типов; точный её размер зависит от реализации. Значение одного из этих трёх типов может быть присвоено переменной u и далее использовано в выражениях, если это правомерно, т.е. если тип взятого ею значения совпадает с типом последнего присвоенного ей значения. Выполнение этого требования в каждый текущий момент – целиком на совести программиста. В случае «рассогласованности» типов результат зависит от реализации.

Синтаксис доступа к членам объединения следующий:

имя-объединения . член

или

указатель-на-объединение -> член

т.е. в точности такой, как в структурах. Если для хранения типа текущего значения u использовать, скажем, переменную utype, то можно написать такой фрагмент программы:

```
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("неверный тип %d в utype\n", utype);
```

Объединения могут входить в структуры и массивы, и наоборот. Запись доступа к члену объединения, находящегося в структуре (как и структуры, находящейся в объединении), такая же, как и для вложенных структур. Например, в массиве структур

```
struct {
    char *name;
    int flags;
    int utype;
    union {
        int ival;
        float fval;
        char *sval;
    } u;
} symtab[NSYM];
```

на ival ссылаются следующим образом:

```
symtab[i].u.ival
```

а к первой литере стринга sval можно обратиться любым из следующих двух способов:

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

Фактически объединение – это структура, все члены которой имеют нулевое смещение относительно её базового адреса, размера, который позволяет поместиться в ней самому большому её члену, и выравнивание которой удовлетворяет всем типам объединения. Операции, применимые к структурам, годятся и для объединений, т.е. законны присваивание объединения и копирование его как единого целого, взятие адреса от объединения и доступ к отдельным его членам.

Инициализировать объединение можно только значением, имеющим тип его первого члена; таким образом, упомянутую выше переменную u можно инициализировать лишь значением типа int.

В гл. 8 (на примере программы, заведующей выделением памяти) мы покажем, как, применяя объединение, можно добиться, чтобы расположение переменной было выровнено по соответствующей границе в памяти.

6.9. Поля битов

При дефиците памяти может возникнуть необходимость запаковать несколько объектов в одно слово машины. Одна из обычных ситуаций, встречающаяся в задачах обработки таблиц символов для компиляторов, – это объединение групп однобитовых флажков. Форматы некоторых данных могут от нас вообще не зависеть и диктоваться, например, интерфейсами с аппаратурой внешних устройств; здесь также возникает потребность адресоваться к частям слова.

Вообразим себе фрагмент компилятора, который заведует таблицей символов. Каждый идентификатор программы имеет некоторую связанную с ним информацию, которая сообщает, например, представляет ли он собой ключевое слово и к какому классу, если это переменная, она принадлежит: внешняя и/или статическая и т.д. Самый компактный способ кодирования такой информации – расположить однобитовые флажки в одном слове типа char или int.

Один из распространённых приёмов работы с битами основан на определении набора «масок», соответствующих позициям этих битов, как, например, в

```
#define KEYWORD  01        /* ключевое слово */
#define EXTERNAL 02        /* внешний */
#define STATIC   04        /* статический */
```

или в

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

Числа должны быть степенями двойки. Тогда доступ к битам становится делом «побитовых операций», описанных в гл. 2 (сдвиг, маскирование, взятие дополнения).

Некоторые виды записи выражений встречаются довольно часто. Так,

```
flags |= EXTERNAL | STATIC;
```

устанавливает 1 в соответствующих битах переменной `flags`,

```
flags &= ~(EXTERNAL | STATIC);
```

обнуляет их, а

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

оценивает условие как истинное, если оба бита нулевые.

Хотя научиться писать такого рода выражения не составляет труда, вместо побитовых логических операций можно пользоваться предоставляемым Си другим способом прямого определения и доступа к полям внутри слова. *Поле-битов* (или для краткости просто *поле*) – это некоторое множество битов, лежащих рядом внутри одной, зависящей от реализации, единице памяти, которую мы будем называть «словом». Синтаксис определения полей и доступа к ним базируется на синтаксисе структур. Например, строки `#define`, фигурировавшие выше при задании таблицы символов, можно заменить на определение трёх полей:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern  : 1;
    unsigned int is_static  : 1;
} flags;
```

Эта запись определяет переменную `flags`, которая содержит три однобитовых поля. Число, следующее за двоеточием, задаёт ширину поля. Поля декларированы как `unsigned int`, чтобы они воспринимались как беззнаковые величины.

На отдельные поля ссылаются так же, как и на члены обычных структур: `flags.is_keyword`, `flags.is_extern`, и т.д. Поля «ведут себя» как малые целые и могут участвовать в арифметических выражениях точно так же, как и другие целые. Таким образом, предыдущие примеры можно написать более естественным образом:

```
flags.is_extern = flags.is_static = 1;
```

устанавливает 1 в соответствующие биты;

```
flags.is_extern = flags.is_static = 0;
```

их обнуляет, а

```
if (flags.is_extern == 0 && flags.is_static == 0)
```

проверяет их.

Почти все технические детали, касающиеся полей, в частности, может ли поле перейти границу слова, зависят от реализации. Поля могут не иметь имени; с помощью безымянного поля (задаваемого только двоеточием и шириной) организуется пропуск нужного количества разрядов. Особая ширина, равная нулю, используется, когда требуется выйти на границу следующего слова.

На одних машинах поля размещаются слева направо, на других – справа налево. Это значит, что при всей полезности работы с ними, если формат данных, с которыми мы имеем дело, дан нам свыше, то необходимо самым тщательным образом исследовать порядок расположения полей; программы, зависящие от такого рода вещей, не переносимы. Поля можно определять только с типом `int`, а для того, чтобы обеспечить переносимость, явно указывая `signed` или `unsigned`. Они не могут быть массивами и не имеют адресов, и, следовательно, оператор `&` к ним не применим.

Глава 7

Ввод-вывод

Возможности ввода-вывода не являются частью самого языка Си, поэтому мы подробно и не рассматривали их до сих пор. Между тем реальные программы взаимодействуют со своим окружением гораздо более сложным способом, чем те, которые были затронуты ранее. В этой главе мы опишем стандартную библиотеку, набор функций, обеспечивающих ввод-вывод, работу со стрингами, управление памятью, стандартные математические функции и разного рода сервисные Си-программы. Но особое внимание уделим вводу-выводу.

Библиотечные функции ввода-вывода точно определяются стандартом ANSI, так что они совместимы на любых установках, где поддерживается Си. Программы, которые в своём взаимодействии с системным окружением не выходят за рамки возможностей стандартной библиотеки, можно без изменений переносить с одной машины на другую.

Свойства библиотечных функций специфицированы в более чем дюжине головных файлов; вам уже встречались некоторые из них, в том числе `<stdio.h>`, `<string.h>` и `<ctype.h>`. Мы не рассматриваем здесь всю библиотеку, так как нас больше интересует написание Си-программ, чем использование библиотечных функций. Стандартная библиотека подробно описана в приложении B.

7.1. Стандартный ввод-вывод

Как уже говорилось в гл. 1, библиотечные функции реализуют простую модель текстового ввода-вывода. Текстовый поток состоит из последовательности строк; каждая строка заканчивается литерой новая-строка. Если система в чём-то не следует принятой модели, библиотека сделает так, чтобы казалось, что эта модель удовлетворяется полностью. Например, пара

литер – возврат-каретки и перевод-строки – при вводе могла бы быть преобразована в одну литеру новая-строка, а при выводе выполнялось бы обратное преобразование.

Простейший механизм ввода – это чтение одной литеры из *стандартного ввода* (обычно с клавиатуры) функцией getchar:

```
int getchar(void)
```

В качестве результата каждого своего вызова функция getchar возвращает следующую литеру ввода или, если обнаружен конец файла, EOF. Именованная константа EOF (аббревиатура от end of file – конец файла) определена в <stdio.h>. Обычно значение EOF равно –1, но, чтобы не зависеть от конкретного значения этой константы, ссылаться на неё следует по имени (EOF).

Во многих системах клавиатуру можно заменить файлом, перенаправив ввод при помощи значка <. Так, если программа prog использует getchar, то командная строка

```
prog <infile
```

предпишет программе prog читать литеры из infile, а не с клавиатуры. Переключение ввода делается так, что сама программа prog не замечает подмены; в частности, стринг "<infile" не будет включён в аргументы командной строки argv. Переключение ввода будет также незаметным, если ввод исходит от другой программы и передаётся через «трубопроводный» механизм. В некоторых системах командная строка

```
otherprog | prog
```

приведёт к тому, что две программы, otherprog и prog, соединятся напрямую, т.е. стандартный вывод otherprog станет стандартным вводом для prog.

Функция

```
int putchar(int)
```

используется для вывода: putchar(c) отправляет литеру c в *стандартный вывод*, под которым по умолчанию подразумевается экран. Функция putchar в качестве результата возвращает посланную литеру или, в случае ошибки, EOF. То же и в отношении вывода: при помощи записи вида >*имя-файла* вывод можно перенаправить в файл. Например, если prog использует для вывода функцию putchar, то

```
prog >outfile
```

будет направлять стандартный вывод не на экран, а в outfile. А командная строка

```
prog | anotherprog
```

соединит стандартный вывод программы prog со стандартным вводом программы anotherprog.

Вывод, осуществляемый функцией printf, также отправляется в стандартный выходной поток. Вызовы putchar и printf могут как угодно чередоваться, при этом вывод будет формироваться в той последовательности, в которой происходили вызовы этих функций.

Любой исходный Си-файл, использующий хотя бы одну функцию библиотеки ввода-вывода, должен содержать в себе строку

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

причём она должна быть расположена до первого обращения к вводу-выводу. Если имя головного файла заключено в угловые скобки < и >, это значит, что поиск головного файла ведётся в стандартном месте (например, в системе UNIX это обычно директория /usr/include).

Многие программы читают только из одного входного потока и пишут только в один выходной поток. Для организации ввода-вывода таким программам вполне хватит функций getchar, putchar и printf, а для начального обучения ознакомления с этими функциями уж точно достаточно. В частности, перечисленных функций достаточно, когда требуется вывод одной программы соединить с вводом следующей. В качестве примера рассмотрим программу lower, переводящую свой ввод на нижний регистр:

```
#include <stdio.h>
#include <ctype.h>
main()  /* lower: переводит ввод на нижний регистр */
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

Функция tolower определена в <ctype.h>. Она переводит буквы верхнего регистра в буквы нижнего регистра, а остальные литеры возвращает без изменений. Как мы уже упоминали, «функции» типа getchar и putchar из библиотеки <stdio.h> и функция tolower из библиотеки <ctype.h> часто реализуют в виде макросов, чтобы исключить накладные расходы от вызова функции на каждую отдельную литеру. В разд. 8.5 мы покажем, как это делается. Независимо от того, как на той или иной машине реализованы функции библиотеки <ctype.h>, использующие их программы могут ничего не знать о кодировке литер.

Упражнение 7.1. Напишите программу, осуществляющую перевод ввода с верхнего регистра на нижний или с нижнего на верхний в зависимости от имени, по которому она вызывается и текст которого находится в argv[0].

7.2. Форматный вывод (printf)

Функция printf переводит внутренние значения в текст.

```
int printf(char *format, arg1, arg2, ...)
```

В предыдущих главах мы использовали printf неформально. Здесь мы покажем наиболее типичные случаи применения этой функции; полное её описание дано в приложении B.

Функция printf преобразует, форматирует и печатает свои аргументы в стандартном выводе под управлением формата. Возвращает она количество напечатанных литер.

Форматный стринг содержит два вида объектов: обычные литеры, которые впрямую копируются в выходной поток, и спецификации преобразования, каждая из которых вызывает преобразование и печать очередного аргумента printf. Любая спецификация преобразования начинается знаком % и заканчивается литерой-спецификатором. Между % и литерой-спецификатором могут быть расположены (в указанном ниже порядке) следующие элементы:

- Знак минус, предписывающий «прижать» преобразованный аргумент к левому краю поля.
- Число, специфицирующее минимальную ширину поля. Преобразованный аргумент будет занимать поле по крайней мере указанной ширины. При необходимости лишние позиции слева (или справа при левостороннем расположении) будут заполнены пробелами.
- Точка, отделяющая ширину поля от величины, устанавливающей точность.
- Число (точность), специфицирующее максимальное количество печатаемых литер стринга, или количество цифр после десятичной точки – для плавающего значения, или минимальное количество цифр – для целого.
- Буква h, если печатаемое целое должно рассматриваться как short, или l (латинская буква эль), если целое должно рассматриваться как long.

Литеры-спецификаторы перечислены в табл. 7.1. Если за % не помещена литера-спецификатор, поведение функции printf будет не определено.

Ширину и точность можно специфицировать при помощи *; значение ширины (или точности) в этом случае берётся из следующего аргумента (который должен быть типа int). Например, чтобы напечатать не более max литер из стринга s, годится следующая запись:

```
printf("%.*s", max, s);
```

ТАБЛИЦА 7.1. ОСНОВНЫЕ ПРЕОБРАЗОВАНИЯ PRINTF

ЛИТЕРА	ТИП АРГУМЕНТА; ВИД ПЕЧАТИ
`d, i`	`int`; десятичное целое.
`o`	`int`; беззнаковое восьмеричное (octal) целое (без ведущего нуля).
`x, X`	`int`; беззнаковое шестнадцатеричное целое (без ведущих `0x` и `0X`), для $10 \ldots 15$ используются `abcdef` или `ABCDEF`.
`u`	`int`; беззнаковое десятичное целое.
`c`	`int`; одиночная литера.
`s`	`char *`; печатает литеры, расположенные до знака `\0`, или в количестве, заданном точностью.
`f`	`double`; `[-]`*`m.dddddd`*, где количество цифр *d* задаётся точностью (по умолчанию равно 6).
`e, E`	`double`; `[-]`*`m.ddddde±xx`* или `[-]`*`m.ddddd`*`E±`*`xx`*, где количество цифр *d* задаётся точностью (по умолчанию равно 6).
`g, G`	`double`; использует `%e` или `%E`, если экспонента меньше, чем -4, или больше или равна точности; в противном случае использует `%f`. «Хвостовые» нули и «хвостовая» десятичная точка не печатаются.
`p`	`void *`; указатель (представление зависит от реализации).
`%`	аргумент не преобразуется; печатается знак `%`.

Бóльшая часть форматных преобразований была продемонстрирована в предыдущих главах. Исключение составляет задание точности для стрингов. Далее приводится перечень спецификаций и показывается их влияние на печать стринга `"hello, world"`, состоящего из 12 литер. Поле специально обрамлено двоеточиями, чтобы была видна его протяжённость.

```
:%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     :
```

Предостережение: функция `printf` использует свой первый аргумент, чтобы определить, сколько ещё ожидается аргументов и какого они будут типа. Вы не получите правильного результата, если аргументов будет не хватать или они будут принадлежать не тому типу. Вы должны также понимать разницу в следующих двух обращениях:

```
printf(s);          /* НЕВЕРНО, если в s есть % */
printf("%s", s);    /* ВЕРНО всегда */
```

Функция sprintf выполняет те же преобразования, что и printf, но вывод запоминает в стринге

```
int sprintf(char *string, char *format, arg1, arg2, ...)
```

Эта функция форматирует arg_1, arg_2 и т.д. в соответствии с информацией, заданной аргументом format так, как мы ранее описывали, но результат помещает не в стандартный вывод, а в string. Заметим, что string должен быть достаточно большим, чтобы в нём поместился результат.

Упражнение 7.2. Напишите программу, которая будет печатать разумным способом любой ввод. Как минимум она должна уметь печатать нерафические литеры в восьмеричном или шестнадцатеричном виде (в форме, принятой на вашей машине), обрывая длинные текстовые строки.

7.3. Списки аргументов переменной длины

Этот раздел содержит реализацию минимальной версии printf. Приводится она для того, чтобы показать, как надо писать функции со списками аргументов переменной длины, причём такие, которые были бы переносимы. Поскольку нас главным образом интересует обработка аргументов, функцию minprintf напишем таким образом, что она в основном будет работать со стрингом, задающим формат, и аргументами; что же касается форматных преобразований, то они будут осуществляться при помощи стандартного printf.

Декларация стандартной функции printf выглядит так:

```
int printf(char *fmt, ...)
```

Многоточие в декларации означает, что число и типы аргументов могут изменяться. Знак многоточие может стоять только в конце списка аргументов. Наша функция minprintf декларируется как

```
void minprintf(char *fmt, ...)
```

поскольку она не будет выдавать число литер, как это делает printf.

Вся сложность в том, каким образом minprintf будет продвигаться вдоль списка аргументов, – ведь у этого списка нет даже имени. Стандартный головной файл <stdarg.h> содержит набор макроопределений, которые определяют, как шагать по списку аргументов. Наполнение этого головного файла может изменяться от машины к машине, но представленный им интерфейс везде одинаков.

Тип va_list служит для описания переменной, которая будет по очереди ссылаться на каждый из аргументов; в minprintf эта переменная имеет

имя ap (от «argument pointer» – указатель на аргумент). Макрос va_start инициализирует переменную ap, чтобы она указывала на первый безымянный аргумент. К va_start нужно обратиться до первого использования ap. Среди аргументов по крайней мере один должен быть именованным; от последнего именованного аргумента этот макрос «отталкивается» при начальной установке.

Макрос va_arg на каждом своём вызове выдаёт очередной аргумент, а ap передвигает на следующий; по имени типа он определяет тип возвращаемого значения и размер шага для выхода на следующий аргумент. Наконец, макрос va_end делает очистку всего, что необходимо. К va_end следует обратиться перед самым выходом из функции.

Перечисленные средства образуют основу нашей упрощённой версии printf.

```
#include <stdarg.h>
/* minprintf: минимальный printf с переменным числом арг. */
void minprintf(char *fmt, ...)
{
    va_list ap; /* указывает на очередн. безымянный арг. */
    char *p, *sval;
    int ival;
    double dval;
    va_start(ap,fmt); /* ap указ-ет на 1-й безымянный арг. */
    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(*p);
            break;
```

```
            }
        }
        va_end(ap);  /* очистка, когда всё сделано */
    }
```

Упражнение 7.3. Дополните `minprintf` другими возможностями `printf`.

7.4. Форматный ввод (scanf)

Функция `scanf`, обеспечивающая ввод, является аналогом `printf`; она выполняет многие из упоминавшихся преобразований, но в противоположном направлении. Её декларация имеет следующий вид:

```
int scanf(char *format, ...)
```

Функция `scanf` читает литеры из стандартного входного потока, интерпретирует их согласно спецификациям стринга `format` и рассылает результаты в свои остальные аргументы. Аргумент-формат мы опишем позже; другие аргументы, *каждый из которых должен быть указателем*, определяют, где будут запоминаться должным образом преобразованные данные. Как и для `printf`, в этом разделе даётся сводка наиболее полезных, но отнюдь не всех возможностей данной функции.

Функция `scanf` прекращает работу, когда оказывается, что исчерпался формат или вводимая величина не соответствует управляющей спецификации. В качестве результата `scanf` возвращает количество успешно введённых элементов данных. По исчерпании файла она выдаёт `EOF`. Существенно то, что значение `EOF` не равно нулю, поскольку нуль `scanf` выдаёт, когда вводимая литера не соответствует первой спецификации форматного стринга. Каждое очередное обращение к `scanf` продолжает ввод с литеры, следующей сразу за последней обработанной.

Существует также функция `sscanf`, которая читает из стринга (а не из стандартного ввода).

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

Функция `sscanf` просматривает `string` согласно формату `format` и рассылает полученные значения в arg_1, arg_2 и т.д. Последние должны быть указателями.

Формат обычно содержит спецификации, которые используются для управления преобразованиями ввода. В него могут входить следующие элементы:

- Пробелы или табуляции, которые игнорируются.
- Обычные литеры (исключая `%`), которые, как ожидается, совпадут с очередными непробельными литерами входного потока.

ТАБЛИЦА 7.2. ОСНОВНЫЕ ПРЕОБРАЗОВАНИЯ SCANF

ЛИТЕРА	ВВОДИМЫЕ ДАННЫЕ; ТИП АРГУМЕНТА
`d`	десятичное целое; `int *`.
`i`	целое; `int *`. Целое может быть восьмеричным (с ведущим `0`) или шестнадцатеричным (с ведущими `0x` или `0X`).
`o`	восьмеричное целое (с ведущим нулём или без него); `int *`.
`u`	беззнаковое десятичное целое; `unsigned int *`.
`x`	шестнадцатеричное целое (с ведущими `0x` или `0X` или без них); `int *`.
`c`	литеры; `char *`. Следующие литеры ввода (по умолчанию одна) размещаются в указанном месте. Обычный пропуск пробельных литер подавляется; чтобы прочесть очередную литеру, отличную от пробельной, используйте `%1s`.
`s`	стринг литер (без обрамляющих кавычек); `char *`, указывающий на массив литер, достаточный для стринга и завершающей литеры `'\0'`, которая будет добавлена.
`e,f,g`	число с плавающей точкой, возможно, со знаком; обязательно присутствие либо десятичной точки, либо экспоненциальной части, а возможно, и обеих вместе; `float *`.
`%`	сам знак `%`, никакое присваивание не выполняется.

- Спецификации преобразования, каждая из которых начинается со знака `%` и завершается литерой-спецификатором типа преобразования. В промежутке между этими двумя литерами в любой спецификации могут располагаться, причём в том порядке, как они здесь указаны: знак `*` (признак подавления присваивания); число, определяющее ширину поля; буква `h`, `l` или `L`, указывающая на размер получаемого значения.

Спецификация преобразования управляет преобразованием следующего вводимого поля. Обычно результат помещается в переменную, на которую указывает соответствующий аргумент. Однако если в спецификации преобразования присутствует `*`, то поле ввода пропускается и никакое присваивание не выполняется. Поле ввода определяется как стринг без пробельных литер; оно простирается до следующей пробельной литеры или же ограничено шириной поля, если она задана. Поскольку литера новая-строка относится к пробельным литерам, это значит, что `scanf` при чтении будет переходить с одной строки на другую. (Пробельными литерами являются литеры пробела, табуляции, новой-строки, возврата-каретки, вертикальной-табуляции и перевода-страницы.)

Литера-спецификатор указывает, каким образом следует интерпретировать очередное поле ввода. Соответствующий аргумент должен быть указателем, как того требует механизм передачи параметров по значению, принятый в Си. Литеры-спецификаторы приведены в табл. 7.2.

Перед литерами-спецификаторами `d`, `i`, `o`, `u` и `x` может стоять буква `h`, указывающая на то, что соответствующий аргумент должен иметь тип `short *` (а не `int *`), или `l` (латинская эль), указывающая на тип `long *`. Аналогично, перед литерами-спецификаторами `e`, `f` и `g` может стоять `l`, указывающая, что тип аргумента – `double *` (а не `float *`).

Чтобы построить первый пример, обратимся к программе калькулятора из гл. 4, в которой организуем ввод с помощью функции `scanf`:

```
#include <stdio.h>
main()      /* программа-калькулятор */
{
    double sum, v;
    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Предположим, что нам нужно прочитать строки ввода, содержащие данные вида

```
25 дек 1988
```

Обращение к `scanf` выглядит следующим образом:

```
int day, year;          /* день, год */
char monthname[20];     /* название месяца */

scanf("%d %s %d", &day, monthname, &year);
```

Знак `&` перед `monthname` не нужен, так как имя массива есть указатель.

В стринге формата могут присутствовать литеры, не участвующие ни в одной из спецификаций; это значит, что эти литеры должны появиться на вводе. Так, мы могли бы читать даты вида `mm/dd/yy` при помощи следующего обращения к `scanf`:

```
int day, month, year;         /* день, месяц, год */
scanf("%d/%d/%d", &day, &month, &year);
```

В своём формате функция `scanf` игнорирует пробелы и табуляции. Кроме того, при поиске следующей порции ввода она пропускает во входном потоке все пробельные литеры (пробелы, табуляции, новые-строки и т.д.). Воспринимать входной поток, не имеющий фиксированного формата, часто оказывается удобнее, если вводить всю строку целиком и для каждого отдельного случая подбирать подходящий вариант `sscanf`. Предположим,

например, что нам нужно читать строки с датами, записанными в любой из приведённых выше форм. Тогда мы могли бы написать:

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year)==3)
        printf("верно: %s\n", line);  /* типа 25 дек 1988 */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year)==3)
        printf("верно: %s\n", line);  /* типа mm/dd/yy */
    else
        printf("неверно: %s\n",line); /* неверн. форма даты */
}
```

Обращения к scanf могут перемежаться с вызовами других функций ввода. Любая функция ввода, вызванная после scanf, продолжит чтение с первой ещё непрочитанной литеры.

В завершение ещё раз напомним, что аргументы scanf и sscanf *должны* быть указателями.

Одна из самых распространённых ошибок состоит в том, что вместо того, чтобы написать

```
scanf("%d", &n);
```

пишут

```
scanf("%d", n);
```

Компилятор о подобной ошибке ничего не сообщает.

Упражнение 7.4. Напишите свою версию scanf по аналогии с minprintf из предыдущего раздела.

Упражнение 7.5. Перепишите основанную на постфиксной записи программу калькулятора из гл. 4 таким образом, чтобы для ввода и преобразования чисел она использовала scanf и/или sscanf.

7.5. Доступ к файлам

Во всех предыдущих примерах мы имели дело со стандартным вводом и стандартным выводом, которые для программы автоматически предопределены операционной системой конкретной машины.

Следующий шаг – научиться писать программы, которые имели бы доступ к файлам, заранее *не* подсоединённым к программам. Одна из программ, в которой возникает такая необходимость – это программа cat, объединяющая несколько именованных файлов и направляющая результат в стандартный вывод. Функция cat часто применяется для выдачи файлов на экран, а также как универсальный «коллектор» файловой информации тех программ, которые не имеют возможности обратиться к файлу по имени. Например, команда

```
cat x.c y.c
```

направит в стандартный вывод содержимое файлов `x.c` и `y.c` (и ничего более).

Возникает вопрос: что надо сделать, чтобы именованные файлы можно было читать; иначе говоря, как связать внешние имена, придуманные пользователем, с инструкциями чтения данных?

На этот счёт имеются простые правила. Для того чтобы можно было читать из файла или писать в файл, он должен быть предварительно *открыт* при помощи библиотечной функции `fopen`. Функция `fopen` получает внешнее имя типа `x.c` или `y.c`, после чего осуществляет некоторые организационные действия и «переговоры» с операционной системой (технические детали которых здесь не рассматриваются) и возвращает указатель, используемый в дальнейшем для доступа к файлу.

Этот указатель, называемый *указателем файла*, ссылается на структуру, содержащую информацию о файле (адрес буфера, положение текущей литеры в буфере, открыт файл на чтение или на запись, были ли ошибки при работе с файлом и встретился ли конец файла). Пользователю не нужно знать подробности, поскольку определения, полученные из `<stdio.h>`, включают описание такой структуры, называемой `FILE`. Единственное, что требуется для определения указателя файла, – это задать декларации такого, например, вида:

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

Из этой записи следует, что `fp` есть указатель на `FILE`, а `fopen` возвращает указатель на `FILE`. Заметим, что `FILE` есть имя типа, наподобие `int`, а не тег структуры. Оно определено при помощи `typedef`. (Детали того, как можно реализовать `fopen` в системе UNIX, приводятся в разд. 8.5.)

Обращение к `fopen` в программе может выглядеть следующим образом:

```
fp = fopen(name, mode);
```

Первый аргумент – стринг, содержащий имя файла. Второй аргумент несёт информацию о *режиме*. Это тоже стринг: в нём указывается, каким образом пользователь намерен использовать файл. Возможны следующие режимы: чтение (read – `"r"`), запись (write – `"w"`) и добавление (append – `"a"`), т.е. запись информации в конец уже существующего файла. В некоторых системах различаются текстовые и бинарные файлы; в случае последних в стринг режима необходимо добавить букву `"b"` (binary – бинарный).

Тот факт, что некий файл, которого раньше не было, открывается на запись или добавление, означает, что он создаётся (если такая процедура физически возможна). Открытие уже существующего файла на запись приводит к выбрасыванию его старого содержимого, в то время как при открытии файла на добавление его старое содержимое сохраняется. Попытка читать несуществующий файл является ошибкой. Могут иметь место и другие

ошибки; например, ошибкой считается попытка чтения файла, который по статусу запрещено читать. При наличии любой ошибки fopen возвращает NULL. (Возможна более точная идентификация ошибки; детальная информация по этому поводу приводится в конце разд. B.1 приложения B.)

Следующее, что нам необходимо знать, – это как читать из файла или писать в файл, коль скоро он открыт. Существует несколько способов сделать это, из которых самый простой состоит в том, чтобы воспользоваться функциями getc и putc. Функция getc возвращает следующую литеру из файла; ей необходимо сообщить указатель файла, чтобы она знала откуда брать литеру.

```
int getc(FILE *fp)
```

Функция getc возвращает следующую литеру из потока, на который ссылаются при помощи *fp; в случае исчерпания файла или ошибки она возвращает EOF.

```
int putc(int c, FILE *fp)
```

Функция putc пишет литеру c в файл fp и возвращает записанную литеру или EOF, в случае ошибки. Аналогично getchar и putchar, программы getc и putc могут быть реализованы в виде макросов, а не функций.

При запуске Си-программы операционная система всегда открывает три файла и обеспечивает три файловые ссылки на них. Этими файлами являются: стандартный ввод, стандартный вывод и стандартный файл ошибок; соответствующие им указатели называются stdin, stdout и stderr; они описаны в <stdio.h>. Обычно stdin соотнесён с клавиатурой, а stdout и stderr – с экраном. Однако stdin и stdout можно связать с файлами или, используя механизм «трубопровода», соединить напрямую с другими программами, как это описывалось в разд. 7.1.

При помощи getc, putc, stdin, stdout функции getchar и putchar теперь можно определить следующим образом:

```
#define getchar()  getc(stdin)
#define putchar(c) putc((c), stdout)
```

Форматный ввод-вывод файлов можно построить на функциях fscanf и fprintf. Они идентичны scanf и printf с той лишь разницей, что первым их аргументом является указатель, ссылающийся на файл, для которого осуществляется ввод-вывод, формат же указывается вторым аргументом.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

Вот теперь мы располагаем теми сведениями, которые достаточны для написания программы cat, предназначенной для конкатенации (последовательного соединения) файлов. Предлагаемая версия функции cat, как оказалось, удобна для многих программ. Если в командной строке присутствуют

аргументы, они рассматриваются как имена последовательно обрабатываемых файлов. Если аргументов нет, то обработке подвергается стандартный ввод.

```
#include <stdio.h>

/* cat: конкатенация файлов, версия 1 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    if (argc == 1)    /* нет арг-тов; копир-ся станд. ввод */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                printf("cat: не могу открыть файл %s\n",
                   *argv);
                return 1;
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    return 0;
}

/* filecopy: копирует файл 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 представляют собой объекты типа FILE *. Это константы, а *не* переменные, следовательно, им нельзя ничего присваивать.

Функция

```
int fclose(FILE *fp)
```

– обратная по отношению к fopen; она разрывает связь между файловым указателем и внешним именем (которая раньше была установлена при помощи fopen), освобождая тем самым этот указатель для других файлов. Так как в большинстве операционных систем количество одновременно открытых одной программой файлов ограничено, то файловые указатели, если они

больше не нужны, лучше освобождать, как это и делается в программе cat. Есть ещё одна причина применить fclose к файлу вывода, – это необходимость «опорожнить» буфер, в котором putc накопила, предназначенные для вывода данные. При нормальном завершении работы программы для каждого открытого файла fclose вызывается автоматически. (Вы можете закрыть stdin и stdout, если они вам не нужны. Воспользовавшись библиотечной функцией freopen их можно создать заново.)

7.6. Управление ошибками (stderr и exit)

Обработку ошибок в cat нельзя признать идеальной. Беда в том, что если файл по какой-либо причине недоступен, сообщение об этом мы получим по окончании конкатенируемого вывода. Это нас устроило бы, если бы вывод отправлялся только на экран, а не в файл или другой программе, напрямую по «трубопроводу».

Чтобы лучше справиться с этой проблемой, программе помимо стандартного вывода stdout придаётся ещё один выходной поток, называемый stderr. Вывод в stderr обычно отправляется на экран, даже если вывод stdout перенаправлен в другое место.

Перепишем cat так, чтобы сообщения об ошибках отправлялись в stderr.

