WWW.BOOKS-SHOP.COM

ПРЕДСТАВЛЯЕТ:

А. Нейбауэр

Моя первая программа на C/C++

2002
WWW.BOOKS-SHOP.COM

Содержание

Как пользоваться данной книгой

Основной текст книги сопровождается материалами, которые помогут вам проверить и закрепить свои знания, получить профессиональный совет или дополнительную информацию. Для удобства читателя такие фрагменты помечены специальными значками. Вот они:

Замечания по Си++

Так отмечается дополнительная информация, в основном касающаяся особенностей языка Си++, которую при первом чтении можно пропустить.

Так выделены замечания, касающиеся использования функций, ключевых слов и различных синтаксических конструкций языка Си/Си++. Обратите особое внимание на подобные замечания, они очень важны!

Этот значок указывает на полезные советы. Они помогут вам сэкономить время и избежать распространенных ошибок.

Контрольные вопросы позволят вам проверить, насколько хорошо вы усвоили содержание раздела. Возможно, кое-что придется повторить.

Как известно, лучший способ чему-то научиться — начать действовать самостоятельно. Приводимые в книге упражнения помогут на практике овладеть конструкциями языка. Правильность своих решений можете сверить с ответами, которые вы найдете в Приложении I.

Кроме перечисленных пометок вы встретите выделения отдельных слов и фраз в основном тексте. Ключевые слова и инструкции языка Си и Си++, а также примеры программ выделены рубленым шрифтом, названия клавиш отмечены полужирным шрифтом, а впервые встречающиеся термины набраны курсивом.

Посвящается Элис и Марти

БЛАГОДАРНОСТИ

Эта книга увидела свет только благодаря помощи многих талантливых людей.

Редакторы Диана Кинг и Гэри Мастерс успешно поддерживали весь этот проект. Гэри с самого начала увидел достоинства этой книги и оказал большую помощь в составлении плана изложения. Особая благодарность Джиму Комптону, литературному редактору, за его пристальное внимание ко всем деталям и способность сглаживать даже самую шероховатую прозу.

Я также признателен замечательным людям из C WARE Corporation за их прекрасные программные продукты, которые использовались мной при подготовке этого издания.

Я благодарен также техническому редактору Эрику Ингенито, наборщице Лизе Джефф, корректору Джанет Бун, составителю алфавитного указателя Теду Локсу и, безусловно, Веронике Эдди и Джеми Райту, создателям макета книги. Усилия дизайнера Алисы Фейнберг и художника Ингрид Оуэн претворили в жизнь идею оформления этого издания. Хочу также выразить признательность доктору Рудольфу Лангеру и другим людям из SYBEX, усилиям которых обязана своим появлением эта книга.

В заключение хочу выразить мою сердечную благодарность Барбаре Нейбауэр. Она читала мою книгу, глава за главой, делала к ней рисунки и организовывала мою жизнь. Пока создавалась эта книга, она овладела навыками программирования на Си и Си++. Вот это жена!

ВВЕДЕНИЕ

Забудьте все, что вы когда-либо слышали о том, как тяжело программировать на компьютере. Не обращайте внимания на байки о всяких трудностях. Не переживайте, если вы не имеете степени кандидата физико-математических наук.

Изучать программирование на компьютере и писать программы очень интересно. Если вы умеете логически мыслить, если вам нравится решать головоломки, или если вы хотите приказывать своему компьютеру, а не просто пользоваться тем, что для вас уже сделали другие, вы являетесь несомненным кандидатом на вступление в ряды программистов.

Сейчас вы держите в руках превосходную книгу, которая облегчит ваши первые шаги. Эта книга написана для начинающих, для самых начинающих. Фактически она предполагает, что вы ничего не знаете о программировании. Эта книга хороша также и для тех, кто хочет перейти к изучению Си и Си++ от других языков, таких как BASIC, Паскаль или языков макропрограммирования, используемых в пакетах типа WordPerfect, Lotus, или Excel.

Фактически единственное, что вам необходимо для того, чтобы изучить программирование, пользуясь этой книгой, это желание.

Книга «Моя первая программа на Си/Си++» основывается на версиях языка Си, известных как стандарты K&R и ANSI Си. Кроме того, в ней освещаются основные аспекты языка Си++. Все приемы и методы, которыми вы овладеете, прочитав эту книгу, вы сможете использовать при программировании как на языке Си, так и Си++. Если вы интересуетесь Си++, обращайте особое внимание на те разделы, советы и замечания, которые посвящены специально этому надмножеству языка Си.

Что вам даст эта книга

В этой книге вы будете изучать программирование постепенно, переходя от самых простых к более сложным задачам. Весь материал изложен максимально ясно и доступно. Текст книги сопровождается большим количеством иллюстраций и примеров. Вы можете не сомневаться, что, прочитав каждую главу, просмотрев рисунки и тексты программ, вы надежно усвоите полученные знания. Каждая глава кончается вопросами для закрепления пройденного материала и заданиями для самостоятельной работы.

Хотя эта книга рассчитана на начинающих программистов, все вопросы в ней освещены очень подробно. Ничто не было упущено. Здесь мы постарались не только описать собственно язык программирования, но уделить особое внимание логике построения программы и принципам решения проблем.

Главы 1 и 2 посвящены основным принципам создания программ и общей структуре программы, написанной на языке Си и Си++. В главе 3 вы узнаете, как общаться с программой посредством использования переменных и констант.

Затем, в главе 4, вы узнаете, как вывести информацию на экран монитора. Глава 5 посвящена вводу информации с клавиатуры. В главе 6 вы познакомитесь с тем, как выполнять математические операции, используя арифметические операторы.

В главе 7, посвященной функциям, вы научитесь структурировать программу, разделяя ее на небольшие, легко управляемые блоки. В главе 8 обсуждается процесс принятия программой решений, а в главе 9 показано, как повторять инструкции, используя циклы.

Овладев основными понятиями, в главе 10 вы познакомитесь с массивами и строками, а в главе 11 — с указателями и структурами. В главе 12 вы узнаете, как происходит чтение данных из дисковых файлов и запись в них. Кроме того, в той же главе описан вывод информации на печатающее устройство.

Для того чтобы свести вместе все приобретенные вами навыки, в главе 13 описан процесс создания завершенной прикладной программы, а именно, построения элементарной базы данных. Изучив эту главу вместе с полным текстом приложенной к ней программы, вы получите некоторое представление о том, как делаются профессиональные программные продукты.

В Приложении I содержатся образцы решений заданий, которые предлагаются в конце каждой главы, а в Приложении II — полный текст программы из главы 13.

Внимательно отнеситесь к примерам и заданиям для самостоятельной работы. Уже после изучения первых двух глав вы почувствуете себя в силах создавать собственные оригинальные программы, которые сможете использовать на своем рабочем месте, в школе или дома.

Программные средства

Если у вас уже есть собственный компилятор языка Си или Си++, вы можете использовать его для того, чтобы изучать приведенные в книге примеры. Помните, все предложенные здесь программы соответствуют стандарту языка ANSI Си и могут работать практически с любым компилятором Си и Си++.

ГЛАВА 1
ОСНОВЫ ПРОГРАММИРОВАНИЯ

Существует множество причин, которые могут побудить человека заняться программированием. Первая из них, конечно, деньги. Большая часть программ написана именно ради того, чтобы обеспечить своему создателю возможность заработать. Авторы могли предназначать свои программы для самой широкой аудитории, или, наоборот, для применения в специальных областях, но и в том, и в другом случае они стремились завоевать рынок. Возможно, вы не ставите перед собой грандиозной задачи добиться шумного коммерческого успеха, но все же не следует забывать, что программирование, из каких бы соображений им ни занимались, в любом случае является способом зарабатывать деньги.

Кто-то начинает писать программы просто для того, чтобы проверить свои возможности и приобрести новый интересный опыт с той же целью, с какой другие люди штурмуют горные вершины, участвуют в марафонских забегах, или ломают голову над кроссвордами. Программисты, относящиеся к этому типу, получают колоссальное удовольствие уже от самого процесса создания программы, ведь каждая программа— это головоломка, своего рода интеллектуальный поединок с компьютером, где вам еще надо доказать, что решения будете принимать вы.

Часто к изучению программирования подводит необходимость решить специальную проблему, при том, что приобрести коммерческий продукт, который позволил бы это сделать, затруднительно из-за его недоступно высокой цены или невозможности найти именно то, что надо. Уровень задачи не играет особой роли— может быть, она будет очень простой, вроде записи кулинарных рецептов, а может, и очень сложной, связанной с решением уникальных коммерческих или инженерных проблем.

Какими бы причинами ни было вызвано ваше личное решение посвятить время программированию, вы увидите, что это занятие наполнит вашу жизнь волнующими ощущениями. Самостоятельно найти удачное решение зачастую бывает не так-то просто, и тот, кто это сделает, может по праву гордиться собой. Удовлетворение, которое вы почувствуете, увидев программу работающей, не поддается описанию. Разочарование при виде неработающей программы может повергнуть вас в отчаяние. Но не сдавайтесь, вы бросили себе вызов, и дело того стоит.

Компьютерная программа

Вероятно, у вас уже есть определенный опыт работы с компьютером: текстовый процессор, электронные таблицы, базы данных, одна-две игры, но когда работаешь с готовым программным продуктом, всегда находишься в положении внешнего наблюдателя. В конце концов, пользователю важен только результат, получаемый при помощи программы, и его не слишком интересует, каким образом программа этот результат производит.

Тот, кто создает программу, смотрит на нее изнутри, видит как она работает, понимает, почему все происходит именно таким образом, а не иначе. Несомненно, такой подход является наиболее интересным, творческим и познавательным. Не имеет решающего значения, насколько большой опыт в работе с компьютером имеет человек, начинающий осваивать программирование. Несущественно и то, хорошо ли он разбирается в литературе, имеющей отношение к компьютерам, так как собственно создание программ будет для него принципиально новым видом деятельности.

Скажите, вам приходилось когда-нибудь объяснять, как найти дом с определенным адресом? А учить другого человека, как ему сделать что-то, чего он не умеет? Именно этим и занимается компьютерная программа. Она представляет собой не что иное, как ряд последовательных инструкций, указывающих компьютеру, *что точно он должен сделать, в какой именно последовательности* нужно действовать и, главное, инструкции эти должны быть *составлены на языке, понятном компьютеру*. Это и есть компьютерная программа.

Если компьютер* не сможет понять ваших указаний, он выдаст сообщение об ошибке, которое появится на экране монитора. Получив такое сообщение, вы должны будете попытаться видоизменить инструкции таким образом, чтобы компьютер смог их выполнить. Может случиться так, что, получив ошибочные команды, компьютер все-таки выполнит их. Эту ситуацию можно сравнить с положением человека, который записал адрес, расспросил, как найти интересующее его место, в точности последовал указаниям и в конце концов обнаружил, что попал совершенно не туда, куда хотел. То же самое произошло и с нашей программой. Такую ошибку особенно трудно исправить, потому что автор программы может даже и не подозревать о ее существовании.

А теперь давайте немного подробнее рассмотрим наше определение программы.

Как дать компьютеру инструкции о том, что точно он должен сделать? Ключевым здесь является слово *точно*: каждая инструкция должна быть точной, нельзя пропустить ни одного шага. Возьмем жизненную ситуацию. Когда прохожий спрашивает у вас дорогу к местному почтовому отделению, вы ему отвечаете: «Пройдите два квартала, а затем поверните налево по Франклин-стрит». Говоря так, вы не сомневаетесь, что он увидит здание почты и зайдет туда.

Замечания по Си++

Си++ менее «процедурный» язык программирования, чем Си. Это означает, что многие инструкции в Си++ в большей степени относятся к «объектам» и «событиям» нежели представляют собой описание ряда последовательных действий. Однако независимо от того, к чему они относятся, инструкции должны быть ясными и исчерпывающими.

Имея дело с компьютером, мы не можем делать таких предположений. Если бы мы объясняли местонахождение почтового отделения компьютеру, инструкции должны были бы выглядеть примерно так:

пройти два квартала на север;

повернуть налево на 90 градусов;

пройти 50 футов прямо;

повернуть налево на 90 градусов;

подняться на четыре ступеньки;

открыть дверь и войти.

Например, когда вы сохраняете созданный в текстовом редакторе документ, редактор должен точно представлять, какие процедуры ему следует выполнить.

*** Точнее, не сам компьютер, а программа-транслятор с языка, на котором составлен исходный текст, в данном случае— компилятор Си или Си++. *(Прим.ред.)***

Он должен знать, куда требуется поместить документ и каким образом сообщить, что, сохраняя новый документ под определенным именем, пользователь рискует уничтожить другой, уже существующий на диске.

Инструкции необходимо задавать в правильном порядке. Когда вы даете компьютеру инструкцию выполнить процедуру А, В, а затем С, то именно в таком порядке он их и выполнит. Компьютер не может подумать «Стоп-стоп! Мне кажется, что тут какая-то ошибка, может мне стоит все-таки сперва повернуть на Франклин-стрит?» Разумеется, компьютер можно заставить принимать решения. Составленная нами программа может решать: «Если зарплата работника больше 50 тысяч долларов, то берем 25% налога, а если меньше, то только 15%», но происходить это будет только в том случае, если мы заранее дадим компьютеру инструкции, когда принимать решение и какой именно вариант допускается выбрать в каждом конкретном случае.

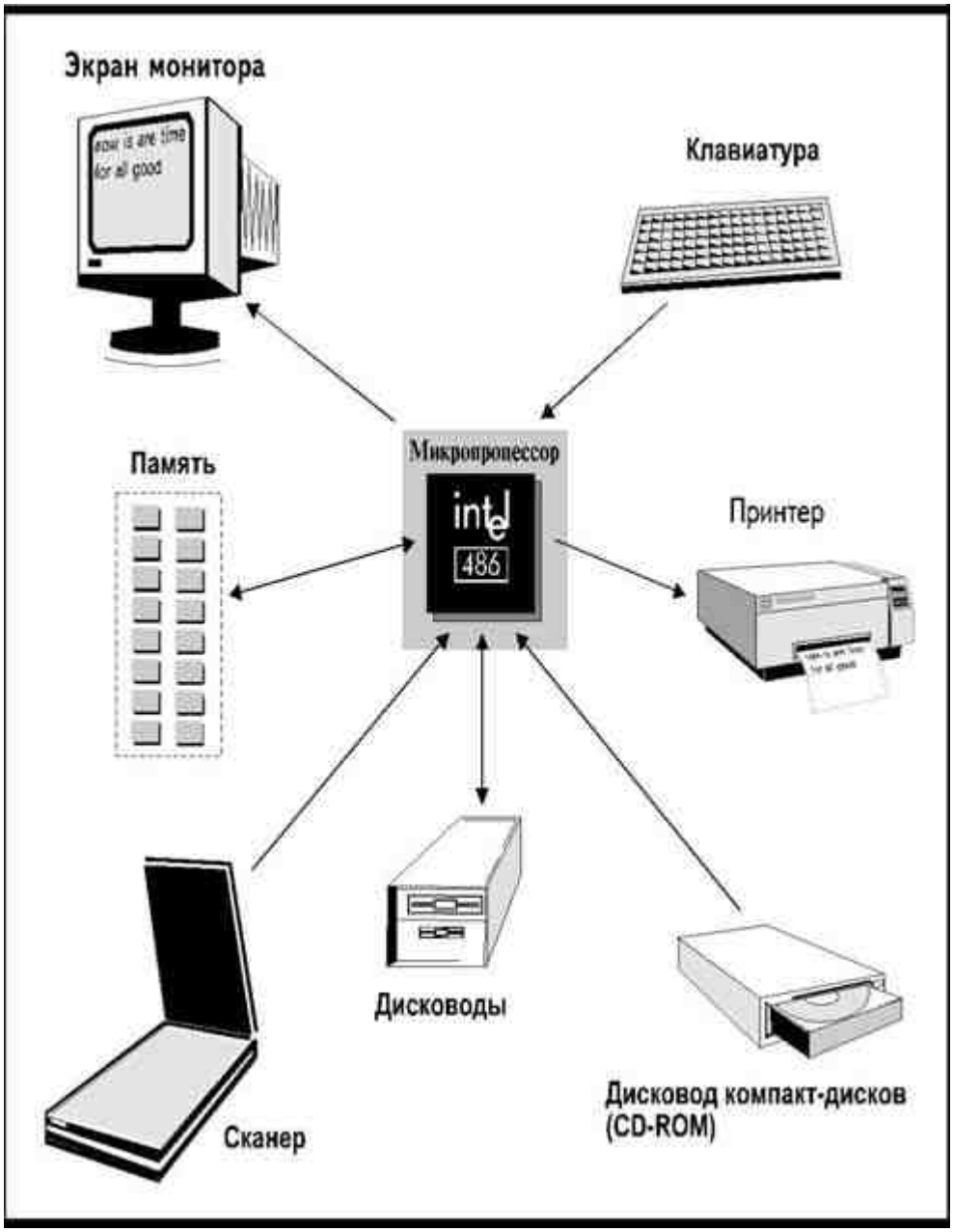

Рис. 1.1. Микропроцессор управляет всем, что происходит в компьютере

Чтобы убедиться, насколько это важно, давайте представим, что созданный нами новый документ мы пытаемся сохранить на диске с тем же именем, что и у другого, уже существующего документа. Будет не очень хорошо, если программа запишет новый файл, удалив уже существовавший, и только потом поинтересуется, хотели ли мы этого. Очевидно, прежде всего программа должна сообщить пользователю о возникшей ситуации, а затем действовать в соответствии с указаниями, полученными от него. Если пользователь ответит «OK», то новый документ будет записан под именем старого, а если откажется сделать это, то его попросят указать другое имя для нового документа.

Языки программирования

Третье требование к программе— это язык, который понятен компьютеру.

Глубоко в недрах компьютера находится микропроцессор. Микропроцессор— это интегральная микросхема, которая управляет всем, что происходит в компьютере (рис.1.1).

Когда программа командует компьютеру отобразить сообщение на экране или напечатать его на принтере, микропроцессор посылает соответствующие электрические сигналы, которые говорят компьютеру, в какой области памяти можно отыскать нужное сообщение и куда его следует отправить. По большей части работа микропроцессора остается недоступной для наблюдателя, он загружает программу и полагает, что микропроцессор знает свое дело. Однако чтобы понять, как работает программа, необходимо иметь хотя бы самое элементарное представление о том, как действует микропроцессор.

Физически микропроцессор выполняет всего четыре действия. Он может перемещать данные из одной области памяти в другую, изменять данные в конкретной области, проверять, содержит ли конкретная об-

ласть памяти определенные данные, и изменять последовательность выполнения инструкций. Все эти действия выполняются путем посылки, приема и отслеживания состояния электрических сигналов.

Электрические сигналы, с которыми имеет дело компьютер, могут иметь только два состояния (в зависимости от уровня напряжения): высокий уровень напряжения— электрический сигнал есть (состояние «включен») либо низкий уровень— электрический сигнал отсутствует (состояние «выключен»). Для того чтобы выполнить любую задачу, мы задаем микропроцессору последовательность сигналов в состоянии «включен» или «выключен». На рис.1.2 для примера приведена последовательность состояний электрических сигналов, необходимая для того, чтобы напечатать символ «А»*.

Для того чтобы задавать инструкции компьютеру на самом низком уровне, используется цифра 0, означающая состояние «выключен», и цифра 1,

*** Здесь и далее на рисунках приведены не конкретные последовательности машинных команд, а лишь дано общее представление о них. *(Прим.ред.)***

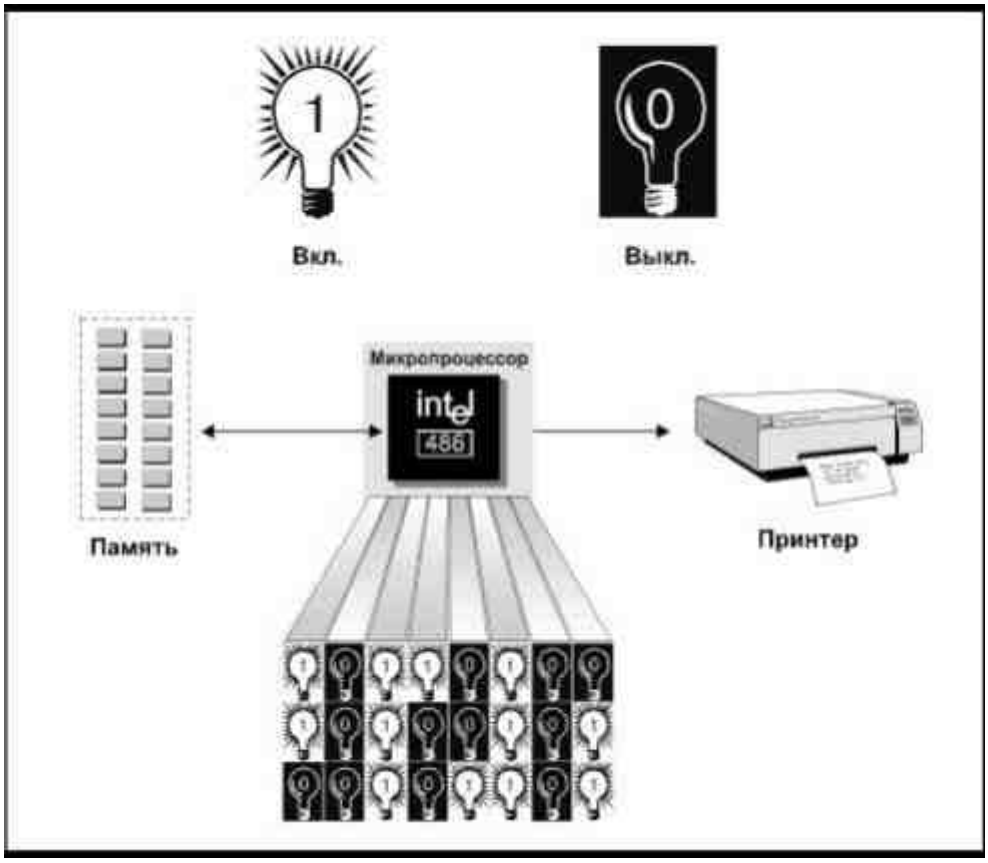

Рис. 1.2. Ряд последовательных сигналов «включен» или «выключен»говорит микропроцессору, что ему делать, в данном случае—взять символ из памяти и послать на принтер

означающая состояние «включен». Мы называем это *двоичными цифрами (битами)** или *двоичными кодами*, так как они основаны на двоичной системе счисления, в которой, как известно, все числа представлены только при помощи комбинаций нулей и единиц.

На заре компьютерных технологий программа выполнялась путем непосредственной манипуляции сигналами «включен», «выключен». Микропроцессоров тогда не существовало, и техник должен был вручную переводить ряды выключателей из одного состояния в другое, действуя, таким образом, как контроллер. Выполнение какой-нибудь задачи требовало правильной установки тысяч отдельных сигналов и, естественно, создание программы отнимало огромное количество времени и сил.

*** От английского *binary digit, bit. (Прим.перев.)***

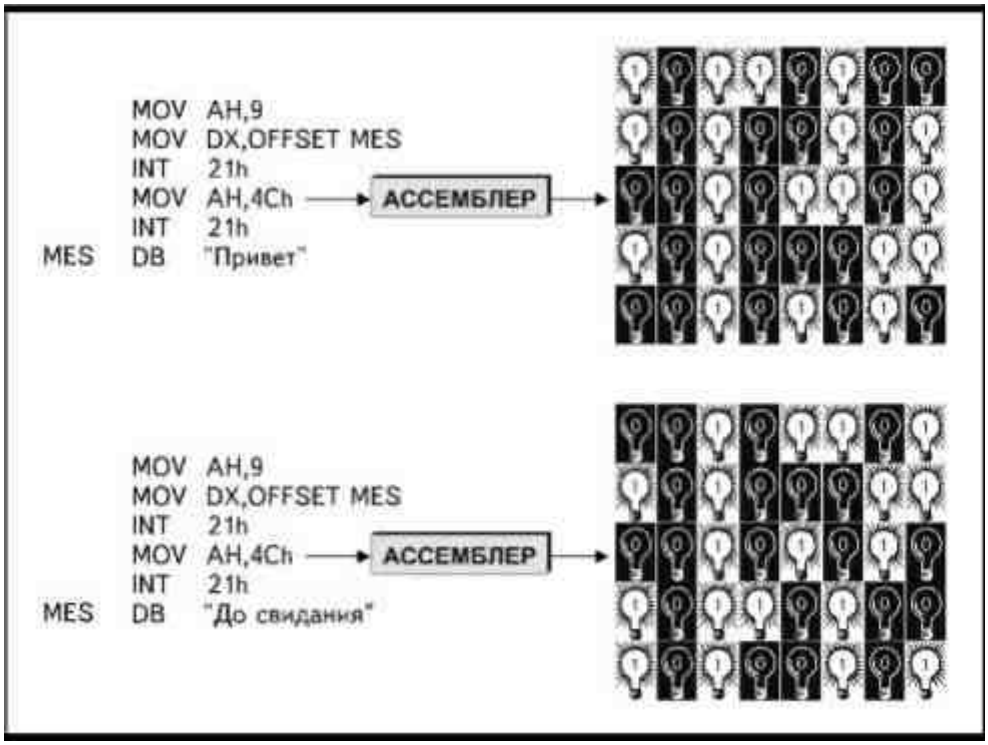

Рис. 1.3. Транслятор с языка ассемблера преобразует инструкции в двоичные коды

По мере развития электронной техники появилась возможность загружать программу в компьютер сразу, а затем заставлять его выполнять содержащиеся в программе инструкции. Такой способ программирования все еще требовал длительной процедуры задания последовательностей тысяч отдельных цифр 0 и 1, и так продолжалось до тех пор, пока не был разработан язык программирования *ассемблер*.

Ассемблер представляет задачу, адресованную непосредственно микропроцессору, используя мнемонические коды. Мнемонические коды— это несложные для запоминания слова или аббревиатуры, представляющие завершенное задание для микропроцессора. Например, код MOV указывает компьютеру на то, что некую информацию следует переместить из одной области памяти в другую, а код JMP указывает, что необходимо перейти в другую область памяти. Таким образом, вместо того, чтобы составлять последовательные ряды 0 и 1, программист на ассемблере может использовать *мнемонические коды*, подобные приведенным выше, каждый из которых представляет собой восемь или более бит.

Как показано на рис. 1.3, транслятор с ассемблера переводит эти коды в информацию о состояниях отдельных электрических сигналов, понятную компьютеру. Так как каждый мнемонический код соотносится непосредственно

На языке Си:

```
puts("SYBEX");
```

На языке ассемблера:

```
        MOV   AH,9
        MOV   DX,OFFSET NAME
        INT   21h
        MOV   AH,4Ch
        INT   21h
NAME    DB    "SYBEX"
```

В двоичных кодах:

```
10110100000010011011101000000010000101 1
11001101001000011011010001001100110011 01
00100001010100110101100101000010010001 01
01011000
```

Рис. 1.4. Сравнение языка Си, ассемблера и двоичных кодов

с внутренними функциями микропроцессора, программа, написанная на ассемблере, выполняется предельно быстро, но само программирование на этом языке отнимает достаточно много времени и требует написания исходных программ большого объема.

В настоящее время большинство программистов работают с языками высокого уровня, где инструкции задают с помощью человеческих слов, а не мнемонических кодов или 0 и 1. Каждое слово представляет практически завершенную операцию, а не одно задание для микропроцессора. Например, функция языка Си puts()* указывает компьютеру, что некая информация должна быть выведена на дисплей. Для выполнения этой же функции может потребоваться использование большого количества мнемонических кодов ассемблера и сотен бит.

Рис. 1.4 демонстрирует простую инструкцию, используемую языком Си и Си++ для вывода какого-нибудь слова на экран, и примерные эквиваленты той же инструкции, написанные на ассемблере и в двоичных кодах. Вы можете сами решить, какой язык программирования кажется более легким для чтения и создания текстов программ.

Разумеется, сам по себе компьютер не понимает, что означает функция puts() и другие инструкции языков высокого уровня, поэтому, прежде чем компьютер реально сможет выполнить задание, оно должно быть переведено на его собственный язык— язык двоичных кодов.

*** От английского *put string. (Прим.перев.)***

Операцию по переводу человеческих слов в двоичные коды можно выполнить двумя способами, которые называют *компиляцией* и *интерпретацией*.

Компиляторы

Компилятор переводит сразу весь текст программы и сохраняет результат на диске, так что программу можно запустить в любое время. Чтобы пояснить работу компилятора, обратимся к реальной жизненной ситуации.

Допустим, у вас есть доклад, написанный по-английски, который вы должны представить во французский парламент. Вы нанимаете переводчика, тот переводит доклад на французский язык, вы делаете нужное количество копий и затем распространяете их. Если переводчик обнаружит грамматическую ошибку, он прекратит работу и пошлет вам соответствующее сообщение. Вам придется исправить эту ошибку, прежде чем он продолжит работу над переводом. Но зато, если вы захотите использовать тот же доклад через год, все что потребуется сделать— это изготовить нужное количество экземпляров уже переведенного доклада и распространить их. Нет никакой необходимости делать перевод заново.

Рис. 1.5 иллюстрирует принцип действия компилятора. В сущности, компилятор— это компьютерная программа, которая (с помощью другой программы, называемой компоновщиком) преобразует все ваши инструкции в двоичные коды таким образом, чтобы программа могла быть выполнена компьютером.

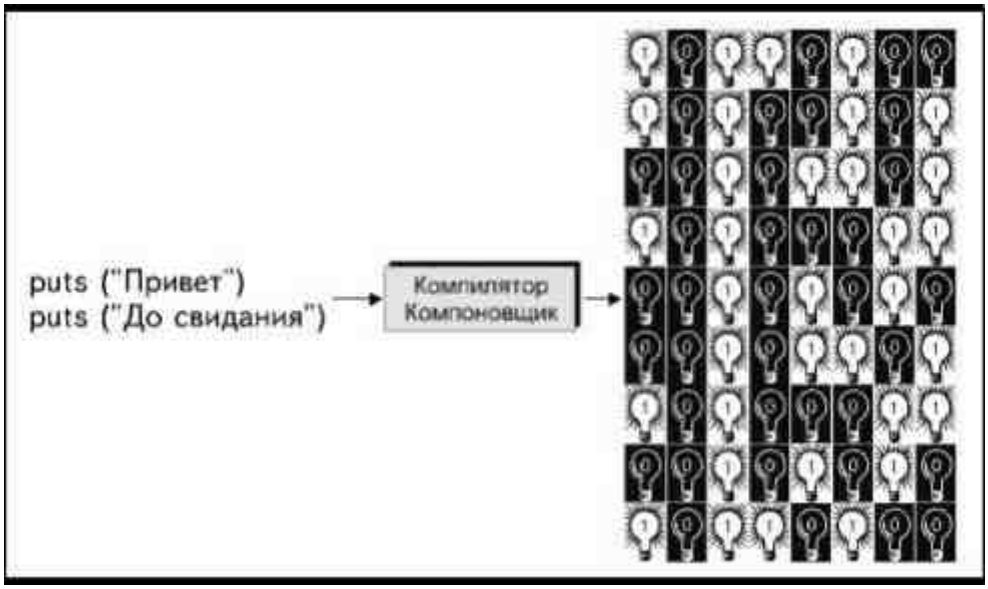

Рис. 1.5. Компилятор и компоновщик преобразуют инструкцииязыка высокого уровня в двоичные коды

Прежде всего компилятор убеждается, что программа написана в соответствии с правилами Си или Си++, затем создает промежуточную форму программы— объектный файл. Если во время работы компилятор встретит непонятную ему инструкцию, он сообщит об этом, и вам придется решить возникшую проблему

Замечания по Си++

Вскоре вы сможете убедиться, что Си++ является надмножеством языка Си. Это означает, что программу, которая компилируется транслятором Си, можно обрабатывать и компилятором Си++.

и повторить компиляцию. *Компоновщик* программ преобразует объектные коды в исполняемую программу (это не означает, что программа сразу же и выполняется).

При работе с компилятором программа существует как бы в трех состояниях. Сначала создается *исходный файл*, который содержит текст программы, написанный на Си. Его можно распечатывать и читать так же, как любой текстовый файл, созданный при помощи текстового процессора. Этот файл можно редактировать, тем самым изменяя программу. Программа, прошедшая

Расширения имен файлов

При работе большинство компиляторов языка Си требуют, чтобы исходный файл с текстом программы имел расширение .C, и присваивают объектному файлу расширение .OBJ, или иногда .O.

компиляцию, содержится в *объектном файле*, а окончательный результат представляет собой *исполняемый файл*, который можно запустить на выполнение.

Си, Си++, Паскаль, Кобол и Фортран— это примеры компилирующих языков.

Интерпретатор

Интерпретатор переводит компьютеру все инструкции непосредственно в момент их выполнения. Вернемся во французский парламент.