```
#include <stdio.h>
/* cat: конкатенация файлов, версия 2 */

main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog = argv[0];   /* имя программы */
    if (argc == 1)    /* нет арг-тов; копир-ся станд. ввод */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
               fprintf(stderr, "%s: не могу откр.файл %s\n",
                    prog, *argv);
               exit(1);
            } else {
               filecopy(fp, stdout);
               fclose(fp);
            }
    if (ferror(stdout)) {
```

```
            fprintf(stderr, "%s: ошибка записи в stdout\n", prog);
            exit(2);
        }
        exit(0);
    }
```

Программа сигнализирует об ошибках двумя способами. Первый – сообщение об ошибке при помощи `fprintf` посылается в `stderr` с тем, чтобы оно попало на экран, а не оказалось в «трубопроводе» или в другом файле вывода. Имя программы, хранящееся в `argv[0]`, мы включили в сообщение, чтобы в случаях, когда данная программа работает совместно с другими, был ясен источник ошибки.

Второй способ указать на ошибку – обратиться к библиотечной функции `exit`, завершающей работу программы. Аргумент функции `exit` доступен некоторому процессу, вызвавшему данный процесс. А следовательно, успешное или ошибочное завершение программы можно проконтролировать с помощью некоей программы, которая рассматривает эту программу в качестве подчинённого процесса. По общей договорённости возврат нуля сигнализирует о том, что работа прошла нормально, в то время как ненулевые значения обычно говорят об ошибках. Чтобы опустошить буфера, накопившие информацию для всех открытых файлов вывода, функция `exit` вызывает `fclose`.

Инструкция главной программы `return` *выр* эквивалентна обращению к функции `exit(`*выр*`)`. Последняя запись (при помощи `exit`) имеет то преимущество, что она пригодна для выхода и из других функций, и, кроме того, её легко обнаружить при помощи программы контекстного поиска, похожей на ту, которую мы рассматривали в гл. 5.

Функция `ferror` выдаёт ненулевое значение, если в файле `fp` была обнаружена ошибка.

```
int ferror(FILE *fp)
```

Хотя при выводе редко возникают ошибки, всё же они встречаются (например, оказался переполненным диск); поэтому в программах широкого пользования они должны тщательно контролироваться.

Функция `feof(FILE *)` аналогична функции `ferror`; она возвращает ненулевое значение, если встретился конец указанного в аргументе файла.

```
int feof(FILE *fp)
```

В наших небольших иллюстративных программах мы не заботились о выдаче статуса выхода, т.е. выдаче некоторого числа, характеризующего состояние программы в момент завершения: работа закончилась нормально или прервана из-за ошибки? Если работа прервана в результате ошибки, то какой? Любая серьёзная программа должна выдавать статус выхода.

7.7. Ввод-вывод строк

В стандартной библиотеке имеется программа ввода fgets, аналогичная программе getline, которой мы пользовались в предыдущих главах.

```
char *fgets(char *line, int maxline, FILE *fp)
```

Функция fgets читает следующую строку ввода (включая и литеру новая-строка) из файла fp в массив литер line, причём она может прочитать не более maxline-1 литер. Переписанная строка дополняется литерой '\0'. Обычно fgets возвращает line, а по исчерпании файла или в случае ошибки – NULL. (Наша getline возвращала длину строки, которой мы потом пользовались, и нуль по концу файла.)

Функция вывода fputs пишет стринг (который может и не заканчиваться литерой новая-строка) в файл.

```
int fputs(char *line, FILE *fp)
```

Эта функция возвращает EOF, если возникла ошибка, и нуль в противном случае.

Библиотечные функции gets и puts подобны функциям fgets и fputs. Отличаются они тем, что оперируют только стандартными файлами stdin и stdout, и, кроме того, gets выбрасывает последнюю литеру '\n', а puts её добавляет.

Чтобы показать, что ничего особенного в функциях типа fgets и fputs нет, мы приводим их здесь в том виде, в каком они существуют в стандартной библиотеке на нашей системе.

```
/* fgets: получает не более n литер 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: посылает стринг s в файл iop */
int fputs(char *s, FILE *iop)
{
    int c;
    while (c = *s++)
```

```
            putc(c, iop);
        return ferror(iop) ? EOF : 0;
    }
```

Упражнение 7.6. Напишите программу, сравнивающую два файла и печатающую первую строку, в которой они различаются.

Упражнение 7.7. Модифицируйте программу поиска по образцу из гл. 5 таким образом, чтобы она брала текст из множества именованных файлов, а если имён файлов в аргументах нет, то – из стандартного ввода. Следует ли печатать имя файла, в котором найдена подходящая строка?

Упражнение 7.8. Напишите программу, печатающую несколько файлов. Каждый файл должен начинаться с новой страницы, предваряться заголовком и иметь свою нумерацию страниц.

7.8. Другие библиотечные функции

В стандартной библиотеке представлен широкий спектр различных функций. Настоящий раздел содержит краткий обзор наиболее полезных из них. Более подробно эти и другие функции описаны в приложении B.

7.8.1. Операции со стрингами

Мы уже упоминали функции `strlen`, `strcpy`, `strcat` и `strcmp`, описание которых даны в `<string.h>`. Далее, до конца подраздела, предполагается, что `s` и `t` имеют тип `char *`, `c` и `n` – тип `int`.

`strcat(s,t)`	конкатенирует `t` в конец `s`
`strncat(s,t,n)`	конкатенирует `n` литер `t` в конец `s`
`strcmp(s,t)`	возвращает отрицательное число, нуль или положительное число для `s < t`, `s == t` или `s > t` соответственно
`strncmp(s,t,n)`	делает то же, что и `strcmp`, но количество сравниваемых литер не может превышать `n`
`strcpy(s,t)`	копирует `t` в `s`
`strncpy(s,t,n)`	копирует не более `n` литер `t` в `s`
`strlen(s)`	возвращает длину `s`
`strchr(s,c)`	возвращает указатель на первое появление литеры `c` в `s` или, если `c` нет в `s`, `NULL`
`strrchr(s,c)`	возвращает указатель на последнее появление литеры `c` в `s` или, если `c` нет в `s`, `NULL`

7.8.2. Анализ класса литер и преобразование литер

Несколько функций из библиотеки <ctype.h> выполняют проверки и преобразование литер. Далее, до конца подраздела, переменная c – это переменная типа int, которая может быть представлена значением unsigned char или EOF. Функции возвращают значения типа int.

isalpha(c)	не нуль, если c – буква; 0 в противном случае
isupper(c)	не нуль, если c – буква верхнего регистра; 0 в противном случае
islower(c)	не нуль, если c – буква нижнего регистра; 0 в противном случае
isdigit(c)	не нуль, если c – цифра; 0 в противном случае
isalnum(c)	не нуль, если isalpha(c), или isdigit(c) истинны; 0 в противном случае
isspace(c)	не нуль, если c – литера пробела, табуляции, новой-строки, возврата-каретки, перевода-страницы, вертикальной-табуляции
toupper(c)	возвращает c, приведённую к верхнему регистру
tolower(c)	возвращает c, приведённую к нижнему регистру

7.8.3. Функция ungetc

В стандартной библиотеке содержится более ограниченная версия функции ungetch по сравнению с той, которую мы написали в гл. 4. Называется она ungetc. Эта функция, имеющая прототип

```
int ungetc(int c, FILE *fp)
```

отправляет литеру c назад в файл fp и возвращает c или EOF, в случае ошибки. Для каждого файла гарантирован возврат не более одной литеры. Функцию ungetc можно использовать совместно с любой из функций ввода типа scanf, getc, getchar и т.д.

7.8.4. Исполнение команд операционной системы

Функция system(char *s) выполняет команду системы, содержащуюся в стринге s, и затем возвращается к выполнению текущей программы. Содержимое s, строго говоря, зависит от конкретной операционной системы. Рассмотрим простой пример: в системе UNIX инструкция

```
system("date");
```

вызовет программу date, которая направит дату и время в стандартный вывод. Функция возвращает зависящий от системы статус выполненной команды. В системе UNIX возвращаемый статус – это значение, переданное функцией exit.

7.8.5. Управление памятью

Функции malloc и calloc получают динамически запрашиваемые ими области памяти. Функция malloc с прототипом

```
void *malloc(size_t n)
```

возвращает указатель на n байт неинициализированной памяти или NULL, если запрос удовлетворить нельзя. Функция calloc с прототипом

```
void *calloc(size_t n, size_t size)
```

возвращает указатель на область, достаточную для хранения массива из n объектов указанного размера (size), или NULL, если запрос не удаётся удовлетворить. Выделенная память обнуляется.

Указатель, возвращаемый функциями malloc и calloc, будет выдан с учётом выравнивания, выполненного согласно указанному типу объекта. Тем не менее к нему должна быть применена операция приведения к соответствующему типу, как это сделано в следующем фрагменте программы:

```
int *ip;
ip = (int *) calloc(n, sizeof(int));
```

Функция free(p) освобождает область памяти, на которую указывает p, – указатель, первоначально полученный с помощью malloc или calloc. Никаких ограничений на порядок, в котором будет освобождаться память, нет, но ужасной ошибкой считается освобождение тех областей, которые не были получены при помощи calloc или malloc.

Нельзя также использовать те области памяти, которые уже освобождены. Следующий пример демонстрирует типичную ошибку в цикле, освобождающем элементы списка.

```
for (p = head; p != NULL; p = p->next)   /* НЕВЕРНО */
    free(p);
```

Правильным будет, если вы до освобождения сохраните то, что вам потребуется, как в следующем цикле:

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
}
```

В разд. 8.7 мы рассмотрим реализацию программы управления памятью типа malloc, позволяющую освобождать выделенные блоки памяти в любой последовательности.

7.8.6. Математические функции

В <math.h> описано более двадцати математических функций. Здесь же приведены наиболее употребительные. Каждая из них имеет один или два аргумента типа double и возвращает результат также типа double.

`sin(x)`	синус x, x в радианах
`cos(x)`	косинус x, x в радианах
`atan2(y,x)`	арктангенс y/x, y/x в радианах
`exp(x)`	экспоненциальная функция e^x
`log(x)`	натуральный (по основанию e) логарифм x ($x > 0$)
`log10(x)`	обычный (по основанию 10) логарифм x ($x > 0$)
`pow(x,y)`	x^y
`sqrt(x)`	корень квадратный x ($x \geqslant 0$)
`fabs(x)`	абсолютное значение x

7.8.7. Генератор случайных чисел

Функция `rand()` вычисляет последовательность псевдослучайных целых в диапазоне от нуля до значения, заданного именованной константой `RAND_MAX`, которая определена в `<stdlib.h>`. Привести случайные числа к значениям с плавающей точкой, бóльшим или равным 0 и меньшим 1, можно по формуле

```
#define frand() ((double) rand() / (RAND_MAX+1.0))
```

(Если в вашей библиотеке уже есть функция для получения случайных чисел с плавающей точкой, вполне возможно, что её статистические характеристики лучше указанной.)

Функция `srand(unsigned)` осуществляет «начальную затравку» для `rand`. Реализации `rand` и `srand`, предлагаемые стандартом и, следовательно, переносимые на различные машины, рассмотрены в разд. 2.7.

Упражнение 7.9. Реализуя функции типа `isupper`, можно либо экономить память, либо время. Напишите оба варианта функции.

Глава 8

Интерфейс с системой UNIX

Свои услуги операционная система UNIX предлагает в виде набора *системных вызовов*, которые фактически являются её внутренними функциями и к которым можно обращаться из программ пользователя. В настоящей главе описано, как в Си-программах можно применять некоторые наиболее важные вызовы. Если вы работаете в системе UNIX, то эти сведения будут вам полезны непосредственно и позволят поднять эффективность работы или получить доступ к тем возможностям, которых нет в библиотеке. Даже если вы используете Си в другой операционной системе, изучение рассмотренных здесь примеров всё равно приблизит вас к пониманию программирования на Си; аналогичные программы (отличающиеся лишь деталями) вы встретите практически в любой операционной системе. Так как библиотека Си-программ, утверждённая в качестве стандарта ANSI, в основном отражает возможности системы UNIX, предлагаемые программы помогут вам лучше понять и библиотеку.

Глава состоит из трёх основных частей, описывающих: ввод-вывод, файловую систему и организацию управления памятью. В первых двух частях предполагается некоторое знакомство читателя с внешними характеристиками системы UNIX.

В гл. 7 мы рассматривали единый для всех операционных систем интерфейс ввода-вывода. В любой конкретной системе программы стандартной библиотеки пишутся с использованием средств именно этой конкретной системы. В следующих нескольких разделах мы опишем вызовы системы UNIX по вводу-выводу и покажем, как с их помощью можно реализовать некоторые разделы стандартной библиотеки.

8.1. Дескрипторы файлов

В системе UNIX любые операции ввода-вывода выполняются посредством чтения и записи файлов, поскольку все внешние устройства, включая клавиатуру и экран, рассматриваются как объекты файловой системы. Это значит, что все связи между программой и внешними устройствами осуществляются в рамках единого однородного интерфейса.

В самом общем случае, прежде чем читать или писать, вы должны информировать систему о действиях, которые вы намереваетесь выполнять в отношении файла; эта процедура называется *открытием* файла. Если вы собираетесь писать в файл, то, возможно, его потребуется создать заново или очистить от хранимой информации. Система проверяет ваши права на эти действия (файл существует? вы имеете к нему доступ?), и, если всё в порядке, возвращает программе небольшое неотрицательное целое, называемое *дескриптором файла*. Всякий раз, когда осуществляется ввод-вывод, идентификация файла выполняется по его дескриптору, а не по имени. (Дескриптор файла аналогичен файловому указателю, используемому в стандартной библиотеке, или хэндлу в MS-DOS.) Вся информация об открытом файле хранится и обрабатывается операционной системой; программа пользователя ссылается на файл только через его дескриптор.

Ввод с клавиатуры и вывод на экран применяются настолько часто, что для удобства работы с ними предусмотрены специальные соглашения. При запуске программы командный интерпретатор (shell) открывает три файла с дескрипторами 0, 1 и 2, которые называются соответственно стандартным вводом, стандартным выводом и стандартным файлом ошибок. Если программа читает из файла 0, а пишет в файлы 1 и 2 (здесь цифры – дескрипторы файлов), то она может осуществлять ввод и вывод, не заботясь об их открытии.

Пользователь программы имеет возможность перенаправить ввод-вывод в файл или из файла при помощи значков < и >, как, например, в

```
prog <infile >outfile
```

В этом случае командный интерпретатор заменит стандартные установки дескрипторов 0 и 1 на именованные файлы. Обычно дескриптор файла 2 остаётся подсоединённым к экрану, чтобы на него шли сообщения об ошибках. Сказанное верно и для ввода-вывода, включённого в «трубопровод». Во всех случаях замену файла осуществляет командный интерпретатор, а не программа. Программа, если она ссылается на файл 0 (в случае ввода) и файлы 1 и 2 (в случае вывода), не знает, ни откуда приходит её ввод, ни куда отправляется её вывод.

8.2. Нижний уровень ввода-вывода (read и write)

Ввод-вывод основан на системных вызовах read и write, к которым Си-программа обращается с помощью функций с именами read и write. Для обеих первым аргументом является дескриптор файла. Во втором аргументе указывается массив литер программы, куда читаются или откуда пишутся данные. Третий аргумент – это количество пересылаемых байтов.

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

Обе функции возвращают число переданных байтов. При чтении количество прочитанных байтов может оказаться меньше числа, указанного в третьем аргументе. Нуль означает конец файла, а –1 сигнализирует о какой-то ошибке. При записи функция возвращает количество записанных байтов, и если это число не совпадает с требуемым, следует считать, что запись не прошла.

За один вызов можно прочитать или записать любое число байтов. Обычно это число равно или 1, что означает политерную передачу «без буферизации», или чему-нибудь вроде 1024 или 4096, соответствующих размеру физического блока внешнего устройства. Эффективнее обмениваться бóльшим числом байтов, поскольку при этом требуется меньше системных вызовов. Используя полученные сведения, мы можем написать простую программу, копирующую свой ввод на свой вывод, эквивалентную программе копирования файла, описанной в гл. 1. При помощи этой программы можно копировать откуда угодно и куда угодно, поскольку всегда существует возможность перенаправить ввод-вывод на любой файл или устройство.

```
#include "syscalls.h"
main() /* копирование ввода на вывод */
{
    char buf[BUFSIZ];
    int n;
    while ((n = read(0, buf, BUFSIZ)) > 0)
        write(1, buf, n);
    return 0;
}
```

Прототипы функций, обеспечивающие системные вызовы, мы собрали в файле syscalls.h, что позволяет нам включать его в программы этой главы. Однако имя данного файла не зафиксировано стандартом.

Параметр BUFSIZ также определён в <syscalls.h>; в каждой конкретной системе он имеет своё значение. Если размер файла не кратен BUFSIZ, то какая-то операция чтения вернёт значение меньшее, чем BUFSIZ, а следующее обращение к read даст в качестве результата нуль.

Полезно рассмотреть, как используются read и write при написании программ более высокого уровня – типа getchar, putchar и т.д. Вот, к примеру, версия программы getchar, которая осуществляет небуферизованный ввод, читая по одной литере из стандартного входного потока.

```
#include "syscalls.h"
/* getchar: небуферизованный ввод одной литеры */
int getchar(void)
{
    char c;
    return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
}
```

Переменная c должна быть описана как char, поскольку read требует указатель на char. Приведение c к unsigned char перед тем, как вернуть её в качестве результата, исключает какие-либо проблемы, связанные с размножением знака.

Вторая версия getchar осуществляет ввод большими кусками, но при каждом обращении выдаёт только одну литеру.

```
#include "syscalls.h"
/* getchar: простая версия с буферизацией */
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int n = 0;
    if (n == 0) { /* буфер пуст */
        n = read(0, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

Если приведённые версии функции getchar компилируются с включением головного файла <stdio.h> и в этом головном файле getchar определена как макрос, то нужно задать строку #undef с именем getchar.

8.3. Системные вызовы open, creat, close, unlink

В отличие от стандартных файлов ввода, вывода и ошибок, которые открыты по умолчанию, остальные файлы нужно открывать явно. Для этого есть два системных вызова: open и creat.

Функция open почти совпадает с fopen, рассмотренной в гл. 7. Разница между ними в том, что первая возвращает не файловый указатель, а дескриптор файла типа int. При любой ошибке open возвращает –1.

```
#include <fcntl.h>
int fd;
int open(char *name, int flags, int perms);
fd = open(name, flags, perms);
```

Как и в fopen, аргумент name – это стринг, содержащий имя файла. Второй аргумент, flags, имеет целый тип и специфицирует, каким образом должен быть открыт файл. Его основными значениями являются:

O_RDONLY	открыть только на чтение
O_WRONLY	открыть только на запись
O_RDWR	открыть и на чтение, и на запись

В системах System V UNIX эти константы определены в <fcntl.h>, а в версиях Berkeley (BSD) – в <sys/file.h>.

Чтобы открыть существующий файл на чтение, можно написать

```
fd = open(name, O_RDONLY, 0);
```

Далее везде, где мы пользуемся функцией open, её аргумент perms равен нулю.

Попытка открыть несуществующий файл считается ошибкой. Создание нового файла или перезапись старого обеспечивает системный вызов creat. Например,

```
int creat(char *name, int perms);
fd = creat(name, perms);
```

Функция creat возвращает дескриптор файла, если файл создан, и –1, если по каким-либо причинам файл создать не удалось. Если файл уже существует, creat «обрежет» его до нулевой длины, что равносильно выбрасыванию предыдущего содержимого данного файла; создание уже существующего файла не является ошибкой.

Если строится действительно новый файл, то creat его создаст с правами доступа, специфицированными в аргументе perms. В системе UNIX с каждым файлом ассоциированы девять бит, содержащие информацию о правах пользоваться этим файлом для чтения, записи и исполнения лицам трёх категорий: собственнику файла, определённой им группе лиц, и всем остальным. Таким образом, права доступа удобно специфицировать с помощью трёх восьмеричных цифр. Например, 0755 специфицирует чтение, запись и право исполнения собственнику файла, а также чтение и право исполнения группе и всем остальным.

Для иллюстрации приведём упрощённую версию программы cp системы UNIX, которая копирует один файл в другой. В нашей версии копируется только один файл, не позволяется во втором аргументе указывать директорий (каталог) и права доступа не копируются, а задаются константой.

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
```

```
#define PERMS 0666  /* RW для собственника, группы и ост-х */
void error(char *, ...);

/* cp: копирование f1 в f2 */
main(int argc, char *argv[])
{
    int f1, f2, n;
    char buf[BUFSIZ];
    if (argc != 3)
        error("Обращение: cp откуда куда");
    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
        error("cp: не могу открыть файл %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("cp: не могу создать файл %s, режим %03o",
            argv[2], PERMS);
    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: ошибка при записи в файл %s", argv[2]);
    return 0;
}
```

Данная программа создаёт файл вывода с фиксированными правами доступа, определяемыми кодом 0666. При помощи системного вызова stat, который будет описан в разд. 8.6, мы можем определить режим использования существующего файла и задать тот же режим для копии.

Заметим, что функция error, вызываемая с различным числом аргументов, во многом похожа на printf. Реализация error иллюстрирует, как пользоваться другими программами семейства printf. Библиотечная функция vprintf аналогична printf, с той лишь оговоркой, что переменная часть списка аргументов заменена в ней одним аргументом, инициализируемым макросом va_start. Подобным же образом соотносятся функции vfprintf с fprintf и vsprintf с sprintf.

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

/* error: печатает сообщение об ошибке и умирает */
void error(char *fmt, ...)
{
    va_list args;
    va_start(args, fmt);
    fprintf(stderr, "ошибка: ");
    vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    va_end(args);
```

```
        exit(1);
    }
```

Имеется ограничение на количество одновременно открытых в программе файлов. (Обычно их число колеблется около 20.) Поэтому любая программа, которая намеревается работать с большим количеством файлов, должна быть готова повторно использовать их дескрипторы. Функция `close(int fd)` разрывает связь между файловым дескриптором и открытым файлом и освобождает дескриптор для его применения с другим файлом. Она аналогична библиотечной функции `fclose` с тем лишь различием, что никакого выталкивания буфера не делает. Завершение программы при помощи `exit` или `return` в главной программе закрывает все открытые файлы.

Функция `unlink(char *name)` удаляет имя файла из файловой системы. Она соответствует функции `remove` стандартной библиотеки.

Упражнение 8.1. Перепишите программу `cat` из гл. 7, используя функции `read`, `write`, `open` и `close`. Замените ими соответствующие функции стандартной библиотеки. Поэкспериментируйте, чтобы сравнить быстродействие двух версий.

8.4. Случайный доступ (lseek)

Ввод-вывод обычно бывает последовательным, т.е. каждая новая операция чтения-записи обрабатывает позицию файла, следующую за обработанной в предыдущей операции (чтения-записи). При желании, однако, файл можно читать в произвольном порядке. Системный вызов `lseek` предоставляет способ передвигаться по файлу, не читая и не записывая данные. Так, функция с прототипом

```
long lseek(int fd, long offset, int origin);
```

в файле с дескриптором `fd` устанавливает текущую позицию, смещая её на величину `offset` относительно места, задаваемого значением `origin`. Значения параметра `origin` 0, 1 или 2 означают, что на величину `offset` отступают соответственно от начала, текущей позиции или конца файла. Если требуется добавить что-либо в файл (когда, например, в командном интерпретаторе shell системы UNIX ввод перенаправлен оператором >> в файл или когда в `fopen` задан аргумент `"a"`), то прежде чем что-либо записывать при помощи вызова функции

```
lseek(fd, 0L, 2);
```

необходимо найти конец файла. Чтобы вернуться назад, в начало файла, надо выполнить

```
lseek(fd, 0L, 0);
```

Следует обратить внимание на аргумент 0L: вместо 0L можно было бы написать (long) 0 или, если функция lseek должным образом декларирована, просто 0.

Благодаря lseek с файлами можно работать так, как будто это большие массивы, правда, с замедленным доступом. Например, следующая функция читает любое число байтов из любого места файла. Она возвращает число прочитанных байтов или –1, в случае ошибки.

```
#include "syscalls.h"

/* get: читает n байт из позиции pos */
int get(int fd, long pos, char *buf, int n)
{
    if (lseek(fd, pos, 0) >= 0) /* установка позиции */
        return read(fd, buf, n);
    else
        return -1;
}
```

Возвращаемое функцией lseek значение имеет тип long и является новой позицией в файле или, в случае ошибки, равно –1. Функция fseek из стандартной библиотеки аналогична lseek; от последней она отличается тем, что в случае ошибки возвращает некоторое ненулевое значение, а её первый аргумент имеет тип FILE *.

8.5. Пример. Реализация функций fopen и getc

Теперь на примере функций fopen и getc из стандартной библиотеки покажем, как описанные выше части согласуются друг с другом.

Напомним, что файлы в стандартной библиотеке описываются файловыми указателями, а не дескрипторами. Указатель файла – это указатель на структуру, содержащую информацию о файле: указатель на буфер, позволяющий читать файл большими кусками; число незанятых байтов буфера; указатель на следующую позицию в буфере; дескриптор файла; флажки, описывающие режим (чтение/запись), ошибочные состояния и т.д.

Структура данных, описывающая файл, содержится в <stdio.h>, который необходимо включать (при помощи #include) в любой исходный файл, если в нём осуществляется стандартный ввод-вывод. Этот же головной файл включён и в исходные тексты библиотеки ввода-вывода.

В следующем фрагменте, типичном для файла <stdio.h>, имена, используемые только в библиотечных функциях, начинаются с подчёркивания. Это сделано для того, чтобы они случайно не совпали с именами, фигурирующими в программе пользователя. Такое соглашение соблюдается во всех программах стандартной библиотеки.

```
#define NULL      0
#define EOF       (-1)
#define BUFSIZ    1024
#define OPEN_MAX  20   /* макс.число одновр.открытых файлов */
typedef struct _iobuf {
    int  cnt;          /* колич.оставшихся литер */
    char *ptr;         /* позиция следующей литеры */
    char *base;        /* адрес буфера */
    int  flag;         /* режим доступа */
    int fd;            /* дескриптор файла */
} FILE;
extern FILE _iob[OPEN_MAX];
#define stdin  (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
enum _flag {
    _READ  = 01,       /* файл открыт на чтение */
    _WRITE = 02,       /* файл открыт на запись */
    _UNBUF = 04,       /* файл не буферизуется */
    _EOF   = 010,      /* в данном файле встретился EOF */
    _ERR   = 020       /* в данном файле встретилась 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)
```

Макрос getc обычно уменьшает счётчик числа литер, находящихся на буфере, и возвращает литеру с последующим продвижением указателя. (Напомним, что длинные #define при помощи обратной наклонной черты можно продолжить на следующих строках.) Когда значение счётчика становится отрицательным, getc вызывает _fillbuf, чтобы снова заполнить буфер, инициализировать содержимое структуры и выдать литеру. Типы возвращаемых литер приводятся к unsigned; это гарантирует, что все они будут положительными.

Хотя в деталях ввод-вывод здесь не рассматривается, мы всё же привели полное определение putc. Сделано это, чтобы показать, что она действует

во многом так же, как и getc, вызывая функцию _flushbuf, когда буфер полон. В тексте имеются макросы, позволяющие получать доступ к флажкам ошибки и конца файла, а также к его дескриптору.

Теперь можно написать функцию fopen. Бóльшая часть инструкций fopen относится к открытию файла, соответствующему его позиционированию и установке флажковых битов, предназначенных для индикации текущего состояния. Сама fopen не отводит места для буфера; это делает _fillbuf при первом чтении файла.

```
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666  /* RW для собственника, группы и ост-х */
/* fopen: открывает файл, возвращает файловый указатель */
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 своб.место для описателя */
    if (fp >= _iob + OPEN_MAX)  /* нет места для описателя */
        return NULL;
    if (*mode == 'w')
        fd = creat(name, PERMS);
    else if (*mode == 'a') {
        if ((fd = open(name, O_RDONLY, 0)) == -1)
            fd = creat(name, PERMS);
        lseek(fd, 0L, 2);
    } else
        fd = open(name, O_RDONLY, 0);
    if (fd == -1)   /* не возможен доступ по имени name */
        return NULL;
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag = (*mode == 'r') ? _READ : _WRITE;
    return fp;
}
```

Приведённая версия fopen не реализует все режимы доступа, оговорённые стандартом; но, мы думаем, их реализация в полном объёме не намного увеличит длину программы. Наша fopen не распознает буквы «b», сигнализирующей о бинарном вводе-выводе (поскольку в системах UNIX это не имеет

смысла), и знака «+», указывающего на возможность одновременно читать и писать.

Для любого файла в момент первого обращения к нему с помощью макровызова getc счётчик cnt равен нулю. Следствием этого будет вызов _fillbuf. Если выяснится, что файл на чтение не открыт, то функция _fillbuf немедленно возвратит EOF. В противном случае она попытается запросить память для буфера (если чтение должно быть с буферизацией).

После получения области памяти для буфера _fillbuf обращается к read, чтобы его наполнить, устанавливает счётчик и указатели и возвращает первую литеру буфера. В следующих обращениях _fillbuf обнаружит, что память для буфера уже выделена.

```
#include "syscalls.h"
/* _fillbuf: запрос памяти и заполнение буфера */
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 ещё нет */
        if ((fp->base = (char *) malloc(bufsize)) == NULL)
            return EOF;     /* нельзя получить 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 невыясненным, – это каким образом организовать начало счёта. Следует так определить и инициализировать массив _iob, чтобы в нём, перед тем как программа начнёт работать, уже была информация о файлах 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 flag как части структуры показывает, что stdin открыт на чтение, stdout – на запись, а stderr – на запись без буферизации.

Упражнение 8.2. Перепишите функции fopen и _fillbuf, работая с флажками как с полями, а не при помощи явных побитовых операций. Сравните размеры и скорости двух вариантов программ.

Упражнение 8.3. Разработайте и напишите функции _flushbuf, fflush и fclose.

Упражнение 8.4. Функция стандартной библиотеки

```
int fseek(FILE *fp, long offset, int origin)
```

идентична функции lseek с теми, однако, отличиями, что fp – это файловый указатель, а не дескриптор, и возвращает она значение int, указывающее на состояние файла, а не позицию в нём. Напишите свою версию fseek. Обеспечьте, чтобы работа вашей fseek по буферизации была согласована с буферизацией, используемой другими функциями библиотеки.

8.6. Пример. Печать каталогов

При разного рода взаимодействиях с файловой системой иногда требуется получить *только* информацию о файле, а не его содержимое. Такая потребность возникает, например, в программе печати каталога файлов, работающей по типу команды ls системы UNIX. Она печатает имена файлов каталога и по желанию пользователя другую дополнительную информацию (размеры, права доступа и т.д.). Аналогичной командой в MS-DOS является dir.

Так как в системе UNIX каталог – это тоже файл, функции ls, чтобы добраться до имён файлов, нужно только его прочитать. Но, чтобы получить другую информацию о файле (например, узнать его размер), необходимо выполнить системный вызов. В других системах (в MS-DOS, например) системным вызовом приходится пользоваться даже для получения доступа к именам файлов. Наша цель – обеспечить доступ к информации по возможности системно-независимым способом несмотря на то, что реализация может быть существенно системно-зависима.

Проиллюстрируем сказанное написанием программы fsize. Функция fsize – частный случай программы ls; она печатает размеры всех файлов, перечисленных в командной строке. Если какой-либо из файлов сам является каталогом, то, чтобы получить информацию о нём, fsize обращается сама к себе. Если аргументов в командной строке нет, то обрабатывается текущий каталог.

Для начала вспомним структуру файловой системы в UNIXe. *Каталог* – это файл, содержащий список имён файлов и некоторую информацию о том, где они расположены. «Место расположения» – это индекс, обеспечивающий доступ в другую таблицу, называемую «списком узлов inode». Для каждого файла имеется свой *inode*, где собрана вся информация о файле, за исключением его имени. Каждый элемент каталога состоит из двух частей: из имени файла и номера inode.

К сожалению, формат и точное содержимое каталога не одинаковы во всех версиях системы. Поэтому, чтобы переносимую компоненту отделить от непереносимой, разобьём нашу задачу на две. Внешний уровень определяет структуру, названную `Dirent`, и три подпрограммы – `opendir`, `readdir` и `closedir`; в результате обеспечивается системно-независимый доступ к имени и номеру узла inode каждого элемента каталога. Мы будем писать программу `fsize`, рассчитывая на такой интерфейс, а затем покажем, как реализовать указанные функции для систем, использующих ту же структуру каталога, что и системы Version 7 и System V UNIX. Другие варианты оставим для упражнений.