Вы написали доклад по-английски и наняли синхронного переводчика. Парламент приступает к слушанию доклада, переводчик переводит первую фразу и читает ее вслух. Затем он переводит вторую фразу, и так до конца, при этом на бумаге доклад существует по-прежнему только в виде английского текста, а французский вариант переводчик держит в уме. Если через год вы

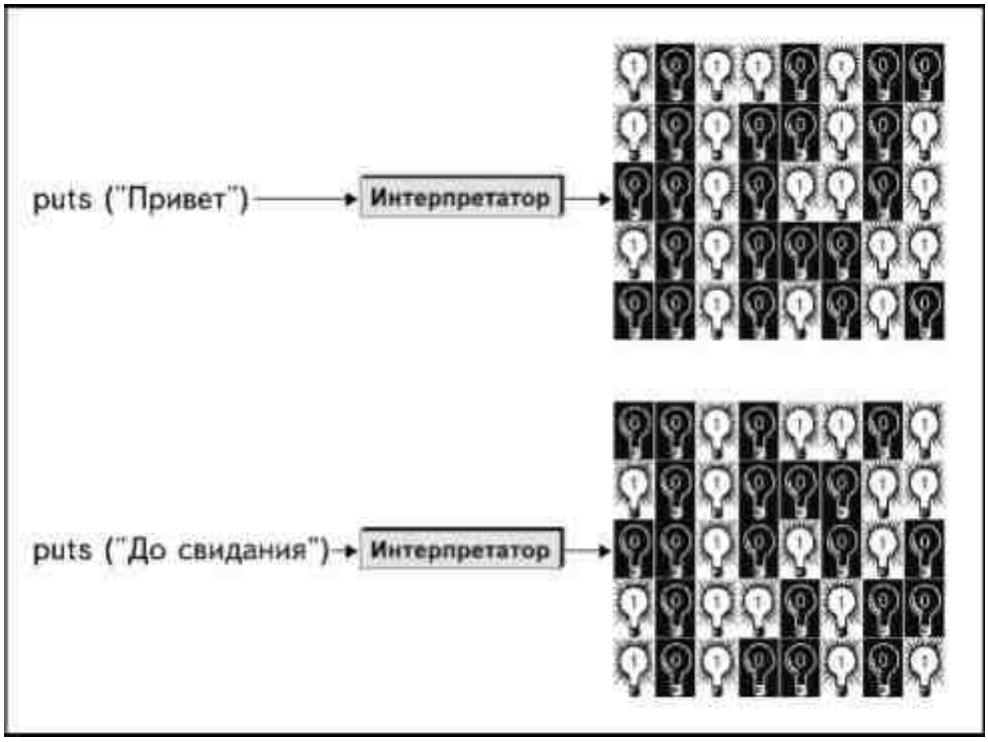

Рис. 1.6. Интерпретатор преобразует инструкции языка высокого уровняв двоичные коды во время каждого запуска программы

захотите, чтобы доклад прочли во франкоязычной Канаде, вам придется снова нанять переводчика и начать все заново.

Компьютерный интерпретатор работает сходным образом. Из рис.1.6 видно, что он переводит каждую инструкцию и сразу выполняет ее. Программа, обрабатываемая интерпретатором, существует только в виде исходного текстового файла. Язык BASIC, который поставляется с операционной системой MS-DOS, является примером интерпретирующего языка.

Почему имеет смысл использовать интерпретаторы? Например, когда вы еще только учитесь программировать, интерпретатор позволяет без труда писать и сразу же тестировать программу строка за строкой. Компилятору же, чтобы приступить к переводу, необходимо иметь полностью завершенный текст всей программы, или, как минимум, отдельной исполняемой ее части.

Почему же в таком случае пользуются компилятором? В силу того, что интерпретатор делает процесс создания программы очень легким, у пользователя появляется искушение пренебречь стадией предварительного планирования и проектирования, необходимой для того, чтобы создать работающую программу. Он уверенно приступает к делу, пытаясь писать программу с налету, а затем проводить долгие бесполезные часы, внося изменения методом проб и ошибок. Это не лучший способ работы, и поскольку вы только приступаете к изучению программирования, вам лучше учиться работать грамотно с самого начала.

Замечания по Си++

Запомните, все что говорится в этой книге относится как к Си, так и к Си++. Вместо того чтобы постоянно ссылаться на оба языка Си/Си++, мы часто будем упоминать только Си. Это не значит, что вы изучаете только Си, вы изучаете оба языка одновременно.

Кроме того, интерпретируемые языки работают медленнее. Необходимо загрузить интерпретатор в память компьютера, затем переводить и выполнять каждую отдельную строку программы. Компилятор преобразует весь текст программы сразу, а после этого откомпилированная запускаемая программа существует в виде двоичных кодов, адресованных непосредственно компьютеру.

Почему Си/Си++?

Си— компилирующий язык программирования. Это набор ключевых слов и функций, представленных привычными словами, которые для выполнения их компьютером должны быть переведены в двоичные коды. За последние годы Си стал наиболее популярным среди всех компьютерных языков, и обусловлено это тремя вескими причинами. Эти причины: скорость, переносимость и структурирование.

Скорость

Можно сказать, что Си является языком более близким к ассемблеру, чем другие языки высокого уровня, так как многие инструкции Си адресованы непосредственно аппаратной части компьютера. Поэтому программа на Си выполняется очень быстро. Фактически, она работает настолько быстро, что Си может быть использован для написания операционных систем, коммуникационных и инженерных приложений и даже компиляторов.

Кроме того, большинство компиляторов Си генерируют высоко оптимизированные коды. Вы помните, что компьютеру необходимы двоичные коды? Чем меньше этих кодов генерирует компилятор, тем более оптимизированным является код и тем быстрее работает программа. Многие компиляторы других языков генерируют менее оптимизированные коды, так что их программы работают медленнее.

Переносимость

Разумеется, можно создавать очень быстро работающие программы, если писать их на ассемблере. Однако мнемонические ассемблерные коды не одинаковы для каждого семейства микропроцессоров. Если бы вы написали на ассемблере программу для IBM PC или совместимого с ним компьютера, а затем решили выполнить те же самые процедуры на Apple Macintosh, вам пришлось бы переписать всю программу.

Си использует стандартные наборы ключевых слов. В общем случае вы пишете программу один раз, безотносительно того на какой платформе (с каким компьютером или операционной системой) собираетесь ее использовать. Тем не менее, хотя исходный файл сохраняется без изменений, необходимы два компилятора: один, чтобы перевести программу в двоичные коды, которые понимает IBM, другой, чтобы перевести программу в двоичные коды для Apple. Но сам текст программы вы создали раз и навсегда.

Вышесказанное также означает, что раз уж вы изучили Си, то вам не надо изучать другие языки, чтобы программировать на других компьютерах. Вы легко перенесете свои навыки с одной платформы на другую без переобучения. В конце концов, трудно предсказать заранее, куда вас заведет только что начатое изучение программирования, и лучше быть готовым ко всему.

Структурирование

Каким бы легким для изучения ни был язык Си, у него есть свои требования. Делая программу на интерпретирующем языке BASIC, можно сидеть у компьютера и писать текст прямо «из головы». В Си это не так-то просто. Язык Си имеет свою структуру и правила создания программы, которые заставляют программиста мыслить логически. Можно обойтись без серьезного структурирования и быстро написать «корявую» программу, сравнительно простую и небольшого размера, но чтобы создать действительно серьезную программу на Си, требуется прежде всего хорошо подумать. Необходимость структурирования, однако, далеко не является обузой. Благодаря этому качеству программу на Си очень легко проектировать, поддерживать и отлаживать.

Библиотеки функций

Язык Си как таковой содержит небольшое количество операций. В отличие от других языков, Си не имеет встроенных средств ввода и вывода информации или работы со строками (строка— это последовательность символов, образующих слово или фразу). Например, первоначальный стандарт языка Си имел только 27 *ключевых слов* (keywords) или команд*.

*** Использование автором термина «команда» в качестве синонима термина «ключевое слово» не является традиционным. *(Прим.перев.)***

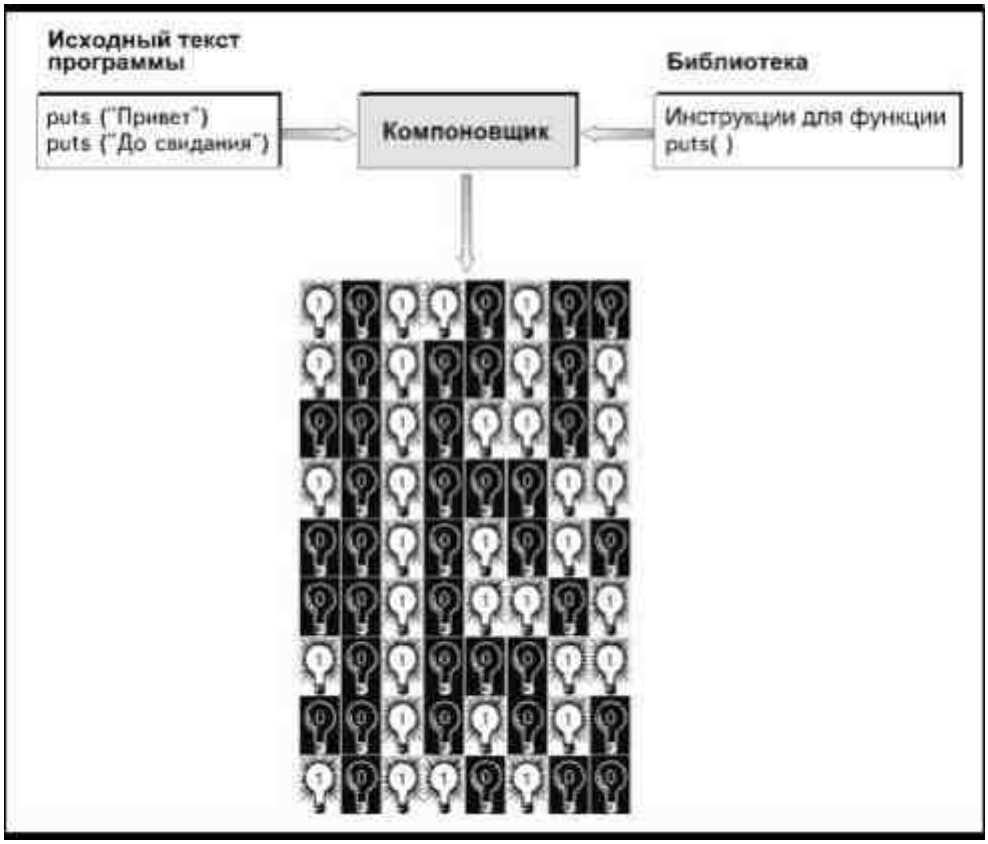

Рис. 1.7. Инструкции, обеспечивающие выполнение функцииберутся из библиотеки в процессе компоновки программы

Вся мощь языка Си обеспечивается библиотеками функций, которые поставляются вместе с компилятором. *Функцией* называют последовательный набор инструкций для решения специальной задачи. *Библиотека* — это отдельный файл, прилагающийся к компилятору и содержащий функции для решения распространенных задач.

Например, Си не имеет средств для отображения информации на экране. В то же время это настолько распространенная задача, что несколько функций вывода информации всегда поставляются в библиотеке Си и вместо того, чтобы писать функцию самостоятельно, можно использовать одну из них.

Ранее уже говорилось, что каждая функция имеет имя, например, функция puts(). Поэтому, вместо того чтобы писать все инструкции по выполнению данной процедуры, достаточно только ввести имя функции с необходимым синтаксисом (позже вы узнаете, как использовать имена функций).

Некоторые библиотечные файлы содержат предварительно откомпилированные коды. Когда компилятор сталкивается с именем функции из такой библиотеки,

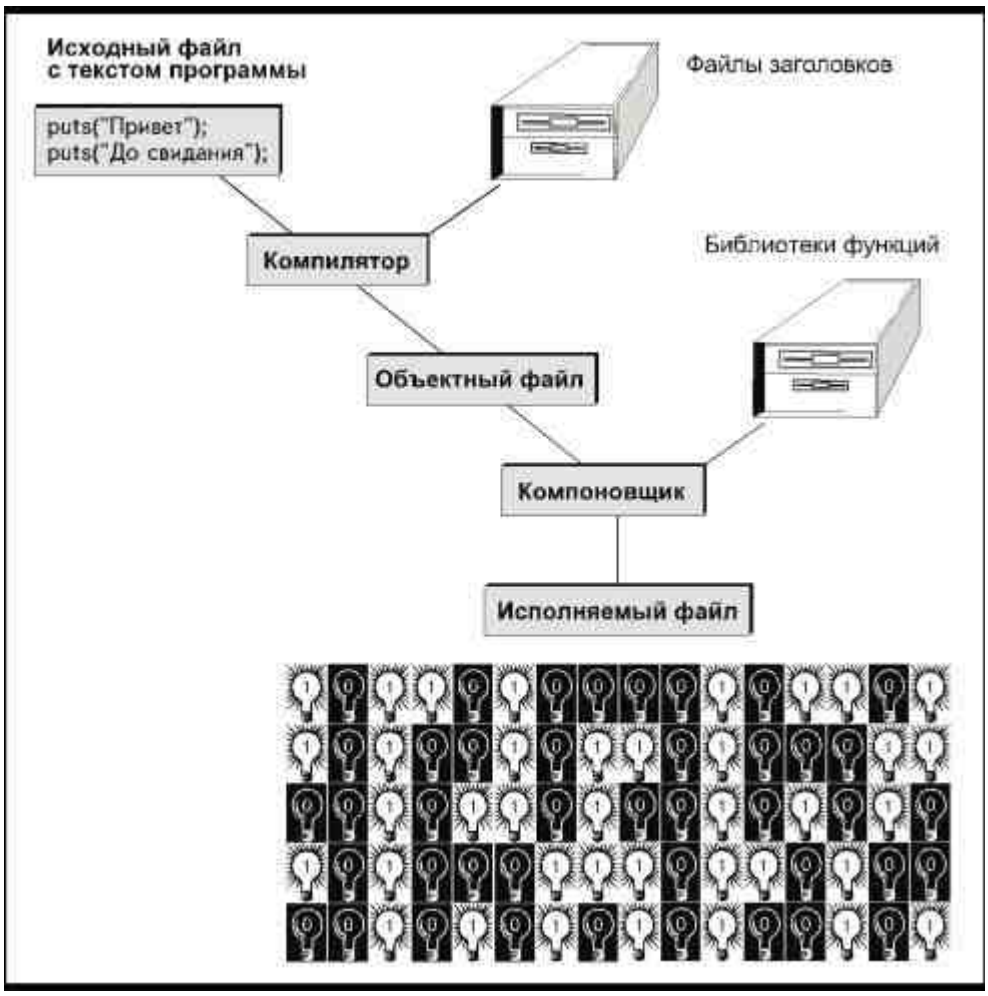

Рис. 1.8. Процесс компиляции/компоновки

ему не приходится заниматься преобразованием информации в двоичные коды, так как это преобразование уже выполнено. Во время процесса *компоновки* программы (проиллюстрированного на рис.1.7) двоичные коды функции (инструкции для выполнения функции, содержащиеся в библиотеке) объединяются с объектным файлом и создается исполняемая программа. Поскольку функции были откомпилированы заранее, они представляют собой очень эффективные коды. Производитель компилятора уже «очистил» функции, так что они полностью оптимизированы.

Существуют функции, которые используются так часто, что вместе со многими компиляторами поставляются их исходные тексты. Они содержатся в файлах, которые называются *файлами заголовков* (header file) и обычно имеют расширение .H. Файлы заголовков также содержат директивы компилятору и инструкции, указывающие использовать конкретные определения. Во время процесса компиляции содержимое файла заголовков добавляется к вашей собственной программе и для него также создаются объектные коды.

Рис. 1.8 кратко иллюстрирует процесс компиляции/компоновки.

Файлы заголовков, в отличие от библиотечных файлов, не откомпилированы. Так же как и ваш исходный файл с текстом программы на Си, их можно читать, печатать на принтере и редактировать. Однако вам следует остерегаться вносить изменения в файлы заголовков, поставляемые с компилятором. Если вы сделаете ошибку, то компилятор больше не сможет генерировать для них объектные коды.

В дополнение к библиотекам, поставляемым с компилятором, можно приобрести специализированные библиотеки, например, построения баз данных, графические, коммуникационные и многие другие. Чем больше функций имеется у вас в библиотеках, тем меньше работы придется делать вам лично.

Вы можете даже создать собственную библиотеку функций, которые часто используются в ваших программах. Например, в нескольких программах, которые вы пишете, нужно выполнить процедуры A, B, а затем C. Вы пишете функции, которые выполняют эти задачи, и помещаете в свою собственную библиотеку, а потом, всякий раз, когда возникнет необходимость использовать их снова, они уже готовы, то есть написаны и откомпилированы.

Именно использование библиотечных файлов делает Си легко переносимым. Компилятор для IBM-совместимых компьютеров поставляется с библиотечными файлами, содержащими двоичные коды для IBM. Компилятор для Mac поставляется с библиотечными файлами, содержащими двоичные коды для Mac. А вы просто пишете программу на Си и используете два компилятора, чтобы создать программы для обеих машин.

Необходимые пояснения

Язык Си был разработан Деннисом М. Ритчи в 1972 году и подробно описан в книге Ритчи и Брайана В. Кернигана «Язык программирования Си». Реализация Си в соответствии с правилами, изложенными в книге, рассматривается как K&R стандарт Си (по именам Кернигана и Ритчи). K&R-Си является, по-видимому, минимальной стандартной реализацией, так что любая программа, написанная с использованием правил K&R-Си, будет успешно транслироваться любым компилятором Си.

Во многих отношениях, однако, стандарт не был исчерпывающе определен, так что разработчики компиляторов стали усовершенствовать и развивать язык, каждый по-своему. Чтобы избежать путаницы, Американский Институт Национальных Стандартов* в 1983 году разработал новый стандарт, названный

*** American National Standards Institute. *(Прим.перев.)***

стандартом ANSI языка Си. ANSI-Си устанавливает правила развития и вводит стандарты для большинства средств языка Си.

Язык программирования, известный как Си++— это *надмножество* языка Си. Реально он не является новым языком, так как включает все операторы и средства языка Си, добавив только некоторые новые. Изучая Си, вы по большей части одновременно изучаете и язык Си++. Преимущество Си++ в том, что он позволяет с большей легкостью разрабатывать большие сложные программы за счет более модульного подхода и некоторых других усовершенствований. Кроме того, Си++ является языком объектно-ориентированного программирования.

Что такое объектно-ориентированное программирование

По правде говоря, почти невозможно быстро и доходчиво описать, что такое объектно-ориентированное программирование, если не имеешь дело с опытным программистом. Однако попытаемся.

Предположим, у нас есть картотека, содержащая информацию о членах некоего клуба: имя, адрес и номер телефона, статус в клубе.

Карточка члена клуба

Имя

Адрес

Телефон

Статус

Если у кого-то изменяется адрес, то мы должны просмотреть картотеку в поисках карточки с его именем, чтобы занести туда новую информацию. То же самое происходит и в случае изменения номера телефона или статуса члена клуба. Если бы мы написали инструкции для выполнения этих трех отдельных действий, они могли бы выглядеть примерно так:

Взять карточки

Найти карточку Смита

Изменить адрес на Западная Авеню, 12

Взять карточки

Найти карточку Доу

Изменить телефон на 555-1234

Взять карточки

Найти карточку Джонса

Изменить статус на «выбыл»

Обратите внимание, все эти три действия относятся к карточкам, карточки же, сами по себе, не имеют отношения к действиям, которые над ними производят. Следовательно, мы имеем дело с четырьмя объектами: карточками и функциями изменения адреса, номера телефона и статуса.

В объектно-ориентированном программировании мы используем наборы данных (карточки), которые комбинируем с производимыми над ними действиями. Отныне мы будем иметь дело с этой комбинацией как с единым объектом. Если изобразить этот объект (назовем его клубная карточка), он будет выглядеть так:

Клубная Карточка

Имя

Адрес

Телефон

Статус

Изменить адрес

Изменить телефон

Изменить статус

Так как объект включает и наборы данных, и функции, больше нет необходимости задавать компилятору каждый шаг по внесению изменений. Теперь компилятор поймет инструкцию, которая выглядит примерно так:

Клубная карточка: Изменить адрес (Смит, Западная Авеню, 12)

Клубная карточка: Изменить телефон (Доу, 555-1234)

Клубная карточка: Изменить статус (Джонс, выбыл)*

Не беспокойтесь, если такое объяснение показалось вам немного абстрактным. Вам не обязательно изучать объектно-ориентированное программирование, чтобы программировать на Си. Однако, изучив Си, вы будете способны с большей легкостью разобраться и в объектно-ориентированном программировании.

Что Си может и чего не может

Если вы заинтересованы в том, чтобы иметь возможность писать программы любых типов и размеров, вы правильно поступаете, выбирая Си. Фактически, нет никаких ограничений на программу, которую можно создать с помощью мощного компилятора Си. Изучение Си позволяет продвинуться в то же время и в изучении Си++.

Но то, что Си— мощный язык, не означает в то же время, что он является единственным или даже просто лучшим средством для решения любой задачи. Например, если вам надо создавать базы данных, то не стоит изучать Си. Существует большое число пакетов и кодовых генераторов, которые могут фактически написать для вас базу. Вы получаете быстро сделанную базу данных и затем добавляете туда созданные вами нужные функции и средства. Си, как таковой, не поможет вам сделать базу данных за один день. Это ограничение не относится именно и только к Си, а является общим для всех универсальных языков программирования. Ни один из них не предназначен для ускоренной разработки специальных прикладных программ.

*** В оригинале:**
Member_cards.change_address(Smiths, 12 West Avenue)
Member_cards.change_phone(Doe, 555-1234)
Member_cards.change_status(Jones, inactive)
что практически выглядит как инструкция Си++. ***(Прим.перев.)***

Этапы программирования

Разработка программы— это логический процесс. Если вы не пожалеете времени и проследите как осуществляется этот процесс с начала и до конца, то сможете успешно программировать на Си. Давайте рассмотрим последовательность этапов создания программы.

План программы

Сядьте перед компьютером и тщательно продумайте, что именно должна делать ваша программа. Опишите задачу как можно подробнее. Большинство программ подчиняются алгоритму, включающему Ввод, Обработку и Вывод.

Для примера, предположим, вы хотите написать программу расчета налога на продажи. Что она должна делать?

Что касается ввода, нам необходимы два параметра: объем продаж и ставка налога. Если вы хотите использовать программу неоднократно, то придется каждый раз вводить новую величину объема продаж. В то же время, вероятно, понадобится только одно значение для ставки налога на продажи— то, которое принято в вашем штате, так что это значение можно ввести прямо в текст программы.

Теперь обработка. В данном случае, для того чтобы рассчитать сумму налога, необходимо умножить сумму продаж на ставку налога.

Теперь рассмотрим вывод. Результат вычислений должен быть выведен на экран.

Итак, последовательные этапы программы таковы:

ВВОД Указать пользователю, что он должен ввести сумму продаж.

Показатель вводится с клавиатуры.
Указать компьютеру величину налога на продажи, взимаемого в вашем штате.

ОБРАБОТКА Умножить сумму продаж на ставку налога.

ВЫВОД Отобразить результат на экране монитора.

Текст программы

Для того чтобы написать текст программы, используется *редактор*. Редактор отличается от текстового процессора отсутствием возможностей для форматирования символов и параграфов. Фактически, исходный файл с текстом программы *не должен* содержать никаких специальных символов форматирования текста, так как компилятор не сможет их понять и выдаст сообщение об ошибке.

Компиляция программы

После того как вы сохранили исходный текстовый файл, надо создать промежуточный объектный файл с помощью компилятора. Если компилятор не может понять какие-либо инструкции, информация об этом появится на экране в виде *предупреждения* или *сообщения об ошибке*. Предупреждение информирует о наличии потенциальной проблемы, которая не препятствует продолжению компиляции. При возникновении ошибки процесс компиляции прекращается и, чтобы продолжить его, вам необходимо загрузить исходный текстовый файл в редактор и исправить ошибку. Наиболее часто встречаются синтаксические ошибки, то есть ошибки в написании, пунктуации или в употреблении ключевых слов и функций языка Си.

Не пугайтесь, получив сообщение об ошибке. Это не повод опускать руки, так как даже наиболее опытные программисты допускают ошибки.

Компоновка программы

Если никаких сообщений об ошибках не получено и процесс компиляции благополучно завершен, остается скомпоновать объектный файл с библиотеками и создать исполняемый файл. При компоновке сообщение об ошибке появляется в том случае, если компоновщик не может найти необходимую ему информацию в библиотеках. В этом случае необходимо проверить исходный текстовый файл, а также убедиться, что вы используете правильные библиотечные файлы.

Тестирование программы

Итак, теперь можно запускать программу. Если все было сделано правильно, программа будет выполняться без проблем. Но могут иметь место ошибки двух типов: ошибки выполнения и логические ошибки.

Ошибка выполнения возникает тогда, когда программа включает инструкции, которые невозможно выполнить. На экране появится соответствующее сообщение, а выполнение программы будет остановлено. Ошибки выполнения обычно случаются при обращении к файлам или к аппаратной части.

Например, программа включает команду открыть файл ACCOUNT.DAT, которого нет на вашем диске. Компилятор и компоновщик полагают, что этот файл будет существовать к моменту запуска программы, так что на этапе создания исполняемого модуля сообщение об ошибке не появится. Однако когда вы запустите программу, она не сможет найти этот файл и выполнить инструкцию.

Логическая ошибка имеет место в том случае, когда программа выполняет заведомо неправильную инструкцию, что, естественно, приводит к получению неправильных результатов. Такая ошибка представляет наибольшую проблему для идентификации, поскольку вы можете даже не знать о ее существовании. Вы должны скрупулезно проверять результаты работы программы.

Рассмотрим снова программу расчета величины налога на продажи. Положим, вы сделали ошибку и указали, что величину объема продаж надо не умножить на ставку налога, а разделить. У компилятора и компоновщика нет возможности узнать, что вы сглупили, так что процесс компиляции и компоновки, по-видимому, пройдет благополучно. Но, к несчастью, запустив программу, вы обнаружите, что величина 6% налога для 100 долларов составила 1666.66 доллара! Ксожалению, многие логические ошибки не столь очевидны, и отыскать их крайне сложно.

Если вы наткнулись на ошибку выполнения или логическую ошибку, необходимо исправить ее и заново провести компиляцию и компоновку.

Изучение основ программирования

Осваивая программирование, вы приобретаете два важных навыка.

Во-первых, вы изучаете синтаксис— слова, грамматику и пунктуацию языка программирования. Вы узнаёте значение каждой команды и каждой функции и учитесь правильно использовать их.

Во-вторых, вы знакомитесь с логикой программирования, то есть с тем, как выполнить какую-то задачу, используя язык программирования. Это универсальный навык, который может быть применен для любого компьютерного языка. Если вы прочувствуете логику программирования на одном языке, то для того, чтобы научиться работать с другим, останется только изучить его синтаксис.

Чтобы создавать компьютерные программы, необходимы оба эти навыка. К счастью, вы изучаете их оба одновременно.

Что нужно, чтобы писать программы

Чтобы написать программу на Си или Си++, необходимы редактор, компилятор и компоновщик.

Для создания исходного файла с текстом программы наряду со специальным редактором можно использовать привычный текстовый процессор, однако необходимо сохранять только неформатированный текст путем записи файла в формате ASCII или DOS TEXT. Большинство текстовых процессоров обладают соответствующими возможностями.

Однако при серьезной длительной работе лучше использовать специальные редакторы, предназначенные для программистов, так как они автоматически сохраняют текстовые файлы без символов формата, что позволяет уменьшить затраты времени. Кроме того, некоторые редакторы позволяют вводить конструкции языка Си путем набора типичных аббревиатур или нажатием специальных комбинаций клавиш.

Компилятор всегда снабжается компоновщиком программ и набором библиотек. Все наборы библиотек языка Си содержат основные функции Си, которые вы изучите, читая эту книгу, однако не все библиотеки одинаковы. Многие из них включают сложные функции для организации баз данных, а также графических, коммуникационных и других приложений. При выборе компилятора для работы поинтересуйтесь, содержит ли он необходимые вам функции*.

Кроме того, существуют дополнительные средства, позволяющие рационализировать процесс программирования. *Отладчик* (debugger) позволяет находить ошибки выполнения в исполняемом файле. Он показывает значения переменных и имена функций, которые выполняются по мере работы программы. Наблюдая за его действиями, вы можете определить, где имеет место ошибка. *Профайлер* (profiler) помогает оптимизировать программу по скорости выполнения отдельных выполняемых модулей. *Ассемблер* (assembler) позволяет добавлять функции, написанные непосредственно на языке ассемблера, если существует необходимость, чтобы программа выполнялась максимально быстро.

Многие компиляторы языка Си представлены в виде интегрированной среды (IDE). Интегрированная среда предоставляет возможность, запустив одну программу, получать доступ к редактору, компилятору, компоновщику и другим вспомогательным средствам путем выбора соответствующих пунктов меню. Для сравнения: если вы не используете интегрированную среду, то сперва надо запустить редактор, ввести текст программы, затем сохранить файл и выйти из редактора, после чего можно запускать компилятор.

Си/Си++ и ваше будущее

В настоящее время создается впечатление, что возможности программиста, владеющего языком Си, безграничны. Это наиболее популярный язык системного программирования и крупномасштабных разработок. Разумеется, Си не является единственным применяемым на сегодняшний день языком программирования, просто он пользуется наибольшей популярностью.

Новые продвинутые компиляторы языка Си разработаны для DOS, Windows и большинства других платформ. Обширные библиотеки и программные средства позволяют рационализировать системные разработки. По-видимому, язык Си будет повсеместно использоваться еще в течение длительного времени, так что его изучение является солидным капиталовложением.

Вопросы

1. В чем состоит различие между компилятором и интерпретатором?
2. Различаются ли между собой компиляторы языка Си?
3. В чем отличие языка ассемблера от языков высокого уровня?
4. Что такое исходный файл с текстом программы?
5. В чем различие ошибок компиляции и ошибок выполнения?
6. В чем преимущества языка Си? Объясните.

7. Какова последовательность этапов создания программы?

Упражнения

1. Составьте детальный план программы для расчета заработной платы и оплаты сверхурочных на основе количества отработанных в неделю часов.
2. Составьте детальный план программы, которая определяет, имеет ли право данная персона уйти на пенсию (пенсионный возраст— 65 лет).

**** Все приведенные в книге примеры могут быть собраны с помощью любого компилятора, поддерживающего стандарт ANSI Си. В том числе, с помощью широко распространенных в нашей стране компиляторов фирмы Borland, начиная с версии Borland C 1.0 и выше.** ***(Прим.перев.)***

ГЛАВА 2
ВВЕДЕНИЕ В СИ/СИ++

Если, имея перед глазами только исходный текстовый файл, вы попытаетесь определить, что именно делает программа, то в первый момент будете обескуражены. Хотя большинство ключевых слов языка Си представляют собой простые английские слова (например, for) и большинство функций также обычно представлены словами или аббревиатурами (например scanf, от английского scan format) при комбинировании ключевых слов и функций с синтаксисом (знаками пунктуации и пробелами), характерным для языка Си, получается программа, которая зачастую выглядит написанной совершенно не по-человечески. Не удивительно, что программисты называют этот текст «кодом». Пусть подобные трудности не пугают вас— как только вы получше познакомитесь с языком, то сможете читать программу в кодах Си так же легко, как приключенческий роман.

В этой главе вы познакомитесь со структурой языка Си и с некоторыми основными понятиями программирования.

Замечания по Си++

Си и Си++ имеют сходную структуру. Научившись писать программу на языке Си, вы сумеете писать и на Си++.

Структура программы

Программа, составленная на языке Си, может содержать одну или больше функций. Вы, наверное, еще помните, что функцией называется ряд последовательных инструкций, говорящих компьютеру, как выполнить определенную задачу. Многие функции, которые могут вам понадобиться, уже написаны, откомпилированы и помещены в библиотеки, так что вам достаточно просто указать компилятору использовать одну из стандартных функций. Необходимость написания собственной функции возникает только в том случае, если подходящей нет в библиотеках.

Все программы на Си (и Си++) должны начинаться с функции, называемой main(). Она выглядит так:

```
main()
```

Круглые скобки являются частью имени функции и ставить их надо обязательно, так как именно они указывают компилятору, что имеется в виду функция, а не просто английское слово *main*. В противном случае компиляция не будет завершена. Фактически каждая функция включает в свое имя круглые скобки, но в большинстве случаев в них содержится некая информация. В дальнейшем в тексте книги, ссылаясь на функцию, мы всегда будем ставить после ее имени круглые скобки.

Следом за main() вводятся инструкции. Инструкции могут быть представлены в виде стандартных команд и имен функций, содержащихся в библиотеках или написанных вами самостоятельно. Прямая, или открывающая фигурная скобка ({) помещается перед первой инструкцией, а обратная, или закрывающая фигурная скобка (}) следует за последней инструкцией*. Таким образом, простейшая структура программы, написанной на языке Си, такова:

```
main()      Функция, означающая начало программы— точку входа
{           Здесь начинается функция
.....;
.....;      Здесь помещаются инструкции,
                  которые должен выполнить компьютер
.....;
}           Здесь функция заканчивается
```

*** Последовательность инструкций, составляющих функцию, часто называют телом функции. *(Прим.перев.)***

Рис. 2.1. Структура программы на Си/Си++

Открывающая и закрывающая фигурные скобки называются ограничителями и служат для выделения части кода в единый блок. Когда вы пишете функцию, она всегда должна начинаться и заканчиваться фигурными скобками. Кроме того, отдельные блоки внутри функции также могут отмечаться при помощи своих собственных пар фигурных скобок.

При запуске программы компьютер начинает ее выполнение с первой инструкции функции main(). Ниже приведена завершенная программа на Си/Си++, которая выводит на экран монитора слово «OK»:

```
main()
        {
        puts("OK");
        }
```

Эта программа содержит всего одну инструкцию, которая, тем не менее, задана в строгом соответствии с правилами языка Си. Кавычки, отмечающие слово внутри круглых скобок не выводятся на экран. В языке Си они означают, что на экран следует вывести заключенную в них последовательность символов, а не константу или переменную, имеющую имя OK (о том, что такое константы и переменные вы узнаете в главе 3). Рис.2.1 иллюстрирует работу каждой части этой простой программы.

Точка с запятой в языке Си является разделителем и отмечает конец инструкции. Разделитель показывает компилятору, что данная инструкция завершена и дальше начинается следующая инструкция или заканчивается программа. Точку с запятой необходимо ставить после каждой отдельной инструкции:

```
main()
        {
        puts("У меня все в порядке");
        puts("А у тебя?");
        }
```

Вышесказанное не означает, что точку с запятой надо ставить в конце каждой строки. Иногда инструкция занимает больше одной строки и в этом случае точку с запятой надо ставить только один раз, отмечая конец команды.

В нашем примере после выполнения функции puts(), компьютер переводит курсор в начало следующей строки, так что при запуске этой программы, содержащей две инструкции puts(), мы увидим на экране две строки сообщений.

Си и Си++ являются языками свободного формата. Это означает, что для них не имеет значения, где будут помещены ограничители и начало строки. С таким же успехом можно написать программу следующим образом:

```
main(){puts("OK");}
```

и компилятор обработает ее так же, как и предыдущую. Но для того, чтобы сделать программу более читабельной, принято следовать определенным правилам:

- помещать функцию main() на отдельной строке;
- помещать фигурные скобки на отдельных строках;
- создавать в тексте программы отступы с помощью табуляции. Когда ваша программа станет достаточно длинной, вы увидите, что с помощью отступов можно сделать более понятной структуру программы и выделить логические единицы.

Некоторые компиляторы выполняют функцию puts() без автоматического перевода строки. Дополнительную информацию можно найти в главе 4.

Старайтесь следовать этим и другим принятым в языке Си правилам. Они кажутся не слишком важными, когда имеешь дело с программами небольшого размера, но зато существенно облегчают работу с большими и сложными программами.

В то время как наличие или отсутствие пробелов не оказывает влияния на код, создаваемый компилятором, правильная расстановка всех знаков пунктуации имеет принципиальное значение. Если вы пропустите скобку, кавычку или точку с запятой, компилятор немедленно остановит работу и сообщит об ошибке. Такие ошибки называются *синтаксическими*, и для того, чтобы компилятор мог создать исполняемый код программы, вам придется исправить их.

Прописные и строчные символы

Другие правила, принятые в Си, регулируют употребление прописных и строчных букв. Команды и имена функций всегда пишутся маленькими буквами, так что следует писать puts(), но не PUTS() или Puts(). Если эта функция определена вами, то вы не получите сообщения об ошибке, но исходный текст программы не будет похож на программу, написанную на языке Си. Заглавные буквы в языке Си обычно употребляются для задания имен *констант* и *макроопределений.*

Инструкция return

Что происходит после того, как компьютер заканчивает выполнение инструкций, заданных в вашей программе? Программа завершается, и компьютер возвращается в исходное состояние, то есть если программа выполнялась из операционной системы MS-DOS, на дисплее вновь появится ее подсказка, если же программа выполнялась из среды Windows, то вы вновь возвратитесь в ее оболочку.

Возврат в исходную среду, как правило, осуществляется автоматически. Исключение составляют отдельные компиляторы языка Си, которые требуют, чтобы вы явно указывали каждый шаг, включая возврат в систему. Для таких компиляторов вводится инструкция return(0), которую помещают непосредственно перед фигурной скобкой, завершающей тело функции main():

```
main()
        {
        puts("У меня все в порядке");
        puts("А у тебя?");
        return(0);
        }
```

Инструкция return(0) указывает компьютеру, что необходимо вернуться назад в исходную среду. При работе с большинством компиляторов включение подобной инструкции в текст программы не является обязательным, вы не получите сообщения об ошибке, даже если она отсутствует. Использование символа 0 также не является обязательным. Большинство компиляторов позволяет вместо записи return(0); использовать сокращенную запись return; без скобок. Однако если вы поставили скобки, то *должны* использовать полную запись, во избежание ошибки компилятора.

Если вы все-таки используете инструкцию return(0), не помещайте никаких инструкций между ней и фигурной скобкой, завершающей программу. Например, ошибкой было бы написать:

```
main()
```

```
{
puts("У меня все в порядке");
return(0);
puts("А у тебя?");
}
```

так как в этом случае компьютер вернется в операционную систему после выполнения первой функции puts(), вы так и не дождетесь появления строки "Ау тебя?".

Использование комментариев

После того как вы напишете и без ошибок откомпилируете свою первую программу, вы лучше освоитесь с принципами программирования. Вы поймете, какие операции выполняет каждая инструкция и в каком случае следует ее использовать.

Но по мере того, как программа усложняется, становится все труднее запоминать, почему вы употребили ту или иную инструкцию, особенно если вы решите снова посмотреть программу по прошествии некоторого времени с момента ее создания. Еще сложнее разобраться в программе, написанной другим человеком.

Добавление *комментариев* сделает любую программу более легкой для понимания. Комментарий— это сообщение для того, кто читает исходный текст программы. Комментарии могут помещаться в любом месте программы. Компилятор и компоновщик игнорируют комментарии, так что их содержимое не включается в объектный и исполняемый файлы.

Тому, кто только начинает программировать, введение комментариев кажется ненужной роскошью, ведь он и так тратит массу времени на создание программы. Действительно, небольшой, простой, легко тестируемой программе, состоящей из небольшого числа строк, комментарии не нужны. Однако, читая чужую программу, вы будете благодарны ее автору за любые пояснения, пусть даже программа не так уж и сложна.

Введение комментария начинается с символов /* и заканчивается символами*/. Выглядит это примерно так:

```
/*Эта программа выводит сообщение на экран*/
main()
{
puts("OK");
return(0);
}
```

Символы /* указывают начало строки комментария, а символы */ отмечают ее конец. Все, что помещено между ними, Си игнорирует. Как правило, в самом начале текста программы программисты помещают строку комментария, чтобы пояснить цель ее создания. Внутри текста программы помещаются комментарии, поясняющие употребление отдельных инструкций или последовательность логических шагов. Такие комментарии обычно помещают после разделителя, то есть точки с запятой:

```
/*Эта программа выводит сообщение на экран */
main()
{
/*На экран выводится сообщение "OK"*/
puts("OK");
return(0); /*Возврат в операционную систему*/
}
```

При записи инструкции и комментария в одной строке принято (исключительно для удобства чтения) разделять их некоторым количеством пробелов.

Комментарий может быть многословным и занимать не одну, а несколько строк. В этом случае символы */, указывающие конец комментария, можно поместить сразу после текста комментария или на отдельной строке:

```
/*Эта программа выводит сообщение на экран, она содержит
инструкцию return(0) для совместимости с компиляторами,
не имеющими автоматического возврата в систему
```

```
*/
```

Некоторые программисты добавляют звездочку в начало каждой дополнительной строки комментария:

```
/*Эта программа выводит сообщение на экран, она содержит

* инструкцию return(0) для совместимости с компиляторами,

* не имеющими автоматического возврата в систему

*/
```

Можно проявить фантазию:

```
/*
* * * * * * * * * * * * * * * * * * * * * * * * * * * *

* Эта программа выводит сообщение на экран, она содержит  *

* инструкцию return(0) для совместимости с компиляторами, *

* не имеющими автоматического возврата в систему         *

* * * * * * * * * * * * * * * * * * * * * * * * * * * *

*/
```

Дополнительные звездочки вводятся исключительно для красоты. Компилятор игнорирует их, как и все остальное, что содержится между символами, ограничивающими строку комментария.

Комментарии в Си++

Весьма распространенным недосмотром, который часто допускают начинающие программисты, является то, что они забывают ставить символы */ в конце комментария, и в результате получают сообщение об ошибке компилятора. В Си++ использование комментария несколько облегчается за счет введения новой пары символов //, указывающих начало строки комментария. В этом случае концом комментария считается конец строки, так что нет необходимости отмечать его специальным символом:

```
//Эта программа выводит сообщение на экран
main()
        {
        puts("OK");
        return(0); //Возврат в операционную систему
        }
```

Однако если комментарий занимает больше одной строки, каждая строка должна начинаться с символов //:

```
//Эта программа выводит сообщение на экран, она содержит
//инструкцию return(0) для совместимости с компиляторами,
//не имеющими автоматического возврата в систему
```

В Си++ можно по-прежнему использовать и символы /* и */.

Понятие параметров

Когда в программе используется функция, такая как puts(), принято говорить, что происходит *вызов функции*. Выражение «вызов функции» означает, что вы указываете Си выполнить некую функцию. Круглые скобки после имени функции могут оставаться пустыми или содержать параметры. *Параметр* — это элемент информации, необходимой для того, чтобы функция могла выполнить задачу*. Например, puts() является библиотечной функцией, она содержит инструкции, указывающие компьютеру вывести строку символов на экран монитора. Но какую именно строку он должен вывести? Мы должны определить это

путем помещения соответствующей информации в круглые скобки. Подобная процедура называется *передачей параметров*. Так, в инструкции

puts("Привет");

слово «Привет» является параметром, который мы передаем функции. Как показано на рис.2.2, мы говорим компилятору, что он должен выполнить функцию puts(), используя при этом слово «Привет». Кавычки являются

*** Список параметров, передаваемых функции при ее вызове, часто называют списком фактических параметров, а его элементы, соответственно, фактическими параметрами.** ***(Прим.перев.)***

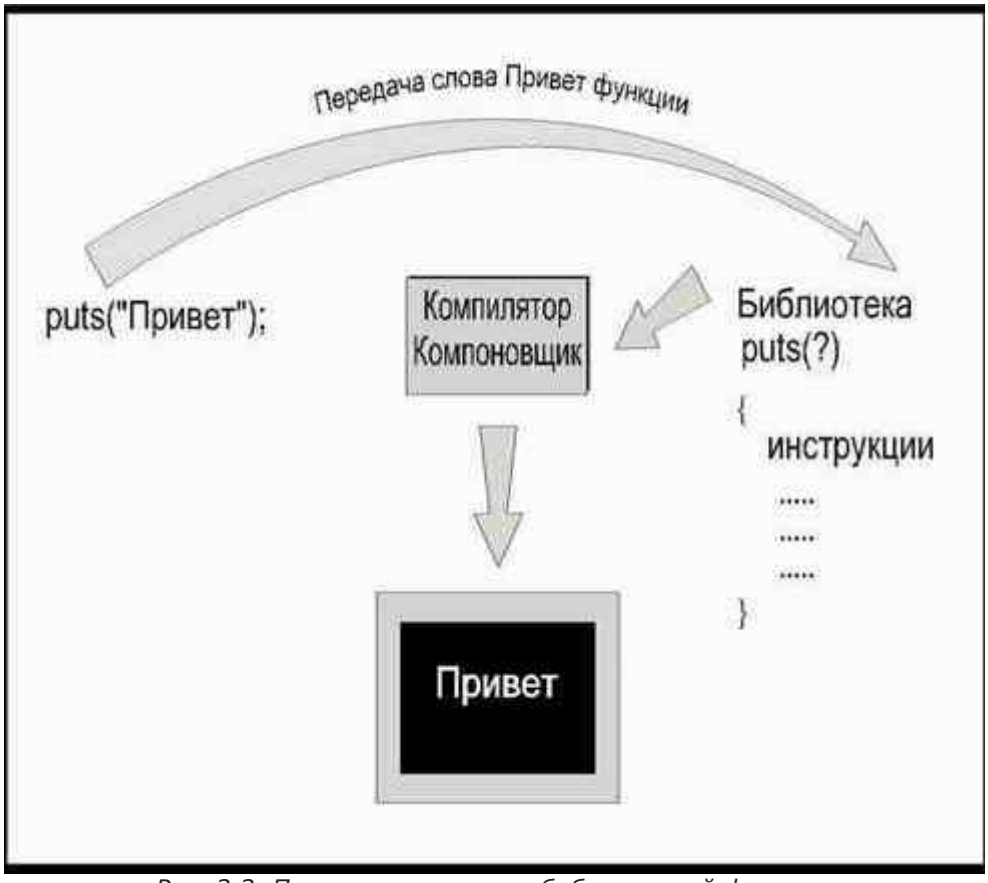

Рис. 2.2. Передача параметра библиотечной функции

индикатором, который указывает, что мы хотим отобразить на экране именно буквы П-р-и-в-е-т, а не какую-либо переменную или константу с таким именем (смотри главу3).

В этой книге в дальнейшем мы будем называть инструкции типа puts("Привет") функциями. В то же время вы должны помнить, что в библиотеке содержится только функция, называемая puts(), так что, строго говоря, puts("Привет")— это инструкция, которая вызывает функцию puts() и передает ей слово «Привет» в качестве параметра.

Слово «Привет» в нашем примере является одним параметром. Функция puts() может иметь только один параметр: символ, слово или фразу, которые мы хотим отобразить на экране. Позднее вы познакомитесь с функциями, которые могут иметь несколько параметров. Для многих функций передача параметров не является обязательной процедурой. По мере знакомства с функциями языка Си вы узнаете о параметрах больше.

Проверьте, правильно ли вы поняли разницу между функцией main() и другими функциями, такими как puts(). Мы используем имя main() для функции, содержащей наши инструкции компьютеру. Мы не вызываем функцию main(), но требуем выполнения содержащихся в ней инструкций, и одна из этих инструкций вызывает функцию puts(). Таким образом, puts()— это функция, вызываемая функцией main().

Директива #include

Если вы пишете программу, которая требует использования дисковых файлов или вывода информации на принтер, вам необходимо включить файл заголовков STDIO.H. Для этого вводятся следующие инструкции:

#include

main()

```
{
puts("OK");
return(0);
}
```

Директива #include указывает компилятору на то, что он должен использовать информацию, содержащуюся в файле заголовков STDIO.H. Сокращение stdio установлено для *стандартного ввода/вывода* (standard input/output). Файл STDIO.H содержит инструкции, необходимые компилятору для работы с дисковыми файлами или принтером.

Инструкции, включающие файлы заголовков, следует помещать перед функцией main().

Где содержатся файлы заголовков?

Символы, окружающие имя файла заголовков (< и >), указывают компилятору, что данный файл может находиться в каталоге INCLUDE. Так называется каталог, куда при инсталляции компилятора помещаются файлы заголовков. Если во время компиляции файл заголовков не будет найден в текущем каталоге, компилятор будет искать его в каталоге INCLUDE. Вы можете также заключить имя файла заголовка в кавычки

```
#include "stdio.h"
```

но тогда компилятор будет искать его только в текущем каталоге и если не обнаружит, выдаст сообщение об ошибке.Посмотрите документацию вашего компилятора, чтобы узнать, в каком случае следует включать в программу тот или иной файл заголовков.

Некоторые встроенные функции Си* для правильной работы нуждаются во включении файла STDIO.H. Для чего он нужен? Например, в языке Си существует функция getc(), которая вводит единичный символ из указанного вами источника, которым может быть и клавиатура, и дисковый файл. Так как существенная часть информации вводится с клавиатуры, Си включает функцию getchar(). Эта функция указывает компилятору «взять символ»* с клавиатуры. Функция getchar() вызывает ту же функцию getc() и сообщает ей, что источником информации является *стандартное устройство ввода*. Мы знаем, что стандартным устройством ввода является клавиатура, но откуда об этом узнает компилятор? Стандартное устройство ввода определено в файле STDIO.H, так что, используя в программе функцию getchar(), мы должны включить и файл STDIO.H с помощью директивы #include. Файл заголовков и библиотека вместе обеспечивают работу функции, так что, если ваш компилятор снабжен файлом STDIO.H, вам следует включать его в каждую программу во избежание ошибки компилятора.

*** Под термином «встроенные функции языка» автор понимает функции, содержащиеся в стандартных библиотеках. *(Прим.перев.)***

Проектирование программы

Изучение Си способствует развитию навыков решения всевозможных проблем. Эти навыки совершенно необходимы для того, чтобы грамотно использовать особенности и структуру языка программирования для выполнения специфических задач. Одним из существенных моментов является умение разделить проблему на составные части.

Документация, поставляемая с компилятором, должна содержать информацию о том, как и где используются файлы заголовков.

Разделение проблемы на части— обычный прием при ее решении. Действительно, разве маленькая задача решается не легче, чем большая? Когда вы сталкиваетесь с большой и трудноразрешимой проблемой, разделите ее на маленькие, удобные для осмысления подпроблемы. Если это необходимо, разбивайте ее на все более мелкие единицы до тех пор, пока решение каждой из них не станет для вас очевидным. Когда вы решите все маленькие подпроблемы, большая разрешится сама собой.

При проектировании программы поступайте аналогичным образом: начните с разделения всего объема работы, который вам необходимо выполнить, на меньшие задачи. Если решение небольшой задачи все еще выглядит слишком запутанным, разделите ее снова. Продолжайте это дробление до тех пор, пока не сможете написать ясные четкие инструкции. Как только вы напишете инструкции для каждой небольшой задачи и объедините их вместе функцией main(), программа будет готова (в главе7 вы познакомитесь с еще более эффективным способом структурирования программы).

*** В оригинале— get a character.** ***(Прим.перев.)***

Подобное деление программы на логические единицы поможет вам с большей легкостью находить ошибки. Вам только придется спросить себя, какая именно часть задачи выполняется неправильно, а затем посмотреть исходный текст соответствующей части программы. Процесс, описанный здесь, называется *диагностированием* и применяется при решении проблем профессионалами всех мастей. Вот, например, когда вы отдаете свой автомобиль в починку, механик задает вам целый ряд вопросов о замеченных неполадках, когда вы приходите к врачу, он спрашивает вас: «Где болит?»— и ваши ответы помогают и тому и другому определить, какая из систем вашего организма или вашего автомобиля является источником проблемы.

Вопросы

1. Какова общая структура программы, написанной на языке Си?
2. Для чего в языке Си служит точка с запятой?
3. Все ли компиляторы Си требуют использования в программе инструкции return?
4. С какой целью в текст программы вводятся комментарии?
5. Для чего при вызове функции используются параметры?
6. Все ли функции требуют передачи параметров при вызове?

Упражнения

1. Напишите программу, которая выводит на экран монитора следующее сообщение:

   Добро пожаловать в мой мир.
   Командовать парадом буду я.

2. Напишите программу, которая выводит в центре экрана ваше имя, адрес и номер телефона.
3. Объясните, почему данная программа написана неверно:

```
4.    main()
5.    (
6.    puts("Меня зовут Алвин");
      }
```

ГЛАВА 3
ПЕРЕМЕННЫЕ И КОНСТАНТЫ

Каждой программе для работы необходима информация. Программе, которая рассчитывает выплату по закладным, требуются сведения о процентной ставке, сумме и сроке выплаты займа. Программа, которая составляет каталог почтовых марок, должна иметь описание каждой марки. Программе, цель которой состоит в том, чтобы отразить нападение космических пиратов, нужны данные о количестве, координатах и скорости движения космических кораблей, заряде фотонной пушки, а также и другие жизненно важные сведения.

Информация, которую мы предоставляем компьютеру, называется *данными*. Данные вводятся в компьютер, он обрабатывает их, следуя вашим инструкциям, и затем выдает результат. Прежде чем вводить данные в компьютер, необходимо описать их тип.

Во-первых, Си должен зарезервировать достаточное количество памяти для хранения введенной информации. Разные типы данных занимают больший или меньший объем памяти. Во-вторых, не все функции языка Си могут работать с данными любого типа. Если вы введете слово, в то время как программа требует введения числа, вы получите ошибку компиляции или ошибку выполнения.

В процессе написания программы необходимо определить все данные, которые будут использоваться, причем сделать это надо и для вводимой информации, и для результата. Определять типы надо правильно с самого начала, так как после запуска программы вы уже не сможете ничего изменить.

Данные классифицируются по типу значений, которые они содержат. Обратите внимание, что *значение* не обязательно означает числовую величину, данные могут быть представлены не только в виде чисел, но и в виде букв, слов и целых фраз.

Замечания по Си++

Си++ имеет те же типы данных, что и Си. Однако некоторые компиляторы Си и Си++ имеют дополнительные типы данных, не определенные в исходном K&R стандарте языка.

Символьные данные

Значением символьных данных (char, от английского character) может быть буква, цифра или другой символ клавиатуры. Для каждого элемента символьных данных программа отводит столько места, сколько необходимо для хранения единичного символа*, так что, если вы используете пять различных элементов символьных данных, ваш компьютер зарезервирует пять элементов памяти (рис.3.1).

Набор употребляемых символов включает в себя латинские буквы, 26 прописных и 26 строчных:

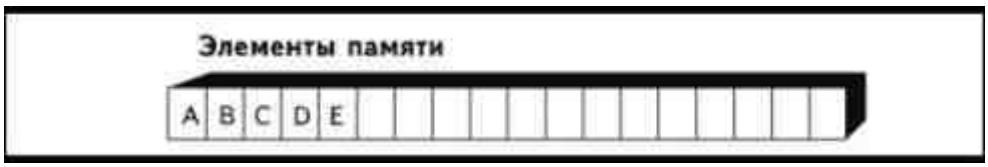

Рис. 3.1. Каждый элемент данных символьного типазанимает один элемент памяти

*** Переменная типа char имеет размер, естественный для хранения символа на данной машине, обычно 1 байт (8 бит). *(Прим.перев.)***

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i g k l m n o p q r s t u v w x y z

десять арабских цифр:

1 2 3 4 5 6 7 8 9 0

и специальные символы клавиатуры:*

! @ # $ % ^ & * ( ) _ + - = | \ } { " ' : ; ? / > . < , ~ `

Например, если вы пишете тест, в котором испытуемый должен выбрать один из четырех вариантов ответа— А, Б, В или Г,— каждая буква будет являться отдельным элементом символьных данных.

Как вы увидите дальше в этой главе, к символьному типу относятся и специальные управляющие коды, для хранения которых Си отводит столько же памяти, сколько и для единичного символа.

Заметьте, что символьные данные могут быть представлены в виде цифр— 1, 2, 3,— однако Си проводит различия между символом «1» и числом 1. *Как символ* единица не может использоваться в математических операциях, поскольку она не рассматривается в этом случае как математическая величина. *Как число* единица участвует в вычислениях, при этом, как вы скоро увидите, для хранения символа «1» Си отводит объем памяти вполовину меньший, чем для хранения числа 1.

Строки

Строкой называют набор символов, слов, фраз или предложений. В отличие от некоторых других языков, в Си строка не выделяется в отдельный тип данных. Язык Си работает со строкой как с последовательностью данных символьного типа, используя так называемый *массив*. Строка может состоять из любой комбинации букв, цифр, знаков препинания и управляющих кодов, которые тоже могут использоваться в качестве символьных данных. Язык Си проводит

Некоторые компиляторы Си и Си++ поддерживают специальный строковый тип данных и имеют библиотеки функций для работы со строками. Некоторые компиляторы могут иметь специальные функции для работы со строками, не выделяя их при этом в отдельный тип данных. Проверьте документацию вашего компилятора.

различия между строкой цифр и числом. Строка «123» будет восприниматься не как математическое значение сто двадцать три, а как комбинация символов «1», «2», «3».

*** Сюда же относятся и буквы русского алфавита: 33 прописных и 33 строчных. *(Прим.перев.)***

В этой книге мы будем оперировать понятием строки, начиная с настоящей главы, хотя массивы как отдельный тип данных подробно рассматриваются только в главе 10. Это обусловлено тем, что строки активно используются в самых разных программах, причем для работы с ними не обязательно иметь детальное представление об их, так сказать, «технической базе».

Целочисленные величины

Если вы хотите производить математические операции, то должны использовать числовой тип данных. Язык Си имеет несколько типов числовых данных в зависимости от значения, которое может быть им присвоено, и занимаемого объема памяти.

Целые числа (int, от английского integer)— это числа, не имеющие дробной части. Их значение может быть положительным, отрицательным или нулевым, но никогда не имеет в своем составе знаков после точки. В языке Си есть простая аксиома, которая гласит: «Используйте для подсчетов целые числа». Используйте целые числа всегда, когда есть возможность представить некое значение в виде целого числа, например, при подсчете количества повторов определенного события.

Как показано на рис. 3.2, каждый элемент целочисленных данных занимает в памяти столько же места, сколько два элемента символьных, независимо от величины самого числа (и число 2, и число 2000 требуют для хранения одинакового объема памяти). Но для того чтобы занимаемое место не превышало двух элементов памяти, величина целочисленных данных в языке Си ограничена. К целочисленным данным (собственно тип int) относятся величины,

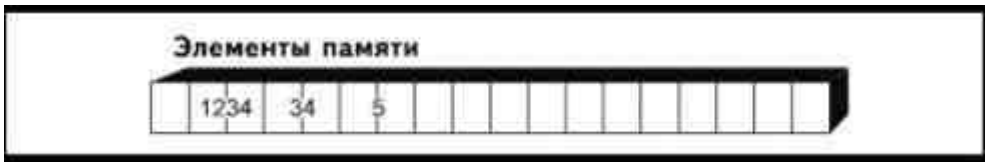

Рис. 3.2. Целочисленные данные занимают два элемента памяти

значение которых находится в промежутке между –32768 и +32767. Величины, значение которых выходит за эти пределы, требуют для хранения больше двух элементов памяти.

Для того чтобы обеспечить возможность работы с числами любой величины, в большинстве компиляторов Си определено несколько типов целочисленных данных. Границы для разных типов могут различаться в зависимости от конкретного компилятора.

short int — короткие целые числа: положительные величины от 0 до 255

int — целые числа: величины от –32768 до +32767

long int длинные целые числа: величины от –2147483648 до +2147483647

unsigned long длинные целые числа без знака: положительные величины от 0 до 4294967295

Вещественные числа

Числа, которые могут содержать десятичную часть (вещественные), называются *числами с плавающей точкой* (floating-point values). Для работы с ними в языке Си используется тип данных с плавающей точкой (float). Так как числа с плавающей точкой могут быть чрезвычайно маленькими или большими, для их записи часто используют экспоненциальную форму, например, значение числа с плавающей точкой может равняться 3.4e+38. Расшифровать это можно следующим образом: «передвинуть точку вправо на 38 пунктов, добавив соответствующее количество нулей». Существуют дополнительные типы данных для работы в очень широких пределах величин:

float величины от 3.4E–38 до 3.4E+38

double величины от 1.7E–308 до 1.7E+308

long double величины от 3.4E–4932 до 1.1E+4932

Тип данных с плавающей точкой имеет предел точности, диапазон которого зависит от компилятора. Например, число 6.12345678912345 в пределах *допустимого диапазона* для чисел типа float может быть записано компьютером только как 6.12345. Этот тип, как принято говорить, является типом с *одинарной точностью*, что означает, что точность его ограничена пятью или шестью знаками после точки. Тип double принято называть типом с *двойной точностью*, он имеет 15–16 знаков после точки.

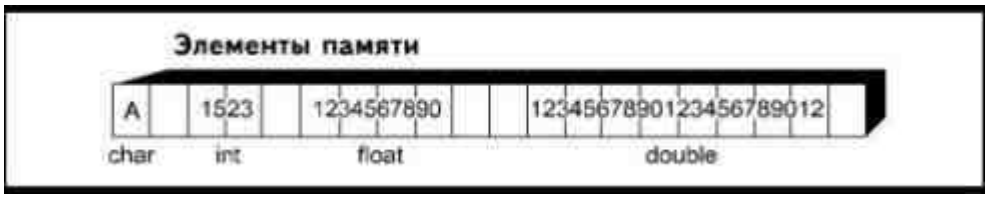

Рис. 3.3. Требования к объему памяти данных различных типов

Для записи данных с одинарной точностью резервируется четыре элемента памяти; двойная точность требует резервирования восьми, *повышенная* (long double)— десяти.

На рис. 3.3 мы просуммировали сведения о типах данных в языке Си и их требованиях к резервированию памяти.

Почему надо использовать целые числа?

В обычной жизни, не связанной с программированием, как правило, никто не обращает внимания на различия между целыми и вещественными числами. Производя подсчеты с помощью калькулятора, просто нажимают соответствующие клавиши и независимо от того, есть у числа десятичная часть или нет, вводят его одинаково— на расчеты это не влияет. Так почему же для языка Си так важно это различие? Почему существует так много разных типов числовых данных? В конце концов, число— оно и есть число!

Частично причина кроется в необходимости резервирования памяти для хранения информации. Если компьютеру не хватает памяти для выполнения вашей программы, он прекращает работу, а память стоит денег. Хорошие программисты стараются экономить память. Чем меньше ее требуется для выполнения программы, тем лучше. Использование типа int вместо float и типа char вместо строк помогает в этом.

Кроме того, операции с целыми числами выполняются быстрее, чем с вещественными. Если вашей программе требуется производить большое количество математических расчетов, разумно использовать целочисленные данные везде, где только это возможно. Конечно, разница в скорости выполнения программы не так заметна, если программа маленькая, а компьютер быстрый, но при работе с большими инженерными и графическими пакетами она становится весьма ощутимой.

Начиная писать программу, всегда выбирайте тот тип данных, который наилучшим образом отвечает вашей непосредственной задаче. Но возьмите себе за правило использовать целые числа везде, где это возможно, то есть всякий раз, когда добавление десятичной части не является совершенно необходимым.

Константы и переменные

Как только вы определили типы всех данных, необходимых программе, надо решить, как вы собираетесь вводить их в компьютер. Поскольку в языке Си существует множество функций для ввода, вы должны, прежде всего, классифицировать каждый элемент данных либо как константу, либо как переменную.

Константа остается неизменной во все время выполнения программы. Фактически вы задаете значение константы уже, когда пишете программу, а не тогда, когда начинается ее выполнение, и значение это нельзя изменить, не изменив исходный текст программы. Если значение некоего элемента данных известно заранее и известно также, что оно не будет меняться, используйте константу. Например, вы живете в штате, где установлен пятипроцентный налог на продажи. Когда вы рассчитываете сумму налога для нескольких облагаемых налогом товаров, ставка налога каждый раз будет составлять 5%. Соответственно, число 0.05, относящееся к типу данных с плавающей точкой, можно определить как константу.

С другой стороны, значение *переменной* вводится после запуска программы и может изменяться в процессе ее выполнения. Вернемся к примеру с налогом на продажи. Как мы уже установили выше, ставка налога не меняется, но сумма налога будет меняться для каждой новой покупки. Покупки совершаются на самые разные суммы, в зависимости от цены и количества купленного товара, и, следовательно, размер стоимости покупки надо определить как переменную.

В большинстве программ используются и константы и переменные. Когда вы пишете свою собственную программу, вы должны четко определить, в каком качестве будет использоваться та или иная часть информации— как константа или как переменная. И это только одно из решений, которые вам придется принимать при проектировании программы.

Имена констант и переменных

Каждой используемой в программе константе и переменной должно быть присвоено имя. Максимальная длина имени зависит от особенностей компилятора: некоторые ограничивают имя восемью символами, другие позволяют давать имена, состоящие из 32 и даже большего количества символов. В некоторых случаях имя переменной может быть гораздо длиннее, но не все составляющие его символы будут иметь значение для компилятора. Например, вы можете использовать имя, состоящее из 32 символов, но определяющими являются только первые восемь, и переменные, названные вами accountspayable и accountsreceivable, компилятор будет воспринимать как одну и ту же переменную.

Имена переменных и констант могут содержать латинские прописные или строчные буквы, а также символ подчеркивания (_). Можно использовать любые сочетания букв и цифр, но начинаться имя должно с буквы. Символ подчеркивания используют для разделения слов, чтобы сделать имя более понятным, например, city_tax вместо citytax.

Старайтесь выбирать имена переменных и констант так, чтобы они указывали на то, как будут использоваться данные. Имя city_tax содержит больше информации, чем ctax, а имя amt_due должно сказать вам больше, чем просто due. Избегайте употребления имен типа A или B кроме тех случаев, когда пишете очень простую программу.

В качестве имен констант и переменных нельзя использовать *ключевые слова* языка Си. При введении в качестве имени ключевого слова вы получите ошибку компиляции. Ниже приведен список всех ключевых слов языка Си и Си++, перечисляющий ключевые слова, определенные в исходном K&R стандарте языка Си, а также те, которые были добавлены в стандарте ANSI, и те, которые добавлены для Си++. Компилятор может иметь дополнительные команды, которые тоже будут относиться к ключевым словам, поэтому, прежде чем писать программу, изучите документацию компилятора. Если у вас произошла ошибка компиляции, а вы уверены, что все инструкции написаны правильно, проверьте, не употребили ли вы случайно ключевое слово в качестве имени константы или переменной.