Структура `Dirent` содержит номер inode и имя. Максимальная длина имени файла есть `NAME_MAX` – значение, являющееся системно-зависимым. Функция `opendir` возвращает указатель на структуру, названную `DIR` (по аналогии с `FILE`), которая используется функциями `readdir` и `closedir`. Эта информация сосредоточена в головном файле `dirent.h`.

```
#define NAME_MAX  14   /* максимальная длина имени файла; */
                       /* системно-зависимая величина     */
typedef struct { /* универс. структура элемента каталога: */
    long ino;              /* номер inode */
    char name[NAME_MAX+1];  /* имя + '\0' */
} Dirent;
typedef struct { /* минимальный DIR: без буфер-ции и т.д. */
    int fd;         /* файловый дескриптор каталога */
    Dirent d;       /* элемент каталога */
} DIR;
DIR *opendir(char *dirname);
Dirent *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

Системный вызов `stat` получает имя файла и возвращает полную о нём информацию, содержащуюся в inode, или –1, в случае ошибки. Так,

```
char *name;
struct stat stbuf;
int stat(char *, struct stat *);
stat(name, &stbuf);
```

заполняет структуру `stbuf` информацией из inode о файле с именем `name`.

Структура, описывающая возвращаемое функцией stat значение, находится в <sys/stat.h> и выглядит примерно так:

```
struct stat {  /* информация из inode, возвращаемая stat */
    dev_t   st_dev;    /* устройство */
    ino_t   st_ino;    /* номер inode */
    short   st_mode;   /* режимные биты */
    short   st_nlink;  /* число связей с файлом */
    short   st_uid;    /* имя пользователя-собственника */
    short   st_gid;    /* имя группы собственника */
    dev_t   st_rdev;   /* для специальных файлов */
    off_t   st_size;   /* размер файла в литерах */
    time_t  st_atime;  /* время последнего использования */
    time_t  st_mtime;  /* время последней модификации */
    time_t  st_ctime;  /* время последнего изменения inode */
};
```

Большинство этих значений объясняется в комментариях. Типы, подобные dev_t и ino_t, определены в файле <sys/types.h>.

Элемент st_mode содержит набор флажков, составляющих дополнительную информацию о файле. Определения флажков также содержатся в <sys/stat.h>; нам потребуется только та его часть, которая имеет дело с типом файла:

```
#define S_IFMT    0160000        /* тип файла */
#define S_IFDIR   0040000        /* каталог */
#define S_IFCHR   0020000        /* символьно-ориентир. */
#define S_IFBLK   0060000        /* блочно-ориентир. */
#define S_IFREG   0100000        /* обычный */
```

Теперь мы готовы приступить к написанию программы fsize. Если режимные биты (st_mode), полученные от stat, указывают, что файл не является каталогом, то можно взять его размер (st_size) и напечатать. Однако если файл – каталог, то мы должны обработать все его файлы, каждый из которых в свою очередь может быть каталогом. Обработка каталога – процесс рекурсивный.

Программа main просматривает параметры командной строки, передавая каждый аргумент функции fsize.

```
#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h>          /* флажки чтения и записи */
#include <sys/types.h>      /* определения типов */
#include <sys/stat.h>       /* структура, возвращаемая stat */
#include "dirent.h"
void fsize(char *);
```

```
/* печатает размеры файлов */
main(int argc, char **argv)
{
    if (argc == 1) /* по умолчанию берётся текущий каталог */
        fsize(".");
    else
        while (--argc > 0)
            fsize(*++argv);
    return 0;
}
```

Функция fsize печатает размер файла. Однако, если файл – каталог, она сначала вызывает dirwalk, чтобы обработать все его файлы. Обратите внимание на то, как используются имена флажков S_IFMT и S_IFDIR из <sys/stat.h> при проверке, является ли файл каталогом. Здесь нужны скобки, поскольку приоритет оператора & ниже приоритета оператора ==.

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));

/* fsize: печатает размер файла с именем "name" */
void fsize(char *name)
{
    struct stat stbuf;
    if (stat(name, &stbuf) == -1) {
        fprintf(stderr, "fsize: нет доступа к %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 – это универсальная программа, применяющая некоторую функцию к каждому файлу каталога. Она открывает каталог, с помощью цикла перебирает содержащиеся в нём файлы, применяя к каждому из них указанную функцию, затем закрывает каталог и осуществляет возврат. Так как fsize вызывает dirwalk на каждом каталоге, в этих двух функциях заложена косвенная рекурсия.

```
#define MAX_PATH 1024

/* dirwalk: применяет fcn ко всем файлам из dir */
void dirwalk(char *dir, void (*fcn)(char *))
{
```

```
    char name[MAX_PATH];
    Dirent *dp;
    DIR *dfd;
    if ((dfd = opendir(dir)) == NULL) {
        fprintf(stderr, "dirwalk: не могу откр.%s\n", dir);
        return;
    }
    while ((dp = readdir(dfd)) != NULL) {
        if (strcmp(dp->name, ".") == 0
         || strcmp(dp->name, "..") == 0)
            continue;  /* пропустить себя и родителя */
        if (strlen(dir)+strlen(dp->name)+2 > sizeof(name))
            fprintf(stderr,"dirwalk: слишк.длин.имя %s/%s\n",
                dir, dp->name);
        else {
            sprintf(name, "%s/%s", dir, dp->name);
            (*fcn)(name);
        }
    }
    closedir(dfd);
}
```

Каждый вызов readdir даёт в результате указатель на информацию о следующем файле или NULL, если все файлы обработаны. Любой каталог всегда хранит в себе информацию о себе самом под именем «.» и о своём родителе под именем «..»; их нужно пропустить, иначе программа зациклится. Обратите внимание: код программы этого уровня не зависит от того, как форматированы каталоги. Следующий шаг – представить минимальные версии opendir, readdir и closedir для некоторой конкретной системы. Здесь приведены программы для систем Version 7 и System V UNIX. Они используют информацию о каталоге, хранящуюся в головном файле <sys/dir.h>, который выглядит следующим образом:

```
#ifndef DIRSIZ
#define DIRSIZ  14
#endif

struct direct   /* элемент каталога */
{
    ino_t d_ino;            /* номер inode */
    char d_name[DIRSIZ];    /* длинное имя не имеет '\0' */
};
```

Некоторые версии системы допускают более длинные имена и имеют более сложную структуру каталога.

Тип ino_t задан при помощи typedef и описывает индекс списка inode. В системе, которой пользуемся мы, этот тип есть unsigned short, но на других системах он может быть иным, поэтому его лучше определять через typedef. Полный набор «системных» типов находится в <sys/types.h>.

Функция opendir открывает каталог, проверяет, является ли он действительно каталогом (в данном случае это делается при помощи системного вызова fstat, который аналогичен stat, но применяется к дескриптору файла), запрашивает пространство для структуры 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 каталог и освобождает пространство.

```
/* closedir: закрывает каталог, открытый opendir */
void closedir(DIR *dp)
{
    if (dp) {
        close(dp->fd);
        free(dp);
    }
}
```

Наконец, readdir с помощью read читает каждый элемент каталога. Если некий элемент каталога в данный момент не используется (соответствующий ему файл был удалён), то номер inode у него равен нулю, и данная позиция пропускается. В противном случае номер inode и имя размещаются в static-структуре, и указатель на неё выдаётся в качестве результата. При каждом следующем обращении новая информация занимает место предыдущей.

```
#include <sys/dir.h>  /* место располож.структуры каталога */

/*readdir: читает последовательно элементы каталога */
Dirent *readdir(DIR *dp)
{
    struct direct dirbuf; /* структура каталога на
                                          данной системе */
    static Dirent d;      /* возвращает унифицированную
                                             структуру */
    while (read(dp->fd, (char *) &dirbuf, sizeof(dirbuf))
                  == sizeof(dirbuf)) {
        if (dirbuf.d_ino == 0)   /* пустой эл-т не использ.*/
            continue;
        d.ino = dirbuf.d_ino;
        strncpy(d.name, dirbuf.d_name, DIRSIZ);
        d.name[DIRSIZ] = '\0';  /* завершающая литера '\0' */
        return &d;
    }
    return NULL;
}
```

Хотя программа `fsize` – довольно специализированная, она иллюстрирует два важных факта. Первый, многие программы не являются «системными»; они просто используют информацию, которую поддерживает операционная система. Для таких программ существенно то, что представление информации сосредоточено исключительно в стандартных головных файлах. Они включают эти файлы, а не держат декларации при себе. Второе наблюдение заключается в том, что при старании системно-зависимым объектам можно создать интерфейсы, которые сами не будут системно-зависимыми. Хорошие тому примеры – функции стандартной библиотеки.

Упражнение 8.5. Модифицируйте `fsize` таким образом, чтобы можно было печатать остальную информацию, содержащуюся в inode.

8.7. Пример. Распределитель памяти

В гл. 5 был описан простой распределитель памяти, основанный на принципе стека. Версия, которую мы напишем здесь, не имеет ограничений: вызовы `malloc` и `free` могут выполняться в любом порядке; `malloc` делает запрос в операционную систему на выделение памяти тогда, когда она требуется. Эти программы иллюстрируют приёмы, позволяющие получать машинно-зависимый код сравнительно машинно-независимым способом,

и, кроме того, они могут служить реальным примером применения таких средств языка, как структуры, объединения и `typedef`.

Никакого ранее скомпилированного массива фиксированного размера, из которого выделяются куски памяти, не будет. Функция `malloc` запрашивает память у операционной системы по мере надобности. Поскольку и другие действия программы могут вызывать запросы памяти, которые удовлетворяются независимо от этого распределителя памяти, пространство, которым заведует `malloc`, не обязательно представляет собой связный кусок памяти. Поэтому свободная память хранится в виде списка блоков. Каждый блок содержит размер, указатель на следующий блок и само пространство. Блоки в списке хранятся в порядке возрастания адресов памяти, при этом последний блок (с самым большим адресом) ссылается на первый.

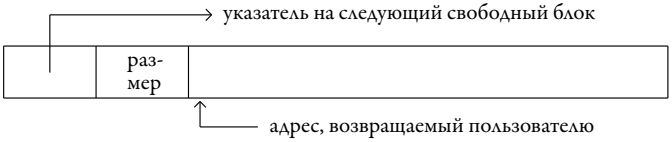
При возникновении запроса на память просматривается список свободных блоков, пока не обнаружится достаточно большой блок. Такой алгоритм называется «поиском первого подходящего» в отличие от алгоритма «поиска наилучшего подходящего», который ищет наименьший блок из числа удовлетворяющих запросу. Если блок оказался в точности требуемого размера, то он отцепляется от списка и отдаётся в пользование. Если размер блока больше, чем требуется, от него отрезается нужная часть – она отдаётся пользователю, а ненужная оставляется в списке свободных блоков. Если блока достаточного размера не оказалось, то у операционной системы запрашивается ещё один большой кусок памяти, который присоединяется к списку свободных блоков.

Процедура освобождения сопряжена с хождением по списку свободных блоков, поскольку нужно найти подходящее место для освобождаемого блока. Если подлежащий освобождению блок примыкает с какой-то стороны к одному из свободных блоков, то он объединяется с ним в один блок бóльшего размера, чтобы по возможности уменьшить раздробленность памяти. Выполнение проверки, примыкают ли блоки друг к другу, не составляет труда, поскольку список свободных блоков всегда упорядочен по возрастанию адресов.

Существует проблема, о которой мы уже упоминали в гл. 5, состоящая

в том, что память, выдаваемая функцией malloc, должна быть соответствующим образом выровнена с учётом объектов, которые в ней будут храниться. Хотя машины и отличаются друг от друга, но для каждой из них существует тип, предъявляющий самые большие требования на выравнивание, и, если по некоему адресу допускается размещение объекта этого типа, то по нему можно разместить и объекты всех других типов. На некоторых машинах таким самым «требовательным» типом является double, на других – это может быть int или long.

Свободный блок содержит указатель на следующий блок в списке, свой размер и собственно свободное пространство.



Указатель и размер представляют собой управляющую информацию и образуют так называемый «заголовок». Чтобы упростить выравнивание, все блоки создаются кратными размеру заголовка, а заголовок соответствующим образом выравнивается. Этого можно достичь, сконструировав объединение, которое будет содержать соответствующую заголовку структуру и самый требовательный в отношении выравнивания тип. Для конкретности мы выбрали тип long.

```
typedef long Align;    /* для выравнивания по границе long */

union header {         /* заголовок блока: */
    struct {
        union header *ptr;  /* на след.блок в списке своб. */
        unsigned size;      /* размер этого блока */
    } s;
    Align x;           /* для принудит. выравнивания блока */
};
typedef union header Header;
```

Поле Align нигде не используется; оно необходимо только для того, чтобы каждый заголовок был выровнен по самому «худшему» варианту границы.

Затребованное число литер округляется в malloc до целого числа единиц памяти размером в заголовок (именно это число и записывается в поле size (размер) в заголовке); кроме того, в блок входит ещё одна единица памяти – сам заголовок. Указатель, возвращаемый функцией malloc, указывает на свободное пространство, а не на заголовок. Со свободным пространством пользователь может делать что угодно, но, если он будет писать что-либо за его пределами, то, вероятно, список будет разрушен. Поскольку па-

мять, управляемая функцией malloc, не обладает связностью, размеры блоков нельзя вычислить по указателям, и поэтому без поля, хранящего размер, нам не обойтись.

Для организации начала работы используется переменная base. Если freep есть NULL (как это бывает при первом обращении к malloc), создаётся «вырожденный» список свободного пространства; он содержит один блок нулевого размера с указателем на самого себя. Поиск свободного блока подходящего размера начинается с этого указателя (freep), т.е. с последнего найденного блока; такая стратегия помогает поддерживать список однородным. Если найденный блок окажется слишком большим, пользователю будет отдана его хвостовая часть; при этом потребуется только в заголовке найденного свободного блока уточнить его размер. В любом случае возвращаемый пользователю указатель является адресом свободного пространства, размещающегося в блоке непосредственно за заголовком.

```
static Header base;    /* пустой список для нач. запуска */
static Header *freep = NULL;     /* начало в спис.своб.бл. */

/* malloc: универсальный распределитель 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) { /* спис.своб.пам.ещё 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) { /* достаточно большой */
            if (p->s.size == nunits) /* точно такого разм. */
                prevp->s.ptr = p->s.ptr;
            else {              /* отрезаем хвостовую часть */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p+1);
        }
        if (p == freep)   /* прошли полный цикл по списку*/
            if ((p = morecore(nunits)) == NULL)
```

```
                return NULL;     /* нет больше памяти */
        }
    }
```

Функция morecore получает память от операционной системы. Детали того, как это делается, могут не совпадать в различных системах. Так как запрос памяти у системы – сравнительно дорогая операция, мы бы не хотели для этого каждый раз обращаться к malloc. Поэтому используется функция morecore, которая запрашивает не менее NALLOC единиц памяти; этот бóльший кусок памяти будет «нарезаться» потом по мере надобности. После установки в поле размера соответствующего значения функция morecore вызывает функцию free и тем самым включает полученный кусок в список свободных областей памяти.

```
#define NALLOC  1024   /* миним.число ед.памяти для запроса*/

/* morecore: запрашивает у системы дополнительную память */
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)  /* нет больше памяти */
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *)(up+1));
    return freep;
}
```

Системный вызов sbrk(n) в UNIXе возвращает указатель на n байт памяти или –1, если требуемого пространства не оказалось, хотя было бы лучше, если бы в последнем случае он возвращал NULL. Константу –1 необходимо привести к типу char *, чтобы её можно было сравнить с возвращаемым значением. Это ещё один пример того, что операция приведения типа делает функцию относительно независимой от конкретного представления указателей на различных машинах. Есть, однако, одна «некорректность», состоящая в том, что сравниваются указатели на различные блоки, выдаваемые функцией sbrk. Такое сравнение не гарантировано стандартом, который позволяет сравнивать указатели лишь в пределах массива. Таким образом, эта версия malloc верна только на тех машинах, в которых допускается сравнение любых указателей.

В заключение рассмотрим функцию free. Она просматривает список свободной памяти, начиная с freep, чтобы подыскать место для вставляемого блока. Искомое место может оказаться или между блоками, или в начале списка, или в его конце. В любом случае, если подлежащий освобождению блок примыкает к соседнему блоку, он объединяется с ним в один блок. О чём ещё осталось позаботиться, – так это о том, чтобы ссылки указывали в нужные места и размеры блоков были правильными.

```
/* free: включает блок в список свободной памяти */
void free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)ap - 1;  /* указ. на заголовок блока */
    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; /* освобожд.блок в начале или в конце */
    if (bp + bp->s.size == p->s.ptr) {  /* слить с верхним */
        bp->s.size += p->s.ptr->s.size; /*         соседом */
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) {   /* слить с нижним соседом */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

Хотя распределение памяти по своей сути – машинно-зависимая проблема, с ней можно справиться, что и иллюстрирует приведённая программа, в которой машинная зависимость упрятана в очень маленькой её части. Что касается проблемы выравнивания, то мы разрешили её при помощи typedef и union (предполагается, что sbrk даёт нормальный в смысле выравнивания указатель). Операции приведения типов позволяют нам сделать явными преобразования типов и даже справиться с плохо спроектированным интерфейсом системы. Несмотря на то что наши рассуждения касались распределения памяти, этот общий подход применим и в других ситуациях.

Упражнение 8.6. Стандартная функция calloc(n, size) возвращает указатель на n элементов памяти размера size, заполненных нулями. Напишите свой вариант calloc, пользуясь функцией malloc или модифицируя последнюю.

Упражнение 8.7. Функция `malloc` допускает любой размер, никак не проверяя его на правдоподобие; `free` предполагает, что размер освобождаемого блока – правильный. Усовершенствуйте эти программы таким образом, чтобы они более тщательно контролировали ошибки.

Упражнение 8.8. Напишите программу `bfree(p,n)`, освобождающую произвольный блок `p`, состоящий из `n` литер, путём включения его в список свободной памяти, поддерживаемый функциями `malloc` и `free`. При помощи `bfree` пользователь должен иметь возможность в любое время добавить статический или внешний массив в список свободной памяти.

Приложение А

Справочное руководство

А.1. Введение

Данное руководство является описанием языка программирования Си, определённого в соответствии с проектом, утверждённым в ANSI в качестве Американского национального стандарта для информационных систем: Язык программирования Си, X3.159–1989 («American National Standard for Information Systems – Programming Language C, X3.159–1989»). Это описание – лишь некоторый вариант предлагаемого стандарта, а не сам стандарт, однако мы специально заботились о том, чтобы оно было надёжным руководством по языку.

Настоящий документ в основном следует общей схеме описания, принятой в стандарте (публикация которого в свою очередь основывалась на первом издании этой книги), однако в организационном плане есть различия. Если не считать отклонений в названиях нескольких продукций и отсутствия формальных определений лексем и препроцессора, грамматики языка здесь и в стандарте эквивалентны.

> Далее всюду примечания (как и это) набираются с отступом от левого края страницы. В основном эти примечания касаются отличий стандарта от версии языка, описанной в первом издании этой книги, и от последующих нововведений в различных компиляторах.

А.2. Соглашения о лексике

Программа состоит из одной или нескольких *компонент трансляции*, хранящихся в виде файлов. Каждая такая компонента проходит ряд фаз трансляции, описанных в А.12. Начальные фазы осуществляют лексические

преобразования нижнего уровня, выполняют директивы, задаваемые в программе строками, начинающимися со знака #, обрабатывают макроопределения и получают макрорасширения. По завершению препроцессирования (А.12) программа представляется в виде последовательности лексем.

А.2.1. Лексемы

Существуют шесть классов лексем: идентификаторы, ключевые слова, константы, стринговые литералы, операторы и прочие разделители. Пробелы, горизонтальные и вертикальные табуляции, новые-строки, переводы-страницы и комментарии (имеющие общее название «пробельные литеры») рассматриваются компилятором только как разделители лексем и в остальном на результат трансляции влияния не оказывают. Любая из пробельных литер годится, чтобы отделить друг от друга соседние идентификаторы, ключевые слова и константы.

Если предположить, что входной поток уже до некоторой литеры разбит на лексемы, то следующей лексемой будет самый длинный стринг, который вообще может быть лексемой.

А.2.2. Комментарий

Литеры /* открывают комментарий, а литеры */ закрывают его. Комментарии нельзя вкладывать друг в друга, их нельзя помещать внутрь стрингов или текстовых литералов.

А.2.3. Идентификаторы

Идентификатор – последовательность букв и цифр. Первой литерой должна быть буква; знак подчёркивания _ считается буквой. Буквы нижнего и верхнего регистров различаются. Идентификаторы могут иметь любую длину; для внутренних идентификаторов значимыми являются первая 31 литера; в некоторых реализациях принято бóльшее число значимых литер. К внутренним идентификаторам относятся имена макросов и все другие имена, не имеющие внешних связей (А.11.2). На идентификаторы с внешними связями могут накладываться бóльшие ограничения: иногда воспринимаются не более шести первых литер и/или не различаются буквы верхнего и нижнего регистров.

А.2.4. Ключевые слова

Следующие идентификаторы зарезервированы в качестве ключевых слов и в другом смысле 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 резервируются также слова `fortran` и `asm`.

Ключевые слова `const`, `signed` и `volatile` впервые появились в стандарте ANSI; `enum` и `void` – новые по отношению к первому изданию, но уже использовались; ранее зарезервированное `entry` нигде не использовалось и поэтому более не резервируется.

А.2.5. Константы

Существует несколько видов констант. Каждая имеет свой тип данных; базовые типы рассматриваются в А.4.2.

константа:
целая-константа
литерная-константа
с-плав-точкой-константа
перечислимая-константа

А.2.5.1. Целые константы

Целая константа, состоящая из последовательности цифр, воспринимается как восьмеричная, если она начинается с `0` (цифры нуль), и как десятичная в противном случае. Восьмеричная константа не содержит цифр `8` и `9`. Последовательность цифр, перед которой стоят `0x` или `0X`, рассматривается как шестнадцатеричное целое. В шестнадцатеричные цифры включены буквы от `a` (или `A`) до `f` (или `F`) со значениями от 10 до 15.

Целая константа может быть записана с буквой-суффиксом `u` (или `U`) для спецификации её как беззнаковой константы. Она также может быть с буквой-суффиксом `l` (или `L`) для указания, что она имеет тип `long`.

Тип целой константы зависит от её вида, значения и суффикса. (О типах см. А.4.) Если константа – десятичная и не имеет суффикса, то она принимает первый из следующих типов, который годится для представления её значения: `int`, `long int` и `unsigned long int`. Восьмеричная или шестнадцатеричная константа без суффикса принимает первый возможный из типов `int`, `unsigned int`, `long int` и `unsigned long int`. Если константа имеет суффикс `u` или `U`, то она принимает первый возможный из типов `unsigned int` и

`unsigned long int`. Если константа имеет суффикс `l` или `L`, то она принимает первый возможный из типов `long int` и `unsigned long int`.

> Типы целых констант получили существенное развитие в сравнении с первой редакцией, в которой большие целые имели просто тип `long`. Суффиксы `U` и `u` введены впервые.

А.2.5.2. Литерные константы

Литерная константа – последовательность из одной или нескольких литер, заключённая в одиночные кавычки (например, `'x'`). Если внутри одиночных кавычек расположена одна литера, значением константы является числовое значение этой литеры в кодировке, принятой на данной машине. Значение константы с несколькими литерами зависит от реализации.

Литерная константа не может содержать в себе одиночную кавычку «`'`» или литеру новая-строка; чтобы изобразить их и некоторые другие литеры, могут быть использованы эскейп-последовательности:

новая-строка (`newline`, `linefeed`)	`NL (LF)`	`\n`
гориз-таб (`horisontal tab`)	`HT`	`\t`
верт-таб (`vertical tab`)	`VT`	`\v`
возврат-на-шаг (`backspace`)	`BS`	`\b`
возврат-каретки (`carriage return`)	`CR`	`\r`
перевод-страницы (`formfeed`)	`FF`	`\f`
сигнал-звонок (`audible alert,bell`)	`BEL`	`\a`
обратная-наклонная-черта (`backslash`)	`\`	`\\`
знак-вопроса (`question mark`)	`?`	`\?`
одиночная-кавычка (`single quote`)	`'`	`\'`
двойная-кавычка (`double quote`)	`"`	`\"`
восьмеричный-код (`octal number`)	*ooo*	`\`*ooo*
шестнадцатеричный-код (`hex number`)	*hh*	`\x`*hh*

Эскейп-последовательность `\`*ooo* состоит из обратной наклонной черты, за которой следуют одна, две или три восьмеричные цифры, специфицирующие значение желаемой литеры. Наиболее частым примером такой конструкции является `\0` (за которой не следует цифра); она специфицирует null-литеру. Эскейп-последовательность `\x`*hh* состоит из обратной наклонной черты с буквой `x`, за которыми следуют шестнадцатеричные цифры, специфицирующие значение желаемой литеры. На количество цифр нет ограничений, но результат будет не определён, если значение полученной литеры превысит значение самой «большой» из допустимых литер. Если в данной реализации тип `char` трактуется как число со знаком, то значение и в восьмеричной, и в шестнадцатеричной эскейп-последовательности получается при помощи «размножения знака», как если бы выполнялась операция приведения к типу `char`. Результат не определён, если за `\` не следует ни одна из перечисленных выше литер.

В некоторых реализациях имеется расширенный набор литер, который не может быть охвачен типом `char`. Константа для такого набора пишется с буквой `L` впереди (например, `L'x'`) и называется «широкой» литерной константой. Такая константа имеет тип `wchar_t` (целочисленный тип, определённый в стандартном головном файле `<stddef.h>`). Как и в случае обычных литерных констант, здесь также возможны восьмеричные и шестнадцатеричные эскейп-последовательности; результат будет не определён, если специфицированное значение превысит тип `wchar_t`.

> Некоторые из приведённых эскейп-последовательностей новые (шестнадцатеричные в частности). Новым является и расширенный тип для литер. Наборам литер, обычно используемым в Америке и Западной Европе, подходит тип `char`, а тип `wchar_t` был добавлен, главным образом для того, чтобы удовлетворить азиатские языки.

А.2.5.3. Константы с плавающей точкой (плавающие константы)

Плавающая константа состоит из целой части, десятичной точки, дробной части, `e` или `E` и целого (возможно, со знаком), представляющего экспоненту, и, возможно, типа-суффикса, задаваемого одной из букв: `f`, `F`, `l` или `L`. И целая, и дробная часть представляют собой последовательность цифр. Либо целая часть, либо дробная часть (но не обе вместе) могут отсутствовать; также могут отсутствовать десятичная точка или `E` с экспонентой (но не обе одновременно). Тип определяется суффиксом; `F` или `f` определяют тип `float`, `L` или `l` – тип `long double`; при отсутствии суффикса подразумевается тип `double`.

> Суффиксы для плавающих констант являются нововведением.

А.2.5.4. Перечислимые константы

Идентификаторы, объявленные как элементы перечисления (А.8.4), являются константами типа `int`.

А.2.6. Стринговые литералы

Стринговый литерал, который также называют стринговой константой, – это последовательность литер, заключённая в двойные кавычки (например, `"..."`). Стринг имеет тип «массив литер» и память класса `static` (А.4), которая инициализируется заданными литерами. Представляются ли одинаковые стринговые литералы одной копией или несколькими, зависит от реализации. Поведение программы, пытающейся изменить стринговый литерал, не определено.

Рядом написанные стринговые литералы объединяются (конкатенируются) в один стринг. После любой конкатенации к стрингу добавля-

ется null-байт (\0), что позволяет программе, просматривающей стринг, найти его конец. Стринговые литералы не могут содержать в себе новую строку или двойную кавычку; в них нужно использовать те же эскейп-последовательности, что и в литерных константах.

Как и в случае с литерными константами, стринговый литерал с литерами из расширенного набора должен начинаться с буквы L (например, L"..."). Стринговый литерал из расширенного набора имеет тип «массив из wchar_t». Конкатенация обычных и «расширенных» стринговых литералов друг с другом не определена.

> То, что стринговые литералы не обязательно представляются разными копиями, запрет на их модификацию, а также конкатенация соседних стринговых литералов являются нововведениями ANSI-стандарта. «Расширенные» стринговые литералы также объявлены впервые.

А.3. Нотация синтаксиса

В нотации синтаксиса, используемой в этом руководстве, синтаксические понятия набираются курсивом, а слова и литеры, воспринимаемые буквально, обычным шрифтом. Альтернативные конструкции обычно перечисляются в столбик (каждая альтернатива на отдельной строке); в редких случаях длинные списки небольших по размеру альтернатив располагаются в одной строке, помеченной словами «один из». Необязательный терминальный или нетерминальный символ снабжается индексом «*необ*». Так, запись

{ *выражение*$_{необ}$ }

обозначает выражение, заключённое в фигурные скобки, которое в общем случае может отсутствовать. Полный перечень синтаксических конструкций приведён в А.13.

> В отличие от грамматики, данной в первом издании этой книги, приведённая здесь грамматика старшинство и порядок выполнения операций в выражениях описывает явно.

А.4. Что обозначают идентификаторы

Существует ряд вещей, на которые ссылаются при помощи идентификаторов, или имён; это – функции; теги структур, объединений и перечислений; члены структур или объединений; typedef-имена и объекты. Объектом (называемым иногда переменной) является часть памяти, интерпретация которой зависит от двух главных характеристик: *класса памяти* и её *типа*. Класс памяти сообщает о времени жизни памяти, связанной с идентифицируемым объектом; тип определяет, какого рода значения находятся в

объекте. С любым именем ассоциируется своя область действия (т.е. тот участок программы, где это имя «видимо») и атрибут связи, определяющий, ссылается ли это имя в другом файле на тот же самый объект или функцию. Область действия и атрибут связи обсуждаются в А.11.

А.4.1. Класс памяти

Существуют два класса памяти: автоматический и статический. Несколько ключевых слов в совокупности с контекстом деклараций объектов специфицируют класс памяти для этих объектов. Автоматические объекты локализованы в блоке (А.9.3), они «исчезают» при выходе из него. Декларация, заданная внутри блока, если в ней отсутствует спецификация класса памяти или указан спецификатор `auto`, создаёт автоматический объект. Объект, помеченный в декларации словом `register`, является автоматическим и размещается по возможности в регистре машины.

Статические объекты могут быть локализованы в блоке или располагаться вне блоков, но в обоих случаях их значения сохраняются после выхода из блока (или функции) до повторного в него входа. Внутри блока (в том числе и в блоке, образующем тело функции) статические объекты в декларациях помечаются словом `static`. Объекты, декларируемые вне всех блоков на одном уровне с определениями функций, всегда статические. При помощи ключевого слова `static` их можно локализовать в пределах транслируемой компоненты (в этом случае они получают атрибут *внутренней связи*), и они становятся глобальными для всей программы, если опустить явное указание класса памяти или использовать ключевое слово `extern` (в этом случае они получают атрибут *внешней связи*).

А.4.2. Базовые типы

Существует несколько базовых типов. Стандартный головной файл `<limits.h>`, описанный в приложении В, определяет самое большое и самое малое значения для каждого типа в данной конкретной реализации. В приложении В приведены минимально возможные величины.

Размер объектов, декларируемых как литеры, позволяет хранить любую литеру из набора литер, принятого в машине. Если объект типа `char` действительно хранит литеру из данного набора, то его значением является код этой литеры, т.е. некоторое неотрицательное целое. Переменные типа `char` могут хранить и другие значения, но тогда диапазон их значений и особенно вопрос о том, знаковые эти значения или беззнаковые, зависит от реализации.

Беззнаковые литеры, декларируемые при помощи слов `unsigned char`, имеют ту же разрядность, что и обычные литеры, но представляют неотрицательные значения; при помощи слов `signed char` можно явно деклариро-

вать литеры со знаком, которые занимают столько же места, как и обычные литеры.

> Тип `unsigned char` не упоминался в первой редакции, но всеми использовался. Тип `signed char` – новый.

Помимо `char` среди целочисленных типов могут быть целые трёх размеров: `short int`, `int` и `long int`. Обычные `int`-объекты имеют естественный размер, принятый в архитектуре данной машины, другие размеры предназначены для специальных нужд. Более длинные целые по крайней мере покрывают все значения более коротких целых, однако в некоторых реализациях обычные целые могут быть эквивалентны коротким (`short`) или длинным (`long`) целым. Все типы `int` представляют значения со знаком, если не оговорено противное.

Для беззнаковых целых в декларациях используется ключевое слово `unsigned`. Такие целые подчиняются арифметике по модулю 2^n, где n – число бит в представлении числа, и, следовательно, в арифметике с беззнаковыми целыми никогда не бывает переполнения. Множество неотрицательных значений, которые могут храниться в знаковых объектах, является подмножеством значений, которые могут храниться в соответствующих беззнаковых объектах; знаковое и беззнаковое представления каждого такого значения совпадают.

Любые два из плавающих типов: с одинарной точностью (`float`), с двойной точностью (`double`) и с повышенной точностью (`long double`) могут быть синонимами, но каждый следующий тип этого списка должен по крайней мере обеспечивать точность предыдущего.

> `long double` – новый тип. В первой редакции синонимом для `double` был `long float`, теперь последний изъят из обращения.