Ключевые слова K&R стандарта:

	entry	
auto		return
	extern	
break		short
	float	
case		sizeof
	for	
char		static
	goto	
continue		struct
	if	
default		switch
	int	
dodouble		typedef
	long	
else		union
	register	

В стандарте ANSI Си добавлены следующие ключевые слова:

const

enum

signed

void

volatile

Рис. 3.4. В языке Си приняты правила определения имен констант и переменных

В языке Си++ добавлены ключевые слова:

catch
cin
class
cout
delete
friend
inline
new
operator
private
protected

В именах констант и переменных учитывается регистр символа. Если вы назвали переменную TAX, вы не можете ссылаться на нее как на переменную Tax или tax. Фактически, вы получите три различные переменные с именами TAX, Tax и tax, каждая из которых имеет свое значение и относится к своему типу, что, разумеется, сильно затруднит чтение и отладку программы. Многие начинающие программисты допускают неточности в употреблении заглавных букв в именах констант и переменных в разных частях программы, что приводит к ошибкам компиляции или выполнения, которые часто бывает очень трудно найти и исправить.

В языке Си принято правило (рис. 3.4) использовать маленькие буквы для имен переменных, а заглавные— для имен констант. Хотя нарушение этого соглашения не приведет к ошибке компиляции, тем не менее существует веская причина подчиняться ему. Соблюдение этого простого правила позволяет легко различать константы и переменные в тексте программы, что упрощает чтение программы и облегчает понимание ее логики.

В программе должны быть определены имена и типы всех используемых переменных и констант.

Рис. 3.5. Синтаксис определения константы

Определение констант

Определить константу— это значит сообщить компилятору Си ее имя и значение. Для определения константы перед функцией main() помещают директиву #define, имеющую следующий синтаксис:

#define NAME VALUE

После директивы точка с запятой не пишется, зато необходимо вставить, по меньшей мере, по одному пробелу между директивой, именем константы и присваиваемым значением. Дополнительные синтаксические символы вводятся в зависимости от типа определяемых данных: числовых, символьных или строк (рис.3.5).

Перед тем как начать генерировать объектные коды, компилятор подставляет на место каждого встреченного им имени константы ее значение (это можно сравнить с функцией автоматического поиска и подстановки, которая имеется в

Директиву #define можно использовать и для создания макроопределений (смотри главу 7).

Рис. 3.6. Определение константы

текстовых процессорах). По сути, само имя константы никогда не преобразуется в объектные коды.

Для определения числовой константы необходимо задать имя и значение константы. Например, директива

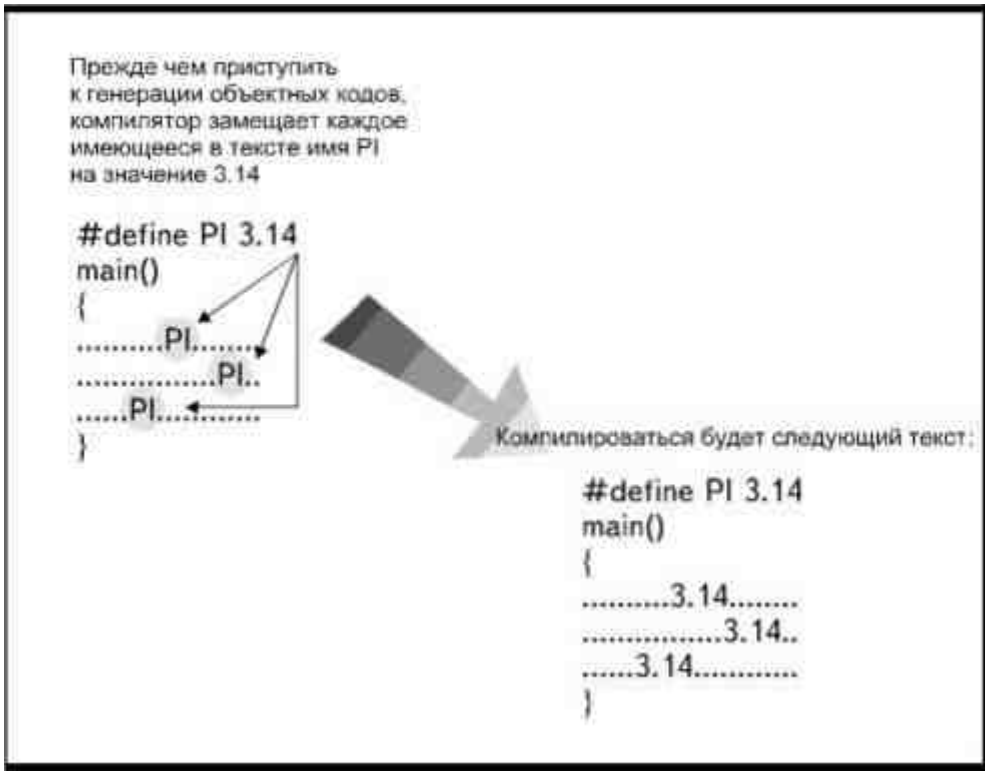

Рис. 3.7. Компилятор подставляет вместо имени константы ее значение

```
#define PI 3.14
```

создает константу, называемую PI, и присваивает ей значение 3.14 (рис.3.6).

Везде, где компилятор в исходном тексте программы (например, в формуле) встретит имя константы PI, он подставит вместо него значение 3.14 (рис.3.7).

При определении константы нет необходимости специально указывать тип данных. Компилятор присвоит его на основании значения, заданного директивой #define. В приводившемся выше примере, константа PI будет отнесена к типу чисел с плавающей точкой, так как ее значение, 3.14, является вещественным числом. В директиве

```
#define COUNT 5
```

компилятор отнесет константу COUNT к типу целых чисел, поскольку 5— это целое число.

Присваивая константе значение, относящееся к типу float, проследите, чтобы ваше число имело, по меньшей мере, по одному знаку слева и справа от точки. Если само значение не имеет десятичной части, добавьте после точки пару нулей:

```
#define RATE 5.00
```

Если этого не сделать, компилятор автоматически отнесет константу к типу int, вместо float. Если вы задаете значение константы меньше единицы, добавьте ноль перед точкой:

```
#define RATE 0.56
```

В противном случае, если будет написано #define RATE .56, неизбежна ошибка компиляции.

Значение символьного типа должно быть заключено в одинарные кавычки:

```
#define UNIT 'A'
```

Аналогично, определяя строку, заключите значение в двойные кавычки:

```
#define MY_FRIEND "George"
main()
        {
        puts(MY_FRIEND);
        }
```

Помните о том, что кавычки не входят в значение константы.

Когда имя константы является параметром, как с функцией puts() в нашем примере, оно не заключается в кавычки. Отсутствие кавычек указывает компилятору, что следует использовать значение, присвоенное

константе с таким именем, а не символы (буквы), из которых это имя состоит. То есть в нашем случае будет использовано значение, присвоенное константе MY_FRIEND, которое соответствует строке "George", а не буквы M-Y-_-F-R-I-E-N-D. Для того чтобы на экране появилось именно слово MY_FRIEND, инструкция должна быть такой:

```
puts("MY_FRIEND");
```

После того как мы ввели директиву #define MY_FRIEND" George", встретив в любом месте программы слово MY_FRIEND, компилятор подставит вместо него слово "George", так что инструкция

```
puts(MY_FRIEND);
```

на самом деле будет вызывать функцию puts() как

```
puts("George");
```

что приведет к появлению на экране имени George. Если ваша программа начинается со строки

```
#define MY_FRIEND "Марина"
```

на экране появится имя Марины. Почему? Потому что директива #define определяет константу с именем MY_FRIEND, имеющую значение "Марина".

Константы в Си++

Компиляторы Си++ и некоторые компиляторы Си имеют дополнительный способ определения констант. Можно определить константу, описать тип данных и присвоить значение, используя ключевое слово const. Такое определение должно помещаться внутри тела функции, в отличие от директивы, помещаемой всегда вне его:

```
main()
        {
        const int CHILDREN = 8;
        const char INIT = 'C';
        ....;
        ....;
        }
```

Использование ключевого слова const обусловлено тем, что оно позволяет программисту создавать константы, которые принадлежат одной определенной функции. Но чтобы обеспечить совместимость компиляторов Си и Си++, многие программисты продолжают использовать для определения констант директиву #define.

Приведенные инструкции определяют целочисленную константу со значением 8 и символьную константу, имеющую значение 'C'. Их можно определить и так:

```
#define CHILDREN 8
#define INIT 'C'
```

Почему используют константы?

Если заранее известно, что некий параметр не будет менять значение в процессе выполнения программы, зачем создавать лишние сложности и вводить константу? Почему нельзя вставить его прямо в текст соответствующей инструкции? Например, если программа начинается с директивы

```
#define PHONE "555-1234"
```

то, чтобы вывести номер телефона на дисплей, надо написать следующую инструкцию:

```
puts(PHONE);
```

Но так как известно, что номер не будет меняться в процессе выполнения программы, то можно было с тем же результатом написать просто

```
puts("555-1234");
```

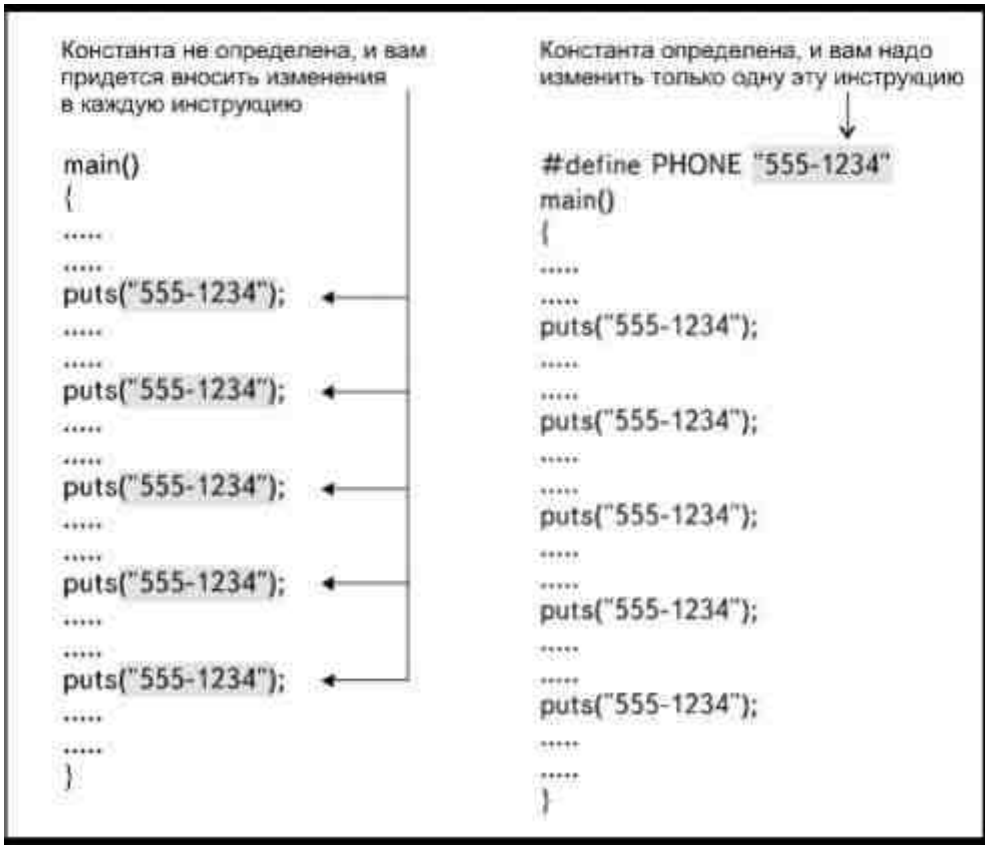

Рис. 3.8. Использование констант позволяет легко вносить изменения в программу

и не тратить время на введение в текст директивы #define и определение имени константы.

Однако использование констант позволяет легче вносить изменения в программу. Предположим для примера, что программа 20 раз использует ваш номер телефона. Может случиться так, что номер изменится, и тогда, если вы не использовали константу, вам придется редактировать 20 функций puts(). Если же вы по ошибке отредактируете только 19 функций puts(), это приведет к тому, что в одном случае будет использован неправильный номер. Введение константы позволит ограничиться изменением только одного места программы, а именно, потребуется вставить новый номер телефона в директиву #define. Как показано на рис.3.8, все функции puts() будут исправлены автоматически.

Все сказанное выше справедливо и для числовых констант. Вместо использования директивы

```
#define TAX 0.06
```

и выполнения вычислений с использованием константы TAX, можно было бы ввести значение 0.06 прямо в формулу. Но, допустим, размер налога в вашем штате изменится с 6 до 6.5 процентов, тогда, если вы не использовали константу, вам придется исправлять эту цифру везде, где она встречается, вместо того, чтобы ограничиться простым изменением директивы #define*.

Определение переменных

Определить переменную— это значит сообщить ее имя и тип компилятору Си, причем, в отличие от определения константы, задание переменной требует явного указания типа присваиваемых переменной значений. В общем виде синтаксис определения переменной выглядит так:

```
type name;
```

количество пробелов между типом и именем переменной может быть произвольным, но обязательно наличие, по меньшей мере, одного. Типичное определение переменной выглядит следующим образом:

```
main()
```

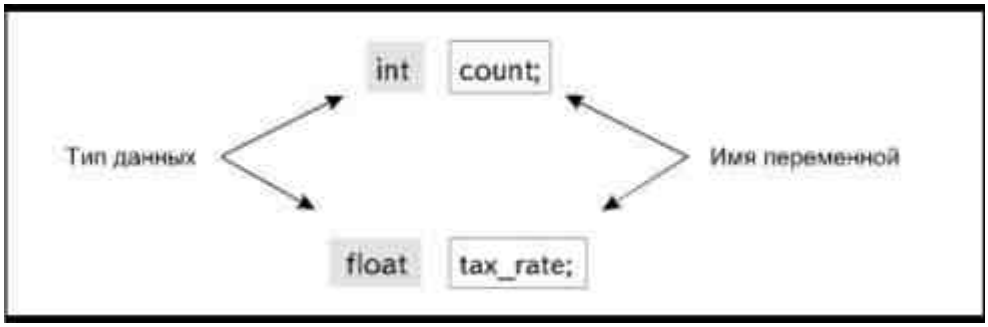

Рис. 3.9. Определение переменной

*** Помимо перечисленных преимуществ, константы дают возможность символьного представления трудных для запоминания числовых величин, а в некоторых случаях сокращают исходный текст (и, как следствие, ускоряют его ввод), если заменить часто встречаемую фразу, например стандартное сообщение, более коротким именем. *(Прим.перев.)***

Внутри функции main() создается целочисленная переменная int и переменная с плавающей точкой, которая называется tax_rate (рис.3.9).

Если вы хотите ввести несколько переменных одного типа, их можно определить в одной инструкции, разделяя имена переменных запятой и обозначив конец определения точкой с запятой:

```
main()
        {
        int count, children, year;
        float tax_rate, discount;
        }
```

В этих инструкциях мы определили пять переменных: три целочисленных и две с плавающей точкой. Определять переменные следует внутри функции main() сразу после открывающей фигурной скобки и перед другими инструкциями. Переменную можно также определить перед main():

```
int count;
main()
{
.....
}
```

В простой программе переменную можно определить и так и так, но когда программа состоит из нескольких множественных функций, правила языка Си точно устанавливают место определения переменных (более подробно этот вопрос обсуждается в главе 7).

Присваивание значения

Некоторые переменные могут иметь начальное значение, то есть значение, которое переменная принимает при запуске программы, но которое, в отличие от значения константы, будет изменяться в процессе выполнения. Начальное значение может быть присвоено либо при определении переменной, либо оформлено в виде отдельной инструкции.

Числовым или символьным переменным (определение строк мы обсудим позже) значение может присваиваться при их определении:

```
main()
        {
        int count = 5;
        char initial = 'A';
        float rate = 0.55;
        }
```

В этих инструкциях мы определили целочисленную переменную count и присвоили ей начальное значение 5. Далее мы определили символьную переменную initial, присвоив ей букву 'A' в качестве начального значения, и переменную типа float с именем rate и начальным значением 0.55. Значение символьной переменной должно быть заключено в одинарные кавычки, а число с плавающей точкой должно иметь хотя бы по одному знаку слева и справа от точки.

Если переменная определена, можно присваивать ей значение в отдельной инструкции, используя оператор присваивания (=):

count = 5;

initial = 'A';

rate = 0.55;

Присвоенное значение является начальным и может изменяться в процессе выполнения программы, иначе переменная не была бы переменной.

Если начальное значение переменной не известно заранее, то после начала выполнения программы оно может быть введено с клавиатуры, из дискового файла или получено в результате определенных вычислений. Со всеми этими способами вы познакомитесь чуть позже.

Определение строковой переменной

Рис. 3.10. Данные строкового типа в памяти

Последовательность \0 называется нулевым символом и рассматривается как один символ.

Как вы помните, язык Си не содержит строкового типа данных и не имеет никаких средств для работы со строками. К счастью, Си позволяет работать с данными строкового типа, используя массивы. Раньше мы уже определяли строку как ряд последовательных символов. Именно этим она и является— рядом символьных переменных, объединенных в нечто, называемое массивом. Элементы строки хранятся вместе в последовательных элементах памяти. В качестве примера на рис.3.10 показано, как выглядит переменная, имеющая значение "Привет". Каждая буква хранится в отдельном элементе памяти так же, как одиночная символьная переменная, но вся строка может быть выведена на экран монитора как единое целое при помощи одной функции puts(). Последовательность \0 является специальным символом, который Си вставляет после каждой строки. Он отмечает конец строки, указывая функции типа puts(), где следует прекратить вывод символов на экран.

Вы уже знаете как определить строковую константу— с помощью директивы #define, заключив символы строки в кавычки:

#define CLIENT "Кейт Тиммонс"

Чтобы определить строковую переменную, необходимо использовать тип char и указать максимальное число символов, которое может содержать строка. Например, так:

char var_name[N];

где var_name означает имя переменной, а N— максимальное количество символов, которое всегда заключается в квадратные скобки: прямую [и обратную].

Число, указанное в скобках, должно быть на единицу больше максимального количества символов, которое вы собираетесь использовать, так как Си нуждается в дополнительном пространстве, чтобы поставить нулевой символ. Например, переменная, содержащая аббревиатуры названий штатов, может быть определена следующим образом:

char state[3];

Рис. 3.11. Определение строковой переменной

Рис. 3.12. Нельзя присвоить строковой переменной значение,имеющее большее количество символов, чем было для нее зарезервировано

при этом, как показано на рис.3.11, создается строковая переменная, которая называется state и содержит два символа, плюс нулевой символ, означающий конец строки (\0).

Вы не можете записать в строку больше символов, чем было указано при ее определении, так как Си отводит ровно столько памяти, сколько необходимо для хранения заданного числа символов. Поэтому, прежде чем определять переменную, хорошенько подумайте, сколько символов может вам потребоваться.

Почему это так важно, можно пояснить на следующем жизненном примере. Допустим, вы вводите переменную, содержащую имена клиентов. Вы определили ее следующим образом:

```
char client[10];
```

Программа работает безупречно до тех пор, пока вы не получаете заказ от господина Флаглехоффена. Когда вы попытаетесь ввести его имя в список, ваша программа остановится из-за ошибки выполнения, что иллюстрирует рис.3.12.

Для того чтобы избежать подобной неприятности, вы могли бы в определении переменной несколько увеличить допустимое количество символов:

```
char client[80];
```

но если в программе будет несколько таких переменных, вы впустую потратите немало памяти компьютера.

Определив строковую переменную, вы тут же можете инициализировать ее, присвоив ей начальное значение в тексте программы. Помимо этого, значение можно присвоить, введя его с клавиатуры или из дискового файла, как это делается для переменных любого другого типа (подробнее о том, как вводить значение строковой переменной с клавиатуры, вы узнаете в следующей главе). В языке Си нельзя присваивать значение строковой переменной так, как это сделано в примере:

```
main()
        {
        char client[15];
        client = "Кейт";
        puts(client);
        }
```

где употребляется не разрешенная инструкция

```
client = "Кейт";
```

Если вы хотите присвоить начальное значение строковой переменной, это можно сделать, определив используемую переменную одним из двух допустимых

Рис. 3.13. Инициализация строки

способов. Вы можете присвоить значение определением переменной перед функцией main():

```
char client[] = "Флаглехоффен";
main()
        {
        puts(client);
        }
```

Заметьте, что в этом случае значение максимального числа символов строковой переменной в квадратных скобках не проставлено. Максимальное число символов строковой переменной при таком способе определения зависит от используемого компилятора, но практически определяется количеством символов в начальном значении плюс один (рис.3.13). Заметим, что в качестве параметра функции puts() используется только имя переменной без каких-либо квадратных скобок.

Другой способ присваивания начального значения внутри функции main() несколько более сложен:

```
main()
{
        static char greet[] = "Привет";
        puts(greet);
}
```

Определение выполняется внутри функции main(), но начинается со слова static, а переменная называется статической. Определяемая таким образом переменная может быть использована только внутри функции, в теле которой она была определена. Если вы желаете присвоить начальное значение строковой переменной, определяемой внутри функции, то ее следует определить как статическую переменную (подробнее смотри в главе 7).

Типы данных и функции

Тип, присвоенный константе или переменной, определяет то, каким образом функции могут их использовать. Большинство функций может использовать в качестве параметров только данные определенного типа. Например, функция puts() может работать только со строками, так что при компиляции программы

```
#define PI 3.14
```

```
main()
        {
        puts(PI);
        }
```

будет возникать ошибка. Функция puts() не может работать с вещественными числами.

Распространенной ошибкой начинающих программистов является использование двойных кавычек при попытке определить символьную константу:

```
#define INITIAL "A"
```

Си воспримет определение как попытку задать строковую константу, и вы получите ошибку компиляции, когда попробуете использовать эту константу при вызове функции, требующей в качестве параметра данные типа char. Даже один символ, заключенный в двойные кавычки, рассматривается Си как строка.

Когда мы будем рассматривать ввод и вывод в следующих главах, обращайте особое внимание на то, данные какого типа использует в качестве параметров каждая функция.

Литералы

*Литералом** называется любой элемент данных, который вводится непосредственно в инструкции языка Си. Литералом может являться любое число, символ или строка, которые вы вводите как начальное значение переменной. В примере:

```
count = 5;
```

число 5 является литералом. Это означает, что вы хотите, чтобы именно это число было присвоено переменной в качестве ее значения. В примере:

```
#define INIT 'C'
rate = 0.55;
client = "Кейт";
puts("555-1234");
```

буква С, число 0.55, слово Кейт и телефонный номер 555-1234 являются литералами.

Проектирование программы

Данные имеют огромное значение для программы на языке Си. Если перед тем как писать программу вы не решите как именно будете использовать данные, вероятность ошибок компиляции и выполнения вырастет во много раз.

В этой главе мы рассматривали константы и переменные в отношении данных, которые вы должны ввести в программу, но данные необходимо рассматривать и в качестве информации, которую вы хотите получить от программы. Для того чтобы это было возможно, надо определить также и переменные для хранения этой информации.

Фактически, многие программисты начинают писать план программы именно с вывода. Определив, какую именно информацию вы хотите получить от программы, легче определить, какую информацию необходимо ввести и как ее следует обрабатывать.

Давайте для примера рассмотрим программу, которая рассчитывает налог на продажи. Вот последовательность шагов определения необходимых констант и переменных.

1. Решить, какая информация необходима программе:
    - Вывод— объем всех продаж плюс налог на продажи.
    - Для того чтобы получить эту информацию, необходимо ввести: стоимость каждого наименования товара и ставку налога. Стоимость товара считается в долларах, следовательно, необходимо определить переменную типа float. Ее можно назвать sale. Ставка налога является для каждого штата фиксированной величиной, так что ее можно определить как константу. Назовем ее TAX_RATE.
    - Программа должна умножить стоимость товара на ставку налога. Например, если цена равна 25.00 доллара, а ставка налога равна 0.05, величина налога на продажи составит 1.25 доллара. Для хранения этого значения нужна переменная типа float. Назовем ее sales_tax.
    - Программа должна прибавить сумму налога к цене товара для того, чтобы получить стоимость с учетом налога. В нашем примере эта стоимость равна 26.25 доллара. Для хранения этого значения необходима переменная типа float. Назовем ее total.
2. Написать директивы #define для определения необходимых констант: #define TAX_RATE 0.05

3. Написать определения переменных: float sale, sales_tax, total;

Теперь, когда вы спланировали все необходимые данные, можно приступать к написанию программы.

*** В литературе чаще используется термин «константное выражение».** ***(Прим.перев.)***

Вопросы

1. Что такое тип данных char?
2. Чем символ '3' отличается от числа 3?
3. Какой тип данных вы используете для записи стоимости товара в долларах?
4. Почему может быть необходимо использование в программе типа данных long int?
5. Для чего используется тип double float?
6. В чем заключается различие между константой и переменной?
7. Как определить константу?
8. Поддерживает ли Си строковый тип данных?
9. Сохраняет ли переменная свое значение в ходе выполнения всей программы?
10. Как изменить значение константы?

Упражнения

1. Решите, какие типы данных вам необходимы, и напишите их определения для программы, которая рассчитывает недельную заработную плату сотрудника, получающего двойную оплату за сверхурочные часы (рабочая неделя— 40 часов).
2. Решите, какие типы данных вам необходимы и напишите их определения для программы, которая рассчитывает сумму и среднее арифметическое значение четырех чисел.
3. Объясните, какие ошибки имеются в следующих инструкциях:
4. char client[3]="Ajax";
5. main()
6. float tax_due;
7. char name(10);
8. int count(5);

   tax_due = "$1635.00";

ГЛАВА 4
ВЫВОД В СИ/СИ++

Выводом называется процедура переноса данных из памяти компьютера в другое место. Данные можно вывести на экран, отпечатать на принтере или сохранить на диске в виде файла. Кроме того, данные можно сохранить на магнитной ленте или послать по телефонной линии через модем или по факсу.

Вывод данных не означает, что они удаляются из памяти компьютера или что изменяется способ их хранения, компьютер просто копирует данные и посылает их куда-то еще.

В этой главе вы научитесь выводить данные на экран монитора.

Все функции вывода языка Си поддерживаются компиляторами Си++.

Функции, используемые для вывода данных, зависят от типа данных и способа их представления. Наиболее прост вывод строк и символьных данных.

Функция puts()

По предыдущим главам вы уже хорошо знакомы с функцией puts(), которая осуществляет вывод информации на экран. Параметр (информация, заключенная в круглые скобки, которая выводится на экран) должен относиться к одному из следующих типов данных:

- Строковый литерал:

  puts("Всем привет!");

- Строковая константа:
- #define MESSAGE "Всем привет"
- main()
- {
- puts(MESSAGE);

  }

- Строковая переменная:
- char greeting[]="Всем привет";
- main()
- {
- puts(geering);

  }

Помните, литерал (или константное выражение)— это конкретный набор символов, который вводится непосредственно в инструкции Си или Си++ вместо имени константы или переменной.

Использование любого другого типа констант, переменных или литералов приведет к ошибке компиляции. Строковый литерал, в отличие от имени константы или переменной, должен быть заключен в двойные кавычки.

Большинство компиляторов выполняют перевод строки после выполнения функции puts(). Это значит, что после того, как данные выведены на экран монитора, курсор автоматически переходит в начало следующей строки.

Однако некоторые компиляторы не выполняют перевод строки. При работе с такими компиляторами для перевода курсора вы должны вставить специальный управляющий код \n в конце строки, предназначенной для вывода на экран (подробнее об управляющих кодах смотри дальше в этой главе). Хотя это свойство компилятора выглядит как недостаток, на самом деле оно может оказаться довольно полезным. Если автоматический перевод строки отсутствует, можно использовать несколько инструкций puts() для вывода на экран одной строки, поставив код \n только в том месте, где вы хотите закончить ее и перейти на следующую.

В примерах, приведенных в этой книге, мы, как правило, вводили код \n в те инструкции, где используется функция puts(). В зависимости от вашего компилятора, вы, конечно, можете его опускать.

Функция putchar()

Функция putchar() предназначена для вывода единичного символа на экран. Параметром функции может являться:

- символьный литерал:

```
putchar('H');
```

- символьная константа:

```
#define INITIAL 'H'
main()
{
putchar(INITIAL);
}
```

- символьная переменная:

```
main()
{
char letter;
letter='G';
putchar(letter);
}
```

С помощью функции putchar() можно отображать только один символ. Инструкция

```
putchar('Hi');
```

приведет к ошибке компиляции.

При выводе на экран символьного литерала или управляющего кода их следует заключать в одинарные кавычки.

Большинство компиляторов Си не имеет автоматического перевода строки после функции putchar(), и курсор остается сразу за выведенным символом, не переходя к началу следующей строки. Для перехода на новую строку вы должны ввести управляющий код \n, который рассматривается дальше в этой главе.

Некоторые компиляторы имеют функцию putch(), которая аналогична функции putchar().

Двойственность символьных переменных

В некоторых системах при использовании функции putchar() вы должны включить в текст программы файл заголовков STDIO.H с помощью директивы #include. В этих системах функция putchar() является производной другой функции — putc(). Функция putc(), как вы увидите в главе 11, может направлять вывод на специальные устройства, такие как диск или принтер. Файл заголовков STDIO.H содержит информацию о том, как использовать функцию putc() для выполнения функции putchar().

Так как функция putc() может направлять вывод в дисковый файл, то ее использование (а соответственно, и применение функции putchar()) требует соблюдения определенных правил. Часть кодов, которые должны быть записаны в дисковый файл, не могут быть помещены в элемент памяти, отводимый для символа. Для осуществления возможности использования этих кодов функции putc() и putchar() были сконструированы таким образом, чтобы они могли работать и с целочисленным типом данных. Компилятор самостоятельно преобразует данные типа int в буквы, так что, имея дело с компиляторами, которые поддерживают стандарт K&R языка Си, можно писать программу следующим образом:

```
main()
```

```
{
int letter;
letter='G';
putchar(letter);
}
```

Даже при том, что переменная letter определена как целочисленная, в качестве начального значения ей может быть присвоен символ. Программа при этом будет компилироваться и выполняться без ошибок.

Некоторые программисты остаются верны стандарту K&R и всегда используют функцию putchar() с параметром типа int. Право выбора остается за вами.

Управляющие коды

Вы можете управлять перемещением курсора на экране и выполнять некоторые другие функции, используя специальные коды, называемые *escape-последовательностями* (escape sequences). Каждая последовательность начинается с символа обратной наклонной черты (\), который указывает на то, что символы, расположенные за ним, являются *escape-символами* (escape character). Обратная косая черта и escape-символы выражают операцию, которую вы хотите произвести. Когда компилятор встречает обратную косую черту, он не отображает следующие символы, а выполняет действие, на которое они указывают.

Код «новая строка»

Последовательность \n, называемая кодом «новая строка», перемещает курсор в начальную позицию следующей строки. Используйте код «новая строка» после вывода на экран символа с помощью функции putchar(). Если вы напишете следующие инструкции:

```
putchar('A');
putchar('\n');
```

то на экране отобразится символ A, а затем курсор перейдет в начало следующей строки. Отметим, что последовательность \n, как и другие символы, передаваемые функции putchar() в качестве параметра, заключена в одинарные кавычки. Так как буква n следует за обратной косой чертой, компилятор обеспечит перевод строки, а не станет отображать этот символ на экране.

При вызове функции puts() вы можете комбинировать escape-последовательность с символами внутри кавычек. Инструкция

```
puts("Уильям Уотсон\n");
```

отобразит имя Уильяма Уотсона на экране, а затем поместит курсор в начало следующей строки. Код «новая строка» должен находиться внутри кавычек.

Если ваш компилятор автоматически выполняет перевод строки после выполнения функции puts(), тогда курсор будет переведен на две строки вниз. Опишем, как это происходит:

1. Отображается имя Уильяма Уотсона.
2. Код \n обеспечивает выполнение escape-последовательности «новая строка», перемещая курсор в начальную позицию строки, расположенной под именем.
3. Выполнение функции puts() завершается. Если ваш компилятор добавляет перевод строки автоматически, то курсор будет перемещен вниз еще на одну строку.

До сих пор код \n использовался нами в конце сроки, но он может располагаться в любом месте внутри кавычек. Инструкция

```
puts("A\nB\nC");
```

отобразит на экране три строки текста:

```
A
B
C
```

Почему? Последовательность действий такова: компилятор отобразит символ A, переместит курсор в начало новой строки, отобразит символ B, переместит курсор в начало следующей строки и отобразит символ C. Если ваш компилятор добавляет перевод строки автоматически, то курсор будет переведен еще на одну строку ниже символа C.

Код «табуляция»

Код «табуляция» \t перемещает курсор в следующую позицию табуляции экрана. Чтобы посмотреть, как работает код «табуляция», воспользуемся следующей программой:

```
main()
{
puts("12345678901234567892123456789312345678941234 5\n");
puts("0\t1\t2\t3\t4\t5\n");
}
```

Она отображает строку цифр, чтобы вы могли рассчитать позиции табуляции, а затем отображает строку с цифрой 0 в крайней левой позиции экрана и цифры от 1 до 5, показывающие позиции табуляции экрана.

```
12345678901234567892123456789312345678941234 5
0           1           2           3           4           5
```

Код «табуляция» можно использовать для создания выровненных колонок текста или чисел, например, так:

```
main()
{
puts("Друзья, суммы долгов и сроки задолженностей:\n");
puts("Алан\t\tДжефф\t\tНэнси\t\tТом\n");
puts("$1.50\t\t$2.45\t\t$6.24\t\t$3.56\n");
puts("10 дней\t\t5 дней\t\t15 дней\t\t1 день\n");
}
```

Результат работы этой программы представлен на рис.4.1. Заметьте, что мы использовали по два кода «табуляция» между колонками, чтобы отчетливо разделить их на экране. Также обратите внимание на отсутствие пробелов между кодом «табуляция» и следующим за ним текстом, так как любые дополнительные пробелы отобразятся на экране, и колонки не будут выровнены правильно.

```
Друзья, суммы долгов и сроки задолженностей:
Алан        Джефф       Нэнси       Том
$1.50       $2.45       $6.24       $3.56
10 дней     5 дней      15 дней     1 день
```

Рис. 4.1. Результаты работы программы, использующей коды «табуляция»

Код «возврат каретки»

Код «возврат каретки» \r выполняет «возврат каретки», перемещая курсор к началу той же строки без перехода на следующую. Если вы вывели на экран какую-то информацию, перевели курсор назад к началу этой строки и затем вывели на экран другую информацию, то старое сообщение стирается, а на его месте появляется новое.

Рис. 4.2 иллюстрирует различия между escape-последовательностями «возврат каретки» и «новая строка». Когда вы работаете с клавиатурой, нажатие клавиши Enter производит действие, аналогичное тому, что в языке Си называется новой строкой. Некоторые программисты называют это комбинацией возврата каретки и перевода строки, сокращенно CR/LF (от английского carriage return/line feed). Использование кода «возврат каретки» в языке Си *не приводит* к переходу на новую строку.

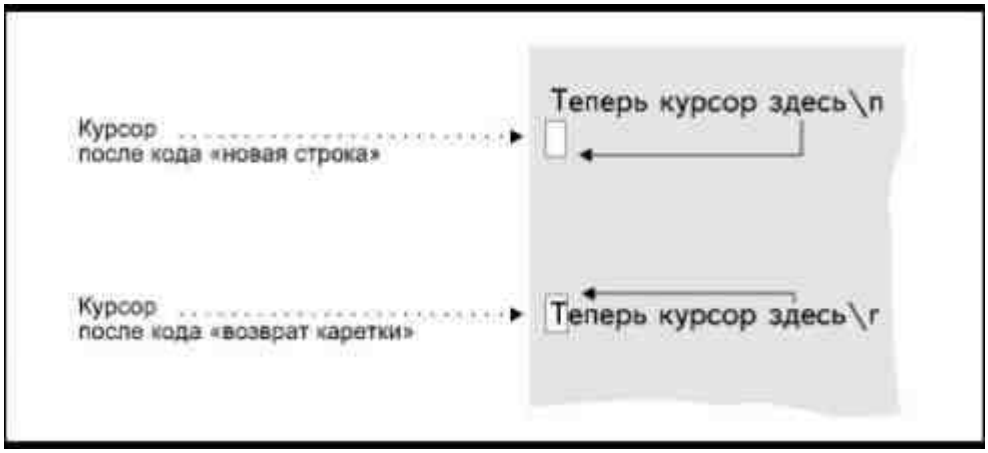

Рис. 4.2. Различия между кодами «возврат каретки» и «новая строка»

Действие кода \r можно видеть на примере следующей инструкции:

```
puts("Левый\rПравый");
```

В результате ее выполнения на экране появится только слово «Правый». И вот почему: после отображения на экране слова «Левый», код \r переводит курсор в начало этой строки, и при отображении слова «Правый» составляющие его символы замещают собой символы слова «Левый». Код \r сам по себе не уничтожает выведенные на экран символы, когда возвращает курсор, и только вывод новых символов после возврата курсора стирает уже существующий текст.

Код «возврат на шаг»

В отличие от кода \r, который возвращает курсор в начало строки, код «возврат на шаг» \b передвигает курсор только на одну позицию влево. При перемещении курсора существующие символы не уничтожаются, как и при возврате каретки.

Если вы используете коды «возврат каретки» или «возврат на шаг», а затем вводите код «новая строка», курсор переходит к следующей строке, не уничтожая существующий текст.

Код «перевод страницы»

Когда вы посылаете информацию на печатающее устройство (как именно это делается, вы узнаете позже), код «перевод страницы» \f вытягивает из него текущую страницу. Этот код распознается большинством принтеров.

После вывода сообщения на печать, большинство программистов использует код «перевод страницы», чтобы быть уверенными в том, что страница с последним сообщением готова и на нее не будет печататься другая информация. Если вывести изображение этого кода на экран при помощи функции puts() или putchar(), он появится в виде маленького графического символа, который никак не влияет на остальную информацию, отображаемую на дисплее.

Отображение специальных символов на экране монитора

Вы можете использовать escape-последовательности для вывода на дисплей специальных символов. Программисты часто используют их для вывода символов, изображение которых невозможно получить иным способом:

Escape-последовательность	Функция
\'	отображает одинарную кавычку
\"	отображает двойную кавычку
\\	отображает обратную косую черту

Например, вы желаете вывести на экран текст:

Мы зовем ее "Наташа"

который содержит кавычки с именем «Наташа». Если вы попробуете отобразить эту строку с помощью инструкции

puts("Мы зовем ее "Наташа"");

то получите от компилятора сообщение об ошибке. Вспомните, параметр, передаваемый функции puts(), должен начинаться и заканчиваться кавычками, чтобы компилятор Си знал, где начинается и где заканчивается строка. В приведенном выше ошибочном примере, компилятор будет интерпретировать параметр как «Мы зовем ее» с дополнительными символами «Наташа"»», которые расположены снаружи по отношению к паре двойных кавычек, но внутри пары скобок, содержащих параметр для вызова функции puts(). Для компилятора в такой строке оказывается слишком много информации.

Приведем теперь правильную запись этой инструкции:

puts("Мы зовем ее \"Наташа\"");

Рис. 4.3. Отображение символов «кавычки»

На рис. 4.3 показано, каким образом компилятор интерпретирует эту строку.

Кроме кавычек и обратной косой черты, вы можете использовать управляющие коды для отображения различных графических символов. IBM PC и совместимые компьютеры могут отображать набор символов, известных как расширенный набор ASCII-символов. Он включает в себя все буквы, цифры и знаки препинания, которые можно ввести с клавиатуры, а также некоторые графические символы и греческие буквы*. Каждый символ в наборе имеет собственный номер, например, номер 3 соответствует символу «сердечко» (по карточной терминологии— «черви»). Чтобы отобразить символ на экране, надо поместить соответствующий ему номер (в виде трех цифр) после знака обратной косой черты:

putchar('\003');

В результате выполнения этой инструкции на дисплей выводится символ «сердечко». Для того чтобы отобразить символы всех карточных мастей, используйте следующую программу:

*** В русифицированных наборах— буквы русского алфавита. *(Прим.перев.)***

```
main()
{
puts("Черви \003\n");
puts("Бубны \004\n");
puts("Трефы \005\n");
puts("Пики \006\n");
}
```

Функция puts() выведет на экран названия всех карточных мастей и соответствующие им графические символы (рис.4.4). В табл.4.1 приведены некоторые символы, которые могут быть изображены только путем использования их кодов.

Бубны	♦
Черви	♥
Трефы	♣
Пики	♠

Рис. 4.4. Отображение графических символов по их кодам

Для отображения графических символов можно использовать и функцию putchar() за счет двойственности символьных переменных. Если вы определили переменную типа int, ей, понятное дело, можно присвоить числовое значение:

```
int count;

count=5;
```

Если затем вы используете эту переменную как параметр, передаваемый функции putchar()

```
putchar(count);
```

символ, ассоциированный со значением переменной, будет отображен на экране.

Таблица 4.1. Некоторые используемые коды символов.

Восьмеричный код	Символ	Восьмеричный код	Символ
003	♥	367	≈
004	◆	370	°
005	♣	371	•
006	♠	372	·
364	⌠	373	√
365	⌡	374	n
366	÷	375	2

Один из символов в наборе ASCII не появляется на экране, зато при попытке вывести его раздается звонок! Использование escape-последовательности \007 приводит к подаче звукового сигнала встроенным динамиком вашего компьютера. Приведенная ниже программа дважды подает звуковой сигнал, привлекая внимание к сообщению на экране:

```
#define BELL '\007'    /* BELL легче запомнить, чем \007 */

main()

{

putchar(BELL);                    /*Подача звукового сигнала*/

putchar(BELL);                    /*Подача звукового сигнала*/

puts("Вниманию покупателей!\n");

puts("В отделе спортивных товаров \

 проводится сезонная распродажа\n");

}
```

Директива #define определяет константу BELL со значением \007. И хотя для определения константы вам пришлось ввести с клавиатуры четыре символа, компилятор воспримет ее как обычную символьную константу. Сигнал динамика слышится тогда, когда вы пытаетесь вывести константу BELL на дисплей с помощью функции putchar().

Замечания по Си++

Если вы чувствуете в себе силы, то можете познакомиться с некоторой дополнительной информацией о специальных символах. Вы можете найти описание набора ASCII-символов во многих руководствах по компьютерам и печатающим устройствам. Обычно номера кодов приведены в десятичной и шестнадцатеричной системах счисления. Некоторые компиляторы Си воспринимают запись в этих форматах, но стандарты Си K&R и ANSI подразумевают использование восьмеричных кодов. Применяйте восьме-

ричные коды (приведенные в табл.4.1) для совместимости с любыми компиляторами Си.

Расширенный набор ASCII-символов содержит все буквы и символы, которые можно *отобразить на экране* монитора, а не только те, которые *вводятся непосредственно с клавиатуры*. В некоторых источниках вы можете найти его подмножество, набор ASCII-символов, который включает только коды от 0 до 127 в десятичной системе счисления. Сюда входят только символы, которые можно ввести с клавиатуры.

Многогранная функция printf()

Функции puts() и putchar() используются довольно часто, но, к сожалению, их возможности несколько ограничены. Ни одна из них не может обеспечить вывод числовых данных, и обе они имеют только один *аргумент* (параметр). Это означает, что обе функции могут отобразить только один объект.

Языки Си и Си++ имеют более многостороннюю функцию, называемую printf(). Она позволяет выводить на дисплей данные всех типов и работать со списком из нескольких аргументов. Кроме того, при вызове функции printf() можно определить способ форматирования данных.

В простейшем случае функцию printf() можно использовать вместо функции puts() для вывода строки:

```
#define MESSAGE "Привет!"
main()
{
printf(MESSAGE);
printf("Добро пожаловать в мой мир, а теперь убирайся");
}
```

Так же как и puts(), функция printf() будет выводить на экран строки, заключенные в кавычки, и значения строковых констант и переменных.

Си++ имеет дополнительное многоцелевое средство вывода cout. Вывод в Си++ рассматривается далее в этой главе.

Вывод чисел

Для того чтобы отобразить числовые данные и иметь возможность форматировать данные всех типов, список параметров, передаваемый функции printf(), делится на две части (рис.4.5).

Рис. 4.5. Две части списка параметров функции printf()

Первый параметр называется *управляющей строкой* или *строкой формата**. Этот параметр заключается в кавычки и указывает компилятору, в какой позиции строки должны появиться данные. Строка формата может содержать любой текст вместе с метками, которые называются *указателями формата*** и определяют тип данных, а также их расположение.

*** Буквальный перевод английских терминов *control string или format string. (Прим.перев.)***

**** В оригинале *format specifier. (Прим.перев.)***

Каждый указатель формата начинается с символа процента (%), после которого стоит буква, указывающая тип данных:

%d целое число

%u беззнаковое целое число

%f вещественное число типа float или double

%e вещественное число в экспоненциальной форме

%g вещественное число, отображаемое по формату %f или %e, в зависимости от того, какая форма записи является более короткой

%c символ

%s строка

Таким образом, первая часть инструкции printf() записывается так:

printf("%d")

Знак процента говорит компилятору, что за ним последует указатель формата (чтобы отобразить сам символ процента, напишите его дважды: printf("%%");).

Буква d указывает компилятору, что следует отобразить целое число, записанное в десятичной системе счисления (рис.4.6).

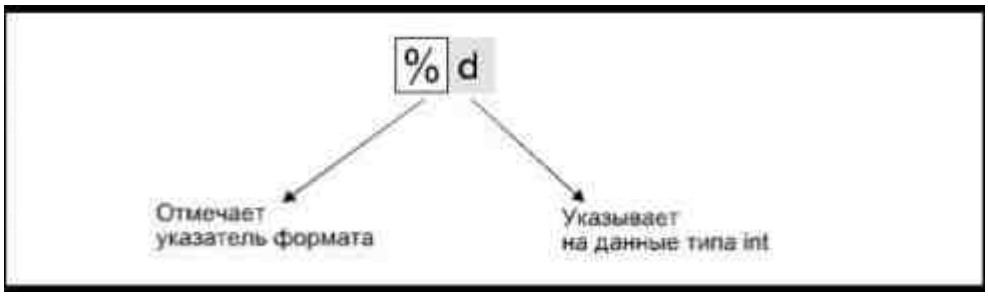

Рис. 4.6. Составляющие указателя формата

Второй частью списка параметров является *список данных*, содержащий литералы, имена констант или имена переменных, значение которых необходимо отобразить на дисплее. Список данных отделяется от строки формата запятой. Между собой все элементы списка данных также разделяются запятыми. Когда компилятор создает объектные коды, он подставляет на место указателей формата значения из списка данных.

Простейший пример использования функции printf() приведен ниже:

printf("%d", 12);

В процессе выполнения этой инструкции значение 12 будет подставлено на место указателя формата (рис.4.7). В нашем примере мы на самом деле передали

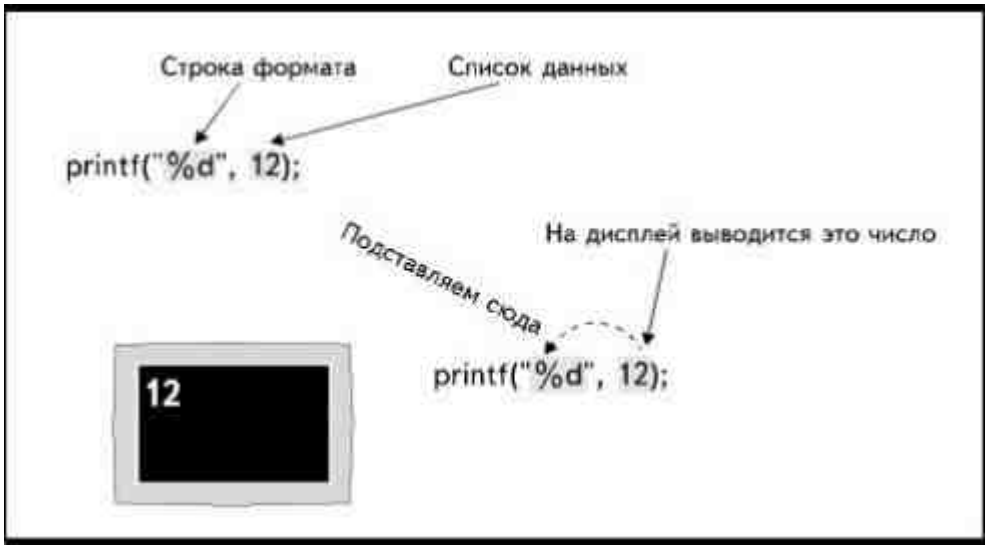

Рис. 4.7. Значение подставляется на место указателя формата

библиотечной функции printf() два параметра: строку формата и числовой литерал12.

Строка формата может содержать и обыкновенный текст с включенными в него указателями формата. Например, взгляните на инструкцию:

printf("Мне исполнилось %d лет", 12);

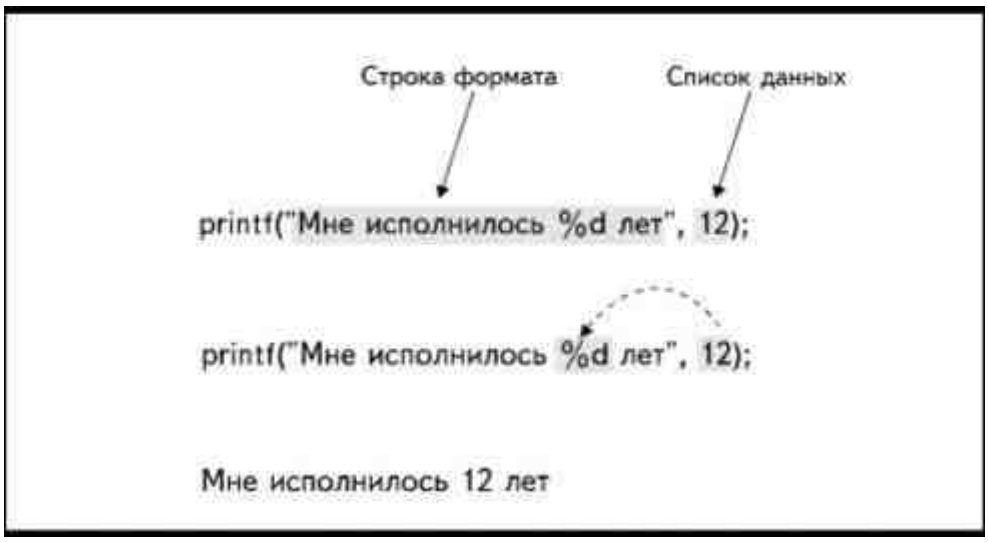

Рис. 4.8. Использование указателя формата внутри строкового литерала

Строкой формата в этом примере является запись

"Мне исполнилось %d лет"

Указатель формата, %d, говорит о том, что мы хотим вставить число между словами "Мне исполнилось" и словом "лет" (рис.4.8). Когда компилятор подставит число12 на место указателя формата, мы увидим следующую фразу:

Мне исполнилось 12 лет

В этом примере функции передается одновременно и строковый литерал, и числовое значение.

В данном случае тот же результат можно получить, передавая всю фразу целиком, как параметр, одной из функций:

```
printf("Мне исполнилось 12 лет");

puts("Мне исполнилось 12 лет");
```

Но чтобы комбинировать текст с числовыми константами или переменными, следует использовать именно функцию printf() и указатели формата, как, например, в программе:

```
main()
        {
        int age;
        age = 12;
        printf("Мне исполнилось %d лет", age);
        }
```

Эта программа отображает на экране строковый литерал и значение целочисленной переменной с помощью одной инструкции (рис.4.9).

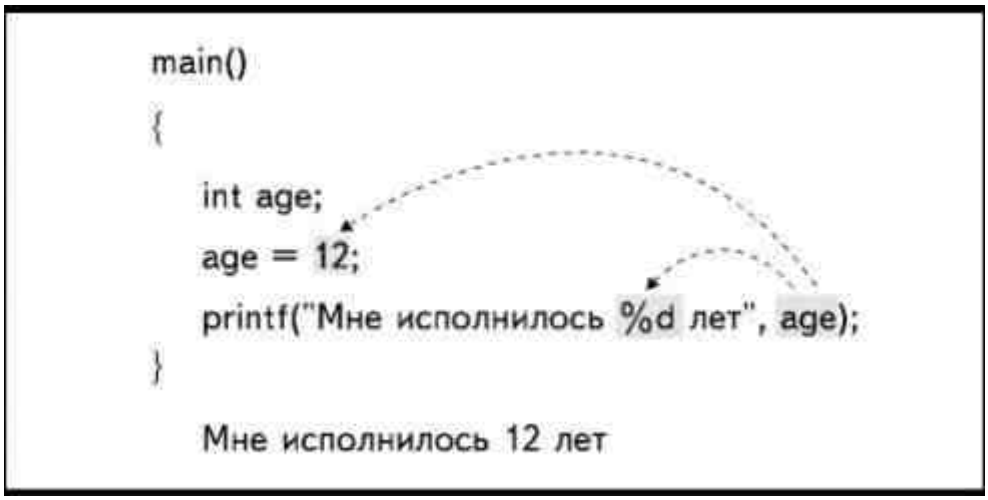

Рис. 4.9. Значение переменной подставляется на место указателя формата

Функции printf() можно передать любое число параметров, чтобы отобразить несколько аргументов. При этом необходимо ставить указатель формата для каждого аргумента. Значения в списке данных должны располагаться в том же порядке, что и соответствующие указатели формата: первый пункт из списка данных подставляется на место первого указателя формата, второй— на место второго и так далее. Взгляните на программу:

```
main()
{
int lucky_1, lucky_2;
lucky_1 = 12;
lucky_2 = 21;
printf("Моими счастливыми номерами являются \
 %d и %d", lucky_1, lucky_2);
}
```

Здесь мы определили две целочисленные переменные lucky_1 и lucky_2 и присвоили им начальные значения. Список данных функции printf() содержит два имени переменных (аргумента), которые мы хотим отобразить, так что строка формата тоже должна иметь два указателя формата. Так как обе переменные относятся к типу int, оба указателя формата одинаковы— %d, как показано на рис.4.10.

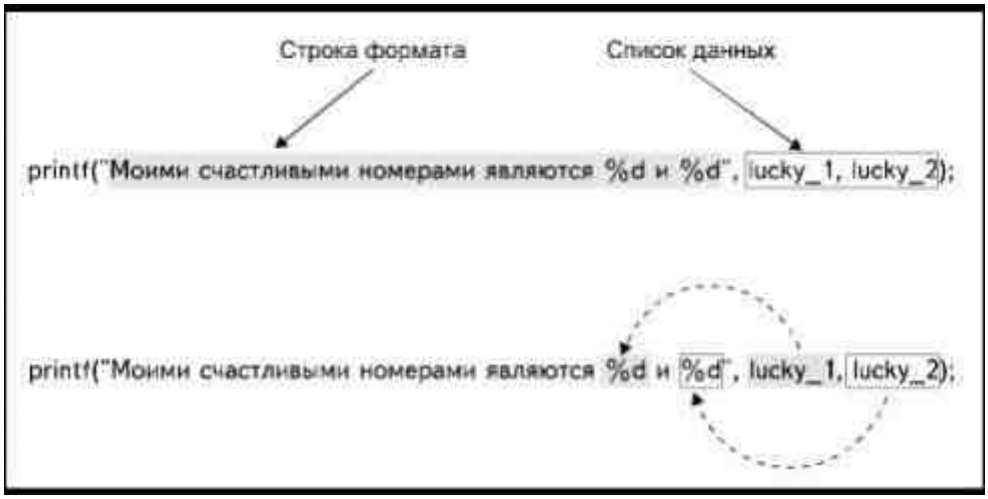

Рис. 4.10. Использование двух указателей формата

Компилятор подставит значения на место указателей формата, и фраза будет выглядеть так:

Моими счастливыми номерами являются 12 и 21

Значение переменной lucky_1, первого пункта в списке данных, займет место первого указателя формата, а второй пункт списка данных (значение переменной lucky_2) будет подставлен на место второго указателя. Если пункты в списке данных поменять местами:

```
printf("Моими счастливыми номерами являются \
 %d и %d", lucky_2, lucky_1);
```

то значения отобразятся в таком порядке:

Моими счастливыми номерами являются 21 и 12

Тип данных должен соответствовать типу указателя формата. Например, следующая программа отображает значения переменной типа float и переменной типа int с помощью одной функции printf():

```
main()
{
int count;
float amount;
count = 5;
amount = 45.48;
printf("Стоимость составляет
                    %f за %d штук", amount, count);
```

}

Первое значение из списка данных— число с плавающей точкой— подставляется на место первого указателя формата %f. Второе значение— целое число— подставляется на место второго указателя формата %d. Программа компилируется и выполняется без ошибок, так как типы переменных и указателей формата совпадают. Как показано на рис.4.11, значение типа float

```
main()
{
   int count;
   float amount;
   count = 5;
   amount = 45.48;
   printf("Стоимость составляет %f за %d штук", amount, count);
}
```

Рис. 4.11. Тип данных должен соответствовать указателю формата

замещает указатель %f, а значение типа int— указатель %d. В результате имеем:

Стоимость составляет 45.580000 за 5 штук

Количество нулей, которое будет проставлено в числе с плавающей точкой зависит от конкретного компилятора (вскоре вы узнаете, почему они появляются). Если поменять местами элементы в списке данных и записать инструкцию следующим образом:

printf("Стоимость составляет \

%f за %d штук", count, amount);

компилятор может не сообщить об ошибке, но в результате появится бессмысленная запись:

Стоимость составляет -2.002149E37 за 16454 штук

К такому результату привело несоответствие типов числовых данных и указателей формата. При вызове функции printf() можно использовать несколько аргументов различных типов, но только при строгом соответствии типов данных и указателей формата.

Перевод строки

Функция printf() не переводит курсор автоматически на новую строку после отображения данных. После того как данные выведены на экран, курсор остается на той же строке, сразу за последним символом.

Если вы хотите перевести курсор на следующую строку, вы должны добавить в строку формата управляющий код «новая строка» \n:

printf("Стоимость составляет %f за %d

штук\n", amount, count);

Управляющий код \n помещают там, где хотят начать новую строку (не обязательно в конце строки формата), например, в результате выполнения инструкции

printf("Стоимость составляет %f\n за

%d штук\n", amount, count);

на экране появятся две строки:

Стоимость составляет 45.580000 за 5 штук

Можно использовать любые другие escape-последовательности для регулирования пробелов, подачи звукового сигнала или изображения специальных символов.

Преобразование типов данных

В Си существует два дополнительных указателя формата, с помощью которых можно переводить данные типа int из десятичной в восьмеричную и шестнадцатеричную системы счисления:

%o перевод в восьмеричную систему, используйте только маленькую букву о, не цифру 0

%x перевод в шестнадцатеричную систему

Для того чтобы перевести число из десятичной в другую систему счисления, надо поместить код %o или код %x в строку формата, при этом в список данных вносится число, записанное в десятичной системе. В качестве примера приведена программа, отображающая число17 в шестнадцатеричной и восьмеричной системах счисления:

```
main()
        {
printf("%d равно %x в шестнадцатеричной и %o в \
 восьмеричной системах счисления\n", 17, 17, 17);
                // так в Си и Си++ можно переносить
                // длинные строки символов,
                // заключенные в двойные кавычки (ред.)
        }
```

Результат работы программы выглядит следующим образом:

```
17 равно 11 в шестнадцатеричной и 21
в восьмеричной системах счисления
```

Значение числа в восьмеричной системе счисления может понадобиться для вывода графических символов. Если соответствующий символу номер известен в десятичной системе, можно использовать программу, подобную приведенной выше, для перевода его в восьмеричную.

Двойственность символьных переменных

При описании функции putchar() уже говорилось, что символьные переменные могут быть заданы и как целочисленные. Следовательно, можно присвоить букву в качестве значения переменной типа int и вывести ее на дисплей при помощи функции putchar() или printf():

```
main()
        {
        int a;
        a = 'A';
        putchar(a);
        putchar('\n');
        printf("%c", a);
        }
```

При выполнении этой программы символ A появится дважды: один раз как символ, отображенный функцией putchar(), второй раз как символ, отображенный функцией printf() с использованием указателя формата %c. Независимо от того, определен тип переменной как char или int, можно преобразовать символ в его ASCII-код:

```
main()
        {
        char a;
        a = 'A';
        printf("ASCII-код символа %c равен %d", a, a);
        }
```

В этом примере одно и то же значение переменной выводится на дисплей с использованием указателей формата %c и %d. В результате получаем следующее сообщение:

ASCII-код символа A равен 65

где присутствует символ A, отображенный с использованием указателя формата %c, и число 65, являющееся ASCII-кодом символа A, выведенное с помощью указателя формата %d.

Программа будет работать точно так же, и в результате мы получим ту же фразу, если переменная a будет определена как int.

Форматированный вывод

Функцию printf() можно использовать для управления форматом данных. Определять величину пробелов и количество выводимых символов можно с помощью *указателей ширины поля**.

Без указателя ширины поля числа с плавающей точкой, например, будут выводиться с шестью знаками после точки. Поэтому в результате выполнения инструкции

```
printf("Стоимость составляет %f за
                    %d штук", amount, count);
```

и появляется строка:

Стоимость составляет 45.580000 за 5 штук

В зависимости от особенностей системы и от того, как ведется расчет среднего значения, может появиться и что-нибудь в таком роде:

Стоимость составляет 45.579998 за 5 штук

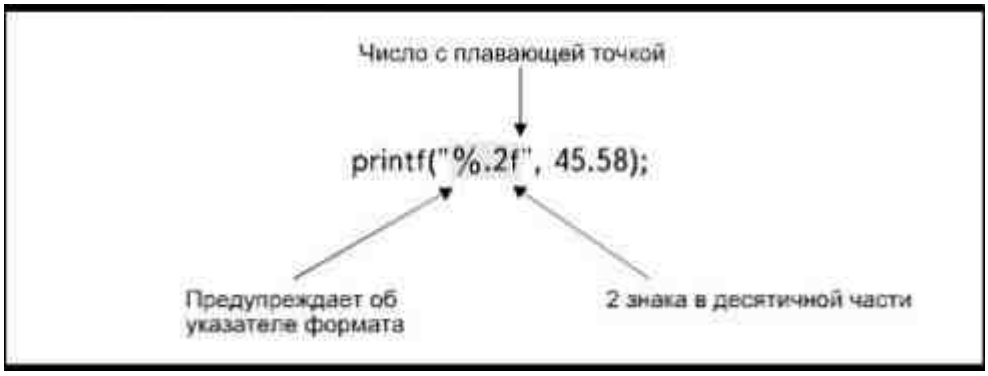

Рис. 4.12. Определение количества знаков после точки

*** Буквальный перевод английского термина *field-width specifiers. (Прим.перев.)***

Мы используем указатель ширины поля для того, чтобы придать желаемый вид числам и тексту, выводимым на экран.

Чтобы определить число знаков после точки, используется указатель %.nf, где число n определяет количество знаков (рис.4.12). Например, если написать:

```
printf("Стоимость составляет %.2f", amount);
```

то при выводе значения переменной с типом float оно будет иметь только два знака после точки:

Стоимость составляет 45.58

Сходным образом, если написать:

```
printf("Стоимость составляет %.3f", amount);
```

значение переменной будет представлено с тремя знаками после точки:

Стоимость составляет 45.580

На рис.4.13 приведено несколько примеров определения количества знаков в десятичной части переменной.

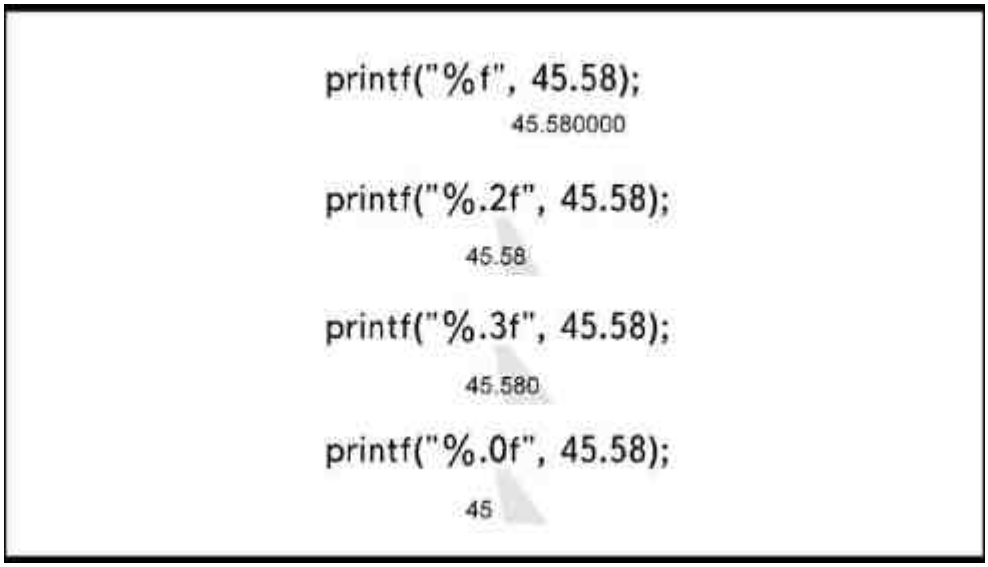

Рис. 4.13. Примеры определения количества знаков в десятичной части

Можно также определить общую ширину поля, то есть размер (в символах) пространства, занимаемого выводимым числом, если использовать следующий формат:

%N.nf

где N— это общая ширина поля.

Если задать инструкцию

printf("Стоимость составляет %8.2f", amount);

появится строка

Стоимость составляет 45.58

с тремя дополнительными пробелами перед числом. Чтобы понять, почему это произошло, взгляните на рис.4.14: указатель ширины поля сообщает компилятору, что числовое значение должно быть как бы втиснуто в «коробочку» размером восемь символов. Само по себе число занимает пять из них, включая точку, а неиспользованные символы отображаются на экран в виде пробелов, создавая перед числом пустое пространство.