Перечисления – единственные в своём роде типы, которым даётся полный перечень значений; с каждым перечислением связывается множество именованных констант (А.8.4). Перечисления ведут себя наподобие целых, но компилятор обычно выдаёт предупреждающее сообщение, если объекту некоторого перечислимого типа присваивается нечто, отличное от его константы, или выражение не этого перечислимого типа.

Поскольку объекты перечислимых типов можно рассматривать как числа, перечисление относят к *арифметическому* типу. Типы `char` и `int` всех размеров, каждый из которых может быть со знаком или без знака, а также перечислимые типы называют *целочисленными*. Типы `float`, `double` и `long double` называются *плавающими*.

Тип `void` специфицирует пустое множество значений. Он используется как «тип возвращаемого функцией значения» в том случае, когда она не генерирует никакого результирующего значения.

A.4.3. Выводимые типы

Помимо базовых типов существует практически бесконечный класс выводимых типов, которые формируются из уже существующих и которые описывают следующие конструкции:

- *массивы* объектов заданного типа;
- *функции*, возвращающие объекты заданного типа;
- *указатели* на объекты заданного типа;
- *структуры*, содержащие последовательность объектов, возможно, различных заданных типов;
- *объединения*, каждое из которых может содержать один из нескольких объектов различных заданных типов.

В общем случае приведённые методы конструирования объектов могут применяться рекурсивно.

A.4.4. Квалификаторы типов

Тип объекта может снабжаться квалификатором. Декларация объекта с квалификатором `const` указывает на то, что его значение далее не будет изменяться; объявляя объект как `volatile` (изменчивый, непостоянный (*англ.*)) мы указываем на его особые свойства в отношении оптимизации, выполняемой компилятором. Ни один из квалификаторов не влияет на диапазоны значений и арифметические свойства объектов. Квалификаторы обсуждаются в A.8.2.

A.5. Объекты и l-значения

Объект – это некоторая именованная область памяти; *l-значение* – выражение, ссылающееся на объект. Очевидным примером l-значения является идентификатор с соответствующим типом и классом памяти. Существуют операции, порождающие l-значения. Например, если `E` – выражение типа указатель, то `*E` есть выражение для l-значения, ссылающегося на объект, на который указывает `E`. Термин «l-значение» произошёл от записи присваивания `E1 = E2`, в которой левый (left – левый (*англ.*), отсюда и буква l) операнд `E1` должен быть выражением l-значения. Описывая каждый оператор, мы сообщаем, ожидает ли он l-значения в качестве операндов и выдаёт ли l-значение в качестве результата.

А.6. Преобразования

Некоторые операторы в зависимости от своих операндов могут вызывать преобразование их значений из одного типа в другой. В этом разделе объясняется, что следует ожидать от таких преобразований. В А.6.5 формулируются правила, по которым выполняются преобразования для большинства обычных операторов. Эти правила могут уточняться при рассмотрении каждого отдельного оператора.

А.6.1. Повышение целочисленного типа

Объект перечислимого типа, литера, короткое целое, целое в поле бит – все они со знаком или без могут использоваться в выражении там, где возможно применение целого. Если тип `int` позволяет «охватить» все значения исходного типа операнда, то операнд приводится к `int`, в противном случае он приводится к `unsigned int`. Эта процедура называется *повышением целочисленности*.

А.6.2. Целочисленные преобразования

Любое целое приводится к некоторому заданному беззнаковому типу путём поиска конгруэнтного (т.е. имеющего то же двоичное представление) наименьшего неотрицательного значения и взятия его по модулю $n_{max} + 1$, где n_{max} – наибольшее число в этом беззнаковом типе. Для двоичного представления в дополнительном коде это означает либо выбрасывание лишних старших разрядов, если беззнаковый тип «у́же» исходного типа, либо заполнение недостающих старших разрядов нулями (для значения без знака) или размноженным знаком (для значения со знаком), если беззнаковый тип «шире» исходного.

В результате приведения любого целого к знаковому типу преобразуемое значение не меняется, если оно представимо в новом типе, в противном случае результат зависит от реализации.

А.6.3. Целые и плавающие

При преобразовании из плавающего типа в целый дробная часть значения отбрасывается; если полученное при этом значение нельзя представить в заданном целом типе, то результат не определён. В частности, не определён результат преобразования отрицательных плавающих значений в беззнаковые целые.

Если значение преобразуется из целого в величину с плавающей точкой и она находится в допустимом диапазоне, но представляется в новом

типе неточно, то результатом будет одно из двух значений нового типа, ближайшего к исходному. Если результат выходит за границы диапазона допустимых значений, поведение программы не определено.

А.6.4. Плавающие типы

При преобразовании из плавающего типа с меньшей точностью в плавающий тип с большей точностью значение не изменяется. Если, наоборот, переход осуществляется от большей точности к меньшей и значение остаётся в допустимых пределах нового типа, то результатом будет одно из двух ближайших значений нового типа. Если результат выходит за границы диапазона допустимых значений, поведение программы не определено.

А.6.5. Арифметические преобразования

Во многих операциях преобразование типов операндов и определение типа результата осуществляются по одним и тем же правилам. Они состоят в том, что операнды приводятся к некоторому общему типу, который также является и типом результата. Эти правила называются *обычными арифметическими преобразованиями*.

- Если какой-либо из операндов имеет тип `long double`, то другой приводится к `long double`.
- В противном случае, если какой-либо из операндов имеет тип `double`, то другой приводится к `double`.
- В противном случае, если какой-либо из операндов имеет тип `float`, то другой приводится к `float`.
- В противном случае для обоих операндов осуществляется повышение целочисленности; затем, если один из операндов имеет тип `unsigned long int`, то и другой преобразуется в `unsigned long int`.
- В противном случае, если один из операндов принадлежит типу `long int`, а другой – `unsigned int`, то результат зависит от того, покрывает ли `long int` все значения `unsigned int`, и если это так, то `unsigned int` приводится к `long int`, если нет, то оба операнда преобразуются в `unsigned long int`.
- В противном случае, если один из операндов имеет тип `long int`, то другой приводится к `long int`.
- В противном случае, если один из операндов – `unsigned int`, то другой приводится к `unsigned int`.

- В противном случае оба операнда имеют тип `int`.

> Здесь два изменения. Во-первых, арифметика с `float`-операндами теперь может быть с одинарной точностью, а не только с двойной; в первой редакции вся плавающая арифметика была с двойной точностью. Во-вторых, более короткий беззнаковый тип в комбинации с более длинным знаковым типом не распространяет свойство беззнаковости на тип результата; в первой редакции беззнаковый тип всегда доминировал. Новые правила немного сложнее, но до некоторой степени уменьшают вероятность появления неожиданных эффектов в комбинациях знаковых и беззнаковых величин. При сравнении беззнакового выражения со знаковым того же размера всё же может возникнуть неожиданный результат.

А.6.6. Указатели и целые

К указателю можно прибавлять (и вычитать из него) выражение целочисленного типа; последнее в этом случае подвергается преобразованию, описанному в А.7.7 при рассмотрении оператора сложения.

К двум указателям на объекты одного типа, принадлежащие одному массиву, может применяться операция вычитания; результат приводится к целому посредством преобразования, описанного в А.7.7 при рассмотрении оператора вычитания.

Целочисленное константное выражение со значением 0 или оно же, но приведённое к типу `void *`, может быть преобразовано в указатель любого типа операторами приведения, присваивания и сравнения. Результатом будет `null`-указатель, который равен любому другому `null`-указателю того же типа, но не равен никакому указателю, ссылающемуся на реальный объект или функцию.

Допускаются и другие преобразования для указателей, но в связи с ними возникает проблема зависимости результата от реализации. Эти преобразования должны быть специфицированы явным оператором преобразования типа или оператором приведения (А.7.5 и А.8.8).

Указатель можно привести к достаточно большому для его хранения целочисленному типу; требуемый размер зависит от реализации. Функция преобразования также зависит от реализации.

Объект целочисленного типа можно явно преобразовать в указатель. Если целое получено из указателя и имеет достаточно большой размер, это преобразование 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 ссылается на объект, должным образом не выров-

ненный по границам слов памяти, то в результате может получиться указатель, адресующий к «исключённому» фрагменту. Если требования на выравнивание у нового типа меньше или совпадают с требованиями на выравнивание первоначального типа, то гарантируется, что преобразование указателя в другой тип и обратно его не изменит; понятие «выравнивание» реализационно-зависимо, однако в любой реализации объекты типа `char` предъявляют минимальные требования на выравнивание. Как описано в А.6.8, указатель может также преобразовываться в `void *` и обратно, значение указателя при этом не изменяется.

Указатель может быть преобразован в другой указатель того же типа с добавлением или удалением квалификаторов (А.4.4, А.8.2) того типа объекта, на который этот указатель ссылается. Новый указатель, полученный добавлением квалификатора, имеет то же значение, но с дополнительными ограничениями, внесёнными новыми квалификаторами. Операция по удалению квалификатора у объекта приводит к тому, что восстанавливается действие его начальных квалификаторов, заданных в декларации этого объекта.

Наконец, указатель на функцию может быть преобразован в указатель на функцию другого типа. Вызов функции по преобразованному указателю зависит от реализации; однако, если указатель ещё раз преобразовать к его исходному типу, результат будет идентичен вызову по первоначальному указателю.

А.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-инструкции (А.9.2) или левого операнда у оператора «запятая» (А.7.18).

Выражение можно привести к типу `void` операцией приведения типа. Например, применительно к вызову функции, используемому в роли выражения-инструкции, операция приведения к `void` явным образом подчёркивает тот факт, что результат функции отбрасывается.

> Тип `void` не фигурировал в первом издании этой книги, однако за прошедшее время стал общеупотребительным.

А.6.8. Указатели на `void`

Любой указатель на объект можно привести к типу `void *` без потери информации. Если результат подвергнуть обратному преобразованию, то

мы получим прежний указатель. В отличие от преобразований указатель-в-указатель (рассмотренных в А.6.6), которые требуют явных операторов приведения к типу, в присваиваниях и сравнениях указатель любого типа может выступать в паре с указателем типа `void *` без каких-либо предварительных преобразований типа.

> Такая интерпретация указателей `void *` – новая; ранее роль обобщённого указателя отводилась указателю типа `char *`. Стандарт ANSI официально разрешает использование указателей `void *` совместно с указателями других типов в присваиваниях и сравнениях; в иных комбинациях указателей стандарт требует явных преобразований типа.

А.7. Выражения

Приоритеты описываемых операторов имеют тот же порядок, что и подразделы данного параграфа (от высших к низшим). Например, для оператора +, описанного в А.7.7, термин «операнды» означает «выражения, определённые в А.7.1–А.7.6». В каждом подразделе описываются операторы, имеющие одинаковый приоритет, и указывается их ассоциативность (левая или правая). Приоритеты и ассоциативность всех операторов отражены в грамматике, приведённой в А.13.

Приоритеты и ассоциативность полностью определены, а вот порядок вычисления выражения не определён за некоторым исключением даже для подвыражений с побочным эффектом. Это значит, что если в определении оператора последовательность вычисления его операндов специально не оговаривается, то в реализации можно свободно выбирать любой порядок вычислений и даже перемежать правый и левый порядок одновременно. Однако любой оператор использует значения своих операндов в точном соответствии с грамматическим разбором выражения, в котором он встречается.

> Это правило отменяет ранее предоставлявшуюся свободу в выборе порядка выполнения операций, которые математически коммутативны и ассоциативны, но которые в процессе вычислений могут таковыми не оказаться. Это изменение затрагивает только вычисления с плавающей точкой, выполняющиеся «на грани точности», и ситуаций, когда возможно переполнение.

Контроль за переполнением, делением на нуль и другими исключительными ситуациями, возникающими при вычислении выражения, в языке не определён. В большинстве существующих реализаций Си при вычислении знаковых целочисленных выражений и присваиваний переполнение игнорируется, но результат таких вычислений не определён. Трактовки деления на нуль и всех исключительных ситуаций, связанных с плавающей точ-

кой, могут не совпадать в разных реализациях; иногда для обработки исключительных ситуаций предоставляется нестандартная библиотечная функция.

А.7.1. Генерация указателя

Если для некоторого типа T тип выражения или подвыражения есть «массив из T», то значением этого выражения является указатель на первый элемент массива, и тип такого выражения заменяется на тип «указатель на T». Такая замена типа не делается, если выражение является операндом унарного оператора `&`, или операндом операций `++`, `--`, `sizeof`, или левым операндом присваивания, или операндом оператора «`.`». Аналогично выражение типа «функция, возвращающая T», исключая случай, когда оно является операндом для `&`, преобразуется в тип «указатель на функцию, возвращающую T».

А.7.2. Первичные выражения

Первичные выражения – это идентификаторы, константы, стринги и выражения в скобках.

первичное-выражение:
идентификатор
константа
стринг
`(` *выражение* `)`

Идентификатор, если он был должным образом декларирован (о том, как это делается, речь пойдёт ниже), – первичное выражение. Тип идентификатора специфицируется в его декларации. Идентификатор есть l-значение, если он обозначает объект (А.5) арифметического типа либо объект типа «структура», «объединение» или «указатель».

Константа – первичное выражение. Её тип зависит от формы записи, которая была рассмотрена в А.2.5.

Стринговый литерал – первичное выражение. Изначально его тип – «массив из `char`» («массив из `wchar_t`» для стринга литер расширенного набора), но в соответствии с правилом, приведённым в А.7.1, указанный тип обычно превращается в «указатель на `char`» («указатель на `wchar_t`») с результирующим значением «указатель на первую литеру стринга». Для некоторых инициализаторов такая замена типа не делается. (См. А.8.7.)

Выражение в скобках – первичное выражение, тип и значение которого идентичны типу и значению этого же выражения без скобок. Наличие или отсутствие скобок не влияет на то, является ли данное выражение l-значением или нет.

А.7.3. Постфиксные выражения

В постфиксных выражениях операторы выполняются 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*

А.7.3.1. Ссылки на элементы массива

Постфиксное выражение, за которым следует выражение в квадратных скобках, есть постфиксное выражение, обозначающее ссылку в индексируемый массив. Одно из этих двух выражений должно иметь тип «указатель на *T*», где *T* – некоторый тип, а другое – принадлежать целочисленному типу; тип результата индексирования есть *T*. Выражение `E1[E2]` по определению идентично выражению `*((E1)+(E2))`. Подробности см. в А.8.6.2.

А.7.3.2. Вызов функции

Вызов функции есть постфиксное выражение (оно называется именователем функции), за которым следуют скобки, содержащие (возможно, пустой) список разделённых запятыми выражений-присваиваний (А.7.17), представляющих собой аргументы этой функции. Если постфиксное выражение – идентификатор, не описанный в текущей области действия, то считается, что этот идентификатор как бы описан декларацией

```
extern  int  идентификатор ( );
```

помещённой в самом внутреннем блоке, содержащем вызов соответствующей функции. Постфиксное выражение (после, возможно, неявного описания и генерации указателя, см. А.7.1) должно иметь тип «указатель на функцию, возвращающую *T*», где *T* – тип возвращаемого значения.

> В первом издании для именователя функции допускался только тип «функция», и, чтобы вызвать функцию через указатель, требовался явный оператор `*`. ANSI-стандарт поощряет практику некоторых существующих компиляторов, разрешающих иметь одинаковый синтаксис для обращения просто к функции и обращения к функции, специфицированной указателем. Возможность применения старого синтаксиса остаётся.

Термин *аргумент* используется для выражения, задаваемого в вызове функции; термин *параметр* – для обозначения в определении или описании функции получаемого ею объекта (или его идентификатора). Вместо этих понятий иногда встречаются термины «фактический аргумент (параметр)» и «формальный аргумент (параметр)», имеющие те же смысловые различия.

При вызове функции копируется каждый её аргумент; передача аргументов осуществляется строго через их значения. Функции разрешается изменять значения своих параметров, которые являются лишь копиями аргументов-выражений, но эти изменения не могут повлиять на значения самих аргументов. Однако можно передать указатель, чтобы дать возможность функции изменить значение объекта, на который ссылается этот указатель.

Имеются два способа декларирования функции. В новом способе типы параметров задаются явно и являются частью типа функции; такая декларация называется прототипом функции. При старом способе типы параметров не указываются. Способы декларирования функций обсуждаются в А.8.6.3 и А.10.1.

Если вызов находится в области действия декларации, написанной по-старому, каждый его аргумент подвергается операции повышения типа: для целочисленных аргументов осуществляется повышение целочисленности (А.6.1), а для `float`-аргументов – преобразование в `double`. Результат работы вызова не определён, если число аргументов не соответствует количеству параметров в определении функции или если типы аргументов после повышения не согласуются с типами соответствующих параметров. Критерий согласованности типов зависит от способа (старого или нового) определения функции. При старом способе сравниваются повышенный тип аргумента в вызове и повышенный тип соответствующего параметра; при новом способе повышенный тип аргумента и тип параметра (без его повышения) должны быть одинаковыми.

Если вызов находится в области действия декларации, написанной по-новому, аргументы преобразуются, как если бы они присваивались переменным, имеющим типы соответствующих параметров прототипа. Число аргументов должно совпадать с числом явно описанных параметров, если только список параметров не заканчивается многоточием (`, ...`). В противном случае число аргументов должно быть больше числа параметров или равно ему; «скрывающиеся» под многоточием аргументы подвергаются операции повышения типа (так, как это было описано в предыдущем абзаце). Если определение функции задано по-старому, то типы параметров в прототипе, которые неявно присутствуют в вызове, должны соответствовать типам параметров в определении функции после их повышения.

> Эти правила особенно усложнились из-за того, что они призваны обслуживать смешанный способ (старого с новым) задания

функций. По возможности его следует избегать.

Порядок вычисления аргументов не определяется, в разных компиляторах он различен. Однако гарантируется, что аргументы и именователь функции вычисляются полностью (включая и побочные эффекты) до входа в неё. Любая функция допускает рекурсивное обращение.

А.7.3.3. Ссылки на члены структуры

Постфиксное выражение, за которым стоит точка с последующим идентификатором, является постфиксным выражением. Выражение первого операнда должно быть структурой или объединением, а идентификатор – именем члена структуры или объединения. Значение – именованный член структуры или объединения, а тип значения – тип члена структуры или объединения. Выражение является l-значением, если первое выражение – l-значение и если тип второго выражения – не «массив».

Постфиксное выражение, за которым указана стрелка (составленная из знаков - и >) с последующим идентификатором, является постфиксным выражением. Выражение первого операнда должно быть указателем на структуру (объединение), а идентификатор – именем члена структуры (объединения). Результат – именованный член структуры (объединения), на которую ссылается указатель, а тип значения – тип члена структуры (объединения); результат – l-значение, если тип не есть «массив».

Таким образом, выражение `E1->MOS` означает то же самое, что и выражение `(*E1).MOS`. Структуры и объединения рассматриваются в А.8.3.

> В первом издании книги уже было приведено правило, по которому имя члена должно принадлежать структуре или объединению, упомянутому в постфиксном выражении. Там, однако, оговаривалось, что оно не является строго обязательным. Последние компиляторы и ANSI делают его обязательным.

А.7.3.4. Постфиксные инкрементирование и декрементирование

Постфиксное выражение, за которым следует ++ или --, есть постфиксное выражение. Значением такого выражения является значение его операнда. После того как значение было взято, операнд увеличивается (++) или уменьшается (--) на 1. Операнд должен быть l-значением; информация об ограничениях, накладываемых на операнд, и деталях операций содержится в А.7.7, где обсуждаются аддитивные операторы, и в А.7.17, где рассматривается присваивание. Результат инкрементирования или декрементирования не есть l-значение.

А.7.4. Унарные операторы

Выражения с унарными операторами выполняются справа налево.

```
унарное-выражение:
    постфиксное-выражение
    ++ унарное-выражение
    -- унарное-выражение
    унарный-оператор  выражение-приведённое-к-типу
    sizeof  унарное-выражение
    sizeof  ( имя-типа )

унарный-оператор: один из
    &   *   +   -   ~   !
```

А.7.4.1. Префиксные инкрементирование и декрементирование

Унарное выражение, перед которым стоит ++ или --, есть унарное выражение. Операнд увеличивается (++) или уменьшается (--) на 1. Значением выражения является значение его операнда после увеличения (уменьшения). Операнд всегда – l-значение; информация об ограничениях на операнд и деталях операции содержится в А.7.7, где обсуждаются аддитивные операторы, и в А.7.17, где рассматривается присваивание. Результат инкрементирования и декрементирования не есть l-значение.

А.7.4.2. Оператор получения адреса

Унарный оператор `&` обозначает операцию получения адреса своего операнда. Операнд должен быть либо l-значением, не ссылающимся ни на поле битов, ни на объект, объявленный как `register`, либо иметь тип «функция». Результат – указатель на объект (или функцию), адресуемый этим l-значением. Если тип операнда есть T, то типом результата является «указатель на T».

А.7.4.3. Оператор косвенности

Унарный оператор `*` обозначает операцию косвенности (раскрытия указателя), возвращающую объект (или функцию), на который указывает её операнд. Результат есть l-значение, если операнд – указатель на объект арифметического типа или на объект типа «структура», «объединение» или «указатель». Если тип выражения – «указатель на T», то тип результата – T.

А.7.4.4. Оператор унарный плюс

Операнд унарного + должен иметь арифметический тип, результат – значение операнда. Целочисленный операнд подвергается повышению целочисленности. Типом результата является повышенный тип операнда.

Унарный + был добавлен для симметрии с унарным –.

А.7.4.5. Оператор унарный минус

Операнд для унарного – должен иметь арифметический тип, результат – значение операнда с противоположным знаком. Целочисленный операнд подвергается повышению целочисленности. Отрицательное значение от беззнаковой величины вычисляется вычитанием приведённого к повышенному типу операнда из максимального числа повышенного типа, увеличенного на 1; однако минус нуль есть нуль. Типом результата будет повышенный тип операнда.

А.7.4.6. Оператор обращения разрядов

Операнд оператора ~ должен иметь целочисленный тип, результат – дополнение операнда до единиц по всем разрядам. Выполняется повышение целочисленности типа операнда. Если операнд беззнаковый, то результат получается вычитанием его значения из самого большого числа повышенного типа. Если операнд знаковый, то результат вычисляется посредством приведения «повышенного операнда» к беззнаковому типу, выполнения операции ~ и обратного приведения его к знаковому типу. Тип результата – повышенный тип операнда.

А.7.4.7. Оператор логического отрицания

Операнд оператора ! должен иметь арифметический тип или быть указателем. Результат равен 1, если сравнение операнда с 0 даёт истину, и равен 0 в противном случае. Тип результата – `int`.

А.7.4.8. Оператор определения размера `sizeof`

Оператор `sizeof` даёт число байтов, требуемое для хранения объекта того типа, который имеет его операнд. Операнд – либо выражение (которое не вычисляется), либо имя типа, записанное в скобках. Применённый к `char` оператор `sizeof` даёт 1. Для массива результат равняется общему количеству байтов в массиве, для структуры или объединения – числу байтов в объекте, включая и байты-заполнители, которые понадобились бы, если бы из элементов составлялся массив. Размер массива из `n` элементов всегда равняется `n`, помноженному на размер отдельного его элемента. Данный оператор нельзя применять к операнду типа «функция», к незавершённому типу и к полю битов. Результат – беззнаковая целочисленная константа; конкретный её тип зависит от реализации. В стандартном головном файле `<stddef.h>` (см. приложение В) этот тип определяется под именем `size_t`.

А.7.5. Оператор приведения типа

Имя типа, записанное перед унарным выражением в скобках, вызывает приведение значения этого выражения к указанному типу.

```
выражение-приведённое-к-типу:
    унарное-выражение
    ( имя-типа ) выражение-приведённое-к-типу
```

Данная конструкция называется *приведением*. Имена типов даны в А.8.8. Результат преобразований описан в А.6. Выражение с приведением типа не является l-значением.

А.7.6. Мультипликативные операторы

Мультипликативные операторы `*`, `/` и `%` выполняются слева направо.

```
мультипликативное-выражение:
    выражение-приведённое-к-типу
    мультипликативное-выражение * выражение-приведённое-к-типу
    мультипликативное-выражение / выражение-приведённое-к-типу
    мультипликативное-выражение % выражение-приведённое-к-типу
```

Операнды операторов `*` и `/` должны быть арифметического типа, оператора `%` – целочисленного типа. Над операндами осуществляются обычные арифметические преобразования, которые приводят их значения к типу результата.

Бинарный оператор `*` обозначает умножение.

Бинарный оператор `/` получает частное, а `%` – остаток от деления первого операнда на второй; если второй операнд есть `0`, то результат не определён. В противном случае всегда выполняется соотношение: `(a/b)*b + a%b` равняется `a`. Если оба операнда не отрицательные, то остаток не отрицательный и меньше делителя; в противном случае гарантируется только, что абсолютное значение остатка меньше абсолютного значения делителя.

А.7.7. Аддитивные операторы

Аддитивные операторы `+` и `–` выполняются слева направо. Если операнды имеют арифметический тип, то осуществляются обычные арифметические преобразования. Для каждого оператора существует ещё несколько дополнительных сочетаний типов.

```
аддитивное-выражение:
    мультипликативное-выражение
    аддитивное-выражение + мультипликативное-выражение
    аддитивное-выражение – мультипликативное-выражение
```

Результат выполнения оператора `+` есть сумма его операндов. Указатель на объект в массиве можно складывать с целочисленным значением. При

этом последнее преобразуется в адресное смещение посредством умножения его на размер объекта, на который ссылается указатель. Сумма является указателем на объект того же типа; только ссылается этот указатель на другой объект того же массива, отстоящий от первоначального соответственно вычисленному смещению. Так, если `P` – указатель на объект в массиве, то `P+1` – указатель на его следующий объект. Если полученный в результате суммирования указатель выводит за границы массива, то за исключением случая, когда он ссылается на место, находящееся непосредственно за концом массива, результат будет неопределённым.

> Возможность для указателя ссылаться на точку, расположенную сразу за концом массива, является новой. Она узаконивает общепринятую практику организации циклического перебора элементов массива.

Результат выполнения оператора – есть разность операндов. Из указателя можно вычитать значение 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 оператор вычитания, то в результате получится целочисленное значение со знаком, представляющее собой расстояние между объектами, на которые ссылаются эти указатели; указатель на следующий объект на 1 больше указателя на предыдущий объект. Тип результата зависит от реализации; в стандартном головном файле `<stddef.h>` он определён под именем `ptrdiff_t`. Значение не определено, если указатели ссылаются на объекты не одного и того же массива; однако если `P` указывает на последний элемент массива, то `(P+1)-P` имеет значение, равное 1.

A.7.8. Операторы сдвига

Операторы сдвига `<<` и `>>` выполняются слева направо. Для обоих операторов каждый операнд должен иметь целочисленный тип, и каждый из них подвергается повышению целочисленности. Тип результата совпадает с повышенным типом левого операнда. Результат не определён, если правый операнд отрицателен или его значение превышает число битов в типе левого выражения или равно ему.

```
сдвиговое-выражение:
    аддитивное-выражение
    сдвиговое-выражение >> аддитивное-выражение
    сдвиговое-выражение << аддитивное-выражение
```

Значение `E1 << E2` равно значению `E1` (рассматриваемому как цепочка битов), сдвинутому влево на `E2` бит; при отсутствии переполнения такая операция эквивалентна умножению на 2^{E2}. Значение `E1 >> E2` равно значению `E1`, сдвинутому вправо на `E2` битовые позиции. Если `E1` беззнаковое или имеет неотрицательное значение, то правый сдвиг эквивалентен делению на 2^{E2},

в противном случае результат зависит от реализации.

А.7.9. Операторы отношения

Операторы отношения выполняются слева направо, однако это свойство едва ли может оказаться полезным; согласно грамматике языка выражение a<b<c трактуется так же, как (a<b)<c, а результат вычисления a<b всегда есть 0 или 1.

```
выражение-отношения:
    сдвиговое-выражение
    выражение-отношения < сдвиговое-выражение
    выражение-отношения > сдвиговое-выражение
    выражение-отношения <= сдвиговое-выражение
    выражение-отношения >= сдвиговое-выражение
```

Операторы: < (меньше), > (больше), <= (меньше или равно) и >= (больше или равно) – все выдают 0, если специфицируемое отношение ложно, и 1, если оно истинно. Тип результата – int. Над арифметическими операндами выполняются обычные арифметические преобразования. Можно сравнивать указатели на объекты одного и того же (без учёта квалификаторов) типа; результат будет зависеть от их относительного расположения в памяти. Допускается, однако, сравнение указателей на разные части одного и того же объекта: если два указателя ссылаются на один и тот же простой объект, то они равны; если они ссылаются на члены одной структуры, то указатель на член с более поздней декларацией в структуре больше; если указатели ссылаются на члены одного итого же объединения, то они равны; если указатели ссылаются на элементы некоторого массива, то сравнение этих указателей эквивалентно сравнению их индексов. Если P указывает на последний элемент массива, то P+1 больше, чем P, хотя P+1 «выводит» нас за границы массива. В остальных случаях результат сравнения не определён.

> Эти правила несколько ослабили ограничения, установленные в первой редакции языка. Они позволяют сравнивать указатели на различные члены структуры и объединения и легализуют сравнение с указателем, ссылающимся на место, которое расположено непосредственно за концом массива.

А.7.10. Операторы равенства

```
выражение-равенства:
    выражение-отношения
    выражение-равенства == выражение-отношения
    выражение-равенства != выражение-отношения
```

Операторы == (равно) и != (не равно) являются аналогами операторов отношения с той лишь разницей, что они имеют более низкий приоритет. (Таким

образом, a<b == c<d есть 1 тогда и только тогда, когда отношения a<b и c<d оба одновременно истинны или ложны.)

Операторы равенства подчиняются тем же правилам, что и операторы отношения. И кроме того, они дают возможность сравнивать указатель с целочисленным константным выражением, значение которого равно нулю, и с указателем на void. (См. А.6.6.)

А.7.11. Оператор побитового И

```
И-выражение:
    выражение-равенства
    И-выражение & выражение-равенства
```

Выполняются обычные арифметические преобразования; результат – побитовое И операндов. Оператор применяется только к целочисленным операндам.

А.7.12. Оператор побитового исключающего ИЛИ

```
исключающее-ИЛИ-выражение:
    И-выражение
    исключающее-ИЛИ-выражение ^ И-выражение
```

Выполняются обычные арифметические преобразования; результат – побитовое исключающее ИЛИ операндов. Оператор применяется только к целочисленным операндам.

А.7.13. Оператор побитового ИЛИ

```
ИЛИ-выражение:
    исключающее-ИЛИ-выражение
    ИЛИ-выражение | исключающее-ИЛИ-выражение
```

Выполняются обычные арифметические преобразования; результат – побитовое ИЛИ операндов. Оператор применяется только к целочисленным операндам.

А.7.14. Оператор логического И

```
логическое-И-выражение:
    ИЛИ-выражение
    логическое-И-выражение && ИЛИ-выражение
```

Операторы && выполняются слева направо. Оператор && выдаёт 1, если оба операнда не равны нулю, и 0 в противном случае. В отличие от &, && гарантирует, что вычисления будут проводиться слева направо: вычисляется первый

операнд со всеми побочными эффектами; если он равен 0, то значение выражения есть 0. В противном случае вычисляется правый операнд, и, если он равен 0, то значение выражения есть 0, в противном случае оно равно 1.

Операнды могут принадлежать разным типам, но либо операнд должен иметь арифметический тип, либо быть указателем. Тип результата – `int`.

А.7.15. Оператор логического ИЛИ

```
логическое-ИЛИ-выражение:
    логическое-И-выражение
    логическое-ИЛИ-выражение || логическое-И-выражение
```

Операторы `||` выполняются слева направо. Оператор `||` выдаёт 1, если по крайней мере один из операндов не равен нулю, и 0 в противном случае. В отличие от `|`, оператор `||` гарантирует, что вычисления будут проводиться слева направо: вычисляется первый операнд, включая все побочные эффекты; если он не равен 0, то значение выражения есть 1. В противном случае вычисляется правый операнд, и, если он не равен 0, то значение выражения есть 1, в противном случае оно равно 0.

Операнды могут принадлежать разным типам, но либо операнд должен иметь арифметический тип, либо быть указателем. Тип результата – `int`.

А.7.16. Условный оператор

```
условное-выражение:
    логическое-ИЛИ-выражение
    логическое-ИЛИ-выражение ? выражение : условное-выражение
```

Вычисляется первое выражение, включая все побочные эффекты; если оно не равно 0, то результат есть 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 к некоторому общему типу, который и будет типом результата. Если оба операнда имеют тип `void`, или являются структурами или 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», этот же тип будет иметь и результат. Если один операнд является указателем на `void`, а второй – указатель другого типа, то последний преобразуется в указатель на `void`, который и будет типом результата.

При сравнении типов указателей квалификаторы типов (А.8.2) объектов, на которые указатели ссылаются, во внимание не принимаются, но тип результата наследует квалификаторы обеих ветвей условного выражения.

А.7.17. Выражения присваивания

Существует несколько операторов присваивания; они выполняются справа налево.

```
выражение-присваивания:
    условное-выражение
    унарное-выражение оператор-присваивания выражение-присваивания

оператор-присваивания: один из
    =   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
```

Операторы присваивания в качестве левого операнда требуют l-значения, причём модифицируемого; это значит, что оно не может быть массивом, или иметь незавершённый тип, или быть функцией. Тип левого операнда, кроме того, не может иметь квалификатор `const`; и, если он является структурой или объединением, в них не должно быть членов или подчленов (для вложенных структур или объединений) с квалификаторами `const`. Тип выражения присваивания – тип его левого операнда, а значение – значение его левого операнда после завершения присваивания.

В простом присваивании с оператором = значение выражения замещает объект, на который ссылается l-значение. При этом должно выполняться одно из следующих условий: оба операнда имеют арифметический тип (если типы операндов разные, правый операнд приводится к типу левого операнда); оба операнда есть структуры или объединения одного и того же типа; один операнд есть 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` у правого операнда).

Выражение `E1` *`op`* `= E2` эквивалентно выражению `E1 = E1` *`op`* `(E2)` с одним исключением: `E1` вычисляется только один раз.

А.7.18. Оператор запятая

```
выражение:
    выражение-присваивания
    выражение , выражение-присваивания
```

Два выражения, разделённые 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 контексте, в котором запятая имеет специальное значение, например в списках аргументов функций (А.7.3.2) или в списках инициализаторов (А.8.7) (здесь в качестве синтаксических единиц фигурируют выражения присваивания), оператор запятая может появиться только в группирующих скобках. Например, в

```
f(a, (t=3, t+2), c)
```

три аргумента, из которых второй имеет значение 5.

A.7.19. Константные выражения

Синтаксически, константное выражение – это выражение с ограниченным подмножеством операторов:

константное-выражение:
    условное-выражение

При указании case-меток в переключателе, задании границ массивов и длин полей битов, на месте значений перечислимых констант и инициализаторов, а также в некоторых выражениях для препроцессора требуются выражения, вычисление которых приводит к константе.

Константные выражения не могут содержать присваиваний, операторов инкрементирования и декрементирования, вызовов функций и операторов-запятых; перечисленные ограничения не распространяются на операнд `sizeof`. Если требуется получить целочисленное константное выражение, то его операнды должны состоять из целых, перечислимых, литерных и плавающих констант; операции приведения должны специфицировать целочисленный тип, а любая плавающая константа – приводиться к целому. Из этого следует, что в константном выражении не может быть массивов, операций раскрытия указателя, получения адреса и доступа к полям структуры. (Однако для `sizeof` возможны операнды любого вида.)

Для константных выражений в инициализаторах допускается бóльшая свобода; операндами могут быть константы любого типа, а к внешним или статическим объектам и внешним и статическим массивам, индексируемым константными выражениями, возможно применять унарный оператор `&`. Унарный оператор `&` может также неявно «присутствовать» при использовании массива без индекса или функции без списка аргументов. Вычисление инициализатора должно давать константу или адрес ранее декларированного внешнего или статического объекта плюс-минус константа.

Меньшая свобода допускается для целочисленных константных выражений, используемых после `#if`: не разрешаются `sizeof`-выражения, перечислимые константы и операции приведения типа. (См. A.12.5.)

A.8. Декларации

То, каким образом интерпретируется каждый идентификатор, специфицируется декларациями; они не всегда резервируют память для описываемых ими идентификаторов. Декларации, резервирующие память, называются *определениями* и имеют следующий вид:

```
декларация:
    спецификаторы-декларации список-иниц-деклараторов_необ ;
```

Декларатор в *список-иниц-деклараторов* содержат декларируемые идентификаторы; *спецификаторы-декларации* представляют собой последовательности, состоящие из спецификаторов типа и класса памяти.

```
спецификаторы-декларации:
    спецификатор-класса-памяти спецификаторы-декларации_необ
    спецификатор-типа спецификаторы-декларации_необ
    квалификатор-типа спецификаторы-декларации_необ

список-иниц-деклараторов:
    иниц-декларатор
    список-иниц-деклараторов , иниц-декларатор
иниц-декларатор:
    декларатор
    декларатор = инициализатор
```

Декларатор содержат подлежащие описанию имена. Мы рассмотрим их позже, в А.8.5. Либо декларация должна иметь по крайней мере один декларатор, либо её спецификатор типа должен определять тег структуры или объединения, либо – задавать члены перечисления; пустая декларация незаконна.

А.8.1. Спецификаторы класса памяти

Класс памяти специфицируется следующим образом:

```
спецификатор-класса-памяти:
    auto
    register
    static
    extern
    typedef
```

Смысл классов памяти обсуждался в А.4.

Спецификаторы `auto` и `register` дают декларируемым объектам класс автоматической памяти, и эти спецификаторы можно применять только внутри функций. Декларации с `auto` и `register` одновременно являются определениями и резервируют память. Спецификатор `register` эквивалентен `auto`, но содержит подсказку, сообщающую, что в программе декларируемые им объекты используются интенсивно. На регистрах может быть размещено лишь небольшое число объектов, причём определённого типа; указанные ограничения зависят от реализации. В любом случае к `register`-объекту нельзя применять (явно или неявно) унарный оператор `&`.

> Новым является правило, согласно которому вычислять адрес объекта класса `register` нельзя, а класса `auto` можно.

Спецификатор `static` даёт декларируемым объектам класс статической памяти, он может использоваться и внутри, и вне функций. Внутри функции этот спецификатор вызывает выделение памяти и служит определением; его роль вне функций будет объяснена в А.11.2.

Декларация со спецификатором `extern`, используемая внутри функции, объявляет, что для декларируемого объекта где-то выделена память; о её роли вне функций будет сказано в А.11.2.

Спецификатор `typedef` не резервирует никакой памяти и назван спецификатором класса памяти из соображений стандартности синтаксиса; речь об этом спецификаторе пойдёт в А.8.9.

Декларация может содержать не более одного спецификатора класса памяти. Если он в декларации отсутствует, то действуют следующие правила: считается, что объекты, декларируемые внутри функций, имеют класс `auto`; функции, декларируемые внутри функций, – класс `extern`; объекты и функции, декларируемые вне функций, – статические и имеют внешние связи (А.10, А.11).

А.8.2. Спецификаторы типа

Спецификаторы типа определяются следующим образом:

спецификатор-типа:
- `void`
- `char`
- `short`
- `int`
- `long`
- `float`
- `double`
- `signed`
- `unsigned`
- *структ-или-объед-спецификатор*
- *епит-спецификатор*
- *typedef-имя*

Вместе с `int` допускается использование ещё какого-то одного слова – `long` или `short`; причём сочетание `long int` имеет тот же смысл, что и просто `long`; аналогично `short int` – то же самое, что и `short`. Слово `long` может употребляться вместе с `double`. С `int` и другими его модификациями (`short`, `long` или `char`) разрешается употреблять одно из слов `signed` или `unsigned`. Любое из последних может использоваться самостоятельно, в этом случае подразумевается `int`. Спецификатор `signed` бывает полезен, когда требуется обеспечить, чтобы `char`-объекты имели знак; его можно применять и к другим целочисленным типам, но в этих случаях он избыточен.

За исключением описанных выше случаев декларация не может содержать более одного спецификатора типа. Если в декларации нет ни одного

спецификатора типа, то имеется в виду тип int.

Для указания особых свойств декларируемых объектов предназначаются квалификаторы:

квалификатор-типа:
    const
    volatile

Квалификаторы типа могут употребляться с любым спецификатором типа. const-объект разрешается инициализировать, однако присваивать ему что-либо в дальнейшем запрещается. Смысл квалификатора volatile зависит от реализации.

> Средства const и volatile (изменчивый) введены ANSI-стандартом. Квалификатор const применяется, чтобы разместить объекты в памяти, открытой только на чтение, или чтобы способствовать возможной оптимизации. Назначение квалификатора volatile – подавить оптимизацию, которая без этого указания могла бы возникнуть. Например, в машинах, где адреса регистров ввода-вывода отображены на адресное пространство памяти, указатель на регистр некоторого устройства мог бы быть декларирован как volatile, чтобы запретить компилятору экономить очевидно избыточную ссылку через указатель. Компилятор может игнорировать указанные квалификаторы, однако обязан сигнализировать о явных попытках изменить значение const-объектов.

А.8.3. Декларации структур и объединений

Структура – это объект, состоящий из последовательности именованных членов различных типов. Объединение – объект, который в каждый момент времени содержит один из нескольких членов различных типов. Декларации структур и объединений имеют один и тот же вид.

структ-или-объед-спецификатор:
    структ-или-объед идентификатор$_{необ}$ { *список-структ-деклараций* }
    структ-или-объед идентификатор

структ-или-объед:
    struct
    union

Список-структ-деклараций является последовательностью деклараций членов структуры или объединения:

список-структ-деклараций:
    структ-декларация
    список-структ-деклараций структ-декларация

```
struкт-декларация:
    список-спецификаторов-квалификаторов список-структ-деклараторов;

список-спецификаторов-квалификаторов:
    спецификатор-типа список-спецификаторов-квалификаторов_необ
    квалификатор-типа список-спецификаторов-квалификаторов_необ

список-структ-деклараторов:
    структ-декларатор
    список-структ-деклараторов , структ-декларатор
```

Обычно *структ-декларатор* – это просто декларатор члена структуры или объединения. Членом структуры может быть также некоторая последовательность битов заданной длины. Такой член называется *полем-бит* или просто *полем*; указатель его длины отделяется от декларатора имени поля двоеточием.

```
структ-декларатор:
    декларатор
    декларатор_необ : константное-выражение
```

Спецификатор типа, имеющий вид

```
структ-или-объед идентификатор { список-структ-деклараций }
```

декларирует *идентификатор* как *тег* структуры (объединения), задаваемый списком. Последующие декларации в той же или более внутренней области действия могут ссылаться в спецификаторе на этот тип при помощи тега (без списка):

```
структ-или-объед идентификатор
```

Если спецификатор с тегом без списка обнаружен в тот момент, когда тег ещё не декларирован, то считается, что специфицирован *незавершённый тип*. На объект с незавершённым типом структуры (объединения) можно ссылаться в контексте, где не нужен его размер, например, в декларациях (но не в определениях), в спецификации указателя или при задании `typedef`, но ни в каких других случаях. Тип становится завершённым при появлении спецификатора с тегом структуры или объединения, содержащего список деклараций. Но и в спецификаторах со списком определяемые ими типы внутри списка остаются незавершёнными до тех пор, пока не встретится завершающая спецификатор фигурная скобка.

В структуру не могут входить члены с незавершённым типом. Следовательно, нельзя декларировать структуру (объединение), содержащую саму себя. Однако, помимо того что теги являются именами типов структур или объединений, они позволяют определять структуры (объединения), ссылающиеся сами на себя; структура (объединение) может содержать указатель на объект этой же структуры (объединения), поскольку указатели с незавершёнными типами декларировать разрешено.

В отношении деклараций вида

структ-или-объед идентификатор ;

которые объявляют структуру или объединение, однако не имеют ни списка, ни деклараторов, действует специальное правило. Даже если этот идентификатор уже был декларирован во внешней области действия (A.11.1) в качестве тега структуры или объединения, в текущей области действия указанная декларация делает его тегом нового незавершённого типа для структуры или объединения.

> Это «неясное» правило введено ANSI-стандартом. Оно позволяет во внутренней области действия программы создавать «взаимно-рекурсивные» декларации структур с тегами, которые уже могли быть объявлены во внешней области действия.

Спецификатор структуры или объединения со списком, но без тега, представляет собой уникальный тип; на него можно сослаться только в той декларации, частью которой он является.

Имена членов и тегов не «конфликтуют» ни между собой, ни с именами обычных переменных. Имя члена нельзя употребить дважды в одной и той же структуре или объединении, но одно и то же имя можно использовать в различных структурах и объединениях.

> В первом издании этой книги имена членов не были локализованы в своей структуре или объединении. Однако такая локализация стала общеупотребительной ещё до принятия ANSI-стандарта для Си.

Член (не поле) может быть объектом любого типа. Поле (которое может и не иметь декларатора и, следовательно, быть без имени) имеет тип `int`, `unsigned int` или `signed int` и интерпретируется как объект целочисленного типа с заданным числом битов; рассматривается ли поле типа `int` как знаковое, зависит от реализации. Также от реализации зависит способ и порядок «упаковки» соседних полей в памяти. Если поле, следующее за другим полем, не помещается в частично заполненной ячейке памяти, то либо оно будет разрезано на части, которые попадут в разные ячейки, либо в структуре возникнут пропуски из неиспользуемых разрядов. Поле без имени длины 0 предписывает пропустить оставшиеся разряды и следующее поле расположить с начала ячейки памяти.

> В сравнении с первым изданием ANSI-стандарт устанавливает ещё большую зависимость полей от реализации. Разумно правила языка, касающиеся размещения полей битов, рассматривать без каких-либо оговорок как «реализационно-зависимые». Структуры с полями битов полезны и как универсальный способ минимизации памяти, требуемой под структуры, возможно, за счёт увеличения количества команд и замедления доступа к полям, и как способ описания побитового расположения информации. Во втором случае необходимо знать особенности конкретной реализации.

Члены структуры имеют адреса, возрастающие в порядке написания их деклараций. Член структуры (не поле) выравнивается по адресуемым границам слов в соответствии со своим типом; следовательно, в структуре могут возникать безымянные «дыры». Если указатель на структуру привести к типу указателя на её первый член, то в результате получится указатель на первый член структуры.

Объединение можно считать как бы структурой, все члены которой имеют смещение 0 от её начала, и которая обладает достаточным размером, чтобы вмещать любой из своих членов. В каждый отдельный момент объединение может хранить значение только одного своего члена. Если указатель на объединение привести к типу какого-то из его членов, то в результате получится указатель на этот член.

Вот простой пример декларации структуры, которая содержит массив из 20 литер, целое и две ссылки на такие же структуры.

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
};
```

Если после указанной декларации поместить запись

```
struct tnode s, *sp;
```

то она будет определять `s` как структуру заданного вида, а `sp` – как указатель на такую структуру. Согласно приведённым определениям выражение

```
sp->count
```

есть ссылка на член `count` в структуре, на которую указывает `sp`;

```
s.left
```

– указатель на левое поддерево в структуре `s`; а

```
s.right->tword[0]
```

есть первая литера из `tword` – члена правого поддерева `s`.

Вообще говоря, нет способа проконтролировать, используется тот ли член объединения, которому последний раз присваивалось значение. Однако гарантируется выполнение правила, облегчающего работу с членами 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 на общую часть данных разрешается ссылаться через любую из указанных структур. Так, правомерен следующий фрагмент программы:

```
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. Перечисления

Перечисления – это уникальный тип, значения которого покрываются множеством именованных констант, называемых перечислителями. Вид спецификатора перечисления заимствован у структур и объединений.

перец-спецификатор:
enum *идентификатор*$_{необ}$ { *список-перечислителей* }
enum *идентификатор*

список-перечислителей:
перечислитель
список-перечислителей , *перечислитель*

перечислитель:
идентификатор
идентификатор = *константное-выражение*

Идентификаторы, входящие в список перечислителей, объявляются константами типа int и могут употребляться везде, где требуется константа. Если в этом списке нет ни одного перечислителя со знаком =, то значения констант начинаются с 0 и увеличиваются на 1 по мере чтения декларации слева направо. Перечислитель со знаком = даёт соответствующему идентификатору значение; последующие идентификаторы продолжают прогрессию от заданного значения.

Имена перечислителей, используемые в одной области действия, должны отличаться друг от друга и от имён обычных переменных, однако их значения могут и совпадать.

Роль идентификатора в *переч-спецификаторе* аналогична роли тега структуры в *структ-спецификаторе*: он является именем некоторого конкретного перечисления. Правила для списков и *переч-спецификаторов* (с тегами и без) те же, что и для спецификаторов структур или объединений, с той лишь оговоркой, что элементы перечислений не бывают незавершённого типа; тег *переч-спецификатора* без списка перечислителей должен отсылать в пределах области действия на спецификатор со списком.

> В первом издании языка перечислений не было, но они уже несколько лет применяются.

А.8.5. Декларатор

Декларатор имеют следующий синтаксис:

```
декларатор:
    указатель_необ собственно-декларатор

собственно-декларатор:
    идентификатор
    ( декларатор )
    собственно-декларатор [ константное-выражение_необ ]
    собственно-декларатор ( список-типов-параметров )
    собственно-декларатор ( список-идентификаторов_необ )

указатель:
    * список-квалификаторов-типа_необ
    * список-квалификаторов-типа_необ указатель

список-квалификаторов-типа:
    квалификатор-типа
    список-квалификаторов-типа квалификатор-типа
```

У структуры декларатора много сходных черт со структурой подвыражений, поскольку в деклараторе, как и в подвыражении, допускаются операции раскрытия указателя, обращения к функции и получения элемента массива (с тем же порядком применения).

А.8.6. Что означают декларатор

Список деклараторов располагается сразу после спецификаторов типа и указателя класса памяти. Главный элемент любого декларатора – это объявляемый им идентификатор; в простейшем случае декларатор из одного его и состоит, что отражено в первой строке продукции грамматики с именем

собственно-декларатор. Спецификаторы класса памяти относятся непосредственно к идентификатору, а его тип зависит от вида декларатора. Декларатор следует воспринимать как утверждение: если в выражении идентификатор появляется в том же контексте, что и в декларатore, то он обозначает объект специфицируемого типа.

Если соединить спецификаторы декларации, относящиеся к типу (А.8.2), и некоторый конкретный декларатор, то декларация примет следующий вид: «`T D`», где `T` – тип, а `D` – декларатор. Эта запись описывает тип для идентификатора любого декларатора индуктивно.

В декларации `T D`, где `D` – просто идентификатор, тип идентификатора есть `T`.

В декларации `T D`, где `D` имеет вид

```
( D1 )
```

тип идентификатора в `D1` тот же, что и в `D`. Скобки не изменяют тип, но могут повлиять на результаты его «привязки» к идентификаторам в сложных декларатorах.

А.8.6.1. Декларатоpы указателей

В декларации `T D`, где `D` имеет вид

```
* список-квалификаторов-типанеоб D1
```

а тип идентификатора декларации `T D1` есть «*модификатор-типа* `T`», тип идентификатора `D` есть «*модификатор-типа список-квалификаторов-типа* указатель на `T`». Квалификаторы, следующие за `*`, относятся к самому указателю, а не к объекту, на который он указывает.

Рассмотрим, например, декларацию

```
int *ap[];
```

Здесь `ap[]` играет роль `D1`; декларацию «`int ap[]`» следует расшифровать (см. ниже) как «массив из `int`»; список квалификаторов типа здесь пуст, а модификатор типа есть «массив из». Следовательно, на самом деле декларация `ap` гласит: «массив из указателей на `int`».

Вот ещё примеры деклараций:

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

В них объявляются целое `i` и указатель на целое `pi`. Значение указателя `cpi` неизменно; `cpi` всегда будет указывать в одно и то же место, даже если значение, на которое он ссылается, станет иным. Целое `ci` есть константа, оно измениться не может (хотя может инициализироваться, как в данном случае). Тип указателя `pci` произносится как «указатель на `const int`»; сам указатель возможно изменить; при этом он будет ссылаться на другое место, но значение, на которое он будет указывать, посредством `pci` изменить нельзя.

A.8.6.2. Деклараторы массивов

В декларации `T D`, где `D` имеет вид

`D1 [` *константное-выражение*$_{необ}$ `]`

и где тип идентификатора декларации `T D1` есть «*модификатор-типа* `T`», тип идентификатора `D` есть «*модификатор-типа* массив из `T`». Если константное выражение присутствует, то оно должно быть целочисленным и больше 0. Если константное выражение, специфицирующее количество элементов в массиве, отсутствует, то массив имеет незавершённый тип.

Массив можно конструировать из объектов арифметического типа, указателей, структур и объединений, а также других массивов (генерируя при этом многомерные массивы). Любой тип, из которого конструируется массив, должен быть завершённым, он не может быть, например, структурой или массивом незавершённого типа. Это значит, что для многомерного массива пустой может быть только первая размерность. Незавершённый тип массива получает своё завершение либо в другой декларации этого массива (A.10.2), либо при его инициализации (A.8.7). Например, запись

```
float fa[17], *afp[17];
```

декларирует массив из `float`-чисел и массив из указателей на `float`-числа. Аналогично

```
static int x3d[3][5][7];
```

декларирует статический трёхмерный массив целых размера $3 \times 5 \times 7$. На самом деле, если быть точными, `x3d` является массивом из трёх элементов, каждый из которых есть массив из пяти элементов, содержащих по 7 целых.

Операция индексирования `E1[E2]` определена так, что она идентична операции `*(E1+E2)`. Следовательно, несмотря на асимметричность записи, индексирование – коммутативная операция. Учитывая правила преобразования, применяемые для оператора `+` и массивов (A.6.6, A.7.1, A.7.7), можно сказать, что если `E1` – массив, а `E2` – целое, то `E1[E2]` обозначает `E2`-й элемент массива `E1`.

Так, `x3d[i][j][k]` означает то же самое, что и `*(x3d[i][j] + k)`. Первое подвыражение, `x3d[i][j]`, согласно A.7.1 приводится к типу «указатель на массив целых»; по A.7.7 сложение включает умножение на размер объекта типа `int`. Из этих же правил следует, что массивы запоминаются «построчно» (последние индексы меняются чаще) и что первая размерность в декларации помогает определить количество памяти, занимаемой массивом, однако в вычислении адреса элемента массива участия не принимает.

A.8.6.3. Деклараторы функций

При новом способе декларация функции `T D`, где `D` имеет вид

`D1(` *список-типов-параметров*`)`

и тип идентификатора декларации `T D1` есть «*модификатор-типа* `T`», тип идентификатора в `D` есть «*модификатор-типа* функция с аргументами *список-типов-параметров*, возвращающая `T`».

Параметры имеют следующий синтаксис:

список-типов-параметров:
    список-параметров
    список-параметров `, ...`

список-параметров:
    декларация-параметра
    список-параметров `,` *декларация-параметра*

декларация-параметра:
    спецификаторы-декларации декларатор
    спецификаторы-декларации абстрактный-декларатор$_{необ}$

При новом способе описания функций список параметров специфицирует их типы, а если функция вообще не имеет параметров, на месте списка типов указывается одно слово – `void`. Если список типов параметров заканчивается многоточием «`, ...`», то функция может иметь больше аргументов, чем число явно описанных параметров. (См. А.7.3.2.)

Типы параметров, являющихся массивами и функциями, заменяются на указатели в соответствии с правилами преобразования параметров (А.10.1). Единственный спецификатор класса памяти, который разрешён в декларации параметра, – это `register`, однако он игнорируется, если декларатор функции не является заголовком её определения. Аналогично, если деклараторы в декларациях параметров содержат идентификаторы, а декларатор функции не является заголовком определения функции, то эти идентификаторы тотчас же выводятся из текущей области действия.

При старом способе декларация функции `T D`, где `D` имеет вид

`D1(`*список-идентификаторов*$_{необ}$`)`

и тип идентификатора декларации `T D1` есть «*модификатор-типа* `T`», тип идентификатора в `D` есть «*модификатор-типа* функция от неспецифицированных аргументов, возвращающая `T`». Параметры, если они есть, имеют следующий вид:

список-идентификаторов:
    идентификатор
    список-идентификаторов `,` *идентификатор*

При старом способе, если декларатор функции не используется в качестве заголовка определения функции (А.10.1), список идентификаторов должен отсутствовать. Никакой информации о типах параметров в декларации не содержится.

Например, декларация

```
int f(), *fpi(), (*pfi)();
```

описывает функцию `f`, возвращающую целое, функцию `fpi`, возвращающую указатель на целое, и указатель `pfi` на функцию, возвращающую целое. Ни для одной функции в декларации не указаны типы параметров; все функции описаны старым способом.

Вот как выглядит декларация в новой записи:

```
int strcpy(char *dest, const char *source), rand(void);
```

Здесь `strcpy` – функция с двумя аргументами, возвращающая значение типа `int`; первый аргумент – указатель на значение типа `char`, а второй – указатель на неизменяющееся значение типа `char`. Имена параметров играют роль хороших комментариев. Вторая функция, `rand`, аргументов не имеет и возвращает `int`.

Прототипы деклараторов функций с параметрами – наиболее важное нововведение ANSI-стандарта. В сравнении со старым способом, принятым в первой редакции языка, они позволяют проверять и приводить к нужному типу аргументы во всех вызовах. Следует однако отметить, что их введение привнесло в язык некоторую сумятицу и необходимость согласования обеих форм. Чтобы обеспечить совместимость, потребовались некоторые «синтаксические уродства» типа `void` для явного указания на отсутствие параметров.

Многоточие «`, ...`» применительно к функциям с варьируемым числом аргументов – также новинка, которая вместе со стандартным головным файлом макросов `<stdarg.h>` формализует неофициально используемый, но официально запрещённый в первой редакции механизм.

Указанные способы записи заимствованы из языка Си++.

А.8.7. Инициализация

При помощи *иниц-декларатора* можно указать начальное значение декларируемого объекта. Инициализатору, представляющему собой выражение или список инициализаторов, заключённый в фигурные скобки, предшествует знак `=`. Этот список может завершаться запятой; её назначение – сделать форматирование более чётким.

```
инициализатор:
    выражение-присваивания
    { список-инициализаторов }
    { список-инициализаторов , }

список-инициализаторов:
    инициализатор
    список-инициализаторов , инициализатор
```

В инициализаторе статического объекта или массива все выражения должны быть константными (A.7.19). Если инициализатор `auto`- и `register`-объекта или массива находится в списке, заключённом в фигурные скобки, то входящие в него выражения также должны быть константными. Однако в случае автоматического объекта с одним выражением инициализатор не обязан быть константным выражением, он просто должен иметь тип, соответствующий объекту.

> В первой редакции не разрешалась инициализация автоматических структур, объединений и массивов. ANSI-стандарт позволяет это; однако, если инициализатор не может быть представлен одним простым выражением, инициализация может быть выполнена только при помощи константных конструкций.

Статический объект, инициализация которого явно не указана, инициализируется так, как если бы ему (или его членам) присваивалась константа 0. Начальное значение автоматического объекта, явным образом не инициализированного, не определено.

Инициализатор указателя или объекта арифметического типа – это единичное выражение (возможно, заключённое в фигурные скобки), которое присваивается объекту.

Инициализатор структуры – это либо выражение того же структурного типа, либо заключённые в фигурные скобки инициализаторы её членов, заданные по порядку. Безымянные поля битов игнорируются и не инициализируются. Если инициализаторов в списке меньше, чем членов, то оставшиеся члены инициализируются нулём. Инициализаторов не должно быть больше числа членов.

Инициализатор массива – это список инициализаторов его членов, заключённый в фигурные скобки. Если размер массива не известен, то он считается равным числу инициализаторов, при этом тип его становится завершённым. Если размер массива известен, то число инициализаторов не должно превышать числа его членов; если инициализаторов меньше, оставшиеся члены обнуляются.

Как особый выделен случай инициализации массива литер. Последний можно инициализировать при помощи стрингового литерала; литеры инициализируют члены массива в том порядке, в каком они заданы в стринговом литерале. Точно так же, при помощи литерала из расширенного набора литер (A.2.6), можно инициализировать массив типа `wchar_t`. Если размер массива не известен, то он определяется числом литер стринга, включающим и завершающую `null`-литеру; если размер массива известен, то число литер стринга, не считая завершающей `null`-литеры, не должно превышать его размера.

Инициализатором объединения может быть либо выражение того же типа, либо заключённый в фигурные скобки инициализатор его первого члена.

> В первой редакции не позволялось инициализировать объединения. Правило «первого члена» не отличается изяществом, однако не требует нового синтаксиса. Стандарт ANSI проясняет ещё и семантику не инициализируемых явно объединений.

Введём для структуры и массива обобщённое имя: *агрегат*. Если агрегат содержит члены агрегатного типа, то правила инициализации применяются рекурсивно. Фигурные скобки в некоторых случаях инициализации можно опускать. Если инициализатор члена агрегата, который сам является агрегатом, начинается с левой фигурной скобки, то этот подагрегат инициализируется последующим списком разделённых запятыми инициализаторов; считается ошибкой, если количество инициализаторов подагрегата превышает число его членов. Если, однако, инициализатор подагрегата не начинается с левой фигурной скобки, то, чтобы его инициализировать, нужно отсчитать соответствующее число элементов из списка; при этом остальные члены инициализируются следующими инициализаторами агрегата, для которого данный подагрегат является частью.

Например,

```
int x[] = { 1, 3, 5 };
```

декларирует и инициализирует `x` как одномерный массив с тремя членами, поскольку размера указано не было, а список состоит из трёх инициализаторов.

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

представляет собой инициализацию с полным набором фигурных скобок: 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. В точности тот же результат был бы достигнут при помощи следующей декларации:

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

Инициализатор для `y` начинается с левой фигурной скобки, но для `y[0]` скобки нет, поэтому из списка будут взяты три элемента. Аналогично по три элемента будут взяты для `y[1]`, а затем и для `y[2]`. В

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

инициализируется первый столбец матрицы y, все же другие элементы остаются нулевыми.

Наконец,

```
char msg[] = "Синтаксическая ошибка в строке %s\n";
```

представляет собой пример массива литер, члены которого инициализируются при помощи стринга; в его размере учитывается и завершающая null-литера.

A.8.8. Имена типов

В ряде случаев возникает потребность в применении имени типа данных (например, при явном приведении к типу, в указании типов параметров внутри деклараций функций, в аргументе оператора sizeof). Эта потребность реализуется при помощи *имени типа*, определение которого синтаксически почти совпадает с декларацией объекта того же типа. Оно отличается от последней лишь тем, что не содержит имени объекта.

имя-типа:
    список-спецификаторов-квалификаторов абстрактный-декларатор$_{необ}$

абстрактный-декларатор:
    указатель
    указатель$_{необ}$ *собственно-абстрактный-декларатор*

собственно-абстрактный-декларатор:
    (*абстрактный-декларатор*)
    собственно-абстрактный-декларатор$_{необ}$ [*константное-выражение*$_{необ}$]
    собственно-абстрактный-декларатор$_{необ}$ (*список-типов-параметров*$_{необ}$)

Можно указать одно-единственное место в абстрактном деклараторе, где мог бы оказаться идентификатор, если бы приведённая конструкция была полноценным декларатором. Именованный тип совпадает с типом этого «невидимого идентификатора». Например,

```
int
int *
int *[3]
int (*)[]
int *()
int (*[])(void)
```

соответственно обозначают типы «целое», «указатель на целое», «массив из трёх указателей на целое», «указатель на массив из неизвестного количества целых», «функция неизвестного количества параметров, возвращающая указатель на целое», «массив неизвестного количества указателей на функции без параметров, каждая из которых возвращает целое».

A.8.9. Декларация typedef

Декларации, в которых спецификатор класса памяти есть typedef, не декларируют объекты – они определяют идентификаторы, представляющие собой имена типов. Эти идентификаторы называются *typedef-именами*.

typedef-имя:
    идентификатор

Декларация typedef приписывает тип каждому имени своего декларатора обычным способом. (См. A.8.6.) С этого момента *typedef-имя* синтаксически эквивалентно ключевому слову спецификатора типа, обозначающему связанный с ним тип. Например, после

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

допустимы следующие декларации:

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

Тип b есть long, тип bp – «указатель на long», тип z – структура заданного вида, а zp – указатель на такую структуру.

Декларация typedef не вводит новых типов, она только даёт имена типам, которые могли бы быть специфицированы и другим способом. Например, b имеет тот же тип, что и любой другой long-объект.

Typedef-имена могут быть перекрыты другими определениями во внутренней области действия, но при условии, что в них присутствует указание типа. Например,

```
extern Blockno;
```

не переопределяет Blockno, а вот

```
extern int Blockno;
```

переопределяет.

A.8.10. Эквивалентность типов

Два списка спецификаторов типа эквивалентны, если они содержат одинаковый их набор с учётом синонимичности названий (например, long и long int считаются одинаковыми типами). Структуры, объединения и перечисления с разными тегами считаются разными, а каждое безтеговое объединение, структура или перечисление представляет собой уникальный тип.

Два типа считаются совпадающими, если их абстрактные декларторы (A.8.8) после замены всех *typedef-имён* их типами и выбрасывания имён параметров функций составят эквивалентные списки спецификаторов типов. При сравнении учитываются размеры массивов и типы параметров функций.