Рис. 4.14. Указатель ширины поля определяет количество пробелов на экране

Если ширина поля, заданного указателем, оказывается меньше количества символов, составляющих число, Си, тем не менее, выведет число целиком, просто игнорируя в данном случае указатель ширины поля. Выполнение инструкции

printf("Стоимость составляет %2.2f", amount);

приведет к появлению сообщения

Стоимость составляет 45.58

Можно также добавить лишнее пространство перед числом, но заполнить его нулями вместо пробелов, как на рис.4.15. Для этого перед числом, определяющим ширину поля, надо поставить символ 0:

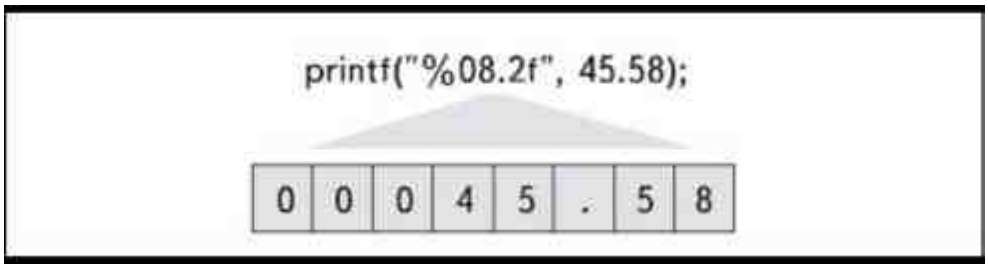

Рис. 4.15. Вывод нулей перед числом

printf("Стоимость составляет %08.2f", amount);

в результате мы увидим сообщение

Стоимость составляет 00045.58

Для того чтобы выровнять значение по левому краю (как бы поместить его слева в нашей воображаемой «коробочке»), в указатель ширины поля после % вводится знак «минус»

%-8.2f

Лишние пробелы в этом случае появятся после выведенного значения. Инструкция

printf("Стоимость составляет %-8.2f

в долларах США", amount);

выводит на экран сообщение

Стоимость составляет 45.58 в долларах США

Как показано на рис. 4.16, число 45.58 опять как бы помещается в «коробочку» из восьми символов, но теперь уже сдвигается к левому краю, а пустое пространство появляется за числом.

Указатель ширины поля может работать как с символьными, так и со строковыми данными. Дополнительные пробелы помещаются перед текстом, сдвигая строку к правому краю воображаемой «коробочки». Например, если строковая переменная, называемая message, имеет значение «Привет», то инструкция

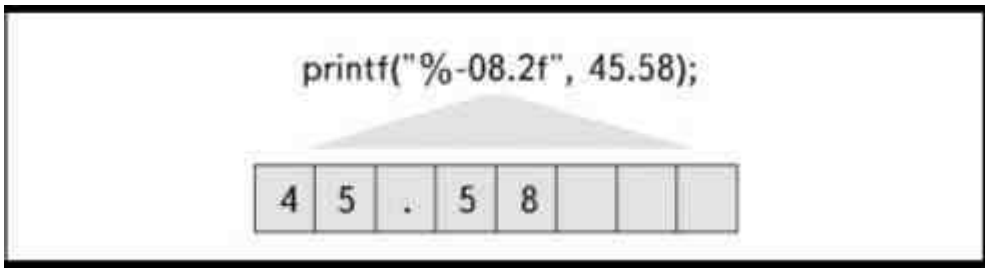

Рис. 4.16. Выравнивание по левому краю

printf("Я позвонил, чтобы сказать %8s", message);

отобразит на экране следующую строку

Я позвонил, чтобы сказать Привет

Как показано на рис. 4.17, при выводе значения строковой переменной message

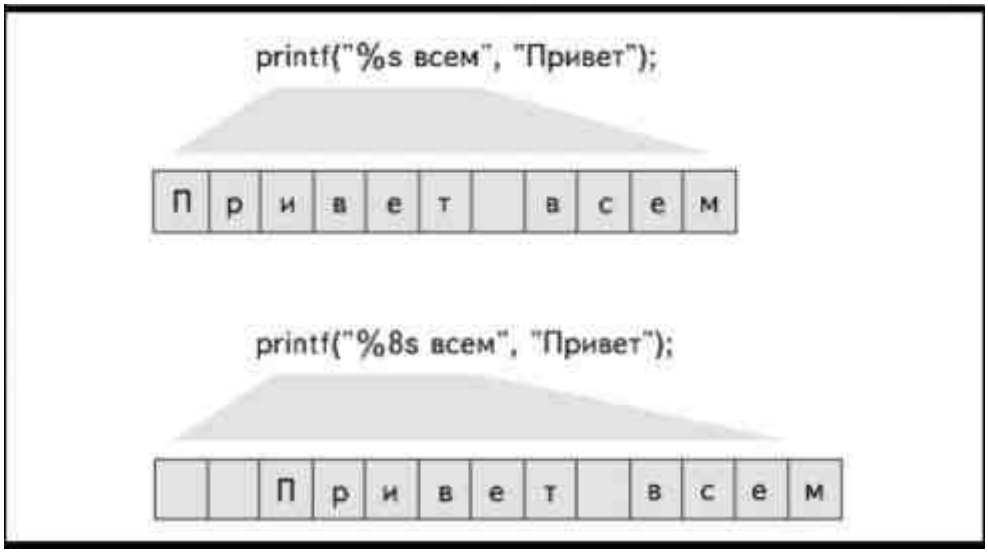

Рис. 4.17. Использование указателя ширины поля при выводе строки

перед ним появятся два дополнительных пробела.

Выбор правильных средств вывода информации

Когда вы планируете способ представления информации в вашей программе, обдумайте, какие именно функции наилучшим образом соответствуют вашим целям.

Чтобы вывести на экран обычный текст или символы, можно использовать функции puts() или putchar(). Так как эти функции не имеют никаких возможностей форматирования данных, они работают быстрее, и их коды занимают меньший объем на диске, чем коды функции printf(). Имея дело с функцией puts(), прежде всего проверьте, добавляет ли компилятор код «новая строка» автоматически. Если он не делает этого, а вы не проверите сразу, потом вам придется потратить довольно много времени на редактирование программы.

Рис. 4.18. Правила использования функции printf()

Кстати, при работе с функцией printf(), пропуск кода «новая строка» тоже является распространенной оплошностью среди начинающих программистов.

Функция printf() работает медленнее и требует большего объема памяти, но она идеально подходит в тех случаях, когда вам требуется выводить числовые данные, форматировать строки или комбинировать текст и числовые переменные в одной строке. Работая с функцией printf(), следует тщательно следить за тем, чтобы указатели формата соответствовали литералам, константам и переменным в списке данных. Рис.4.18 иллюстрирует наиболее важные моменты, необходимые, чтобы правильно написать инструкцию, использующую функцию printf().

Вывод в Си++

Все обсуждавшиеся ранее приемы программирования относятся к выводу данных как в языке Си, так и Си++. Однако язык Си++ имеет дополнительный способ вывода данных всех типов.

В Си++ существует стандартный поток вывода cout, позволяющий в сочетании с двумя символами «меньше»

(<<)

, которые называются оператором вставки,* отображать литералы или значения констант и переменных без использования указателей формата.

Если у вас есть компилятор Си++, посмотрите документацию к нему. Не исключено, что необходим специальный файл заголовков для того, чтобы иметь

*** В оригинале *insertion operator*. Также часто используется термин «оператор вывода». *(Прим.перев.)***

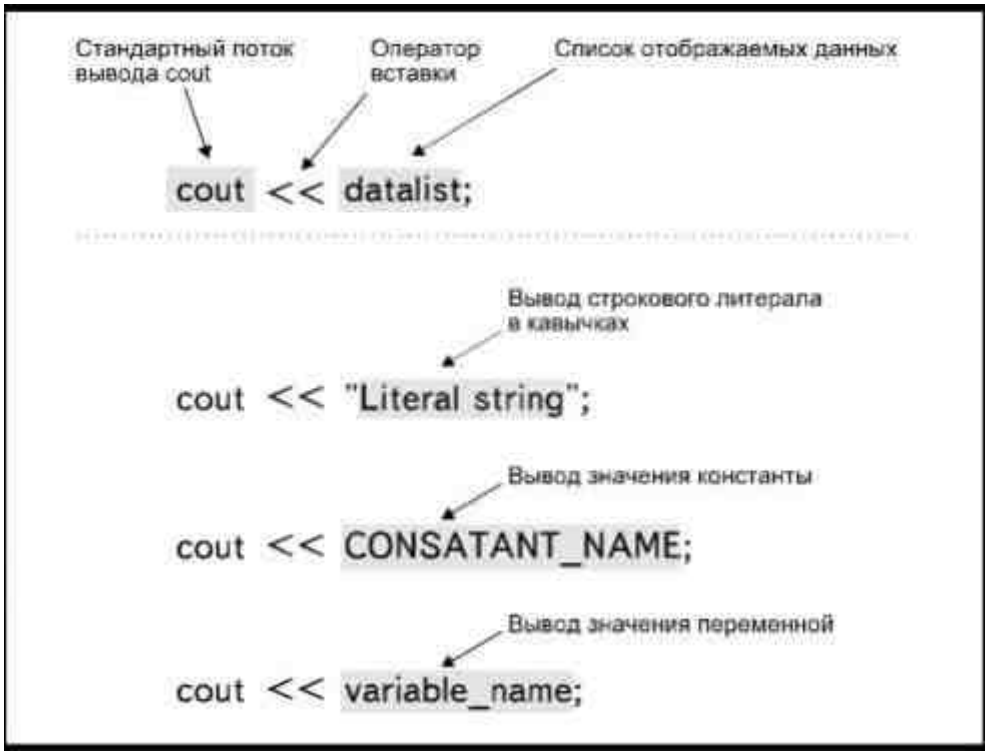

Рис. 4.19. Стандартный поток вывода cout

возможность использовать преимущества стандартного потока вывода cout и стандартного потока ввода cin (о нем вы прочтете в следующей главе). Для некоторых компиляторов, например, необходимо включить файл STREAM.H с помощью директивы #include в начале программы (директива #include и использование файлов заголовков рассматривались в главе 2).

Структура инструкции, использующей cout, показана на рис.4.19. После cout надо поставить два знака

<

. Они указывают cout отобразить помещенную после них информацию. Информация может быть представлена в виде литерала (тогда ее заключают в кавычки) либо имени константы или переменной.

Рассмотрим инструкцию

```
cout << "Привет, меня зовут Сэм. Мы
                    с вами уже встречались";
```

При ее выполнении произойдет вывод на дисплей строки, заключенной в кавычки. Инструкция

```
int count;
count = 4509;
cout << count
```

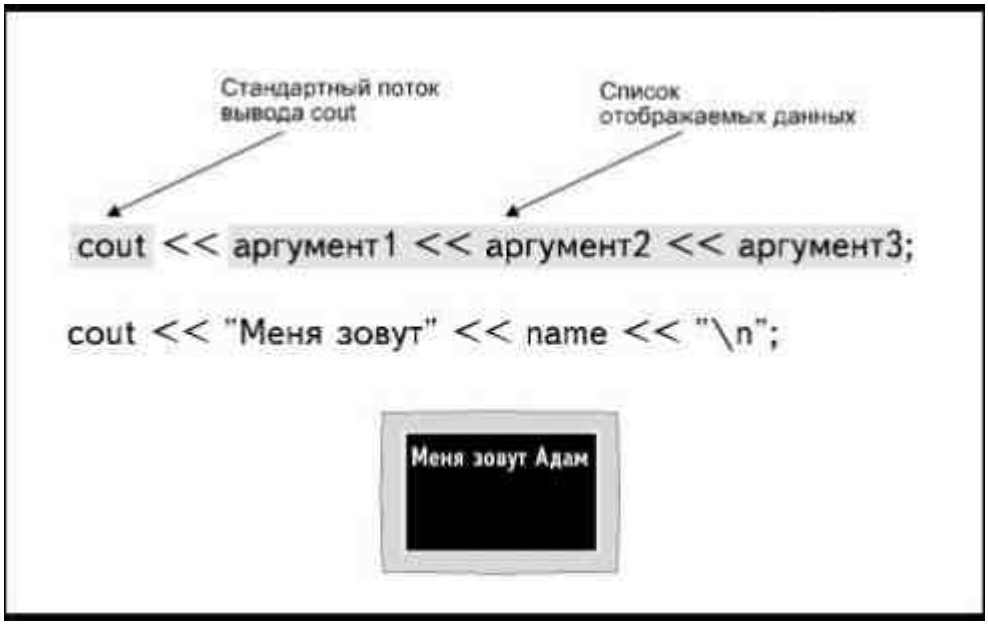

Рис. 4.20. Вывод нескольких аргументов с помощью cout

отобразит значение переменной с именем count— число 4509.

Используя один стандартный поток вывода cout, можно отобразить несколько аргументов. Между собой аргументы разделяются операторами вставки, как это продемонстрировано на рис.4.20. Например, инструкция

```
int age;
age = 43;
cout << "Вам исполнилось " << age << " года.";
```

отображает текст

Вам исполнилось 43 года.

Стандартный поток вывода cout отображает каждый пункт, указанный с помощью оператора вставки, в том порядке, в каком они записаны в инструкции.

Так же, как и функция printf(), cout не добавляет никаких команд новой строки после отображения данных. Чтобы перейти к новой строке, там, где вы хотите ее начать, надо добавить управляющий код \n, как это показано на рис.4.20.

Стандартный поток вывода cout не требует обязательного использования указателей формата, но позволяет их ввести. С помощью указателей формата можно определять ширину поля, количество пробелов и число знаков после точки в вещественных числах. Описание форматирования вывода при использовании cout не входит в задачу этой книги. Если у вас есть компилятор Си++, вы можете подробно прочитать об этом в его описании. Не забудьте

Замечания по Си++

Одна из особенностей языка Си++— так называемая *перегрузка* — делает необязательным использование указателей формата. В отличие от функции printf(), которая требует обязательного указания формата данных, cout при передаче параметров сам определяет формат на основании типа получаемых данных. Этот процесс и называется перегрузкой.

посмотреть в документации, надо ли включать специальные файлы заголовков при использовании cout.

Проектирование программы

Вывод, то есть отображение информации на экране, принтере или каком-либо другом устройстве, является важнейшей составляющей любой программы, так что планировать вывод необходимо самым тщательным образом.

Начните программу с вывода инструкций, которые объясняют цель ее создания:

```
puts("Добро пожаловать\n");

puts("Эта программа рассчитывает платежи по закладным.\n");

puts("Введите сумму займа,\n");

puts("проценты по закладным и срок выплаты в годах.\n");
```

В следующей главе рассказывается о вводе информации в программу. Прежде чем предложить ввести данные, убедитесь, что вы точно объяснили пользователю (даже если этот пользователь— вы сами), что именно он должен ввести. Используйте функцию, выводящую строку с комментарием, для каждого ввода информации:

```
puts("Пожалуйста, введите сумму полученного займа:");
```

Когда вам нужно вывести результат, делайте информацию максимально доступной для чтения и понимания:

```
printf("Основные месячные выплаты:        %7.2f\n", princ);

printf("Проценты:                         %7.2f\n", interest);

printf("Общие месячные выплаты:                   %7.2f\n", total);
```

Дополнительные пробелы между двоеточиями и указателями формата, выравнивание указателей и указатели сами по себе приводят к тому, что числовые значения выравниваются следующим образом:

```
Основные месячные выплаты:    256.25

Проценты:                     92.12

Общие месячные выплаты:       34.37
```

Такой способ представления информации кажется более удобным, чем, например, следующий:

```
printf("%f %f %f", prins, inter, total);
```

в результате чего имеем:

```
256.25 92.12 34.37
```

Старайтесь все время поступать в соответствии с предлагаемыми правилами. Процесс создания программы отнимет несколько больше времени, но результат того стоит. Программа будет выглядеть более профессиональной и создаст максимум удобства для всех, кто будет ею пользоваться.

В табл. 4.2 подведен краткий итог всем способам вывода данных, которые обсуждались в этой главе.

Таблица 4.2. Средства вывода информации в языке Си/Си++.

Функция или ключевое слово	Тип данных	Комментарии
puts()	только строки	Си и Си++. Используется только для вывода строк. Выводит строковые литералы в кавычках, значения переменных и констант. Может автоматически перемещать курсор на следующую строку после вывода (не забудьте проверить документацию).
putchar()	единичный символ	Си и Си++. Используется для вывода символа, определенного как тип char или int. Не переводит курсор на следующую строку автоматически (проверьте документацию). Используйте для вывода литералов в одинарных кавычках, констант, переменных типа char, специальных символов или escape-последовательностей.
printf()	все типы данных	Си и Си++. Необходимы указатели формата для каждого элемента данных. Может обеспечить вывод нескольких аргументов. Не переводит курсор автоматически, используйте escape-последовательность \ n.
cout	все типы данных	Только Си++. Отделяйте аргументы знаками <<. Не переводит курсор автоматически. Указатели формата не обязательны.

Вопросы

1. Что такое вывод?
2. Какие три типа аргументов можно использовать с функцией puts()?
3. Какие три типа аргументов можно использовать с функцией putchar()?
4. Что такое escape-последовательности?
5. В чем заключаются различия между escape-последовательностями \n и \r?
6. Как вывести на экран символ «кавычка»?
7. Из каких двух частей состоит список параметров функции printf()?
8. Какие преимущества имеет функция printf() по сравнению с puts()?
9. Что такое указатель формата?
10. Как вывести на экран значение числовой переменной?

Упражнения

1. Напишите программу вывода вашего имени и адреса на экран с ис-

пользованием функции puts().

2. Напишите программу вывода вашего имени и адреса на экран с использованием функции printf().
3. Напишите функцию puts(), которая выводит слово «Заглавие» в середине экрана. Ширина экрана 80 символов.
4. Напишите функцию printf(), которая выводит слово «Страница» с правой стороны экрана.
5. Напишите функцию printf(), которая выводит значения следующих переменных:

   float length, width, height, volume;

6. Программа должна отображать имя и возраст субъектов. Напишите функцию printf(), которая выводила бы значения переменных:

7. char name[12];

   int age;

8. Программа содержит следующие переменные:

9. char item[] = "Дискеты";

10. float cost = 3.55;

    float markup = 0.75;

    Напишите функцию printf(), которая выводит на экран следующие сообщения:

    Наименование товара: Гибкий диск

    Цена за 1 упаковку: 3.55

    Наценка: 0.75

    Обратите внимание на выравнивание.

11. Программа содержит следующую переменную:

    int count = 30;

    Используя значение переменной count для вывода числа в последней строке, напишите программу, которая подает звуковой сигнал и выводит на экран следующее сообщение:

    Внимание! Внимание! Внимание! Внимание!

    Нежелательное отклонение параметров среды.

    У вас есть 30 секунд, чтобы покинуть помещение.

ГЛАВА 5
ВВОД В СИ/СИ++

Вводом называется процесс предоставления компьютеру информации, необходимой для работы программы. Информация вводится в переменные, это означает, что данные, которые пользователь вводит в ответ на соответствующую подсказку, определяются как значение переменной, хранящейся в памяти. Затем переменная используется в выполняемых программой операциях. Из рисунка 5.1 видно, что ввод информации может происходить из различных источников.

Ввод данных — это процесс, который определяет всю дальнейшую работу программы. Достоверность получаемых от программы результатов не может быть больше, чем достоверность вводимой в нее информации. Как говорится, чистый вход— чистый выход. Если введенные в программу данные были ошибочными, все полученные результаты тоже окажутся неверными.

В этой главе мы подробно рассмотрим ввод данных с клавиатуры. В главе 12 вы узнаете, как программа берет данные из дискового файла.

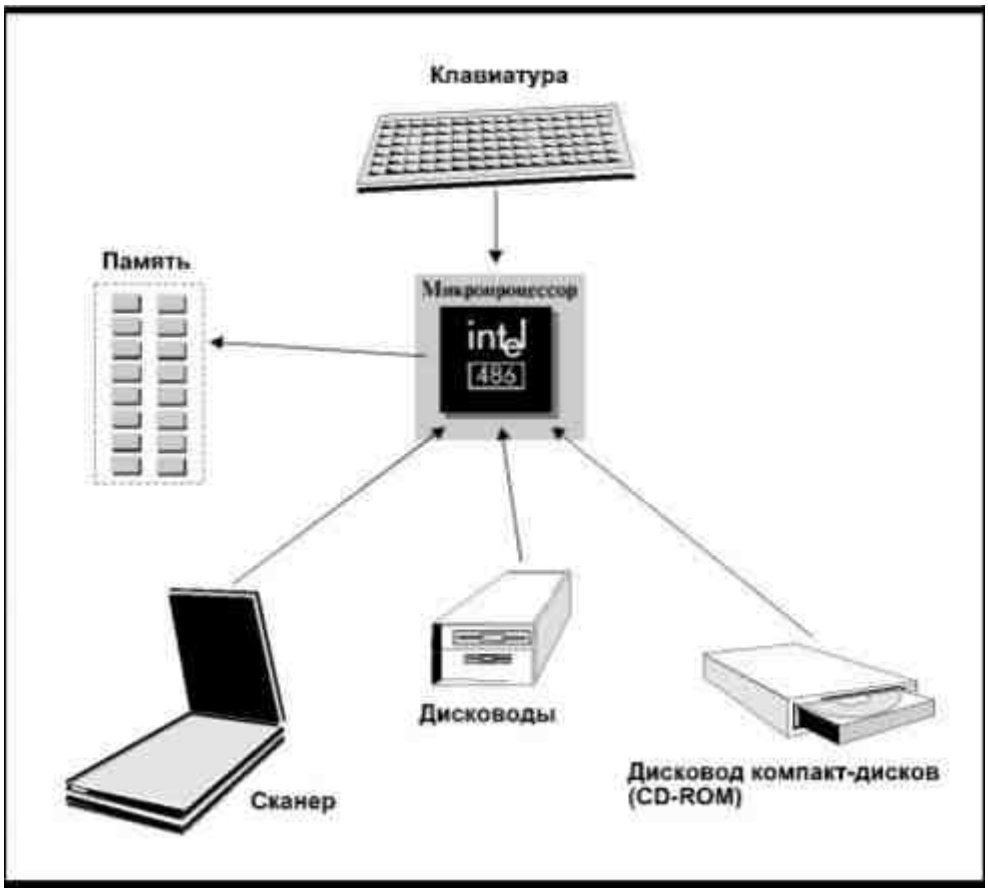

Рис. 5.1. Откуда бы ни поступали данные,компьютер хранит их в памяти как переменные

Как вы увидите, слово *ввод* употребляется в двух значениях: как существительное, означающее данные, полученные компьютером, и как глагол, обозначающий сам процесс ввода информации.

Замечания по Си++

Си++ может работать со всеми функциями ввода Си.

Данные вводятся только как значения переменных, но не констант. Константы всегда сохраняют присвоенные им начальные значения. Когда вы вводите в переменную некоторые данные, они помещаются в отведенную для этой переменной область памяти. Если переменной уже присвоено какое-то значение, новая информация заместит прежнюю, уничтожив ее (рис.5.2).

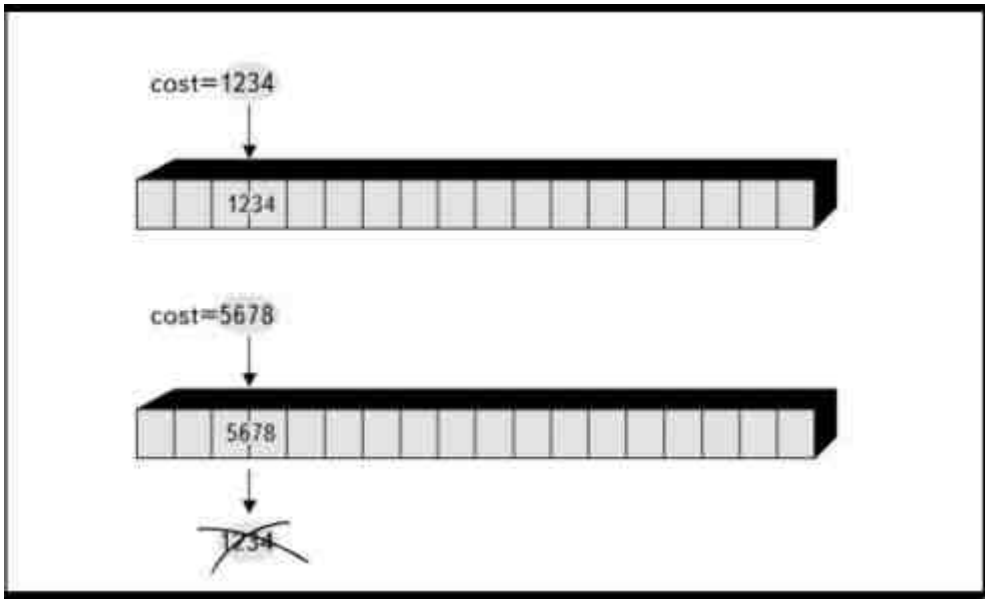

Рис. 5.2. При вводе данных прежнее значение переменной теряется

Функция gets()

Функция gets() вводит строку в переменную. Параметром функции является имя переменной. Рассмотрим такую программу:

```
main()
        {
        char name[15];
        gets(name);
        puts(name);
        }
```

Функция gets() будет рассматривать первые 14 символов, введенные с клавиатуры, как значение строковой переменной с именем name. Вы помните, что Си отводит строковой переменной столько элементов памяти, сколько указано в максимальном значении при определении переменной, а так как один элемент необходим для нулевого символа (\0), реально можно ввести на один символ меньше. Если вы хотите ввести в переменную name строку, состоящую из 15 символов, то укажите в квадратных скобках максимальное значение 16:

```
char name[16];
```

На время работы функции gets() выполнение программы приостанавливается. Она ждет, пока пользователь что-то напечатает. Для того чтобы напечатанные данные стали значением переменной, после ввода информации надо нажать клавишу **Enter**. Как только это сделано, строка, введенная пользователем, присваивается переменной в качестве значения, а курсор переходит на следующую строку на экране. При нажатии **Enter** Си добавляет в строку нулевой символ.

Когда вы вводите символы при выполнении инструкции gets(), они отображаются на экране монитора (в режиме эха), при этом не используются никакие функции вывода, и символы на самом деле не будут введены в компьютер, пока не нажата клавиша **Enter**.

Рассмотрим более подробный пример:

```
main()
        {
        char name[25];
        printf("Пожалуйста, введите Ваше имя: ");
        gets(name);
        printf("Подтвердите, Ваше имя: %s", name);
```

```
}
```

Когда программа будет выполняться, вы увидите на экране подсказку:

Пожалуйста, введите Ваше имя:

Возьмите себе за правило использовать функции printf() или puts() для вывода на экран подсказок, говорящих пользователю перед каждой процедурой ввода данных, что именно ему надо сейчас ввести. Как вы сами убедитесь, четкие подсказки очень важны для правильного ввода информации.

Поскольку функция printf() в данном случае не содержит кода «новая строка», курсор останавливается сразу за подсказкой, на расстоянии в один пробел от двоеточия, так как в строку формата функции printf() мы ввели пробел между двоеточием и закрывающими кавычками. Итак, пока вы сидите перед компьютером и смотрите на экран, ничего не произойдет. Программа ждет, что вы введете некую информацию, в данном случае напечатаете свое имя.

Пока вы печатаете имя, символы отображаются на экран в режиме эха. Если вы допустите ошибку, ее можно исправить до того, как нажата клавиша **Enter**, уничтожив неправильные символы клавишей **Backspace** и напечатав новые. В некоторых системах можно использовать клавишу **Esc** для того, чтобы удалить все введенные символы и начать процедуру заново.

После нажатия клавиши **Enter** Си присваивает введенные символы переменной name и вставляет нулевой символ в конце строки. Затем программа переходит к выполнению второй функции printf() и на экране появляется вторая надпись (предположим, что вас зовут Петр Иванов):

Пожалуйста, введите Ваше имя: Петр Иванов

Подтвердите, Ваше имя: Петр Иванов

Помните, символы, составляющие ваше имя, отображались на экране после первой подсказки только в режиме эха, они не были введены в программу до того, как вы нажали Enter. После второй подсказки имя появляется на экране в качестве значения переменной name в результате выполнения второй функции printf().

Таким образом, функция gets() прекрасно подходит для ввода в программу строк.

Функция getchar()

Функция getchar() вводит с клавиатуры единичный символ. Для большинства компиляторов безразлично, к какому типу (char или int) вы отнесете вводимый символ, что обусловлено способом определения символьной переменной в K&R-стандарте языка Си (двойственность символьных данных подробно обсуждалась в главе 4).

Для ввода символа можно использовать оба этих формата:

```
int letter;          char letter;
letter = getchar();  letter = getchar();
```

Обратите внимание на то, что вызов функции getchar() осуществляется не так, как вызов функций, которые мы рассматривали раньше. Вместо того чтобы поставить имя функции в начало строки инструкции, мы относим его к переменной с помощью знака «равно». Эта запись означает: «Присвоить переменной letter значение, полученное в результате выполнения функции getchar()». Практически, функция getchar() рассматривается программой как значение переменной (рис.5.3). При выполнении этой инструкции вызывается функция getchar() и осуществляется ввод символа, который присваивается переменной. *Функция* getchar() *не имеет аргумента*. Поэтому круглые скобки после имени функции остаются пустыми.

Когда пользователь нажимает клавишу, getchar() отображает введенный символ на экране. В данном случае нажимать **Enter** нет необходимости, так как getchar()

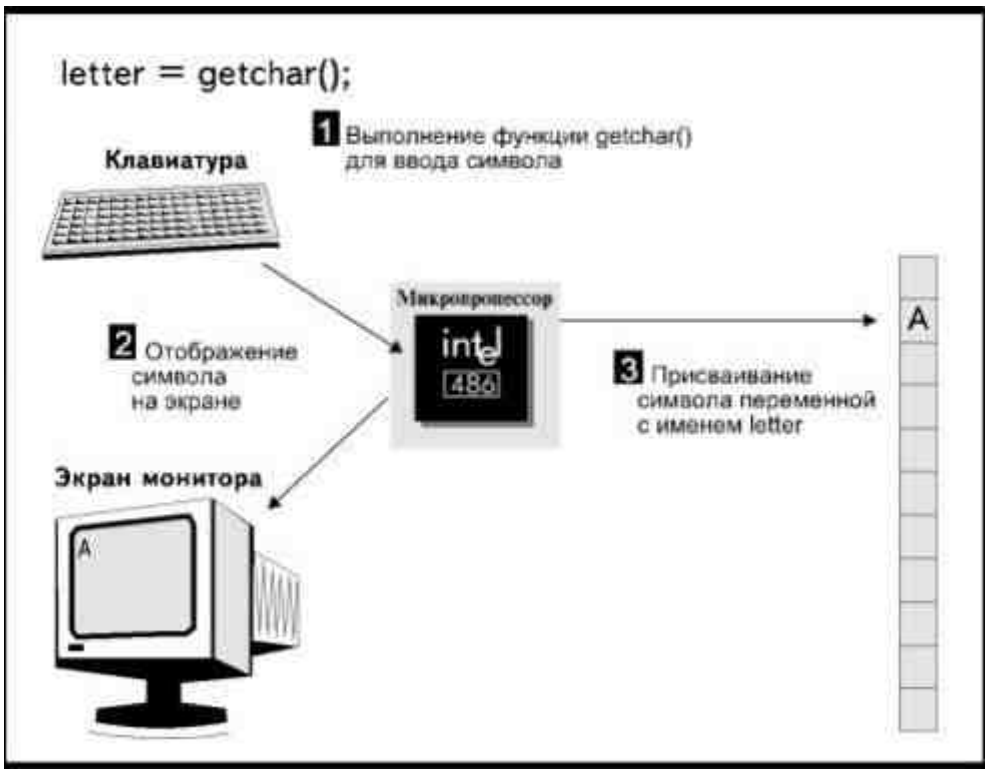

Рис. 5.3. Функция getchar()

вводит только один символ, после чего программа немедленно продолжает работу. Символ присваивается переменной, как только вы нажали какую-либо клавишу.

Некоторые компиляторы Си и Си++ используют функцию getch(), которая вводит символ без последующего нажатия **Enter**. Может оказаться, что при работе с getchar(), эти компиляторы требуют нажатия **Enter** после ввода символа. Проверьте документацию вашего компилятора.

Для чего может понадобиться ввод единичного символа? Вероятно, вам приходилось видеть программы, в которых необходимо ответить «Да» или «Нет» в ответ на запрос, или выбрать один из пунктов предложенного меню. Функция getchar() идеально подходит в этих случаях, ведь при работе с ней нет необходимости нажимать **Enter**, ввод одного из предложенных на выбор символов позволяет немедленно продолжить выполнение программы.

Если значение переменной введено с помощью функции getchar(), оно, независимо от того, определена переменная как int или как char, может быть отображено с помощью функции putchar() или printf(). В последнем случае используется указатель формата %c:

```
/*getchar.c*/
main()
        {
        int initial;
        puts("Пожалуйста, введите следующий инициал.");
        initial=getchar();
        putchar('\n');
        putchar(initial);
        putchar('\n');
        printf("%c", initial);
        }
```

Данная программа вводит символ в переменную initial, а затем отображает значение переменной на дисплее, используя функции putchar() и printf(). Если в качестве инициала вы ввели букву 'A', то на экране увидите эту букву, повторенную трижды на отдельных строках. Первый раз она появляется при вводе ее с

клавиатуры, еще два раза— в результате выполнения инструкций программы. Как правило, после ввода символа большинство компиляторов не выполняет автоматический перевод на новую строку.

Используя указатель формата %d, можно отобразить на экране ASCII-код символа:

```
/*ascii.c*/
main()
        {
        int letter;
        letter=getchar();
        printf("ASCII-кодом символа \
 %c является %d\n", letter, letter);
        }
```

Здесь мы присваиваем значение переменной letter, используя функцию getchar(), а затем, используя функцию printf(), выводим на экран значение переменной, записанное в двух форматах. Указатель формата %c обеспечивает отображение символа в таком виде, как он был введен с клавиатуры, а указатель формата %d преобразует значение переменной в целое число, ASCII-код данного символа.

Значения, введенные с помощью функции getchar(), всегда рассматриваются Си как символ, а не как числовое значение, так что, даже если определить переменную как int и вводить цифру, ее нельзя будет использовать в математических операциях.

«Для продолжения нажмите Enter»

Функцию getchar() можно использовать для приостановки выполнения программы, что может оказаться весьма полезно в некоторых ситуациях. Приведем простой пример. На экран можно одновременно вывести ограниченное количество строк (обычно 25). Если в программе используется длинный ряд инструкций puts(), при выполнении которых надо вывести больше строк, чем может поместиться на экране, первые появившиеся строки информации быстро уйдут за верхний край, и пользователь просто физически не успеет ознакомиться с их содержанием до того, как на этом месте появятся другие.

Справиться с подобной проблемой можно, если разделить длинную последовательность инструкций на блоки, включающие какое-то количество функций puts(), так, чтобы сообщения, появившиеся в результате их выполнения, заняли только часть экрана, и в конце каждого блока добавить:

```
printf("Для продолжения нажмите Enter");
getchar();
```

В этом примере функция getchar() приостанавливает выполнение программы до тех пор, пока пользователь не нажмет клавишу (рис.5.4). Совершенно необязательно этой клавишей должна быть именно Enter, но нажатие любой другой приведет к появлению на экране соответствующего символа, а это может запутать пользователя. Функция getchar() в данном случае не связана ни с какой переменной. Если для продолжения выполнения программы будет нажата, например, клавиша Y, соответствующий символ появится на экране, но не будет присвоен в качестве значения ни одной переменной.

Рис. 5.4. Использование функции getchar() для создания паузы в программе

Оператор получения адреса &

Вы уже знаете, что для каждой переменной, используемой в программе, отводится определенный объем памяти, который зависит от типа переменной (см.главу3). Каждый элемент памяти пронумерован от 0 и далее. Если компьютер имеет 2Мегабайта оперативной памяти, элементы будут пронумерованы от 0 до 2097151, как показано на рис.5.5. Эти номера и называют *адресами элементов памяти*.

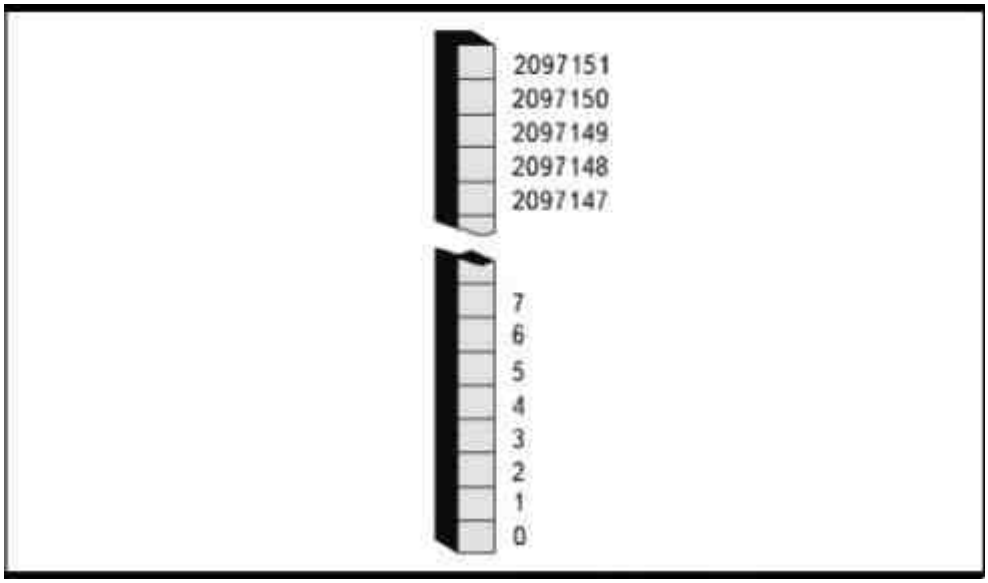

Рис. 5.5. Адреса элементов памяти

Когда вы определяете переменные, Си резервирует память для хранения их значений. Например, для целочисленной переменной с именем count, Си отведет два элемента (байта) памяти. Первый из них определяет адрес переменной.

На рис. 5.6 показана переменная count, которая имеет значение 2341. Это значение хранится в элементах, имеющих номера 21560 и 21561, так что адрес переменной count (не значение!)— это число 21560. В качестве адреса достаточно указать номер первого элемента, поскольку Си знает, сколько памяти отводится для каждого типа данных. Если count относится к типу int, для нее требуется два элемента памяти, и если первый из них имеет номер 21560, то номер последнего— 21561.

Актуальный адрес переменной может изменяться на другом компьютере и даже на том же самом при каждом запуске программы. Адрес 21561 взят просто для примера.

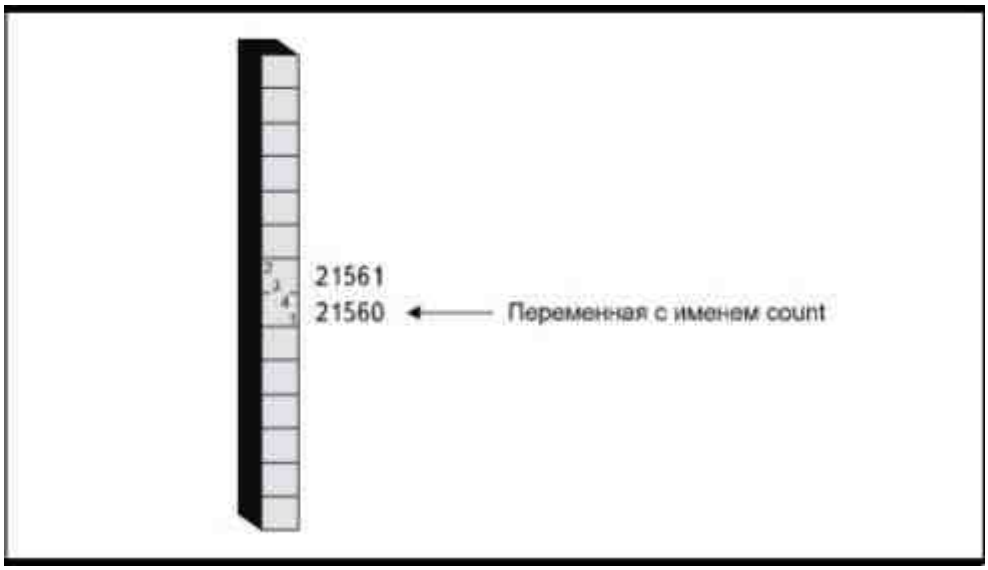

Рис. 5.6. Переменная и ее адрес

При ссылке на переменную (например, при выводе ее значения на экран) используется имя переменной. Можно также сослаться и на ее адрес, если поставить символ амперсанда (&) перед именем переменной, скажем, так: &count. Символ & называется *оператором получения адреса*. Он указывает Си, что вас в настоящий момент интересует адрес элемента памяти, где зарезервировано место для переменной, а не значение переменной, хранящееся в этом элементе. Поэтому, хотя переменная count имеет значение 2341, &count имеет значение 21560.

Оператор получения адреса не используется со строковыми переменными. Строки определяются с помощью специального класса переменных, называемого массив, о котором мы будем говорить в главе 10.

Оператор получения адреса используется при вводе данных с помощью функции scanf().

Функция scanf()

Функция scanf() является многоцелевой функцией, дающей возможность вводить в компьютер данные любых типов. Название функции отражает ее назначение— функция сканирует (просматривает) клавиатуру, определяет, какие клавиши нажаты, и затем интерпретирует ввод, основываясь на указателях формата (SCAN Formatted characters). Так же, как и функция printf(), scanf() может иметь несколько аргументов, позволяя тем самым вводить значения числовых, символьных и строковых переменных в одно и то же время.

Си++ работает со всеми функциями ввода языка Си, но имеет свое дополнительное средство ввода— стандартный поток ввода cin. Ввод в Си++ описан дальше в этой же главе.

Так же, как список параметров printf(), список параметров функции scanf() состоит из двух частей: строки формата и списка данных (рис.5.7). Строка формата содержит указатели формата, здесь они носят название *преобразователей символов**, которые определяют то, каким образом должны быть интерпретированы вводимые данные. Список данных содержит переменные, в которые должны быть занесены вводимые значения.

Рис. 5.7. Список параметров функции scanf() делится на две части

Указатели формата аналогичны тем, которые используются функцией printf():

%d целые числа

%u беззнаковые целые числа

%f вещественные числа, float

%e вещественные числа в экспоненциальной форме

%g вещественные числа в наиболее коротком из форматов %f или %e

%c символы

%s строки

%o целые числа в восьмеричной системе счисления

%x целые числа в шестнадцатеричной системе счисления

Вводя числовые или символьные данные, следует указывать в списке данных функции scanf() адрес переменной, а не просто ее имя:

```
main()
        {
        float amount;
        scanf("%f", &amount);
        }
```

*** В оригинале conversion characters. *(Прим.перев.)***

В этом примере функция scanf() вводит число с плавающей точкой и вносит его в область памяти, зарезервированную для переменной amount. Как только число помещается в эту область памяти, оно автоматически становится значением переменной.

На время работы функции scanf(), выполнение программы приостанавливается, и программа ожидает ввода данных. Ввод заканчивается нажатием клавиши **Enter**.

Принцип работы с данными функции scanf() в корне отличается от работы функций gets() и getchar(). Для того чтобы понять, что именно происходит при вводе с помощью scanf(), необходимо детально рассмотреть эти отличия.

Входной поток

Когда данные вводятся при помощи функции gets(), все символы, которые были набраны на клавиатуре до нажатия **Enter**, становятся значением переменной. Когда символ вводится с помощью функции getchar(), нажатие клавиши автоматически приводит к присвоению соответствующего символа переменной.

Функция scanf() работает по-другому. Вместо того чтобы просто взять данные и присвоить их переменной, scanf() прежде всего с помощью указателей формата определяет, каким образом следует трактовать введенные символы.

Принято говорить, что scanf() получает данные из *входного потока*. Входным потоком называется последовательность символов, поступающих из некоторого источника. В случае функции scanf() источником служит клавиатура. После нажатия клавиши **Enter** все данные, которые были введены к этому времени, передаются функции scanf() в виде пока еще бессмысленного набора символов, в том же порядке, в каком их набирали. Затем scanf() определяет, какие символы соответствуют типу, заданному указателем формата, а какие следует игнорировать. Указатели формата называют преобразователями символов, так как они берут исходные символы из входного потока и преобразуют их в данные, относящиеся к определенному типу. Рис.5.8 иллюстрирует этот процесс.

Функция scanf() игнорирует не содержащие информации знаки: пробелы, символы табуляции, символы новой строки, кроме тех случаев, когда текущий тип данных определен как char. Рассмотрим программу:

```
main()
        {
        int count;
        puts("Пожалуйста, введите число: ");
        scanf("%d", &count);
        printf("Число равно %d", count);
        }
```

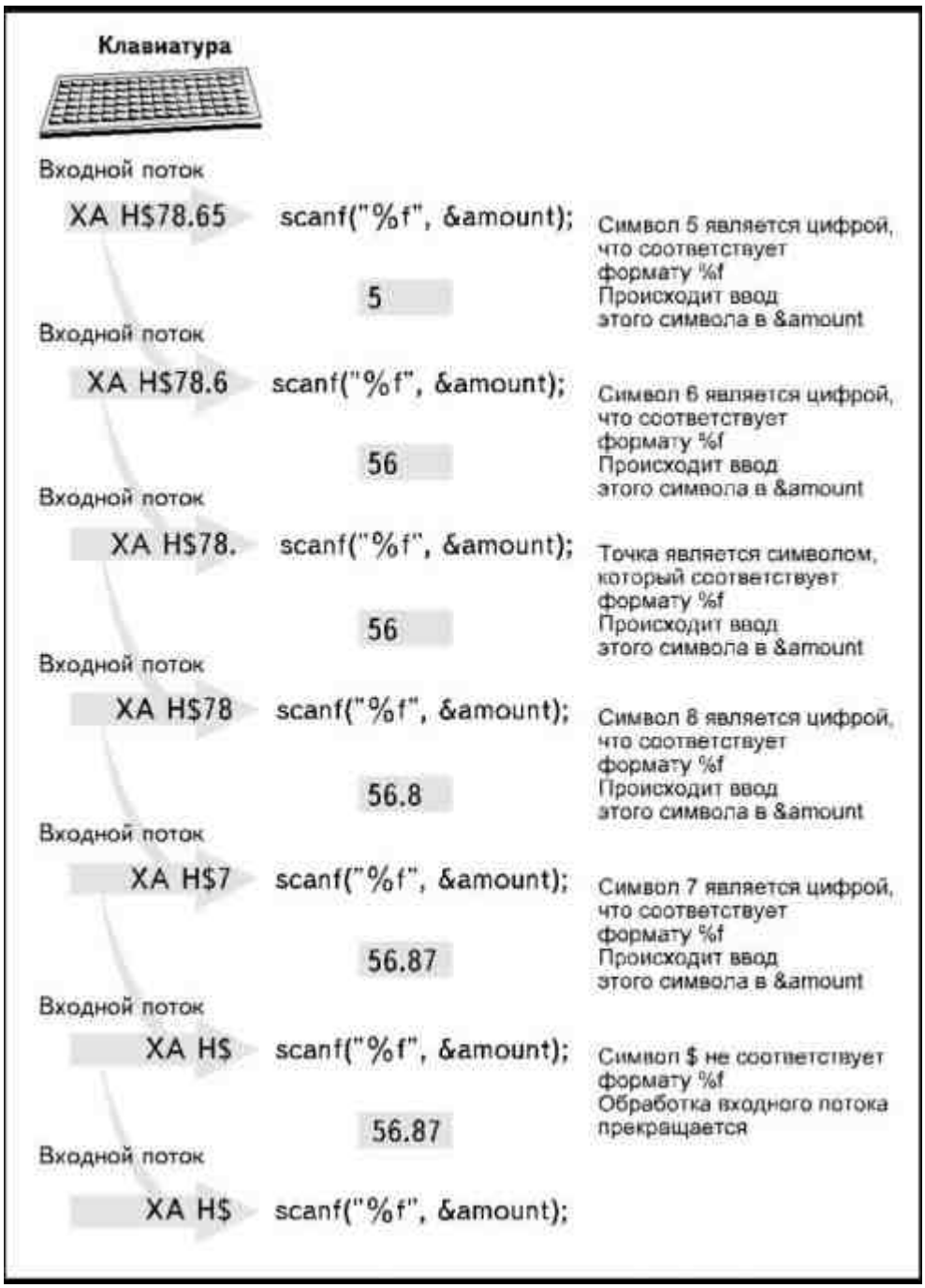

Рис. 5.8. Функция scanf() читает входной поток и определяет,какие данные следует ввести, а какие— игнорировать

Перед тем как ввести число, вы можете нажимать на клавишу пробела столько, сколько хотите— Си будет игнорировать пробелы, в ожидании первого значимого символа. Затем Си попытается преобразовать символы в соответствии с указателями формата в строке формата функции scanf(). Если эти символы соответствуют формату (в данном случае— если это цифры), они будут внесены в переменную. Ввод данных прекратится, если встретится символ, формат которого не соответствует ожидаемому, то есть он не является цифрой. Например, если набрать на клавиатуре последовательность «123abc», то число 123 в нашем примере будет присвоено переменной count, а буквы «abc»— проигнорированы, как это показано на рис.5.9. Остановку ввода данных может вызвать пробел. Например, если напечатать «123», то переменной будет присвоено значение 12, а число 3— проигнорировано.

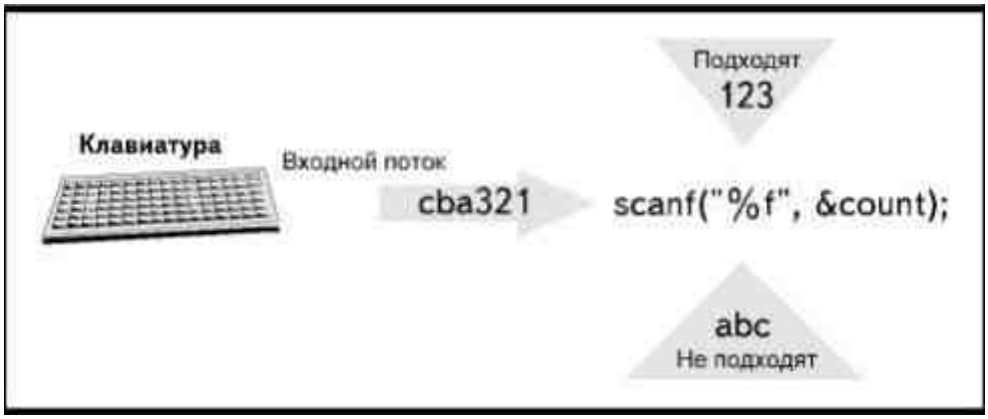

Рис. 5.9. Функция scanf() прекращает ввод данных,встретив первый не цифровой символ

Первый значимый символ должен соответствовать указанному в аргументе функции scanf() формату. Так, если напечатать последовательность «ABC123», программа проигнорирует ее всю целиком, и вы останетесь в неведении относительно значения переменной.

Какие символы программа расценивает как «подходящие», зависит от указателей формата. Если стоит указатель %d, то «подходящими» являются только цифры и знак «минус». Если поставить указатель %x, то соответствующими формату окажутся символы 0123456789ABCDE, так как все они используются при записи чисел в шестнадцатеричной системе счисления. Если же стоит указатель %c, принимаются любые символы, даже пробел внутри входного потока функция scanf() в этом случае не игнорирует. Если написать инструкцию:

```
char letter;

scanf("%c", &letter);
```

и нажать клавишу пробела в начале последовательности значимых символов, scanf() присвоит переменной значение пробел, игнорируя последующие символы. Поэтому, имея дело с типом char, нельзя помещать пробелы перед другими символами.

Вводя строку, функция scanf() начнет присваивание значения с первого значимого символа, игнорируя пробелы впереди, и остановит присваивание, встретив первый пробел среди значимых символов. Взгляните на программу:

```
main()
        {
        char name[25];
        puts("Пожалуйста, введите Ваше имя: ");
        scanf("%s", name);
        puts(name);
        }
```

Обратите внимание, оператор получения адреса не используется с именем строковой переменной. Если вы наберете на клавиатуре «Нэнси» и нажмете Enter, эти символы будут присвоены переменной name. Даже если набрать «Нэнси Чезин», scanf() начнет присвоение символов с первого значимого символа и остановит, встретив первый пробел, так что значением переменной все равно станет только имя Нэнси, а остальное программа проигнорирует (рис.5.10).

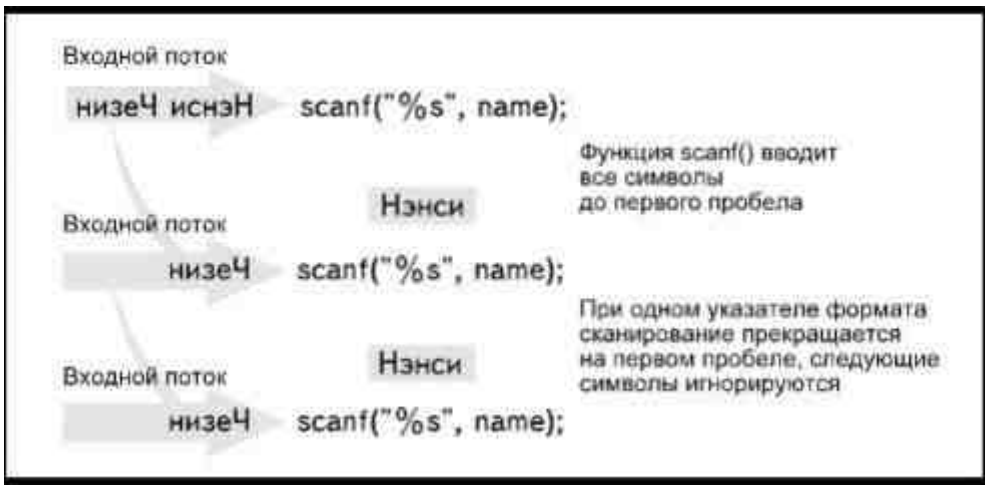

Рис. 5.10. Функция scanf() прекращает чтение символов в строкепри появлении первого пробела

Из-за этой особенности функции scanf(), она является не слишком удобной для ввода строк. В этих случаях лучше использовать функцию gets().

Использование функции scanf()

Давайте познакомимся с функцией scanf() поближе. Рассмотрим следующую программу:

```
main()
 {
 int count;
 printf("Пожалуйста, введите целое
 число и нажмите Enter: ");
 scanf("%d", &count);
 printf("Подтвердите ввод числа: %d", count);
 }
```

Что произойдет во время работы программы?

1. Функция printf() выведет на экран подсказку:

   Пожалуйста, введите целое число и нажмите Enter:

   Код «новая строка» отсутствует, поэтому курсор остановится в конце строки подсказки.
2. Программа ждет, пока пользователь введет информацию.
3. Вы набираете число и нажимаете **Enter**.
4. Функция scanf() просматривает входной поток и определяет, какие символы можно ввести, а какие следует игнорировать. Если цифровые символы предшествуют нецифровым, она продолжает сканирование до тех пор, пока не встретится нецифровой символ или пробел. Цифровые символы преобразуются в число и заносятся по адресу переменной. Если нецифровые символы предшествуют цифровым, функция прекращает работу и вся напечатанная последовательность игнорируется.
5. Курсор переходит к началу следующей строки.
6. Функция printf() выводит на экран значение, присвоенное переменной, сопровождая его сообщением: «Подтвердите ввод числа».

Вы можете использовать столько функций scanf(), сколько необходимо для ввода данных. В качестве примера Листинг 5.1 демонстрирует текст программы, в которой вводятся переменные четырех типов: int, float, char и строка. Обратите внимание, что оператор получения адреса используется для всех переменных, кроме строки.

Листинг 5.1. Программа, в которой вводятся четыре переменные. /*scanf3.c*/

```
main()
        {
        int count;
        float amount;
```

```
char letter;
char name[15];
puts("Введите целое число и нажмите Enter");
scanf("%d", &count);
puts("Введите вещественное
число и нажмите Enter");
scanf("%f", &amount);
puts("Введите символ и нажмите Enter");
scanf("%c", &letter);
puts("Введите строку и нажмите Enter");
scanf("%s", name);
printf("%d %6.2f %c %s", count,
amount, letter, name);
}
```

Функцию scanf() можно использовать для ввода нескольких переменных в одной инструкции. Однако если пользователь не будет вводить данные точно в том порядке, в каком их ожидает функция, она может проигнорировать всю введенную информацию. Пока вы еще только изучаете Си, старайтесь избегать нескольких аргументов при вызове функции scanf(), лучше используйте отдельные инструкции для каждого элемента данных.

Разрешается определить символьную переменную как int, но для ввода ее значения следует использовать указатель %c, и ни в коем случае не %d. Указатель %d можно использовать, чтобы *отобразить* ASCII-код какого-либо символа с помощью функции printf().

Выбор соответствующих средств ввода данных

Раньше уже говорилось, что особенности работы функции scanf() делают весьма вероятной ситуацию, когда, из-за несоответствия форматов, ввод игнорируется. Поэтому, используя scanf(), очень важно давать пользователю четкие инструкции перед каждой процедурой ввода.

Допустим, вы пишете программу, где надо ввести данные об объеме продаж:

```
main()
{
float amount;
puts("Введите сумму продаж: ");
scanf("%f", &amount);
printf("\nСумма составляет %f", amount);
}
```

Так как суммы обычно указывают в долларах, пользователь может, вводя данные в эту программу, поставить знак доллара: $45.65. Этот знак, естественно, включается во входной поток. Но в данном случае он не соответствует формату, ожидаемому функцией scanf(), а так как именно этот знак стоит на первом месте во входном потоке, scanf() проигнорирует весь ввод целиком. Не самый лучший способ вести дела!

Именно для предотвращения подобных казусов и следует вводить специальные инструкции перед каждой функцией scanf():

```
puts("Пожалуйста, введите сумму продаж. Вводите\n");
puts("только сумму. Не используйте знак доллара или\n");
puts("запятую между цифрами. Правильный
 формат: 4567.87\n");
puts("Неправильный формат: $4567,87\n");
```

puts(" Спасибо\n");

Однако существует возможность ввести набор данных, даже если он включает символы, которые не должны быть присвоены переменной. Помните, когда мы рассматривали функцию printf(), то говорили, что строка формата может содержать не только указатели формата. В строку формата функции printf() может включаться текст, который также будет отображен на экране, причем на место указателей формата, будут подставлены соответствующие значения переменных. В строку формата функции scanf() тоже можно ввести текст. Положим, вы написали следующую функцию scanf():

scanf("$%f", &amount);

Знак $ в данном случае говорит программе, что во входном потоке первым значимым символом должен быть знак доллара, так что, когда пользователь вводит число $45.65, программа ожидает этого, и функция scanf() не прекращает работу, а просто отбрасывает знак доллара и заносит следующее числовое значение в адрес переменной amount.

Следует помнить, что любой текст, внесенный в строку формата, *должен быть* непременно введен и во входной поток, причем строго на том же месте по отношению к указателям формата. Если символы не появятся в ожидаемом месте, все введенные в этом пункте данные будут проигнорированы. Например, когда знак $ внесен в строку формата, ввод игнорируется, если этот символ отсутствует в начале входного потока. Если вы используете какие-нибудь литералы в строке формата, позаботьтесь о том, чтобы пользователь был поставлен в известность относительно того, что именно он должен ввести:

puts("Пожалуйста, введите сумму продаж. Начните\n");

puts("число со знака доллара.

Правильный формат: $4567.87\n");

puts("Не используйте запятую внутри числа.\n");

Знак доллара не вводится в значение переменной и не выводится на экран, когда вы используете функцию printf(). Он появляется на экране только в момент ввода данных, так же, как и остальные символы, в режиме эха. Если вы хотите, чтобы знак доллара отображался вместе со значением переменной в дальнейшем, следует задать его в строке формата функции printf():

printf("Сумма продаж составляет $%f", amount);

но теперь это будет уже другой символ $, выведенный на дисплей с помощью функции printf(). Он появится перед значением переменной, которое будет подставлено на место указателя формата %f.

Будьте осторожны при использовании scanf()

Во многих системах символы, входящие во входной поток, помещаются в область памяти, называемую *буфером*. Если функция scanf() прерывает работу преждевременно, символы не будут внесены в значение переменной, а останутся в буфере. Существует опасность, что при следующей процедуре ввода данных функция начнет чтение символов с тех, которые уже содержатся в буфере, а не с тех, которые вводятся в ответ на новый запрос.

Поэтому функцию scanf() следует использовать с осторожностью. Лучше всего вводить только один набор данных с одной функцией и использовать подсказки, четко объясняющие пользователю, какой именно ввод от него ожидается.

Ввод в Си++

Компиляторы языка Си++ поддерживают функции gets(), getchar() и scanf(), о которых мы говорили в этой главе. Кроме того, Си++ имеет собственное многоцелевое средство ввода для всех типов данных. Стандартный поток ввода cin в сочетании с двумя символами «больше» (>>), которые называются *оператором извлечения**, служит для считывания данных с клавиатуры. Программа

int count;

cin >> count;

вносит целочисленные данные с клавиатуры в значение переменной count. Стандартный поток ввода cin не требует указания адреса переменной для числовых и символьных данных, указывается только имя переменной.

При использовании cin нет необходимости определять формат с помощью указателей формата. Поток ввода cin имеет возможность определять тип данных самостоятельно на основании вводимой информации. Это свойство называется перегрузкой. Из-за этой особенности оператора cin большинство программистов предпочитают использовать именно его, а не функцию scanf().

При работе с несколькими аргументами необходимо отделять каждое имя переменной оператором извлечения:

*** В оригинале extraction operator. *(Прим.перев.)***

Рис. 5.11. Оператор вставки и оператор извлечения

cin >> amount >> count >> age >> name;

Возможно, первое время вы будете испытывать затруднения в использовании операторов извлечения >> и вставки <<. На рис.5.11 показан простой способ запоминания этих операторов, изображенных в виде стрелок, которые указывают направление вперед или назад относительно переменной из списка данных.

Оператор извлечения >> изображен в виде стрелки, направленной в сторону переменной. Это означает, что данные поступают к ней в виде ввода. Когда вы применяете cin для ввода данных, пользуйтесь оператором, который как бы указывает направление к переменной.

Оператор вставки << изображен в виде стрелки, направленной от переменной. Это значит, что данные выводятся из переменной на экран. Со стандартным потоком вывода cout используйте, соответственно, тот оператор, который как бы указывает направление от переменной.

Неинициализированные переменные

Может получиться так, что переменной, которая определена в тексте программы, так никогда и не присваивается никакое значение. Это может произойти по причине того, что функция scanf() прервала работу, либо потому, что вы забыли написать инструкцию ввода значения для этой переменной. Могут быть и другие причины. Если переменную, которой не присвоено значение, использовать в инструкциях вывода данных, компилятор может не сообщить об ошибке, но отображенная на дисплее информация будет лишена смысла. Почему это происходит?

При включении компьютера области памяти, которые не используются операционной системой, заполнены случайными данными. Когда Си резервирует память для переменных при их определении, то содержимое этих областей памяти не меняется до тех пор, пока переменная не инициализирована или пока ей не присвоены введенные данные. Поэтому, если в программе в качестве параметра функции вывода используется переменная, которой не было присвоено к этому моменту никакого значения, на экране появятся те случайные данные, что все еще хранятся в памяти. Но это еще не самое худшее. Если подобная переменная используется при расчетах, компилятор Си тоже не сообщит об ошибке, но при этом все полученные результаты будут лишены смысла, а поскольку вы не были предупреждены о том, что произошло, вы можете принять их как вполне достоверные.

Чтобы избежать неприятностей, многие программисты присваивают начальные значения всем переменным. Можно присвоить ноль в качестве начального значения числовой переменной и пробел символьной и строковой переменным:

int count = 0;

char initial = ' ';

float rate = 0.0;

Разумеется, логика программы может потребовать присваивания других начальных значений.

Поступив подобным образом, можно, как минимум, избежать появления на экране бессмысленных символов, если с функцией вывода будет использована переменная, которой не присвоено никакого значения. Позже вы узнаете, как протестировать список переменных, чтобы убедиться, что они имеют правильные значения. Помните: *чистый вход— чистый выход*.

В табл. 5.1 собраны все средства ввода данных, о которых говорилось в этой главе.

Таблица 5.1. Основные средства ввода данных в языке Си/Си++.

Функция	Тип данных	Комментарии
gets()	Только строки	Си и Си++. Используется для ввода строк, возможно наличие пробелов. После ввода данных необходимо нажать Enter.
getchar()	Единичные символы	Си и Си++. Используется для ввода единичных символов в переменную, определенную как int или char. После ввода символа не надо нажимать Enter. Используется без параметров в качестве переменной (например, letter= getchar()) или сама по себе для временной остановки программы.
scanf()	Все типы данных	Си и Си++. Требует указателей формата для каждого элемента вводимых данных. Не используется для ввода строк, имеющих пробелы. Необходимо следить за правильностью формата вводимых данных. Может иметь несколько аргументов. Перед переменной типа char нельзя ставить пробел.
cin	Все типы данных	Только Си++. Не нуждается в указателях формата и операторе получения адреса. Может вводить значения для нескольких аргументов. Аргументы отделяются операторами >>.

Большинство интерпретаторов (например, языка BASIC) и многие компиляторы автоматически присваивают числовым переменным ноль в качестве начального значения.

Используемые алгоритмы ввода

Алгоритмом называют способ выполнения определенной задачи. Изучая программирование, вы в то же время учитесь разрабатывать алгоритмы, то есть определять, каким именно образом решить ту или иную задачу, используя язык Си. Может показаться, что существует бесчисленное множество способов решения, столько же, сколько и задач, но на самом деле есть основной набор алгоритмов, которые используются в 90 процентах случаев. Изучив наиболее часто используемые алгоритмы, вы сможете легко проектировать программы, основываясь на уже известных вам методах.