А.9. Инструкции

За исключением оговорённых случаев инструкции выполняются в том порядке, как они написаны. Инструкции не имеют значений и выполняются, чтобы произвести определённые действия. Все виды инструкций можно разбить на несколько групп:

инструкция:
    помеченная-инструкция
    инструкция-выражение
    составная-инструкция
    инструкция-выбора
    циклическая-инструкция
    инструкция-перехода

А.9.1. Помеченные инструкции

Инструкции может предшествовать метка.

помеченная-инструкция:
    идентификатор : *инструкция*
    case *константное-выражение* : *инструкция*
    default : *инструкция*

Метка, состоящая из идентификатора, одновременно служит и декларацией этого идентификатора. Единственное назначение идентификатора-метки – указать место перехода для goto. Областью действия идентификатора-метки является текущая функция. Так как метки имеют своё собственное пространство имён, они не «конфликтуют» с другими идентификаторами и не могут быть перекрыты. (См. А.11.1.)

case-метки и default-метки используются в инструкции switch (А.9.4). Константное выражение в case должно быть целочисленным.

Сами по себе метки не изменяют порядка вычислений.

А.9.2. Инструкция-выражение

Наиболее употребительный вид инструкции – это *инструкция-выражение*.

инструкция-выражение:
    выражение$_{необ}$;

Чаще всего *инструкция-выражение* – это присваивание или вызов функции. Все действия, реализующие побочный эффект выражения, завершаются, прежде чем начинает выполняться следующая инструкция. Если выражение в инструкции опущено, то она называется пустой; пустая инструкция часто используется для обозначения пустого тела циклической инструкции или в качестве места для метки.

А.9.3. Составная инструкция

Так как в местах, где по синтаксису полагается одна инструкция, иногда возникает необходимость выполнить несколько, предусматривается возможность задания составной инструкции (которую также называют блоком). Тело определения функции есть составная инструкция:

составная-инструкция:
{ *список-деклараций*$_{необ}$ *список-инструкций*$_{необ}$ }

список-деклараций:
декларация
список-деклараций декларация

список-инструкций:
инструкция
список-инструкций инструкция

Если идентификатор из списка деклараций находился в области действия объемлющего блока, то действие внешней декларации при входе внутрь данного блока приостанавливается (А.11.1), а после выхода из него возобновляется. Внутри блока идентификатор может быть декларирован только один раз. Для каждого отдельного пространства имён эти правила действуют независимо (А.11); идентификаторы из разных пространств имён всегда различны.

Инициализация автоматических объектов осуществляется при каждом входе в блок и продолжается по мере продвижения по декларатором. При передаче управления внутрь блока инициализации не выполняются. Инициализации статических объектов осуществляются только один раз перед запуском программы.

А.9.4. Инструкции выбора

Инструкции выбора осуществляют отбор одной из нескольких альтернатив, определяющих порядок выполнения инструкций.

инструкция-выбора:
if (*выражение*) *инструкция*
if (*выражение*) *инструкция* else *инструкция*
switch (*выражение*) *инструкция*

Оба вида `if`-инструкций содержат выражение, которое должно иметь арифметический тип или тип указателя. Сначала вычисляется выражение со всеми его побочными эффектами, результат сравнивается с 0. В случае несовпадения с 0 выполняется первая подинструкция. В случае совпадения с 0 для второго типа `if` выполняется вторая подинструкция. Связанная со словом `else` неоднозначность разрешается тем, что слово `else` соотносят с последней не имеющей `else` `if`-инструкцией, расположенной в одном с этим `else` блоке и на одном уровне вложенности блоков.

Инструкция switch вызывает передачу управления на одну из нескольких инструкций в зависимости от значения выражения, которое должно иметь целочисленный тип. Управляемая при помощи switch подинструкция обычно составная. Любая инструкция внутри этой подинструкции может быть помечена одной или несколькими case-метками (A.9.1). Управляющее выражение подвергается целочисленному повышению (A.6.1), а case-константы приводятся к повышенному типу. После такого преобразования никакие две case-константы в одной инструкции switch не должны иметь одинаковых значений. Со switch-инструкцией может быть связано не более одной default-метки. Конструкции switch допускается вкладывать друг в друга; case и default-метки относятся к самой внутренней switch-инструкции из тех, которые их содержат.

Инструкция switch выполняется следующим образом. Вычисляется выражение со всеми побочными эффектами, и результат сравнивается с каждой case-константой. Если одна из case-констант равна значению выражения, управление переходит на инструкцию с соответствующей case-меткой. Если ни с одной из case-констант нет совпадения, управление передаётся на инструкцию с default-меткой, если такая имеется, в противном случае ни одна из подинструкций switch не выполняется.

> В первой редакции требовалось, чтобы выражение и case-константы в switch были целого типа.

A.9.5. Циклические инструкции

Циклические инструкции специфицируют циклы.

циклическая-инструкция:
    while (*выражение*) *инструкция*
    do *инструкция* while (*выражение*)
    for ($выражение_{необ}$; $выражение_{необ}$; $выражение_{необ}$) *инструкция*

В инструкциях while и do выполнение подинструкции повторяется до тех пор, пока значение выражения не станет нулём. Выражение должно иметь арифметический тип или тип указателя. В while вычисление выражения со всеми побочными эффектами и проверка осуществляются перед каждым выполнением инструкции, а в do – после.

В инструкции for первое выражение вычисляется один раз, тем самым осуществляется инициализация цикла. На тип этого выражения никакие ограничения не накладываются. Второе выражение должно иметь арифметический тип или тип указателя; оно вычисляется перед каждой итерацией. Как только его значение становится равным 0, for прекращает свою работу. Третье выражение вычисляется после каждой итерации и, следовательно, выполняет повторную инициализацию цикла. Никаких ограничений на его тип нет. Побочные эффекты всех трёх выражений заканчиваются по завершении их вычислений. Если подинструкция не содержит в себе continue,

то

```
for ( выражение1 ; выражение2 ; выражение3 ) инструкция
```

эквивалентна конструкции

```
выражение1 ;
while ( выражение2 ) {
    инструкция
    выражение3 ;
}
```

Любое из трёх выражений цикла может быть опущено. Считается, что отсутствие второго выражения равносильно сравнению с ненулевой константой.

А.9.6. Инструкции перехода

Инструкции перехода осуществляют безусловную передачу управления.

```
инструкция-перехода:
    goto идентификатор ;
    continue ;
    break ;
    return выражение_необ ;
```

В goto-инструкции идентификатор должен быть меткой (А.9.1), расположенной в текущей функции. Управление передаётся на помеченную инструкцию.

Инструкцию continue можно располагать только внутри цикла. Она вызывает переход к следующей итерации самого внутреннего содержащего её цикла. Говоря более точно, для каждой из конструкций

```
while (...) {          do {                for (...) {
    ...                  ...                   ...
contin: ;              contin: ;           contin: ;
}                      } while (...);      }
```

инструкция continue, если она не «погружена» в ещё более внутренний цикл, делает то же самое, что и goto contin.

Инструкция break встречается в циклической или в swhitch-инструкции, и только в них. Она завершает работу самой внутренней циклической или switch-инструкции, содержащей break, после чего управление переходит к следующей инструкции.

При помощи return функция возвращает управление в программу, откуда она была вызвана. Если за return следует выражение, то его значение возвращается вызвавшей эту функцию программе. Значение выражения

приводится к типу так, как если бы оно присваивалось переменной, имеющей тот же тип, что и функция.

Ситуация, когда «путь» вычислений приводит в конец функции (т.е. на последнюю закрывающую фигурную скобку), равносильна выполнению return-инструкции без выражения. В последнем случае, а также в случае явного задания return без выражения возвращаемое значение не определено.

А.10. Внешние декларации

То, что подготовлено в качестве ввода для Си-компилятора, называется компонентой трансляции. Она состоит из последовательности внешних деклараций, каждая из которых представляет собой либо декларацию, либо определение функции.

```
компонента-трансляции:
    внешняя-декларация
    компонента-трансляции внешняя-декларация

внешняя-декларация:
    определение-функции
    декларация
```

Область действия внешних деклараций простирается до конца компоненты трансляции, в которой они декларированы, точно так же, как область действия деклараций в блоке распространяется до конца этого блока. Синтаксис внешней декларации не отличается от синтаксиса любой другой декларации за одним исключением: код функции можно определять только при помощи внешней декларации.

А.10.1. Определение функции

Определение функции имеет следующий вид:

```
определение-функции:
    спецификаторы-декларации_необ декларатор
                    список-деклараций_необ составная-инструкция
```

Из спецификаторов класса памяти в *спецификаторах-декларации* возможны только extern и static; различия между последними рассматриваются в А.11.2.

Типом возвращаемого функцией значения может быть арифметический тип, структура, объединение, указатель и void, но не «функция» и не «массив». Декларатор в декларации функции должен явно указывать на то, что описываемый им идентификатор имеет тип «функция», т.е. он должен иметь одну из следующих двух форм (А.8.6.3):

собственно-декларатор (*список-типов-параметров*)
собственно-декларатор (*список-идентификаторов*$_{необ}$)

где *собственно-декларатор* есть идентификатор или идентификатор, заключённый в скобки. Заметим, что на тип «функция» посредством typedef ссылаться нельзя.

Первая форма соответствует определению функции новым способом, для которого характерно декларирование параметров в *списке-типов-параметров* вместе с их типами; *список-деклараций*, располагаемый за декларатором, должен отсутствовать. Если *список-типов-параметров* не состоит из одного-единственного слова void, показывающего, что параметров у функции нет, то в каждом декларатore в *списке-типов-параметров* обязан присутствовать идентификатор. Если *список-типов-параметров* заканчивается знаками «, ...», то вызов функции может иметь аргументов больше, чем параметров; в таком случае, чтобы ссылаться на дополнительные аргументы, следует пользоваться механизмом макроса va_arg из головного файла <stdarg.h>, описанного в приложении В. Функции с переменным числом аргументов должны иметь по крайней мере один именованный параметр.

Вторая форма – определение функции старым способом. *Список-идентификаторов* содержит имена параметров, а *список-деклараций* приписывает им типы. В *списке-деклараций* разрешено декларировать только именованные параметры, инициализация запрещается, и из спецификаторов класса памяти возможен только register.

И в том и другом способе определения функции мыслится, что все параметры как бы декларированы в самом начале составной инструкции, образующей тело функции, и совпадающие с ними имена здесь декларироваться не должны (хотя, как и любые идентификаторы, их можно переопределить в более внутренних блоках). Декларацию параметра «массив из *тип*» можно трактовать как «указатель на *тип*»; аналогично декларацию параметра «функция, возвращающая *тип*» – как «указатель на функцию, возвращающую *тип*». В момент вызова функции её аргументы соответствующим образом преобразуются и присваиваются параметрам. (См. А.7.3.2.).

> Новый способ определения функций введён ANSI-стандартом. Есть также небольшие изменения в операции повышения типа; в первой редакции параметры типа float следовало читать как double. Различие между float и double становилось заметным, лишь когда внутри функции генерировался указатель на параметр.

Ниже приведён пример определения функции новым способом:

```
int max(int a, int b, int c)
{
    int m;
    m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Здесь int – *спецификаторы-декларации*; max(int a, int b, int c) – декларатор функции, а { ... } – блок, задающий её код. Определение старым способом той же функции выглядит следующим образом:

```
int max(a, b, c)
int a, b, c;
{
    /* ... */
}
```

где max(a, b, c) – декларатор, а int a, b, c – *список-деклараций* для параметров.

A.10.2. Внешние декларации

Внешние декларации специфицируют характеристики объектов, функций и других идентификаторов. Термин «внешний» здесь используется, чтобы подчеркнуть тот факт, что декларации расположены вне функций; впрямую с ключевым словом extern («внешний») он не связан. Класс памяти для объекта с внешней декларацией либо вообще не указывается, либо специфицируется как extern или static.

В одной компоненте трансляции для одного идентификатора может содержаться несколько внешних деклараций, если они согласуются друг с другом по типу и способу связи и если для этого идентификатора существует не более одного определения.

Две декларации объекта или функции считаются согласованными по типу в соответствии с правилами, рассмотренными в A.8.10. Кроме того, если декларации отличаются лишь тем, что в одной из них тип структуры, объединения или перечисления незавершён (A.8.3), а в другой соответствующий ему тип с тем же тегом завершён, то такие типы считаются согласованными. Если два типа массива (A.8.6.2) отличаются лишь тем, что один завершённый, а другой незавершённый, то такие типы также считаются согласованными. Наконец, если один тип специфицирует функцию старым способом, а другой – ту же функцию новым способом (с декларациями параметров), то такие типы также считаются согласованными.

Если первая внешняя декларация функции или объекта помечена спецификатором static, то у декларируемого идентификатора – *внутренняя связь*; в противном случае у него – *внешняя связь*. Способы связей обсуждаются в A.11.2.

Внешняя декларация объекта считается определением, если она имеет инициализатор. Внешняя декларация, в которой нет инициализатора и нет спецификатора `extern`, считается *пробным определением*. Если в компоненте трансляции появится определение объекта, то все его пробные определения просто станут избыточными декларациями. Если никакого определения для этого объекта в компоненте трансляции не обнаружится, то все его пробные определения будут трактоваться как одно определение с инициализатором 0.

Каждый объект должен иметь ровно одно определение. Для объекта с внутренней связью это правило относится к каждой отдельной компоненте трансляции, поскольку объекты с внутренними связями в каждой компоненте уникальны. В случае объектов с внешними связями указанное правило действует в отношении всей программы в целом.

> Хотя правило одного определения формулируется несколько иначе по сравнению с первой редакцией, по существу оно совпадает с прежним. Некоторые реализации его ослабляют, более широко трактуя понятие пробного определения. В другом варианте указанного правила, который распространён в системах UNIX и признан как общепринятое расширение стандарта, все пробные определения объектов с внешними связями из всех транслируемых компонент программы рассматриваются вместе, а не отдельно в каждой компоненте. Если где-то в программе обнаруживается определение, то пробные определения становятся просто декларациями, но, если никакого определения не встретилось, то все пробные определения становятся одним-единственным определением с инициализатором 0.

A.11. Область действия и связи

Каждый раз компилировать всю программу целиком нет необходимости. Исходный текст можно хранить в нескольких файлах, представляющих собой компоненты трансляции. Ранее скомпилированные программы могут загружаться из библиотек. Связи между функциями программы могут осуществляться через вызовы и внешние данные.

Следовательно, существуют два вида областей действия: первая – это *лексическая область* идентификатора: т.е. область в тексте программы, где имеют смысл все его характеристики; вторая область – это область, ассоциируемая с объектами и функциями, имеющими внешние связи, устанавливаемые между идентификаторами из раздельно компилируемых компонент трансляции.

А.11.1. Лексическая область действия

Каждый идентификатор попадает в одно из нескольких пространств имён. Эти пространства никак не связаны друг с другом. Один и тот же идентификатор может использоваться в разных смыслах даже в одной области действия, если он принадлежит разным пространствам имён. Ниже через точку с запятой перечислены классы объектов, имена которых представляют собой отдельные независимые пространства: объекты, функции, `typedef`-имена и `enum`-константы; метки инструкций; теги структур, объединений и перечислений; члены каждой отдельной структуры или объединения.

> Сформулированные правила несколько отличаются от прежних, описанных в первом издании. Метки инструкций не имели раньше собственного пространства; теги структур и теги объединений (а в некоторых реализациях и теги перечислений) имели отдельные пространства. Размещение тегов структур, объединений и перечислений в одном общем пространстве – это дополнительное ограничение, которого раньше не было. Наиболее существенное отклонение от первой редакции в том, что каждая отдельная структура (или объединение) создаёт своё собственное пространство имён её членов. Таким образом, одно и то же имя может использоваться в нескольких различных структурах. Это правило широко применяется уже несколько лет.

Лексическая область действия идентификатора объекта (или функции), объявленного во внешней декларации, начинается с места, где заканчивается его декларатор, и простирается до конца компоненты трансляции, в которой он декларирован. Область действия параметра в определении функции начинается с начала блока, представляющего собой тело функции, и распространяется на всю функцию; область действия параметра в описании функции заканчивается в конце этого описания. Область действия идентификатора, декларируемого в начале блока, начинается от места, где заканчивается его декларатор, и продолжается до конца этого блока. Областью действия метки является вся функция, где эта метка встречается. Область действия тега структуры, объединения или перечисления начинается от его появления в спецификаторе типа и продолжается до конца компоненты трансляции для декларации внешнего уровня и до конца блока для декларации внутри функции.

Если идентификатор явно декларирован в начале некоторого блока (в том числе тела функции), то любая декларация того же идентификатора, находящаяся снаружи этого блока, временно перестаёт действовать вплоть до конца блока.

А.11.2. Связи

Если встречается несколько деклараций, имеющих одинаковый идентификатор и описывающих объект (или функцию), то все эти декларации в случае внешней связи относятся к одному объекту (функции) – уникальному для всей программы; если же связь внутренняя, то свойство уникальности распространяется только на компоненту трансляции.

Как говорилось в А.10.2, если первая внешняя декларация имеет спецификатор `static`, то она описывает идентификатор с внутренней связью, если такого спецификатора нет, то с внешней связью. Если декларация находится внутри блока и не содержит `extern`, то соответствующий идентификатор ни с чем не связан и уникален для данной функции. Если декларация содержит `extern` и блок находится в области действия внешней декларации этого идентификатора, то последний имеет ту же связь и ссылается на тот же объект (функцию). Однако, если ни одной внешней декларации для этого идентификатора нет, то он имеет внешнюю связь.

А.12. Препроцессирование

Препроцессор выполняет макроподстановку, условную компиляцию, включение именованных файлов. Строки, начинающиеся со знака `#` (перед которым возможны пробельные литеры), устанавливают связь с препроцессором. Их синтаксис не зависит от остальной части языка; они могут появляться где угодно и оказывать влияние (независимо от области действия) вплоть до конца транслируемой компоненты. Границы строк принимаются во внимание; каждая строка анализируется отдельно (однако есть возможность «склеивать» строки, см. А.12.2). Лексемами для препроцессора являются все лексемы языка и последовательности литер, задающие имена файлов, как, например, в директиве `#include` (А.12.4). Кроме того, любая литера, не определённая каким-либо другим способом, воспринимается как лексема. Влияние пробельных литер, отличающихся от пробелов и горизонтальных табуляций, внутри строк препроцессора не определено.

Само препроцессирование проистекает в нескольких логически последовательных фазах. В отдельных реализациях некоторые фазы объединены.

1. Трёхзнаковые последовательности, описанные в А.12.1, заменяются их эквивалентами. Между строками вставляются литеры новая-строка, если того требует операционная система.

2. Выбрасываются пары литер, состоящие из обратной наклонной черты с последующей литерой новая-строка; тем самым осуществляется «склеивание» строк (А.12.2).

3. Программа разбивается на лексемы, разделённые литерами пропусков. Комментарии заменяются на единичные пробелы. Затем выполняются директивы препроцессора и макроподстановки (А.12.3–А.12.10).

4. Эскейп-последовательности в литерных константах и стринговых литералах (А.2.5.2, А.2.6) заменяются на литеры, которые они обозначают. Соседние стринговые литералы конкатенируются.

5. Результат транслируется. Затем устанавливаются связи с другими программами и библиотеками посредством сбора необходимых программ и данных и соединения ссылок на внешние функции и объекты с их определениями.

А.12.1. Трёхзнаковые последовательности

Множество литер, из которых набираются исходные Си-программы, основано на семибитовом ASCII-коде. Однако он шире, чем инвариантный код литер ISO 646-1983 (ISO 646-1983 Invariant Code Set). Чтобы дать возможность пользоваться сокращённым набором литер, все указанные ниже трёхзнаковые последовательности заменяются на соответствующие им единичные литеры. Замена осуществляется до любой иной обработки.

```
??=    #        ??(    [        ??<    {
??/    \        ??)    ]        ??>    }
??'    ^        ??!    |        ??-    ~
```

Никакие другие замены, кроме указанных, не делаются.

Трёхзнаковые последовательности введены ANSI-стандартом.

А.12.2. Склеивание строк

Строка, заканчивающаяся обратной наклонной чертой, соединяется со следующей, поскольку литера \ и следующая за ней литера новая-строка выбрасываются. Это делается перед «разбиением» текста на лексемы.

А.12.3. Макроопределение и макрорасширение

Управляющая строка вида

```
# define идентификатор последовательность-лексем
```

заставляет препроцессор заменять идентификатор на последовательность лексем; пробельные литеры в начале и в конце последовательности лексем

выбрасываются. Повторная строка `#define` с тем же идентификатором считается ошибкой, если последовательности лексем неидентичны (несовпадения в разделяющих пробельных литерах при сравнении во внимание не принимаются).

Строка вида

```
# define идентификатор( список-идентификаторов ) последовательность-лексем
```

где между первым идентификатором и знаком `(` не должно быть ни одной пробельной литеры, представляет собой макроопределение с параметрами, задаваемыми списком идентификаторов. Как и в первом варианте, пробельные литеры в начале и в конце последовательности лексем выбрасываются, и макрос может быть повторно определён только с тем же списком параметров и той же последовательностью лексем.

Управляющая строка вида

```
# undef идентификатор
```

предписывает препроцессору «забыть» определение, данное идентификатору. Применение `#undef` к неизвестному идентификатору ошибкой не считается.

Если макроопределение было задано вторым способом, то текстуальная последовательность, состоящая из его идентификатора с, возможно, следующими за ним пробельными литерами, знака `(`, списка лексем, разделённых занятыми, и знака `)`, представляет собой вызов макроса. Аргументами вызова макроса являются лексемы, разделённые запятыми (запятые, «закрытые» кавычками или вложенными скобками, в разделении аргументов не участвуют). Аргументы при их выделении макрорасширениям не подвергаются. Количество аргументов в вызове макроса должно соответствовать количеству параметров макроопределения. После выделения аргументов окружающие и пробельные литеры выбрасываются. Затем в замещающей последовательности лексем макроса идентификаторы-параметры (если они не окружены кавычками) заменяются на соответствующие им аргументы. Если в замещающей последовательности перед параметром не стоит знак `#` и ни перед ним, ни после него нет знака `##`, то лексемы аргумента проверяются: не содержат ли они в себе макровызова, и если это так, то прежде чем аргумент будет подставлен, производится соответствующее ему макрорасширение.

На процесс подстановки влияют два специальных оператора. Первый – это оператор `#`, который ставится перед параметром. Он требует, чтобы подставляемый вместо параметра и знака `#` (перед ним) текст был заключён в двойные кавычки. При этом в аргументе в стринговых литералах и литерных константах перед каждой двойной кавычкой `"` (включая и обрамляющие стринг), а также перед каждой обратной наклонной чертой `\` вставляется `\`.

Второй оператор записывается как ##. Если последовательность лексем в любого вида макроопределении содержит оператор ##, то сразу после подстановки параметров он вместе с окружающими его пробельными литерами выбрасывается, благодаря чему «склеиваются» соседние лексемы, образуя тем самым новую лексему. Результат не определён при получении неправильных лексем или когда генерируемый текст зависит от порядка применения операторов ##. Кроме того, ## не может стоять ни в начале, ни в конце замещающей последовательности лексем.

В макросах обоих видов замещающая последовательность лексем повторно просматривается на предмет обнаружения там новых define-имён. Однако, если некоторый идентификатор уже был заменён в данном расширении, повторное появление такого идентификатора не вызовет его замены.

Если полученное расширение начинается со знака #, оно не будет воспринято как директива препроцессора.

> ANSI-стандарт описывает процесс макрорасширения более точно, чем первое издание. Наиболее важные изменения касаются введения операторов # и ##, которые предоставляют возможность осуществлять расширения внутри стрингов и конкатенацию лексем. Некоторые из новых правил, особенно касающиеся конкатенации, могут показаться несколько странными. (См. приведённые ниже примеры.)

Описанные возможности можно использовать для показа смысловой сущности констант, как, например, в

```
#define TABSIZE 100
int table[TABSIZE];
```

Определение

```
#define ABSDIFF(a, b)   ((a)>(b) ? (a)-(b) : (b)-(a))
```

задаёт макрос, возвращающий абсолютное значение разности его аргументов. 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 иметь побочный эффект, вычисляются дважды: один раз – при проверке, другой раз – при вычислении результата.

Если имеется определение

```
#define tempfile(dir)    #dir "/%s"
```

то макровызов `tempfile(/usr/tmp)` даст в результате

```
"/usr/tmp" "/%s"
```

Далее эти два стринга превратятся в один стринг. По макросу

```
#define cat(x, y)    x ## y
```

вызов `cat(var, 123)` сгенерирует `var123`. Однако `cat(cat(1,2),3)` не даст желаемого, так как оператор `##` воспрепятствует получению правильных аргументов для внешнего вызова `cat`. В результате будет выдана следующая цепочка лексем:

```
cat ( 1 , 2 )3
```

где `)3` (результат «склеивания» последней лексемы первого аргумента с первой лексемой второго аргумента) не является правильной лексемой. Если второй уровень макроопределения задан в виде

```
#define xcat(x,y)    cat(x,y)
```

то никаких коллизий здесь не возникает; `xcat(xcat(1,2),3)` в итоге даст `123`, поскольку сам `xcat` не использует `##`.

Аналогично сработает и `ABSDIFF(ABSDIFF(a,b),c)`, и мы получим правильный результат.

A.12.4. Включение файла

Управляющая строка

```
# include <имя-файла>
```

заменяется на содержимое файла с именем *имя-файла*. Среди литер, составляющих *имя-файла*, не должно быть знака `>` и литеры новая-строка. Результат не определён, если *имя-файла* содержит любую из литер `"`, `'`, `\` или пару литер `/*`. Порядок поиска указанного файла зависит от реализации.

Подобным же образом выполняется управляющая строка

```
# include "имя-файла"
```

Сначала поиск осуществляется по тем же правилам, по каким компилятор ищет первоначальный исходный файл (механизм этого поиска зависит от реализации), а в случае неудачи осуществляется методом поиска, принятым в `#include` первого типа. Результат остаётся неопределённым, если имя файла содержит `'`, `\` или `/*`; использование знака `>` разрешается.

Наконец, директива

```
# include последовательность-лексем
```

не совпадающая ни с одной из предыдущих форм, рассматривает последовательность лексем как текст, который в результате всех макроподстановок должен дать `#include` с `<...>` или с `"..."`. Сгенерированная таким образом директива далее будет интерпретироваться в соответствии с полученной формой.

Файлы, вставляемые при помощи `#include`, сами могут содержать в себе директивы `#include`.

А.12.5. Условная компиляция

Части программы могут компилироваться условно, если они оформлены в соответствии со следующим схематично изображённым синтаксисом:

```
условная-конструкция-препроцессора:
     if-строка текст elif-части else-часть_необ #endif

if-строка:
     # if константное-выражение
     # ifdef идентификатор
     # ifndef идентификатор

elif-части:
     elif-строка текст
     elif-части_необ

elif-строка:
     # elif константное-выражение

else-часть:
     else-строка текст

else-строка:
     # else
```

Каждая из директив (*if-строка*, *elif-строка* и `#endif`) записывается на отдельной строке. Константные выражения в `#if` и последующих `#elif`-строках вычисляются по порядку, пока не обнаружится выражение с ненулевым значением; текст, следующий за строкой с нулевым значением, выбрасывается. Текст, расположенный за директивой с ненулевым значением, обрабатывается обычным образом. Под словом «текст» здесь имеется в виду любая последовательность строк, включая строки препроцессора, которые не являются частью условной структуры; текст может быть и пустым. Если `#if` или `#elif`-строка с ненулевым значением выражения найдена и её текст обработан, то последующие `#elif` и `#else`-строки вместе со своими текстами выбрасываются. Если все выражения имеют нулевые значения и присутствует строка `#else`, то следующий за ней текст обрабатывается обычным образом. Тексты «неактивных» ветвей условных конструкций, за исключением тех, которые заведуют вложенностью условных конструкций, игнорируются.

Константные выражения в `#if` и `#elif` являются объектами для обычной макроподстановки. Более того, прежде чем просматривать выражения вида

```
defined идентификатор
```

и

```
defined ( идентификатор )
```

на предмет наличия в них макровызова, они заменяются на `1L` или `0L` в зависимости от того, был или не был определён препроцессором указанный в них идентификатор. Все идентификаторы, оставшиеся после макрорасширения, заменяются на `0L`. Наконец, предполагается, что любая целая константа всегда имеет суффикс `L`, т.е. вся арифметика имеет дело с операндами только типа `long` или `unsigned long`.

Константное выражение (A.7.19) здесь используется с ограничениями: оно должно быть целочисленным, не может содержать в себе перечислимых констант, преобразований типа и операторов `sizeof`.

Управляющие строки

```
#ifdef идентификатор
#ifndef идентификатор
```

эквивалентны соответственно строкам

```
# if defined идентификатор
# if ! defined идентификатор
```

> Строки `#elif` не было в первой редакции, хотя она и использовалась в некоторых препроцессорах. Оператор препроцессора `defined` – также новый.

A.12.6. Нумерация строк

Для удобства работы с другими препроцессорами, генерирующими Си-программы, можно использовать одну из следующих директив:

```
# line константа "идентификатор"
# line константа
```

которая предписывает компилятору «считать», что указанные десятичное целое и идентификатор являются номером следующей строки и именем текущего файла соответственно. Если имя файла отсутствует, то ранее запомненное имя не изменяется. Расширения макровызовов в `line`-директиве выполняются до интерпретации последней.

A.12.7. Генерация сообщения об ошибке

Строка препроцессора вида

```
# error последовательность-лексем_необ
```

приказывает ему выдать диагностическое сообщение, включающее заданную последовательность лексем.

А.12.8. Прагма

Управляющая строка вида

```
# pragma последовательность-лексем_необ
```

призывает препроцессор выполнить зависящие от реализации действия. Неопознанная прагма игнорируется.

А.12.9. Пустая директива

Строка препроцессора вида

```
#
```

не вызывает никаких действий.

А.12.10. Заранее определённые имена

Препроцессор «понимает» несколько заранее определённых идентификаторов; их он заменяет специальной информацией. Эти идентификаторы (и оператор препроцессора `defined` в том числе) нельзя повторно переопределять, к ним нельзя также применять директиву `#undef`. Это следующие идентификаторы:

`__LINE__` Номер текущей строки исходного текста, десятичная константа.

`__FILE__` Имя компилируемого файла, стринг.

`__DATE__` Дата компиляции в виде `"Ммм дд гггг"`, стринг.

`__TIME__` Время компиляции в виде `"чч:мм:сс"`, стринг.

`__STDC__` Константа 1. Предполагается, что этот идентификатор определён как 1 только в тех реализациях, которые следуют стандарту.

Строки `#error` и `#pragma` впервые введены ANSI-стандартом. Заранее определённые макросы препроцессора также до сих пор не описывались, хотя и использовались в некоторых реализациях.

А.13. Грамматика

Ниже приведены сведённые воедино грамматические правила, описанные в данном приложении. Они имеют то же содержание, но даны в ином порядке.

Здесь не приводятся определения следующих терминальных символов: *целая-константа*, *литерная-константа*, *с-плав-точкой-константа*, *идентификатор*, *стринг* и *перечислимая-константа*. Слова, набранные обычным латинским шрифтом, и знаки рассматриваются как терминальные символы и используются буквально в том виде, как записаны. Данную грамматику можно механически трансформировать в текст, понятный си-

стеме автоматической генерации грамматического распознавателя. Для этого помимо добавления некоторых синтаксических пометок, предназначенных для указания альтернативных продукций, потребуется расшифровка конструкции со словами «один из» и дублирование каждой продукции, использующей символ с индексом *необ*, причём один вариант продукции должен быть написан с этим символом, а другой – без него. С одним изменением, а именно удалением продукции *typedef-имя: идентификатор* и объявлением *typedef-имя* терминальным символом, данная грамматика будет понятна генератору грамматического распознавателя YACC. Ей присуще лишь одно противоречие, вызываемое неоднозначностью `if-else`-конструкции.

компонента-трансляции:
    внешняя-декларация
    компонента-трансляции внешняя-декларация

внешняя-декларация:
    определение-функции
    декларация

определение-функции:
    спецификаторы-декларации$_{необ}$ *декларатор*
        список-деклараций$_{необ}$ *составная-инструкция*

декларация:
    спецификаторы-декларации список-иниц-деклараторов$_{необ}$;

список-деклараций:
    декларация
    список-деклараций декларация

спецификаторы-декларации:
    спецификатор-класса-памяти спецификаторы-декларации$_{необ}$
    спецификатор-типа спецификаторы-декларации$_{необ}$
    квалификатор-типа спецификаторы-декларации$_{необ}$

спецификатор-класса-памяти: один из
    auto register static extern typedef

спецификатор-типа: один из
    void char short int long float double signed unsigned
        структ-или-объед-спецификатор енит-спецификатор typedef-имя

квалификатор-типа: один из
    const volatile

структ-или-объед-спецификатор:
    структ-или-объед идентификатор$_{необ}$ { *список-структ-деклараций* }
    структ-или-объед идентификатор

структ-или-объед: один из
    struct union

список-структ-деклараций:
    структ-декларация
    список-структ-деклараций структ-декларация

список-иниц-деклараторов:
    иниц-декларатор
    список-иниц-деклараторов , иниц-декларатор

иниц-декларатор:
    декларатор
    декларатор = инициализатор

структ-декларация:
    список-спецификаторов-квалификаторов список-структ-деклараторов ;

список-спецификаторов-квалификаторов:
    спецификатор-типа список-спецификаторов-квалификаторов$_{необ}$
    квалификатор-типа список-спецификаторов-квалификаторов$_{необ}$

список-структ-деклараторов:
    структ-декларатор
    список-структ-деклараторов , структ-декларатор

структ-декларатор:
    декларатор
    декларатор$_{необ}$ *: константное-выражение*

переч-спецификатор:
    enum *идентификатор*$_{необ}$ { *список-перечислителей* }
    enum *идентификатор*

список-перечислителей:
    перечислитель
    список-перечислителей , перечислитель

перечислитель:
    идентификатор
    идентификатор = константное-выражение

декларатор:
    указатель$_{необ}$ *собственно-декларатор*

собственно-декларатор:
    идентификатор
    (декларатор)
    собственно-декларатор [константное-выражение$_{необ}$ *]*
    собственно-декларатор (список-типов-параметров)
    собственно-декларатор (список-идентификаторов$_{необ}$ *)*

указатель:
    ** список-квалификаторов-типа*$_{необ}$
    ** список-квалификаторов-типа*$_{необ}$ *указатель*

список-квалификаторов-типа:
    квалификатор-типа
    список-квалификаторов-типа квалификатор-типа

список-типов-параметров:
    список-параметров
    список-параметров , ...

список-параметров:
    декларация-параметра
    список-параметров , декларация-параметра

декларация-параметра:
    спецификаторы-декларации декларатор
    спецификаторы-декларации абстрактный-декларатор$_{необ}$

список-идентификаторов:
    идентификатор
    список-идентификаторов , идентификатор

инициализатор:
    выражение-присваивания
    { *список-инициализаторов* }
    { *список-инициализаторов* , }

список-инициализаторов:
    инициализатор
    список-инициализаторов , *инициализатор*

имя-типа:
    список-спецификаторов-квалификаторов *абстрактный-декларатор*$_{необ}$

абстрактный-декларатор:
    указатель
    указатель$_{необ}$ *собственно-абстрактный-декларатор*

собственно-абстрактный-декларатор:
    (*абстрактный-декларатор*)
    собственно-абстрактный-декларатор$_{необ}$ [*константное-выражение*$_{необ}$]
    собственно-абстрактный-декларатор$_{необ}$ (*список-типов-параметров*$_{необ}$)

typedef-имя:
    идентификатор

инструкция:
    помеченная-инструкция
    инструкция-выражение
    составная-инструкция
    инструкция-выбора
    циклическая-инструкция
    инструкция-перехода

помеченная-инструкция:
    идентификатор : *инструкция*
    case *константное-выражение* : *инструкция*
    default : *инструкция*

инструкция-выражение:
    выражение$_{необ}$;

составная-инструкция:
    { *список-декларации*$_{необ}$ *список-инструкции*$_{необ}$ }

список-инструкций:
    инструкция
    список-инструкций *инструкция*

инструкция-выбора:
    if (*выражение*) *инструкция*
    if (*выражение*) *инструкция* else *инструкция*
    switch (*выражение*) *инструкция*

циклическая-инструкция:
    while (*выражение*) *инструкция*
    do *инструкция* while (*выражение*)
    for (*выражение*$_{необ}$; *выражение*$_{необ}$; *выражение*$_{необ}$) *инструкция*

инструкция-перехода:
    goto *идентификатор* ;
    continue ;
    break ;
    return *выражение*$_{необ}$;

```
выражение:
    выражение-присваивания
    выражение , выражение-присваивания

выражение-присваивания:
    условное-выражение
    унарное-выражение оператор-присваивания выражение-присваивания

оператор-присваивания: один из
    =   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=

условное-выражение:
    логическое-ИЛИ-выражение
    логическое-ИЛИ-выражение ? выражение : условное-выражение

константное-выражение:
    условное-выражение

логическое-ИЛИ-выражение:
    логическое-И-выражение
    логическое-ИЛИ-выражение || логическое-И-выражение

логическое-И-выражение:
    ИЛИ-выражение
    логическое-И-выражение && ИЛИ-выражение

ИЛИ-выражение:
    исключающее-ИЛИ-выражение
    ИЛИ-выражение | исключающее-ИЛИ-выражение

исключающее-ИЛИ-выражение:
    И-выражение
    исключающее-ИЛИ-выражение ^ И-выражение

И-выражение:
    выражение-равенства
    И-выражение & выражение-равенства

выражение-равенства:
    выражение-отношения
    выражение-равенства == выражение-отношения
    выражение-равенства != выражение-отношения

выражение-отношения:
    сдвиговое-выражение
    выражение-отношения < сдвиговое-выражение
    выражение-отношения > сдвиговое-выражение
    выражение-отношения <= сдвиговое-выражение
    выражение-отношения >= сдвиговое-выражение

сдвиговое-выражение:
    аддитивное-выражение
    сдвиговое-выражение >> аддитивное-выражение
    сдвиговое-выражение << аддитивное-выражение

аддитивное-выражение:
    мультипликативное-выражение
    аддитивное-выражение + мультипликативное-выражение
    аддитивное-выражение - мультипликативное-выражение

мультипликативное-выражение:
    выражение-приведённое-к-типу
    мультипликативное-выражение * выражение-приведённое-к-типу
    мультипликативное-выражение / выражение-приведённое-к-типу
    мультипликативное-выражение % выражение-приведённое-к-типу
```

```
выражение-приведённое-к-типу:
    унарное-выражение
    ( имя-типа ) выражение-приведённое-к-типу
унарное-выражение:
    постфиксное-выражение
    ++ унарное-выражение
    -- унарное-выражение
    унарный-оператор выражение-приведённое-к-типу
    sizeof  унарное-выражение
    sizeof  ( имя-типа )
унарный-оператор: 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
    стринг
    ( выражение )
список-аргументов-выражений:
    выражение-присваивания
    список-аргументов-выражений , выражение-присваивания
константа:
    целая-константа
    литерная-константа
    с-плав-точкой-константа
    перечислимая-константа
```

Грамматика языка препроцессора приводится в виде перечня структур управляющих строк. Для механического получения программы грамматического разбора она не годится. Грамматика включает символ *текст*, который означает текст обычной программы, неусловные управляющие строки препроцессора и его законченные условные конструкции.

```
управляющая-строка:
    # define идентификатор последовательность-лексем
    # define идентификатор( идентификатор , ... , идентификатор )
                                                  последовательность-лексем
    # undef идентификатор
    # include <имя-файла>
    # include "имя-файла"
    # include последовательность-лексем
    # line константа "идентификатор"
    # line константа
    # error последовательность-лексем_необ
    # pragma последовательность-лексем_необ
    #
    условная-конструкция-препроцессора
```

условная-конструкция-препроцессора:
    if-строка текст elif-части else-часть$_{необ}$ # endif

if-строка:
    # if *константное-выражение*
    # ifdef *идентификатор*
    # ifndef *идентификатор*

elif-части:
    elif-строка текст
    elif-части$_{необ}$

elif-строка:
    # elif *константное-выражение*

else-часть:
    else-строка текст

else-строка:
    # else

Приложение B

Стандартная библиотека

Настоящее приложение представляет собой краткое изложение библиотеки, утверждённой в качестве ANSI-стандарта. Сама по себе библиотека не является частью языка, однако заложенный в ней набор деклараций функций, а также определений типов и макросов составляет системную среду, поддерживающую стандарт Си. Мы не приводим несколько функций с ограниченной областью применения – те, которые легко синтезируются из других функций, а также всё то, что касается многобайтовых литер и специфики, обусловленной языком, национальностью и культурой.

Функции, типы и макросы декларируются в следующих *головных 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>
```

Доступ к головному файлу осуществляется при помощи строки препроцессора

```
#include <головной-файл>
```

Головные файлы можно включать в любом порядке и сколько угодно раз. Строка `#include` не должна быть внутри внешней декларации или определения и должна встретиться раньше, чем что-нибудь будет востребовано из включаемого головного файла. В конкретной реализации головной файл может и не быть исходным файлом.

Для использования в библиотеке зарезервированы внешние идентификаторы, начинающиеся со знака подчёркивания, а также все другие идентификаторы, начинающиеся с двух знаков подчёркивания или с подчёркивания и заглавной буквы.

B.1. Ввод-вывод: <stdio.h>

Определённые в <stdio.h> функции ввода-вывода, а также типы и макросы составляют приблизительно треть библиотеки.

Поток – это источник или получатель данных; его можно связать с диском или с каким-то другим внешним устройством. Библиотека поддерживает два вида потоков: текстовый и бинарный, хотя на некоторых системах, в частности в UNIXe, они не различаются. Текстовый поток – это последовательность строк; каждая строка имеет нуль или более литер и заканчивается литерой `'\n'`. Операционная среда может потребовать коррекции текстового потока (например, перевода `'\n'` в литеры возврат-каретки и перевод-строки).

Бинарный поток – это последовательность непреобразуемых байтов, представляющих собой некоторые промежуточные данные, которые обладают тем свойством, что, если их записать, а затем прочесть той же системой ввода-вывода, то мы получим информацию, совпадающую с исходной.

Поток соединяется с файлом или устройством посредством его *открытия*; указанная связь разрывается путём *закрытия* потока. Открытие файла возвращает указатель на объект типа `FILE`, который содержит всю информацию, необходимую для управления этим потоком. Если не возникает двусмысленности, мы будем пользоваться терминами «файловый указатель» и «поток» как равнозначными.

Когда программа начинает работу, уже открыты три потока: `stdin`, `stdout` и `stderr`.

B.1.1. Операции над файлами

Ниже перечислены функции, оперирующие с файлами. Тип `size_t` – беззнаковый целочисленный тип, используемый для описания результата оператора `sizeof`.

`FILE *fopen(const char *filename, const char *mode)`

`fopen` открывает файл с заданным именем и возвращает поток или `NULL`, если попытка открытия оказалась неудачной. Режим `mode` допускает следующие значения:

- `"r"` текстовый файл открывается для чтения (от read (англ.) – читать)
- `"w"` текстовый файл создаётся для записи; старое содержимое (если оно было) выбрасывается (от write (англ.) – писать)
- `"a"` текстовый файл открывается или создаётся для записи в конец файла (от append (англ.) – добавлять)

- `"r+"` текстовый файл открывается для исправления (т.е. для чтения и для записи)
- `"w+"` текстовый файл создаётся для исправления; старое содержимое (если оно было) выбрасывается
- `"a+"` текстовый файл открывается или создаётся для исправления уже существующей информации и добавления новой в конец файла

Режим «исправления» позволяет читать и писать в один и тот же файл; при переходах от операций чтения к операциям записи и обратно должны осуществляться обращения к `fflush` или к функции позиционирования файла. Если указатель режима дополнить буквой `b` (например, `"rb"` или `"w+b"`), то это будет означать, что файл бинарный. Ограничение на длину имени файла задано константой `FILENAME_MAX`. `FOPEN_MAX` ограничивает число одновременно открытых файлов.

```
FILE *freopen(const char *filename, const char *mode,
                        FILE *stream)
```

`freopen` открывает файл с указанным режимом и связывает его с потоком `stream`. Она возвращает `stream` или, в случае ошибки, `NULL`. Обычно `freopen` используется для замены файлов, связанных с `stdin`, `stdout` или `stderr`, другими файлами.

```
int fflush(FILE *stream)
```

Применяемая к потоку вывода `fflush` производит дозапись всех оставшихся на буфере (ещё не записанных) данных; для потока ввода эта функция не определена. Возвращает `EOF` в случае возникшей при записи ошибки или нуль в противном случае. Обращение вида `fflush(NULL)` выполняет указанные операции для всех потоков вывода.

```
int fclose(FILE *stream)
```

`fclose` производит дозапись ещё незаписанных буферизованных данных, сбрасывает нечитанный буферизованный ввод, освобождает все автоматически запрошенные буфера, после чего закрывает поток. Возвращает `EOF` в случае ошибки и нуль в противном случае.

```
int remove(const char *filename)
```

`remove` удаляет файл с указанным именем; последующая попытка открыть файл с этим именем вызовет ошибку. Возвращает ненулевое значение в случае неудачной попытки.

```
int rename(const char *oldname, const char *newname)
```

`rename` заменяет имя файла; возвращает ненулевое значение в случае,

если попытка изменить имя оказалась неудачной. Первый параметр задаёт старое имя, второй – новое.

`FILE *tmpfile(void)`
: `tmpfile` создаёт временный файл с режимом доступа `"wb+"`, который автоматически удаляется при его закрытии или обычном завершении программой своей работы. Эта функция возвращает поток или, если она не смогла создать файл, `NULL`.

`char *tmpnam(char s[L_tmpnam])`
: `tmpnam(NULL)` создаёт стринг, который не совпадает ни с одним из имён существующих файлов, и возвращает указатель на внутренний статический массив. `tmpnam[s]` запоминает стринг в `s` и возвращает его в качестве значения функции; длина `s` должна быть не менее `L_tmpnam`. При каждом вызове `tmpnam` генерируется новое имя; при этом гарантируется не более `TMP_MAX` различных имён за один сеанс работы программы. Заметим, что `tmpnam` создаёт имя, но не файл.

`int setvbuf(FILE *stream, char *buf, int mode, size_t size)`
: `setvbuf` управляет буферизацией потока; к ней следует обращаться прежде, чем будет выполняться чтение, запись или какая-либо другая операция. Режим `mode` со значением `_IOFBF` вызывает полную буферизацию, с `_IOLBF` – «построчную» буферизацию текстового файла, а режим `_IONBF` отменяет всякую буферизацию. Если параметр `buf` не есть `NULL`, то его значение – указатель на буфер, в противном случае под буфер будет запрашиваться память. Параметр `size` задаёт размер буфера. Функция `setvbuf` в случае ошибки выдаёт ненулевое значение.

`void setbuf(FILE *stream, char *buf)`
: Если `buf` есть `NULL`, то для потока `stream` буферизация выключается. В противном случае вызов `setbuf` приведёт к тем же действиям, что и вызов `(void) setvbuf(stream, buf, _IOFBF, BUFSIZ)`.

B.1.2. Форматный вывод

Функции `printf` осуществляют вывод информации по формату.

`int fprintf(FILE *stream, const char *format, ...)`

`fprintf` преобразует и пишет вывод в поток `stream` под управлением `format`. Возвращаемое значение – число записанных литер или, в случае ошибки, отрицательное значение.

Форматный стринг содержит два вида объектов: обычные литеры, копируемые в выводной поток, и спецификации преобразования, которые

вызывают преобразование и печать остальных аргументов в том порядке, как они перечислены. Каждая спецификация преобразования начинается с `%` и заканчивается литерой-спецификатором преобразования. Между `%` и литерой-спецификатором в порядке, в котором они здесь перечислены, могут быть расположены следующие элементы информации:

- Флаги (в любом порядке), модифицирующие спецификацию:

`-`	указывает на то, что преобразованный аргумент должен быть прижат к левому краю поля.
`+`	предписывает печатать число всегда со знаком.
`пробел`	если первая литера – не знак, то числу должен предшествовать пробел.
`0`	указывает, что числа должны дополняться ведущими нулями до всей ширины поля.
`#`	указывает на одну из следующих форм вывода: для `o` первой цифрой должен быть 0; для `x` или `X` ненулевому результату должны предшествовать `0x` или `0X`; для `e`, `E`, `f`, `g` и `G` вывод должен всегда содержать десятичную точку; для `g` и `G` хвостовые нули не отбрасываются.

- Число, специфицирующее минимальную ширину поля. Преобразованный аргумент будет напечатан в поле, размер которого не меньше указанной ширины, а если потребуется, в поле большего размера. Если число литер преобразованного аргумента меньше ширины поля, то поле будет дополнено слева (или справа, если число прижимается к левому краю). Обычно поле дополняется пробелами (или нулями, если присутствует флаг дополнения нулями).

- Точка, отделяющая указатель ширины поля от указателя точности.

- Число, задающее точность, которое специфицирует максимальное количество литер, печатаемых из стринга, или количество цифр после десятичной точки в преобразованиях `e`, `E` или `f`, или количество значащих цифр для `g` или `G`-преобразования, или минимальное количество цифр при печати целого (до необходимой ширины поля число дополняется ведущими нулями).

- Модификаторы `h`, `l` (буква эль) или `L`. «`h`» указывает, что соответствующий аргумент должен печататься как `short` или `unsigned short`; «`l`» сообщает, что аргумент имеет тип `long` или `unsigned long`; «`L`» информирует, что аргумент принадлежит типу `long double`.

Ширина, или точность, или обе характеристики могут быть специфицированы при помощи `*`; в этом случае необходимое число «извлекается» из сле-

ТАБЛИЦА B.1. ПРЕОБРАЗОВАНИЯ PRINTF

ЛИТЕРА	ТИП АРГУМЕНТА; ВИД ПЕЧАТИ
`d, i`	`int`; знаковая десятичная запись.
`o`	`int`; беззнаковая восьмеричная запись (без ведущего `0`).
`x, X`	`int`; беззнаковая шестнадцатеричная запись (без ведущих `0x` и `0X`), в качестве цифр от 10 до 15 используются `abcdef` для `x` и `ABCDEF` для `X`.
`u`	`int`; беззнаковое десятичное целое.
`c`	`int`; единичная литера после преобразования в `unsigned char`.
`s`	`char *`; литеры стринга печатаются, пока не встретится `'\0'` или не исчерпается количество литер, указанное точностью.
`f`	`double`; десятичная запись вида `[-]`*`mmm`*`.`*`ddd`*, где количество *d* специфицируется точностью. По умолчанию точность равна 6; нулевая точность подавляет печать десятичной точки.
`e, E`	`double`; десятичная запись вида `[-]`*`m`*`.`*`dddddd`*`e±`*`xx`* или запись вида `[-]`*`m`*`.`*`dddddd`*`E±`*`xx`*, где количество *d* специфицируется точностью. По умолчанию точность равна 6; нулевая точность подавляет печать десятичной точки.
`g, G`	`double`; используется `%e` или `%E`, если экспонента меньше -4 или больше или равна точности; в противном случае используется `%f`. Хвостовые нули и точка в конце не печатаются.
`p`	`void *`; печатает в виде указателя (представление зависит от реализации).
`n`	`int *`; число литер, напечатанных к данному моменту данным вызовом `printf`, *записывается* в аргумент. Никакие другие аргументы не преобразуются.
`%`	никакие аргументы не преобразуются; печатается `%`.

дующего аргумента, который должен иметь тип `int` (в случае двух звёздочек используются два аргумента).

Литеры-спецификаторы и разъяснение их смысла приведены в табл. B.1. Если за `%` нет правильной литеры-спецификатора, результат не определён.

```
int printf(const char *format, ...)
```
`printf(...)` полностью эквивалентна `fprintf(stdout, ...)`

```
int sprintf(char *s, const char *format, ...)
```
`sprintf` действует так же, как и `printf`, только вывод осуществляет в стринг `s`, завершая его литерой `'\0'`. Стринг `s` должен быть достаточ-

но большим, чтобы вмещать результат вывода. Возвращает количество записанных литер, в число которых литера `'\0'` не входит.

```
vprintf(const char *format, va_list arg)
vfprintf(FILE *stream, const char *format, va_list arg)
vsprintf(char *s, const char *format, va_list arg)
```

Функции `vprintf`, `vfprintf` и `vsprintf` эквивалентны соответствующим `printf`-функциям с той лишь разницей, что переменный список аргументов представлен параметром `arg`, инициализированным макросом `va_start` и, возможно, вызовами `va_arg`. (См. в B.7 описание `<stdarg.h>`.)

B.1.3. Форматный ввод

Функции `scanf` имеют дело с форматным преобразованием при вводе.

```
int fscanf(FILE *stream, const char *format, ...)
```

`fscanf` читает данные из потока `stream` под управлением `format` и преобразованные величины присваивает по порядку аргументам, *каждый из которых должен быть указателем*. Завершает работу, если исчерпался формат. Выдаёт `EOF` по исчерпании файла или перед любым преобразованием, если возникла ошибка; в остальных случаях она возвращает количество преобразованных и введённых элементов.

Форматный стринг обычно содержит спецификации преобразования, которые используются для управления вводом. В форматный стринг могут входить:

- Пробелы и табуляции, которые игнорируются.
- Обычные литеры (но не `%`), которые ожидаются в потоке ввода среди литер, отличных от пробельных.
- Спецификации преобразования, состоящие из `%`; необязательного знака `*`, подавляющего присваивание; необязательного числа, специфицирующего максимальную ширину поля; необязательных `h`, `l` или `L`, указывающих размер присваиваемого значения, и литеры-спецификатора преобразования.

Спецификация преобразования определяет преобразование следующего поля ввода. Обычно результат размещается в переменной, на которую указывает соответствующий аргумент. Однако если присваивание подавляется при помощи `*`, как, например, в `%*s`, то поле ввода просто пропускается, и никакого присваивания не происходит. Поле ввода определяется как стринг непробельных литер; при этом ввод стринга прекращается при выполнении любого из двух условий: если встретилась пробельная литера или

если ширина поля (в случае, когда она указана) исчерпана. Из этого следует, что при переходе к следующему полю scanf может «перешагивать» через границы строк, поскольку литера новая-строка является пробельной. (Под пробельными понимаются литеры пробела, табуляции, новой-строки, возврата-каретки, вертикальной-табуляции и смены-страницы.)

Литера-спецификатор указывает на способ интерпретации поля ввода. Соответствующий аргумент должен быть указателем. Список допустимых литер-спецификаторов приводится в табл. B.2.

Литерам-спецификаторам d, i, n, o, u и x может предшествовать h, если аргумент есть указатель на short (а не int) или l (буква эль), если аргумент есть указатель на long. Литерам-спецификаторам e, f и g может предшествовать l, если аргумент – указатель на double (а не float), или L, если аргумент – указатель на long double.

`int scanf(const char *format, ...)`
: `scanf(...)` делает то же, что и `fscanf(stdin, ...)`.

`int sscanf(char *s, const char *format, ...)`
: `sscanf(s,...)` делает то же, что и `scanf(...)`, только ввод литер осуществляет из стринга s.

B.1.4. Функции ввода-вывода литер

`int fgetc(FILE *stream)`
: fgetc возвращает следующую литеру из потока stream в виде unsigned char (переведённую в int) или EOF, если исчерпан файл или обнаружена ошибка.

`char *fgets(char *s, int n, FILE *stream)`
: fgets читает не более n–1 литер в массив s, прекращая чтение, если встретилась литера новая-строка, которая включается в массив; кроме того, записывает в массив '\0'. fgets возвращает s или, если исчерпан файл или обнаружена ошибка, NULL.

`int fputc(int c, FILE *stream)`
: fputc пишет литеру c (переведённую в unsigned char) в stream. Возвращает записанную литеру или EOF в случае ошибки.

`int fputs(const char *s, FILE *stream)`
: fputs пишет стринг s (который может не иметь '\n') в stream; возвращает неотрицательное целое или EOF в случае ошибки.

ТАБЛИЦА B.2. ПРЕОБРАЗОВАНИЯ SCANF

ЛИТЕРА	ДАННЫЕ НА ВВОДЕ; ТИП АРГУМЕНТА
d	десятичное целое; int *.
i	целое; int *. Целое может быть восьмеричным (с ведущим нулём) или шестнадцатеричным (с ведущими 0x или 0X).
o	восьмеричное целое (с ведущим нулём или без него); int *.
u	беззнаковое десятичное целое; unsigned int *.
x	шестнадцатеричное целое (с ведущими 0x или 0X или без); int *.
c	литеры; char *. Литеры ввода размещаются в указанном массиве в количестве, заданном шириной поля; по умолчанию это количество равно 1. Литера '\0' не добавляется. Пробельные литеры здесь рассматриваются как обычные литеры и поступают в аргумент. Чтобы прочесть следующую непробельную литеру, используйте %1s.
s	стринг непробельных литер (записывается без кавычек); char *, указывающий на массив размера достаточного, чтобы вместить стринг и добавляемую к нему литеру '\0'.
e,f,g	число с плавающей точкой; float *. Формат ввода для float состоит из необязательного знака, стринга цифр, возможно, с десятичной точкой и необязательной экспоненты, состоящей из E или e и целого, возможно, со знаком.
p	значение указателя в виде, в котором printf("%p") его напечатает; void *.
n	записывает в аргумент число литер, прочитанных к этому моменту в этом вызове; int *. Никакого чтения ввода не происходит. Счётчик числа введённых элементов не увеличивается.
[...]	выбирает из ввода самый длинный непустой стринг, состоящий из литер, заданных в квадратных скобках; char *. В конец стринга добавляется '\0'. Спецификатор вида []...] включает] в задаваемое множество литер.
[^...]	выбирает из ввода самый длинный непустой стринг, состоящий из литер, *не входящих* в заданное в скобках множество. В конец добавляется '\0'. Спецификатор вида [^]...] включает] в задаваемое множество литер.
%	обычная литера %; присваивание не делается.

`int getc(FILE *stream)`

getc делает то же, что и fgetc, но в отличие от последней, если она – макрос, stream может быть вычислен более одного раза.

`int getchar(void)`
: `getchar()` делает то же, что `getc(stdin)`.

`char *gets(char *s)`
: `gets` читает следующую строку ввода в массив `s`, заменяя литеру новая-строка на `'\0'`. Возвращает `s` или, если исчерпан файл или обнаружена ошибка, `NULL`.

`int putc(int c, FILE *stream)`
: `putc` делает то же, что и `fputc`, но в отличие от последней, если она – макрос, `stream` может быть вычислен более одного раза.

`int putchar(int c)`
: `putchar(c)` делает то же, что `putc(c, stdout)`.

`int puts(const char *s)`
: `puts` пишет стринг `s` и литеру новая-строка в `stdout`. Возвращает `EOF`, в случае ошибки, или неотрицательное значение, если запись прошла нормально.

`int ungetc(int c, FILE *stream)`
: `ungetc` отправляет литеру `c` (переведённую в `unsigned char`) обратно в `stream`; при следующем чтении из `stream` она будет получена снова. Для каждого потока вернуть можно не более одной литеры. Нельзя возвращать `EOF`. В качестве результата `ungetc` выдаёт отправленную назад литеру или, в случае ошибки, `EOF`.

B.1.5. Функции прямого ввода-вывода

`size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)`
: `fread` читает из потока `stream` в массив `ptr` не более `nobj` объектов размера `size`. Она возвращает количество прочитанных объектов, которое может быть меньше заявленного. Для индикации состояния после чтения следует использовать `feof` и `ferror`.

`size_t fwrite(const void *ptr, size_t size, size_t nobj,`
`FILE *stream)`
: `fwrite` пишет из массива `ptr` в `stream` `nobj` объектов размера `size`; возвращает число записанных объектов, которое в случае ошибки меньше `nobj`.

B.1.6. Функции позиционирования файла

`int fseek(FILE *stream, long offset, int origin)`
: `fseek` устанавливает позицию для `stream`; последующее чтение или запись будет производиться с этой позиции. В случае бинарного файла позиция устанавливается со смещением `offset` – относительно начала, если `origin` равен `SEEK_SET`; относительно текущей позиции, если `origin` равен `SEEK_CUR`; и относительно конца файла, если `origin` равен `SEEK_END`. Для текстового файла `offset` должен быть нулём или значением, полученным при помощи вызова функции `ftell`. При работе с текстовым файлом `origin` всегда должен быть равен `SEEK_SET`.

`long ftell(FILE *stream)`
: `ftell` возвращает текущую позицию потока `stream` или `-1L`, в случае ошибки.

`void rewind(FILE *stream)`
: `rewind(fp)` делает то же, что и `fseek(fp, 0L, SEEK_SET); clearerr(fp)`.

`int fgetpos(FILE *stream, fpos_t *ptr)`
: `fgetpos` записывает текущую позицию потока `stream` в `*ptr` для последующего использования её в `fsetpos`. Тип `fpos_t` позволяет хранить такого рода значения. В случае ошибки `fgetpos` возвращает ненулевое значение.

`int fsetpos(FILE *stream, const fpos_t *ptr)`
: `fsetpos` устанавливает позицию в `stream`, читая её из `*ptr`, куда она была записана ранее при помощи `fgetpos`. В случае ошибки `fsetpos` возвращает ненулевое значение.

B.1.7. Функции обработки ошибок

Многие функции библиотеки в случае ошибки или конца файла устанавливают индикаторы состояния. Эти индикаторы можно проверять и изменять. Кроме того, целое выражение `errno` (декларированное в `<errno.h>`) может содержать номер ошибки, который даёт дополнительную информацию о последней из обнаруженных ошибок.

`void clearerr(FILE *stream)`
: `clearerr` очищает индикаторы конца файла и ошибки потока `stream`.

`int feof(FILE *stream)`
: `feof` возвращает ненулевое значение, если для потока `stream` установлен индикатор конца файла.

`int ferror(FILE *stream)`
: `ferror` возвращает ненулевое значение, если для потока `stream` установлен индикатор ошибки.

`void perror(const char *s)`
: `perror(s)` печатает `s` и реализационно-зависимое сообщение об ошибке, соответствующее целому значению в `errno`, т.е. делает то же, что и обращение к функции `fprintf` вида

```
fprintf(stderr, "%s: %s\n", s, "сообщение об ошибке")
```

См. `strerror` в разд. B.3.

B.2. Проверки класса литеры: <ctype.h>

Головной файл <ctype.h> декларирует функции, предназначенные для проверок литер. Аргумент каждой из них имеет тип `int` и должен либо представлять собой `EOF`, либо быть значением `unsigned char`, приведённым к `int`; возвращаемое значение тоже имеет тип `int`. Функции возвращают ненулевое значение («истина»), когда аргумент `c` удовлетворяет описанному условию или принадлежит указанному классу литер, и нуль в противном случае.

`isalnum(c)`	`isalpha(c)` или `isdigit(c)` есть истина
`isalpha(c)`	`isupper(c)` или `islower(c)` есть истина
`iscntrl(c)`	управляющая литера
`isdigit(c)`	десятичная цифра
`isgraph(c)`	печатаемая литера кроме пробела
`islower(c)`	буква нижнего регистра
`isprint(c)`	печатаемая литера, включая пробел
`ispunct(c)`	печатаемая литера кроме пробела, буквы или цифры
`isspace(c)`	пробел, смена-страницы, новая-строка, возврат-каретки, табуляция, вертикальная-табуляция
`isupper(c)`	буква верхнего регистра
`isxdigit(c)`	шестнадцатеричная цифра

В наборе семибитовых ASCII-литер печатаемые литеры находятся в диапазоне от `0x20` (`' '`) до `0x7E` (`'~'`); управляющие литеры – от `0` (`NUL`) до `0x1F` (`US`) и `0x7F` (`DEL`).

Помимо перечисленных есть две функции, приводящие буквы к одному из регистров:

`int tolower(int c)`	переводит `c` на нижний регистр
`int toupper(int c)`	переводит `c` на верхний регистр

Если c – буква на верхнем регистре, то tolower(c) выдаст эту букву на нижнем регистре; в противном случае она вернёт c. Если c – буква на нижнем регистре, то toupper(c) выдаст эту букву на верхнем регистре; в противном случае она вернёт c.

B.3. Функции, оперирующие со стрингами: <string.h>

Имеются две группы функций, оперирующих со стрингами. Они определены в головном файле <string.h>. Имена функций первой группы начинаются с str, второй – с mem. Если копирование имеет дело с объектами, перекрывающимися по памяти, то, за исключением memmove, поведение функций не определено. Функции сравнения рассматривают аргументы как массивы элементов типа unsigned char.

В следующей таблице переменные s и t принадлежат типу char *, cs и ct – типу const char *, n – типу size_t, а c – значение типа int, приведённое к типу char.

`char *strcpy(s,ct)`	копирует стринг ct в стринг s, включая '\0'; возвращает s.
`char *strncpy(s,ct,n)`	копирует не более n литер стринга ct в s; возвращает s. Дополняет результат литерами '\0', если литер в ct меньше n.
`char *strcat(s,ct)`	конкатенирует ct к s; возвращает s.
`char *strncat(s,ct,n)`	конкатенирует не более n литер ct к s, завершая s литерой '\0'; возвращает s.
`int strcmp(cs,ct)`	сравнивает cs с ct; возвращает < 0, если cs<ct, 0, если cs==ct, и > 0, если cs>ct.
`int strncmp(cs,ct,n)`	сравнивает не более n литер 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, если такового не оказалось, NULL.
`char *strrchr(cs,c)`	возвращает указатель на последнее вхождение c в cs или, если такового не оказалось, NULL.
`size_t strspn(cs,ct)`	возвращает длину начала cs, состоящего из литер, входящих в стринг ct.
`size_t strcspn(cs,ct)`	возвращает длину начала cs, состоящего из литер, *не* входящих в стринг ct.
`char *strpbrk(cs,ct)`	возвращает указатель в cs на первую литеру, которая совпала с одной из литер, входящих в ct, или, если такой не оказалось, NULL.
`char *strstr(cs,ct)`	возвращает указатель на первое вхождение ct в cs или, если такового не оказалось, NULL.

`size_t strlen(cs)`	возвращает длину `cs`.
`char *strerror(n)`	возвращает указатель на реализационно-зависимый стринг, соответствующий номеру ошибки `n`.
`char *strtok(s,ct)`	`strtok` ищет в `s` лексему, ограниченную литерами из `ct`; (более подробное описание этой функции см. ниже).

Последовательные вызовы `strtok` разбивают стринг `s` на лексемы. Ограничителем лексемы может быть любая литера из числа входящих в `ct`. В первом вызове указатель `s` не есть `NULL`. Функция находит в стринге `s` первую лексему, состоящую из литер, не входящих в `ct`; её работа заканчивается тем, что поверх следующей литеры пишется `'\0'` и возвращается указатель на лексему. Каждый последующий вызов, в котором указатель `s` равен `NULL`, выдаёт указатель на следующую лексему, которую функция будет искать сразу за концом предыдущей. Функция `strtok` возвращает `NULL`, если далее никакой лексемы не обнаружено. Параметр `ct` от вызова к вызову может варьироваться.

Функции `mem...` предназначены для манипулирования с объектами как с массивами литер; их назначение – получить интерфейсы к эффективным программам. В приведённой ниже таблице `s` и `t` принадлежат типу `void *`; `cs` и `ct` – `const void *`; `n` – `size_t`; а `c` – значение типа `int`, приведённое к типу `char`.

`void *memcpy(s,ct,n)`	копирует `n` литер из `ct` в `s` и возвращает `s`.
`void *memmove(s,ct,n)`	делает то же самое, что и `memcpy`, но работает и в случае «перекрывающихся» объектов.
`int memcmp(cs,ct,n)`	сравнивает первые `n` литер `cs` и `ct`; выдаёт тот же результат, что и функция `strcmp`.
`void *memchr(cs,c,n)`	возвращает указатель на первое вхождение литеры `c` в `cs` или, если среди первых `n` литер `c` не встретилась, `NULL`.
`void *memset(s,c,n)`	размещает литеру `c` в первых `n` позициях стринга `s` и возвращает `s`.

B.4. Математические функции: <math.h>

В головном файле <math.h> описываются математические функции и определяются макросы.

Макросы `EDOM` и `ERANGE` (находящиеся в <errno.h>) задают отличные от нуля целочисленные константы, используемые для фиксации ошибки области и ошибки диапазона; `HUGE_VAL` определена как положительное `double`-значение. *Ошибка области* возникает, если аргумент выходит за область значений, для которой определена функция. Фиксация ошибки области осу-

ществляется установкой EDOM в errno; возвращаемое значение зависит от реализации. *Ошибка диапазона* возникает в том случае, когда результат функции не может быть представлен в виде double. В случае переполнения функция возвращает HUGE_VAL с правильным знаком и в errno устанавливает ERANGE. При исчезновении порядка функция возвращает нуль, а устанавливается ли в этом случае в errno ERANGE, зависит от реализации.

Далее x и y имеют тип double, n – тип int, и все функции возвращают значения типа double. Углы в тригонометрических функциях 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 $\log_{10}(x)$, $x > 0$

`pow(x,y)` x^y. Ошибка области, если $x = 0$ и $y \leqslant 0$ или $x < 0$ и y – не целое

`sqrt(x)` $\sqrt{x}$, $x \geqslant 0$

`ceil(x)` наименьшее целое в виде double, которое $\geqslant x$

`floor(x)` наибольшее целое в виде double, которое $\leqslant x$

`fabs(x)` абсолютное значение $|x|$

`ldexp(x,n)` $x \cdot 2^n$

`frexp(x, int *exp)`
разбивает x на два сомножителя, первый из которых – нормализованная дробь в интервале $[1/2, 1)$, которая возвращается, а второй – степень двойки, показатель которой запоминается в `*exp`. Если x – нуль, то обе части результата равны нулю.

`modf(x, double *ip)`
x разбивается на целую и дробную части, обе имеют тот же знак, что и x. Целая часть запоминается в `*ip`, дробная часть выдаётся как результат.

`fmod(x,y)` остаток от деления x на y в виде числа с плавающей точкой. Знак результата совпадает со знаком x. При $y = 0$, результат зависит от реализации.

B.5. Функции общего назначения: <stdlib.h>

Головной файл <stdlib.h> декларирует функции, предназначенные для преобразования чисел, запроса памяти и других задач.

`double atof(const char *s)`
atof переводит s в double; эквивалентна strtod(s, (char**)NULL).

`int atoi(const char *s)`
переводит s в int; эквивалентна (int)strtol(s, (char**)NULL, 10).

`int atol(const char *s)`
переводит s в long; эквивалентна strtol(s, (char**)NULL, 10).

`double strtod(const char *s, char **endp)`
strtod преобразует первые литеры s в double, игнорируя начальные пробельные литеры; запоминает указатель на непреобразованный конец в *endp (если endp не NULL). В случае переполнения она выдаёт HUGE_VAL с соответствующим знаком, в случае исчезновения порядка – 0; в обоих случаях в errno устанавливается ERANGE.

`long strtol(const char *s, char **endp, int base)`
strtol преобразует первые литеры s в long, игнорируя начальные пробельные литеры; запоминает указатель на непреобразованный конец в *endp (если endp не NULL). Если base находится в диапазоне от 2 до 36, то преобразование делается в предположении, что на входе – запись числа по основанию base. Если base равен нулю, то основанием числа считается 8, 10 или 16; число, начинающееся с цифры 0, считается восьмеричным, а с 0x или 0X – шестнадцатеричным. Цифры от 10 до base–1 записываются начальными буквами латинского алфавита в любом регистре. В случае основания 16 в начале числа разрешается помещать 0x или 0X. При переполнении функция возвращает LONG_MAX или LONG_MIN (в зависимости от знака), а в errno устанавливается ERANGE.

`unsigned long strtoul(const char *s, char **endp, int base)`
strtoul работает так же, как и strtol, с той только разницей, что выдаёт результат типа unsigned long, а в случае переполнения – ULONG_MAX.

`int rand(void)`
rand выдаёт псевдослучайное число в диапазоне от 0 до RAND_MAX; RAND_MAX не меньше 32767.

`void srand(unsigned int seed)`
srand использует seed в качестве «затравки» для новой последовательности псевдослучайных чисел. Изначально параметр seed равен 1.

`void *calloc(size_t nobj, size_t size)`
calloc возвращает указатель на место в памяти, отведённое для массива nobj объектов, каждый из которых размера size, или, если памяти запрашиваемого объёма нет, NULL. Выделенная область памяти обнуляется.

`void *malloc(size_t size)`
malloc возвращает указатель на место в памяти для объекта размера size или, если памяти запрашиваемого объёма нет, NULL. Выделенная область памяти не инициализируется.

`void *realloc(void *p, size_t size)`
realloc заменяет размер объекта, на который указывает p, на size. Для части, размер которой равен наименьшему из старого и нового размеров, содержимое не изменяется. Если новый размер больше старого, дополнительное пространство не инициализируется. realloc возвращает указатель на новое место памяти или, если требования не могут быть удовлетворены, NULL (*p при этом не изменяется).

`void free(void *p)`
free освобождает область памяти, на которую указывает p; эта функция ничего не делает, если p есть NULL. В p должен стоять указатель на область памяти, ранее выделенную одной из функций: calloc, malloc или realloc.

`void abort(void)`
abort вызывает аварийное завершение программы, её действия эквивалентны вызову raise(SIGABRT).

`void exit(int status)`
exit вызывает нормальное завершение программы. Функции, зарегистрированные при помощи atexit, выполняются в порядке, обратном их регистрации. Производится выталкивание буферов открытых файлов, открытые потоки закрываются, и управление возвращается в среду, из которой был произведён запуск программы. Значение status, передаваемое в среду, зависит от реализации, однако при успешном завершении программы это всегда нуль. Можно также использовать значения EXIT_SUCCESS (в случае успешного завершения) и EXIT_FAILURE (в случае ошибки).

```
int atexit(void (*fcn)(void))
```
atexit регистрирует fcn в качестве функции, которая будет вызываться при нормальном завершении программы; возвращает ненулевое значение, если регистрация не может быть сделана.

```
int system(const char *s)
```
system передаёт стринг s операционной среде для выполнения. Если s есть NULL и существует командный процессор, то system возвращает ненулевое значение. Если s не NULL, то возвращаемое значение зависит от реализации.

```
char *getenv(const char *name)
```
getenv возвращает стринг среды, связанный с name, или, если никакого стринга не существует, NULL. 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))
```
bsearch среди base[0]...base[n-1] ищет элемент с подходящим ключом *key. Функция cmp сравнивает первый аргумент (ключ поиска) со своим вторым аргументом (значением ключа в таблице) и в зависимости от результата сравнения выдаёт отрицательное число, нуль или положительное значение. Элементы массива base должны быть упорядочены в возрастающем порядке. bsearch возвращает указатель на элемент с подходящим ключом или, если такого не оказалось, NULL.

```
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *, const void *))
```
qsort сортирует массив base[0]...base[n-1] объектов размера size в возрастающем порядке. Функция сравнения cmp – такая же, что и в bsearch.

```
int abs(int n)
```
abs возвращает абсолютное значение int-аргумента.

```
long labs(long n)
```
labs возвращает абсолютное значение long-аргумента.

```
div_t div(int num, int denom)
```
div вычисляет частное и остаток от деления num на denom. Результаты запоминаются в int-членах quot и rem структуры div_t.

```
ldiv_t ldiv(long num, long denom)
```
ldiv вычисляет частное и остаток от деления num на denom. Результаты запоминаются в long-членах quot и rem структуры ldiv_t.

B.6. Диагностика: <assert.h>

Макрос assert используется для включения в программу диагностических сообщений.

```
void assert(int выражение)
```

Если *выражение* есть нуль, то

```
assert(выражение)
```

напечатает в stderr сообщение следующего вида:

```
Assertion failed: выражение, file имя-файла, line ппп
```

после чего будет вызвана функция abort, которая завершит вычисления. Имя исходного файла и номер строки будут взяты из макросов __FILE__ и __LINE__.

Если в момент включения файла <assert.h> было определено имя NDEBUG, то макрос assert игнорируется.

B.7. Списки аргументов переменной длины: <stdarg.h>

Головной файл <stdarg.h> предоставляет средства для перебора аргументов функции, количество и типы которых заранее не известны.

Пусть *посларг* – последний именованный параметр функции f с переменным числом аргументов. Внутри f декларируется переменная ap типа va_list, предназначенная для хранения указателя на очередной аргумент:

```
va_list ap;
```

Прежде чем будет возможен доступ к безымянным аргументам, необходимо один раз инициализировать ap, обратившись к макросу va_start:

```
va_start(va_list ap, посларг);
```

С этого момента каждое обращение к макросу:

```
тип va_arg(va_list ap, тип);
```

будет давать значение очередного безымянного аргумента указанного типа, и каждое такое обращение будет вызывать автоматическое продвижение указателя ap, чтобы последний был готов к выдаче следующего аргумента. Один раз после перебора аргументов, но до выхода из f необходимо обратиться к макросу

```
void va_end(va_list ap);
```

B.8. Далёкие переходы: <setjmp.h>

Декларации в <setjmp.h> предоставляют способ отклониться от обычной последовательности «вызов – возврат»; типичная ситуация – необходимость вернуться из «глубоко вложенного» вызова функции на верхний уровень, минуя промежуточные возвраты.

`int setjmp(jmp_buf env)`

Макрос setjmp сохраняет текущую информацию о вызовах в env для последующего её использования в longjmp. Возвращает нуль, если возврат осуществляется непосредственно из setjmp, и не нуль, если – от последующего вызова longjmp. Обращение к setjmp возможно только в определённых контекстах; в основном это проверки в if, switch и циклах, причём только в простых выражениях отношения.

```
if (setjmp(env) == 0)
    /* после прямого возврата */
else
    /* после возврата из longjmp */
```

`void longjmp(jmp_buf env, int val)`

longjmp восстанавливает информацию, сохранённую в самом последнем вызове setjmp, по информации из env; счёт возобновляется, как если бы функция setjmp только что отработала и вернула ненулевое значение val. Результат будет непредсказуемым, если в момент обращения к longjmp функция, содержащая вызов setjmp, уже «отработала» и осуществила возврат. Доступные ей объекты имеют те значения, которые они имели в момент обращения к longjmp; setjmp не сохраняет значений.

B.9. Сигналы: <signal.h>

Головной файл <signal.h> предоставляет средства для обработки исключительных ситуаций, возникающих во время выполнения программы, таких, как прерывание, вызванное внешним источником или ошибкой в вычислениях.

`void (*signal(int sig, void (*handler)(int)))(int)`

signal устанавливает, как будут обрабатываться последующие сигналы. Если параметр handler имеет значение SIG_DFL, то используется реализационно-зависимая «обработка по умолчанию»; если значение handler равно SIG_IGN, то сигнал игнорируется; в остальных случаях будет выполнено обращение к функции, на которую указывает handler с типом сигнала в качестве аргумента. В число допустимых видов сигналов входят:

SIGABRT	аварийное завершение, например, от abort
SIGFPE	арифметическая ошибка: деление на 0 или переполнение
SIGILL	неверный код функции (недопустимая команда)
SIGINT	запрос на взаимодействие, например, прерывание
SIGSEGV	неверный доступ к памяти, например, выход за границы
SIGTERM	требование завершения, посланное в программу

signal возвращает предыдущее значение handler – в случае специфицированного сигнала, или SIG_ERR – в случае возникновения ошибки.

Когда в дальнейшем появляется сигнал sig, сначала восстанавливается готовность поведения «по умолчанию», после чего вызывается функция, заданная в параметре handler, т.е. как бы выполняется вызов (*handler)(sig). Если handler-функция вернёт назад управление, то вычисления возобновятся с того места, в каком застал программу пришедший сигнал.

Начальное состояние сигналов зависит от реализации.

`int raise(int sig)`

raise посылает сигнал sig в программу. В случае неудачи возвращает ненулевое значение.

B.10. Функции даты и времени: <time.h>

Головной файл <time.h> декларирует типы и функции, связанные с датой и временем. Некоторые функции имеют дело с *местным временем*, которое может отличаться от календарного, например, в связи с зонированием времени. Типы clock_t и time_t – арифметические типы для представления времени, а 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 после 1900
int tm_wday;	дни после воскресенья (0, 6)
int tm_yday;	дни после 1 января (0, 365)
int tm_isdst;	признак светлого времени

Значение tm_isdst – положительное, если время приходится на светлую часть суток, нуль в противном случае и отрицательное, если информация не доступна.

`clock_t clock(void)`

clock возвращает время, фиксируемое процессором от начала счёта

программы, или –1, если оно не известно. Для выражения этого времени в секундах применяется формула `clock()/CLOCKS_PER_SEC`.

`time_t time(time_t *tp)`
: `time` возвращает текущее календарное время или –1, если время не известно. Если `tp` не `NULL`, то возвращаемое значение записывается и в `*tp`.

`double difftime(time_t time2, time_t time1)`
: `difftime` возвращает разность `time2-time1`, выраженную в секундах.

`time_t mktime(struct tm *tp)`
: `mktime` преобразует местное время, заданное структурой `*tp`, в календарное, выдавая его в том же виде, что и функция `time`. Компоненты будут иметь значения в указанных диапазонах. Функция `mktime` возвращает календарное время или –1, если оно не представимо.

Следующие четыре функции возвращают указатели на статические объекты, каждый из которых может быть изменён другими вызовами.

`char *asctime(const struct tm *tp)`
: `asctime` переводит время в структуре `*tp` в стринг вида

```
Sun Jan  3  15:14:13  1988\n\0
```

`char *ctime(const time_t *tp)`
: `ctime` переводит календарное время в местное, что эквивалентно выполнению

```
asctime(localtime(tp))
```

`struct tm *gmtime(const time_t *tp)`
: `gmtime` переводит календарное время во Всемирное координированное время (Coordinated Universal Time – UTC). Выдаёт `NULL`, если UTC не известно. Имя этой функции, `gmtime`, происходит от Greenwich Mean Time (среднее время по Гринвичскому меридиану).

`struct tm *localtime(const time_t *tp)`
: `localtime` переводит календарное время `*tp` в местное.

```
size_t strftime(char *s, size_t smax, const char *fmt,
                const struct tm *tp)
```
: `strftime` форматирует информацию о дате и времени из `*tp` в стринг `s` согласно формату `fmt`, который имеет много общих черт с форматом, задаваемым в функции `printf`. Обычные литеры (включая и завершающую литеру `'\0'`) копируются в `s`. Каждая пара, состоящая из `%` и

буквы, заменяется, как показано ниже, с использованием значений по форме, соответствующей локальной среде. В `s` размещается не более `smax` литер. `strftime` возвращает число литер без учёта `'\0'` или нуль, если число сгенерированных литер больше `smax`.

- `%a` сокращённое название недели.
- `%A` полное название недели.
- `%b` сокращённое название месяца.
- `%B` полное название месяца.
- `%c` локальное представление даты и времени.
- `%d` день месяца (01–31).
- `%H` час (24-часовое время) (00–23).
- `%I` час (12-часовое время) (01–12).
- `%j` день года (001–366).
- `%m` месяц (01–12).
- `%M` минута (00–59).
- `%p` локальное представление AM и PM (до и после полудня).
- `%S` секунда (00–61).
- `%U` неделя по счёту от начала года (воскресенье – 1-й день недели) (00–53).
- `%w` день недели (0–6, номер воскресенья есть 0).
- `%W` неделя по счёту от начала года (понедельник – 1-й день недели) (00–53).
- `%x` локальное представление даты.
- `%X` локальное представление времени.
- `%y` год без указания века (00–99).
- `%Y` год с указанием века.
- `%Z` название временной зоны, если оно есть.
- `%%` %.

B.11. Реализационно-зависимые пределы: <limits.h> и <float.h>

Головной файл <limits.h> определяет константы для размеров целочисленных типов. Ниже перечислены минимальные приемлемые величины, но в конкретных реализациях могут использоваться и большие значения.

`CHAR_BIT`	8	бит в значении `char`
`CHAR_MAX`	`UCHAR_MAX` *или* `SCHAR_MAX`	максимальное значение `char`
`CHAR_MIN`	0 *или* `SCHAR_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, приведённые в следующей таблице, взяты из <float.h> и являются константами, имеющими отношение к арифметике с плавающей точкой. Значения (если они есть) представляют собой минимальные значения для соответствующих величин. В каждой реализации устанавливаются свои значения.

FLT_RADIX	2	основание представления экспоненты: например, 2, 16
FLT_ROUNDS		способ округления при сложении чисел с плавающей точкой
FLT_DIG	6	количество верных десятичных цифр
FLT_EPSILON	1E−5	минимальное x, такое, что $1.0 + x \neq 1.0$
FLT_MANT_DIG		количество цифр по основанию FLT_RADIX в мантиссе
FLT_MAX	1E+37	максимальное число с плавающей точкой
FLT_MAX_EXP		максимальное n, такое, что $\text{FLT\_RADIX}^n - 1$ представимо
FLT_MIN	1E−37	минимальное нормализованное число с плавающей точкой
FLT_MIN_EXP		минимальное n, такое, что 10^n представимо в виде нормализованного числа
DBL_DIG	10	количество верных десятичных цифр для типа double
DBL_EPSILON	1E−9	минимальное x, такое, что $1.0 + x \neq 1.0$, где x принадлежит типу double
DBL_MANT_DIG		количество цифр по основанию FLT_RADIX в мантиссе для чисел типа double
DBL_MAX	1E+37	максимальное число с плавающей точкой типа double
DBL_MAX_EXP		максимальное n, такое, что $\text{FLT\_RADIX}^n - 1$ представимо в виде числа типа double
DBL_MIN	1E−37	минимальное нормализованное число с плавающей точкой типа double

`DBL_MIN_EXP`	минимальное n, такое, что 10^n представимо в виде нормализованного числа типа `double`

Приложение С

Перечень изменений

С момента публикации первого издания этой книги определение языка Си претерпело изменения. Почти все нововведения – это расширения исходной версии языка, выполненные так, чтобы сохранилась совместимость с существующими программами; некоторые изменения касаются устранения двусмысленностей первоначального описания, а некоторые представляют собой модификации, привнесённые существующей практикой. Многие из новых возможностей, впоследствии принятые другими разработчиками Си-компиляторов, были первоначально объявлены в документах, прилагаемых к компиляторам. Комитет ANSI, подготавливая стандарт языка, включил большинство этих изменений, а также ввёл другие значительные модификации. Некоторые коммерческие компиляторы реализовали их ещё до выпуска официального Си-стандарта.

В этом приложении сведены воедино различия между языком, определённым в первой его редакции, и той его версии, которая принята в качестве стандарта. Здесь рассматривается только сам язык, вопросы, относящиеся к его окружению и библиотеке, не затрагиваются. Хотя последние и являются важной частью стандарта, но, поскольку в первом издании не делалось попытки описать окружение и библиотеку, сравнить их с соответствующими стандартными элементами оказывается невозможным.

- В стандарте более тщательно, по сравнению с первым изданием, определено и расширено препроцессирование: в его основу явно положены лексемы; введены новые операторы для «склеивания» лексем (`##`) и создания стрингов (`#`), а также новые управляющие строки типа `#elif` и `#pragma`; разрешено повторное определение макроса с той же последовательностью лексем; отменена подстановка параметров внутри стрингов. Разрешено «склеивание» строк при помощи \ в любом месте, но только не в стрингах и макроопределениях.

- Минимальное число значимых литер всех внутренних идентификаторов доведено до 31; для идентификаторов с внешней связью оно остаётся равным 6 с неразличимостью букв на нижнем и верхнем регистрах. (Многие реализации допускают бóльшее число значимых литер.)

- Для знаков `#\^[]{}|~`, которых может не быть в некоторых наборах литер, введены трёхзнаковые последовательности, начинающиеся с `??` (См. A.12.1). Следует заметить, что введение трёхзнаковых последовательностей может «испортить» значения стрингов, в которых содержатся `??`.

- Введены новые ключевые слова (`void`, `const`, `volatile`, `signed`, `enum`), а мертворождённое слово `entry` из обращения изъято.

- Для литеральных констант и стринговых литералов определены новые эскейп-последовательности. Объявлено, что появление за `\` «неправильных» литер приводит к непредсказуемому результату. (См. A.2.5.2.)

- Узаконено полюбившееся всем тривиальное изменение: 8 и 9 не являются восьмеричными цифрами.

- Введён расширенный набор суффиксов для явного указания типов констант: `U` и `L` – для целых и `F` и `L` – для плавающих. Уточнены также правила определения типа для констант без суффиксов (A.2.5).

- Объявлено, что соседние стринги конкатенируются.

- Предоставлены средства, позволяющие записывать стринговые литералы и литерные константы из расширенного набора литер (A.2.6).

- Объект типа `char` (как и объект другого типа) можно специфицировать явно со знаком или без знака. Исключается использование словосочетания `long float` в смысле `double`, но вводится тип `long double` для чисел с плавающей точкой повышенной точности.

- С некоторых пор доступен тип `unsigned char`. Стандарт вводит ключевое слово `signed` для явного указания знаковости `char` и других целочисленных объектов.

- Уже несколько лет в большинстве реализаций доступен тип `void`. Стандарт вводит `void *` в качестве типа обобщённого указателя; раньше для этой цели использовали `char *`. Одновременно вступают в силу правила, по которым запрещается без преобразования типа «смешивать» указатели и целые или указатели разных типов.

- Стандарт устанавливает минимальные пределы диапазонов арифметических типов, предусматривает головные файлы `<limits.h>` и `<float.h>`, в которых помещаются эти характеристики для каждой конкретной реализации.

- Перечисление – новый тип, которого не было в первой редакции.

- Стандарт заимствует из Си++ способ записи квалификатора типа, в частности квалификатора `const` (A.8.2).

- Вводится запрет на модификацию стрингов; это значит, что их разрешается размещать в памяти, доступной только на чтение.

- Изменены «обычные арифметические преобразования», по существу, осуществлён переход от «для целых всегда превалирует `unsigned`; для плавающей точки всегда используется `double`» к «повышению до минимального достаточно вместимого типа». (См. A.6.5.)

- Отменены старые операторы присваивания типа `=+`. Каждый оператор присваивания теперь представляется одной отдельной лексемой. В первом издании оператор присваивания мог изображаться парой литер, возможно, разделённых пробельными литерами.

- Компиляторам более не разрешается трактовать математическую ассоциативность операторов как вычислительную ассоциативность.

- Введён унарный оператор `+` для симметрии с унарным `–`.

- Разрешено использовать указатель на функцию в качестве её именователя без явного оператора `*`. (См. A.7.3.2.)

- Со структурами разрешено оперировать как с целыми объектами при присваиваниях, передачах их в качестве аргументов функциям и получении их в качестве результата от функций.

- Разрешено применять оператор получения адреса `&` к массиву; результатом будет указатель на массив.

- В первой редакции результат `sizeof`-операции имел тип `int`; во многих реализациях он заменён на `unsigned`. Стандарт официально объявляет его реализационно-зависимым, но требует, чтобы он был определён в головном файле `<stddef.h>` под именем `size_t`. Аналогичное изменение было сделано в отношении типа разности указателей, который теперь выступает под именем `ptrdiff_t`. (См. A.7.4.8 и A.7.7)

- Запрещено применять оператор получения адреса `&` к `register`-объекту даже тогда, когда данный компилятор не располагает его на регистре.

- Типом результата операции сдвига является тип её левого операнда; тип правого операнда на повышение типа результата влияния не оказывает. (См. A.7.8.)

- Стандарт разрешает адресоваться при помощи указателей на место, лежащее сразу за последним элементом массива, и позволяет оперировать с такими указателями, как с обычными, см. A.7.7.

- Стандарт вводит (заимствованный из Си++) способ записи прототипа функции с включением в него типов параметров и явного указания о варьируемости их числа и формализует метод работы с переменным списком аргументов. (См. A.7.3.2, A.8.6.3, B.7.) С некоторыми ограничениями доступен старый способ записи.

- Стандартом запрещены пустые декларации, т.е. такие, в которых нет деклараторов и не декларируется ни одной структуры, объединения или перечисления. Однако декларация с одним тегом структуры (или объединения) передекларирует её даже в том случае, если она была объявлена во внешней области действия.

- Запрещены декларации внешних данных, не имеющие спецификаторов и квалификаторов (т.е. декларации с одним «голым» декларатором).

- В некоторые реализациях, когда `extern`-декларация расположена во внутреннем блоке, её действие распространяется на остальную часть файла. Стандарт вносит ясность в эту ситуацию и объявляет, что область действия такой декларации ограничена блоком.

- Область действия параметров «вставляется» в составную инструкцию, представляющую собой тело функции, так что декларации на верхнем уровне функции не могут их «затенить».

- Несколько изменены именные пространства идентификаторов. Всем тегам структур, объединений и перечислений стандарт выделяет одно именное пространство; для меток инструкций вводится отдельное именное пространство; см. A.11.1. Кроме того, имена членов связаны со структурой или объединением, частью которой они являются. (С некоторых пор это общепринятая практика.)

- Допускается инициализация объединения; инициализатор относится к первому члену объединения.

- Разрешается инициализация автоматических структур, объединений и массивов, хотя и с некоторыми ограничениями.

- Разрешается инициализация массива литер при помощи стрингового литерала по точному количеству указанных литер (без `\0`).
- Управляющее выражение и `case`-метки в `switch` могут иметь любой целочисленный тип.

Предметный указатель

В

Г

Д

З

И

П

Р

С

Т

Оглавление

Научное издание

Бриан В. Керниган, Деннис М. Ритчи

Язык программирования Си

Книга одобрена на заседании секции редсовета
по электронной обработке данных в экономике 24.02.89 г.

Зав. редакцией *А. В. Рассказов*
Редактор *А. И. Павловская*
Художественный редактор *Ю. И. Артюхов*
Технический редактор *Г. А. Полякова*
Корректоры *Г. А. Башарина*, *Г. В. Хлопцева*
Переплёт художника *Е. К. Самойлова*

ИБ № 2555

Оригинал-макет книги изготовлен
в Институте прикладной математики АН СССР
Вик. С. Штаркманом
с помощью текстового процессора ChiWriter

Подписано в печать 25.12.91. Формат 70×100 1/16. Бум. офсетная.
Гарнитура "Литературная" Печать офсетная. Усл. п. л. 22,1. Усл. кр.-отт. 22,1.
Уч.-изд. л. 19,78. Тираж 100 000 экз. Заказ 13. Цена договорная.

Издательство "Финансы и статистика"
101000, Москва, ул. Чернышевского, 7.

Типография им. И. Е. Котлякова
издательства "Финансы и статистика"
Министерства информации и печати РСФСР.
195273, Санкт-Петербург,
ул. Руставели, 13

Кернига́н Брайа́н В.
Ри́тчи Де́ннис М.

Язык программирования Си

Издание второе, переработанное и дополненное

перевод с английского языка Вик. С. Штаркман
1992 год

Перенабор книги выполнен в рамках проекта ReType (https://retype.r4in.tk)
Исходный код для индивидуальной сборки или типографской вёрстки может быть получен по адресу: https://ggs.retype.r4in.tk/rt-releases/rt-g0_1fcd69d8-KnR_C-russian

Оригинал

ed2k -
ed2k://|file|Kernigan_B.W.,Ritchie_D.M.-Yazyk_programmirovaniya_Si[2nd_ed][1992y][273p]-russian.scans.pdf|12940945|D5E8C0A31208E386BEEDEF99E5ED889D|/

torrent - magnet:?xt=urn:btih:8819eea62b5ef892440e028c14346c327c0af8f5

Замечания по соответствию оригиналу

1. Восстановлена буква "ё".
2. Исправлены замеченные опечатки и ошибки. Эти изменения относительно оригинала отмечены в исходном коде комментариями под абзацами, в которых произведены изменения.
3. Схема дерева в разделе 6.5 книги в оригинале была выполнена моноширинным шрифтом. Её оригинальное исполнение присутствует в исходном коде, но закомментировано и заменено схемой, выполненной с использованием пакета Xy-pic (http://tug.org/applications/Xy-pic/). Это сделано в связи с тем, что оригинальное представление дерева было сочтено крайне неясным и шрифтозависимым, однако, в своих сборках ничего не мешает вам раскомментировать моноширинное представление дерева и использовать его.
4. Для отображения основного текста использован шрифт "Adobe Garamond Premier Pro", однако в оригинале использовалась гарнитура "Литературная" (ГОСТ-3489.33-72). Если удастся получить набор всех начертаний гарнитуры "Литературная", этот недостаток будет исправлен.
5. Не удалось точно установить каким шрифтом в оригинале набраны исходные тексты программ, но вероятнее всего это шрифт ремингтоновских печатных машинок. Принципиально этот шрифт доступен под именем "LTC Remington Typewriter Pro", но так как реализован шрифт с ошибками при описании глифов, использовать его не представляется возможным до исправления ошибок. В связи с этим в данном коммите использована несколько избыточная система замен и подстановки моноширинных шрифтов с уменьшением кернинга. Со временем этот недостаток будет исправлен.
6. При восстановлении предметного указателя были найдены откровенные ошибки и ряд неясностей, для разрешения которых приходилось обращаться к англоязычному изданию. Кроме того, список ключей в переводе был асимметричен, то есть мог быть ключ "ААА, ББВ", но не быть ключа "ББВ, ААА", что, если было замечено, то исправлено. Поэтому предметный указатель имеет значительные расхождения с оригиналом, но его полнота и точность не снижены, а наоборот увеличены.
7. Восстановлена шутка присутствующая в оригинальном англоязычном издании, но утраченная при переводе. Шутка заключается в том, что в предметном указателе (англ. Index) в списке номеров страниц по ключевому слову "рекурсия" (англ. recursion) последним указан номер страницы, где и напечатано ключевое слово "рекурсия" предметного указателя.