Один из таких алгоритмов используется для изменения значения переменной. Речь идет о присваивании нового значения переменной, которая уже инициализирована. Если использовать просто функцию ввода и имя переменной в качестве параметра, исходное значение при вводе нового будет потеряно. А что делать, если вы не хотите, чтобы оно пропало? Например, вам нужно сравнить старое и новое значения (соответствующие примеры приведены в главе8). Один из возможных способов— это ввести дополнительную переменную для сохранения прежнего значения:

```
cache = amount;
```

Здесь значение переменной amount передается в переменную cache, и до тех пор, пока в amount не записана новая информация, обе переменные имеют одну и ту же величину. Переменная cache обеспечивает хранение значения в памяти, пока в нем не возникнет необходимость. Когда оно вам потребуется, используйте переменную, которая служила для хранения.

В программе, приведенной в Листинге 5.2, показан пример присваивания переменной нового значения (в следующих главах вы увидите различные варианты практического применения этой процедуры).

Листинг 5.2. Программа присваивания переменной нового значения с сохранением старого.

```
/*storage.c*/
main()
        {
```

```
int number, storage;
puts("Введите значение переменной");
scanf("%d", &number);
storage = number;
puts("Введите значение переменной");
scanf("%d", &number);
printf("Исходное значение: %d\n", storage);
printf("Новое значение: %d", number);
}
```

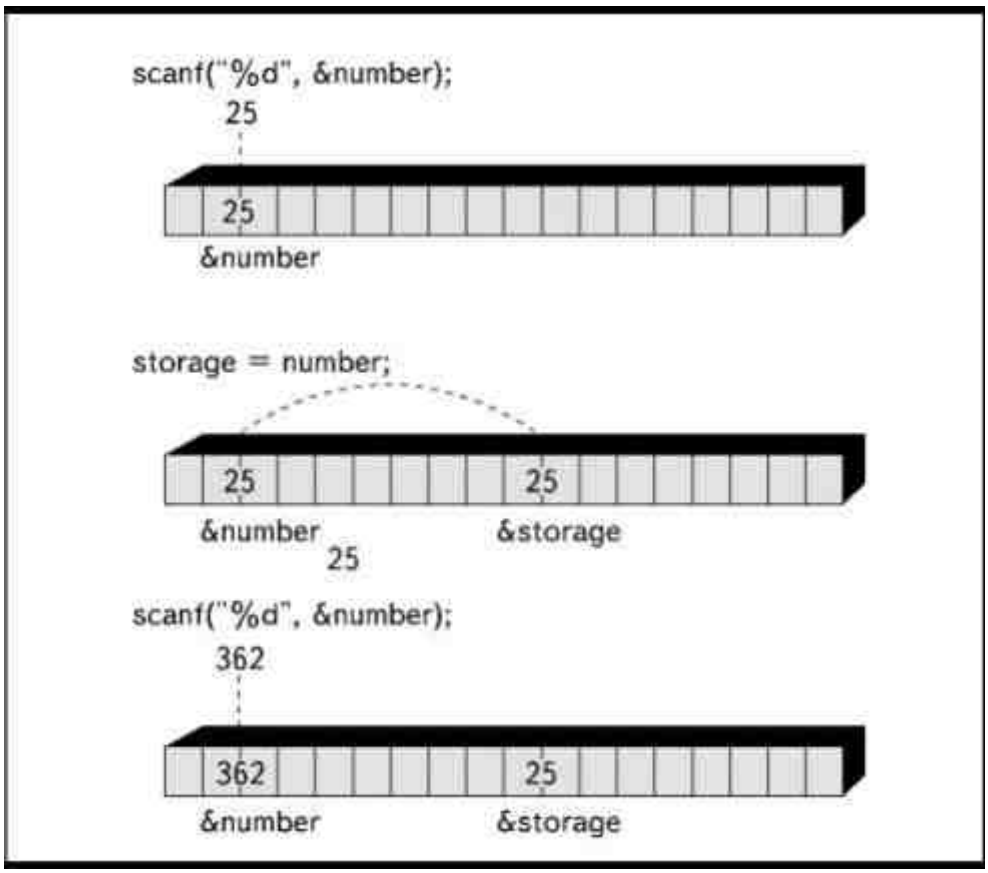

Рис. 5.12. Присваивание переменной нового значения

Ввод значения переменной number обеспечивается следующими инструкциями:

```
puts("Введите значение переменной");
scanf("%d", &number);
```

Сохранение значения переменной number в переменной storage выполняет инструкция:

```
storage = number;
```

Для ввода второго значения переменной number используются точно такие же инструкции, как и в первом случае. Оба значения, старое и новое, позже могут быть отражены на экране. Прежнее содержимое переменной не было утеряно просто потому, что было сохранено в другой области памяти, как показано на рис.5.12.

Вопросы

1. В чем смысл выражения «чистый вход— чистый выход»?
2. Почему после ввода с использованием функции getchar() нет необходимости нажимать клавишу Enter?

3. Назовите два способа использования функции getchar().
4. Что такое оператор получения адреса?
5. Какие преимущества дает использование функции scanf()?
6. Что такое преобразователи символов?
7. Почему для ввода строки лучше использовать функцию gets(), а не scanf()?
8. В чем преимущества и недостатки форматированного ввода при помощи scanf()?
9. Является ли необходимым присваивание всем переменным начальных значений?
10. Какие трудности могут возникнуть при использовании функции scanf() для ввода значения переменной типа char?

Упражнения

1. Напишите программу, в которой вводится, а затем отображается на экране монитора в одной строке ваше имя и номер телефона.
2. Напишите программу, в которой вводится число, а затем на экран выводится адрес области памяти, куда было записано это число.
3. Напишите программу, в которой вводятся три числа, а затем эти числа отображаются на экране в порядке, обратном тому, в котором их вводили.
4. Напишите программу, в которой используются функции getchar(), gets() и scanf().
5. Объясните, почему следующая программа написана неправильно:

```
6.    main()
7.    {
8.    char initial;
9.    initial = gets();
10.   puts(initial);
      }
```

ГЛАВА 6

ОПЕРАТОРЫ

Не возникло ли у вас впечатления, что кое-что пропущено? В пятой главе вы узнали, как данные вводятся в компьютер, а в четвертой— как информация выводится на дисплей. Вероятно, пора рассказать о том, что происходит в промежутке между этими двумя событиями— об обработке данных.
В процессе *обработки* программа превращает *данные*, которые мы вводили в компьютер, в *информацию*, которую компьютер представляет нам. Различие между данными и информацией трудноуловимо, но очень существенно. «Данные»— это, так сказать, исходный материал, символы и числа, которые не могут быть использованы как конечный продукт. «Информация»— это конечный продукт, ради получения которого и была написана программа.
Превращение данных в информацию может осуществляться различными способами. Для чисел, например, обработка зачастую включает некоторые математические операции. Скажем, вам надо умножить стоимость заказанных товаров на ставку налога, а затем добавить полученную сумму налога к сумме заказа. Эти две операции показаны на рис.6.1, они могут быть описаны как:
налог на продажи = стоимость заказа Ф ставка налога
общая сумма заказа = стоимость заказа + сумма налога

Рис. 6.1. Процесс вычисления общей суммы заказа

Для того чтобы выполнить вычисления, превращающие данные в информацию, необходимы *операторы*. Оператором называется символ, который говорит компьютеру, как следует обрабатывать данные. В этой главе вы познакомитесь с *арифметическими операторами, операторами приращения и операторами присваивания.*

Замечания по Си++

Си и Си++ используют математические операторы сходным образом. Однако благодаря *перегрузке*, Си++ может использовать определенный оператор с различными типами данных. В главе 10 вы узнаете, как использовать оператор + в Си++ для соединения двух строк.

Арифметические операторы

При математических вычислениях используются следующие арифметические операторы:

Оператор	Функция оператора
+	сложение
–	вычитание
*	умножение
/	деление
%	получение остатка от деления нацело

Обратите внимание, что процедура умножения обозначается символом «звездочка» (а не символом «x»), а процедура деления— символом «прямая косая черта» (а не обратная «\»).
Операторы часто используются в инструкциях, требующих выполнения математических операций и присваивания полученного результата переменной в качестве значения. Ниже приводится пример расчета

накладных расходов с помощью арифметических операторов. В этом примере не имеет смысла присваивать переменной точное значение в виде литерала, например:

```
sales_tax = 4500;
```

напротив, величину накладных расходов требуется вычислить. При записи математических операций имя переменной всегда помещают слева от знака «равно», а арифметические операторы— справа:

```
sales_tax = amount * tax_rate;
price = cost + shipping + insurance;
per_unit = total / count;
```

Эти инструкции говорят компилятору, что следует выполнить три операции:

- присвоить переменной sales_tax значение, полученное в результате умножения значения переменной amount на значение переменной tax_rate;
- присвоить переменной price значение, полученное из суммы значений трех переменных: cost, shipping и insurance;
- присвоить переменной per_unit значение, полученное в результате деления total на count.

Компьютер выполнит математические операции, указанные в правой части, а потом присвоит полученное значение переменной в левой части. Что касается правой части выражения, то она может содержать любые комбинации переменных, констант и литералов, например:

```
sales_tax = amount * 0.06;
price = 56.90 + shipping + 7.87;
per_unit = 156.65 / 12.50;
```

Листинг 6.1 демонстрирует использование операторов в программе. В приведенном фрагменте необходимо ввести имя и адрес покупателя, а также стоимость его заказа. Затем программа рассчитывает стоимость транспортировки (10процентов от суммы заказа) и налог на продажи (6 процентов от суммы заказа). Наконец, весь список накладных расходов отображается на экране с помощью функций puts() и printf().

Листинг 6.1. Программа расчета и отображения накладной.

```
/*invoice.c*/
#define TAX_RATE 0.06
#define SHIPPING 0.10
main()
{
char name[15], address[20], city[15],
 state[3], zip[6];
float order, total, tax, ship;
printf("Имя покупателя: ");
gets(name);
printf("Адрес: ");
gets(address);
printf("Город: ");
gets(city);
printf("Штат: ");
gets(state);
printf("Индекс:");
gets(zip);
printf("Сумма заказа: ");
scanf("%f", &order);
tax = order * TAX_RATE;
ship = order * SHIPPING;
total = order + tax + ship;
puts("\n\n\n\n");
puts(" НАКЛАДНАЯ\n");
printf("%s\n%s\n%s, %s %s\n", name,
 adress, city, state, zip);
printf("\t\t\t\t%-10s\t%10.2f\n", "Заказ:", order);
printf("\t\t\t\t%-10s\t%10.2f\n", "Налог:", tax);
printf("\t\t\t\t%-10s\t%10.2f\n",
 "Транспортировка:", ship);
printf("\t\t\t\t\t\t_______________\n);
printf("\t\t\t\t%-10s\t%10.2f","Всего: ",total);
}
```

Деление нацело

Оператор % используется для расчета остатка от деления нацело. Если вы используете оператор деления (/) для целочисленных данных, то результат деления тоже всегда будет целым числом. Например, при делении 12 на 5 (12/5) вы получите 2, а не 2.4. Дробная часть, равная в данном случае 0.4, при делении целых чисел отбрасывается.
Разумеется, нередко возникает необходимость узнать значение остатка от деления. Пока мы имеем дело с целыми числами, мы не можем использовать значение 0.4, так как это число относится к типу float. Мы же определили результат деления как целое. В этом случае получается, что число 12 состоит из двух чисел 5, а лишняя двойка просто игнорируется. Число 2 в данном случае является остатком от деления нацело, для получения которого и используется оператор %. Остаток от деления нацело также всегда является целым числом.
В качестве примера приведен Листинг 6.2. В этой программе подсчитывается количество банкнот достоинством в двадцать, десять, пять и один доллар, необходимых для уплаты определенной суммы. Наиболее существенным, на что следует обратить внимание в этом примере, является алгоритм, то есть способ использования операции деления нацело при выполнении задачи, решение которой в другом случае потребовало бы произведения более сложных расчетов. Как и все прочие алгоритмы, данный алгоритм выглядит очень простым, стоит только понять его основную идею.

Листинг 6.2. Программа, в которой используется оператор получения остатка от деления нацело.

```
/*change.c*/
main()
{
int amount, twenties, tens, fives, ones, r20, r10;
printf("Введите необходимую сумму: ");
scanf("%d", &amount);

twenties = amount / 20;
r20 = amount % 20; /*r20 - остаток от деления на двадцать*/
tens = r20 / 10;
r10 = r20 % 10;          /*r10 - остаток от деления на десять*/
fives = r10 / 5;
ones = r10 % 5;
putchar('\n');
printf("Для того чтобы дать %d \
 долларов сдачи, используйте:", amount);
printf("%d банкнот(ы) достоинством 20 долларов", twenties);
printf("%d банкнот(ы) достоинством 10 долларов", tens);
printf("%d банкнот(ы) достоинством 5 долларов", fives);
printf("%d банкнот(ы) достоинством 1 доллар", ones);
}
```

Если вы ввели, например, значение 57, то получите следующий результат:

```
Для того чтобы дать 57 долларов сдачи, используйте:
2 банкнот(ы) достоинством 20 долларов
1 банкнот(ы) достоинством 10 долларов
1 банкнот(ы) достоинством 5 долларов
2 банкнот(ы) достоинством 1 доллар
```

Рис. 6.2 иллюстрирует работу этой программы. Количество двадцатидолларовых банкнот рассчитывается с помощью инструкции twenties = amount/20. Так как и переменная amount, и переменная twenties относятся к типу int, результат деления окажется целым числом. Он показывает, сколько раз число 20 содержится в значении переменной amount. Для компьютера результат— это просто некая величина, которую следует занести в память. Для нас же значение переменной twenties представляет значимую информацию, так как сообщает, сколько надо взять банкнот достоинством 20 долларов, чтобы дать сдачу.

$57		
twenties = amount / 20;	57/20=2	2 банкноты по 20 долларов
r20 = amount % 20;	57%20=17	Остаток от деления на 20 равен 17
tens = r20 / 10;	17/10=1	1 банкнота по 10 долларов
r10 = r20 % 10;	17%10=7	Остаток от деления на 10 равен 7
fives = r10 / 5;	7/5=1	1 банкнота по 5 долларов
ones = r10 % 5;	7%5=2	Остаток от деления на 5 равен 2 раза по 1 доллару

Рис. 6.2. Как работает программа, использующаяоператор остатка от деления нацело

Теперь, когда мы знаем, что нужны две банкноты по 20 долларов, встает вопрос: как определить нужное количество десятидолларовых купюр? Вспомните, как поступают в таких случаях в реальной жизни. После того как мы отняли от общей суммы два раза по 20 долларов, надо взять сумму остатка и посмотреть, сколько банкнот по 10 долларов могло бы в нее войти. Программа поступает аналогичным образом. Для того чтобы определить размер остатка, используется оператор %. В инструкции r20 = amount % 20 переменной r20 присваивается значение, полученное в остатке после деления переменной amount на 20. Для нас это число означает сумму, оставшуюся после того, как из общей суммы вычли столько раз по 20, сколько это было возможно. Два раза по 20 долларов составляет сумму 40 долларов, вычитаем ее из общей суммы и получаем 17 долларов. Значение 17 присваивается переменной tens.

Та же процедура, которая описана для переменной twenties, повторяется для переменных tens и fives. Значение переменной ones равно остатку после деления на 5.

Типы данных и операторы

Как правило, при выполнении математических расчетов по обе стороны от знака равенства используют данные одного типа. Например, если складывают два числа типа float, тип переменной, которой присваивают результат, тоже должен быть определен как float. Это показано в следующем примере:

```
main()
{
float cost, shipping, total;
cost = 56.09;
shipping = 4.98;
total = cost + shipping;
printf("Общая стоимость
 составляет сумму %.2f \
 долларов", total);
}
```

Результат работы программы, представленный на экране монитора, выглядит так:

Общая стоимость составляет сумму 61.07 долларов

Можно использовать и данные разных типов в правой и левой частях выражения. Отображаемое на экране значение будет определяться в зависимости от типа переменной в левой части выражения. Для примера приведен слегка измененный вариант той же программы:

```
main()
{
int total;
float cost, shipping;
cost = 56.09;
shipping = 4.98;
total = cost + shipping;
printf("Общая стоимость составляет
 сумму %d долларов", total);
}
```

В операции сложения участвуют две переменные типа float (cost и shipping), но полученный результат присваивается целочисленной переменной total. Если сложить эти числа на калькуляторе, то в результате получим 61.07, но так как переменная total— целочисленная, то и результат будет преобразован в целое число. Использование указателя формата %d задает отображение на экране целого числа 61.

Рис. 6.3. Математические операции выполняются до того, как данные преобразуются к заданному типу переменной

Обратите внимание, вначале выполняется математическое действие, а затем происходит присваивание значения (рис.6.3). Если бы заданные значения преобразовывались в целые числа до их сложения, то результат оказался бы равен 60 (56+4).

Можно складывать два целых числа, определив сумму как float:

```
main()
{
int cost, shipping;
float total;
cost = 56;
shipping = 4;
total = cost + shipping;
printf("Общая стоимость составляет сумму %.2f \
 долларов", total);
}
```

В этом случае, в соответствии с типом переменной total и указателем формата %f, результат сложения будет представлен числом с плавающей точкой. Но вследствие того, что оба слагаемых являлись целыми числами, в десятичной части будут проставлены нули— 60.00.

Данные float и int можно объединять и в правой части уравнения, причем в случае сложения и вычитания полученные результаты будут отображаться в соответствии с заданным типом переменной в левой части. Посмотрите на следующую программу:

```
main()
{
int shipping;
float total, cost;
cost = 56.09;
shipping = 4;
total = cost + shipping;
printf("Общая стоимость составляет сумму \
 %.2f долларов", total);
}
```

Здесь выполняется сложение двух переменных типа int и float, с последующим отображением полученного результата в виде значения типа foat (60.09).

Аналогичные правила соблюдаются и при выполнении деления. Но имейте в виду: если вы хотите, чтобы переменная, содержащая результат деления и определенная как float, имела значимые цифры в десятичной части, необходимо, чтобы хотя бы у одного из участвующих в делении чисел (литералов) также имелась дробная часть. На рис.6.4 продемонстрировано, как одно и то же действие выполняется трижды. Каждый раз результат деления определен как тип float. Однако в первом случае ни одно из чисел не имеет знаков после точки, так что при отображении результата в десятичной части стоят нули.

```
main()
{
   float c;
   c = 12 / 5;
   printf("%f\n", c);
   c = 12.0 / 5;
   printf("%f\n", c);
   c = 12 / 5.0;
   printf("%f\n", c);
}
```

Результат

```
2.000000

2.400000

2.400000
```

Рис. 6.4. Деление литералов

Выражения

Выражение находится справа от знака равенства, и это понятие не включает переменную в левой части. Выражения можно использовать и в инструкциях вывода. Например:

```
main()
        {
        int count;
        count = 5;
        printf("Значение равно %d", count+19);
        }
```

В результате выполнения программы появится сообщение:

Значение равно 24

В этом примере вторым аргументом функции printf() является выражение count+19. При вызове функции значение выражения оценивается (вычисляется) в первую очередь. Результат вычисления выводится на дисплей при помощи указателя формата %d.

При использовании выражений значение переменной не изменяется. В нашем примере, после того как было вычислено и выведено на экран значение выражения count+19, содержимое переменной count по-прежнему осталось равным 5. Помните, что выражение является правой частью уравнения, и рассчитанная таким образом величина не присваивается никакой переменной.

Выражения могут состоять из любой комбинации констант, переменных, константных выражений и операторов:

```
printf("%d", count+number);
printf("%d", 16-4);
printf(%f", amount*TAX_RATE);
```

В общем виде выражения можно использовать в тех же ситуациях, что и переменные. Однако в каждом конкретном случае следует хорошенько подумать, имеет ли смысл использовать выражение, или лучше занести полученный результат в значение переменной. Посмотрите на следующую программу:

```
main()
        {
        float cost, shipping;
        printf("Введите стоимость единицы товара: ");
        scanf("%f", &cost);
        printf("Введите величину транспортных расходов: ");
        scanf("%f", &shipping);
        printf("Общая сумма составляет %f", cost+shipping);
        }
```

Выполнение инструкции

```
printf("Общая сумма составляет %f", cost+shipping);
```

приведет к результату, который мог быть получен и другим способом:

```
total = cost + shipping;
/*total следует определить как тип float*/
printf("Общая сумма составляет %f", total);
```

Обе инструкции выводят на экран монитора общую сумму затрат. В первом случае используется выражение, так что нет необходимости определять переменную, которой будет присвоено полученное значение, и нет необходимости писать уравнение. В процессе работы программы выполняется математическая операция сложения, и итог сразу же отображается на экране с помощью функции printf(). Результат, полу-

ченный таким образом, не заносится в память компьютера и, следовательно, если возникнет необходимость снова вывести на экран сумму затрат, придется опять воспользоваться выражением cost+shipping. Если результат расчетов присваивается переменной в качестве значения, как это сделано во втором примере, необходимо определить переменную и написать уравнение. Зато в этом случае результат сложения хранится в памяти компьютера и к нему можно без труда обратиться снова, используя только имя соответствующей переменной вместо целого выражения.
Если вы предполагаете, что результат какой-либо математической операции может понадобиться не один раз, присвойте его значение какой-либо переменной. Выражения имеет смысл использовать только в том случае, если результат может понадобиться только однажды.

Приоритет операторов и порядок вычислений

Когда компьютер встречает уравнение, он выполняет математические действия не просто слева направо. Прежде всего он просматривает строку и определяет порядок выполнения операций, основываясь на приоритете операторов. *Приоритет операторов* означает, что одни операторы выполняются раньше других независимо от того, в какой последовательности они записаны в уравнении. Приоритет операторов умножения и деления выше, чем приоритет сложения и вычитания, что совпадает с последовательностью выполнения математических действий в уравнении (вспомните школьные примеры на «порядок действий»). Компьютер просматривает уравнение и выполняет все операции умножения и деления в том порядке, в каком они записаны в инструкции, не отдавая предпочтения ни одному из двух действий. Если оператор деления стоит перед оператором умножения, деление будет выполнено первым. Промежуточные результаты помещаются в специальной, временно отведенной для этого области памяти*. После выполнения всех операций умножения и деления, компьютер возвращается к началу выражения и выполняет сложение и вычитание в порядке, соответствующем расположению операторов.
Рассмотрим простое уравнение:
a = 1 + 4 / 2 * 5 + 3;
*** Автор имел в виду машинный стек, но на самом деле современные компиляторы чаще сохраняют промежуточные результаты вычислений на регистрах центрального процессора, а значения выражений, не содержащих имен переменных, вычисляют еще на стадии компиляции. *(Прим.перев.)***

Если выполнять действия просто слева направо, не отдавая предпочтения ни одному из них, полученный результат будет равен 15.5 (рис.6.5). Однако компьютер учитывает порядок выполнения математических операций, и результат поэтому получится равным 14 (на рис.6.6 показано, как вычисляется этот результат). Когда вы пишете программу, следует внимательно следить за тем, выполняются ли математические операции в желаемом порядке.

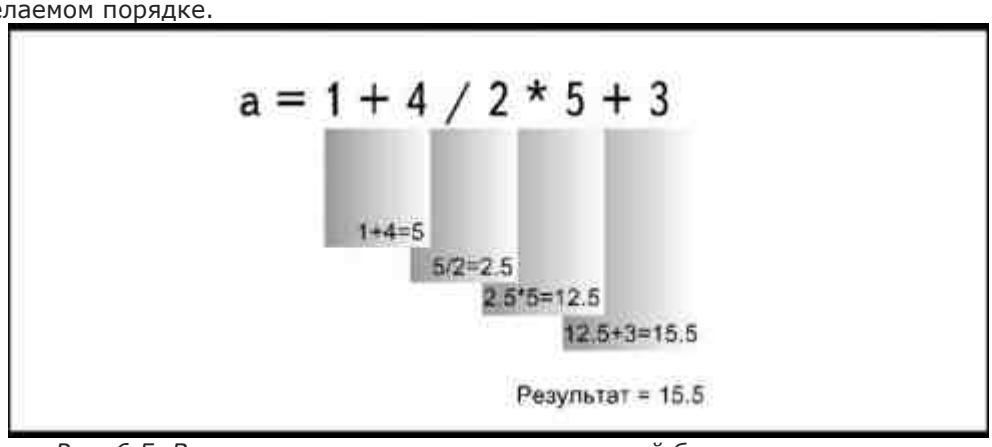

Рис. 6.5. Выполнение математических операций без учета приоритета

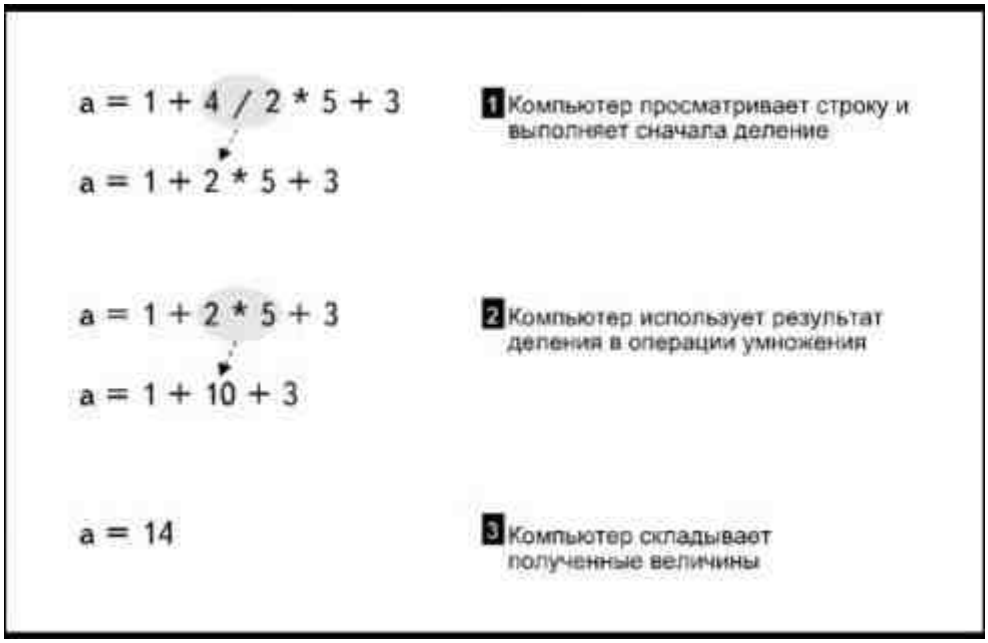

Рис. 6.6. Компьютер выполняет математические операциис учетом приоритета операторов

Очень часто начинающие программисты допускают ошибку при расчетах средних значений. В Листинге 6.3 приведен текст программы, в которой сделана попытка ввести три числа и вычислить их среднее арифметическое. Мы говорим «попытка», так как полученное в результате работы этой программы значение будет неправильным. Проверьте, сможете ли вы определить, почему?

Листинг 6.3. Неправильная программа расчета среднего арифметического значения трех чисел.

```
/*average.c*/
main()
        {
        float number_1, number_2, number_3, average;
        printf("Введите первое число: ");
        scanf("%f", &number_1);
        printf("Введите второе число: ");
        scanf("%f", &number_2);
        printf("Введите третье число: ");
        scanf("%f", &number_3);
        average = number_1 + number_2 + number_3 / 3;
        printf("Среднее арифметическое равно %f", average);
        }
```

Ошибка содержится в инструкции

average = number_1 + number_2 + number_3 / 3;

Дойдя до этой строки программы, компьютер будет выполнять математические действия в следующем порядке:

1. Поделит значение переменной number_3 на 3.
2. Сложит значения переменных number_1 и number_2 и прибавит к ним результат деления.

Если вы присвоите каждой переменной значение 100, выведенное на экран среднее арифметическое будет равняться 233.33. Вот правильная запись этой строки инструкций:

average = (number_1 + number_2 + number_3) / 3;

Круглые скобки изменяют порядок приоритетов. Просматривая эту строку, компьютер прежде всего выполнит действия в скобках, затем вернется к началу строки и выполнит остальные операции в соответствии с обычным приоритетом операторов, о котором мы только что говорили. Таким образом, в нашей программе будет сперва выполнено сложение трех чисел в скобках, а потом результат поделен на 3 (рис.6.7). Вуаля! Получился правильный ответ.

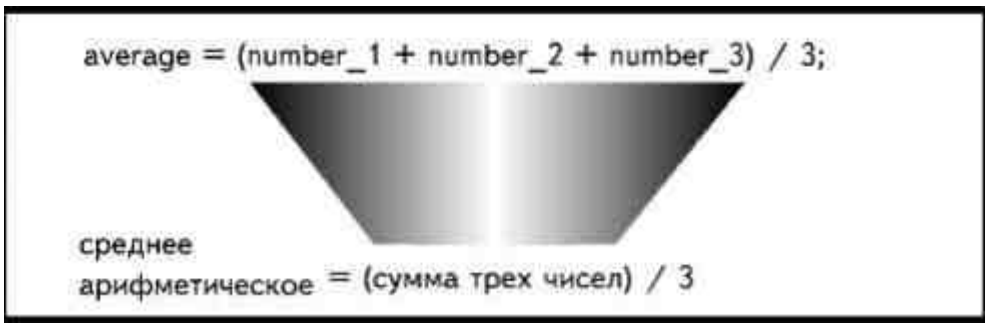

Рис. 6.7. Правильная формула расчета среднего арифметического значения

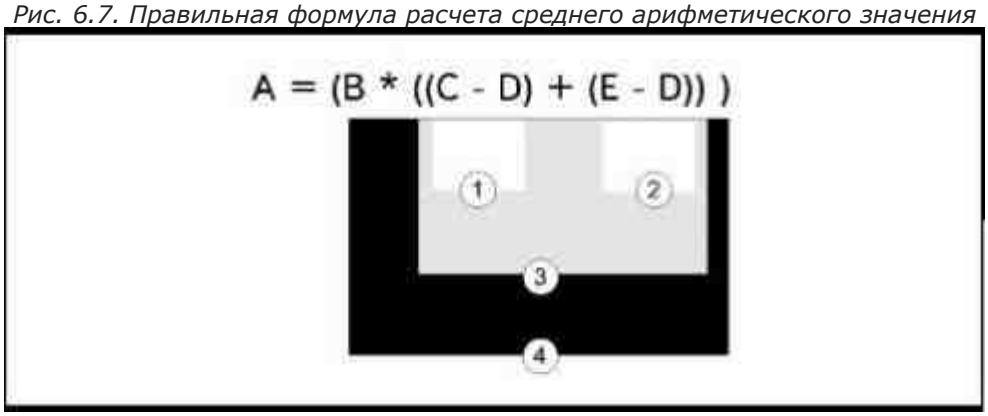

Рис. 6.8. Использование нескольких уровней круглых скобок

Для того чтобы установить нужный порядок выполнения операций, можно использовать несколько уровней скобок (рис.6.8). Порядок выполнения операций установлен от самых внутренних скобок к наружным. Рассмотрим пример. Допустим, работник получает оплату в двойном размере за каждый час, проработанный сверх 40-часовой рабочей недели. Полагая, что он работает как минимум 40 часов и, соответственно, получает плату как минимум за 40 рабочих часов, расчет недельного заработка состоит из следующих частей:

```
40 * rate          /* обычный недельный заработок */
hours - 40  /* сверхурочные часы */
rate * 2           /* оплата сверхурочных */
```

где rate— плата за 1 час работы, а hours— общее количество проработанных часов. В этом уравнении надо умножить количество сверхурочных часов на оплату в двойном размере, а затем прибавить полученный результат к обычной недельной оплате. Если вы напишете уравнение без учета приоритета операторов, оно будет выглядеть так:

total = 40 * rate + hours - 40 * rate * 2

Допустим теперь, некто проработал 48 часов в неделю, причем его обычный заработок составляет 10 долларов в час. В приведенном уравнении операции будут выполняться в следующем порядке:

Операция	Результат
40 * rate	400
40 * rate	400
400 * 2	800
400 + 48	448
448 – 800	–352

Согласитесь, что получить недельный заработок в размере –352 доллара несколько обидно. Чтобы написать правильное уравнение, используйте скобки: total = (40 * rate) +((hours - 40) * (rate * 2)) Теперь уравнение состоит из двух логических частей (полный текст программы приведен в Листинге6.4). В первых скобках вычисляется обычная заработная плата, во вторых— оплата сверхурочных. Расчет сверхурочных тоже состоит из двух частей, заключенных в собственные скобки. Си прежде всего выполнит операции во внутренних скобках. Порядок выполнения действий теперь станет таким:

Операция	Результат
40 * rate	400
hours – 40	8
rate * 2	20
8 * 20	160
400 + 160	560

Листинг 6.4. Программа расчета недельного заработка с учетом сверхурочных.

```
/*payroll1.c*/
main()
        {
```

```
float rate, hours, total;
printf("Введите оплату одного часа работы: ");
scanf("%f", &rate);
printf("Введите количество отработанных часов: ");
scanf("%f", &hours);
total = (40 * rate) + ((hours -40) * (rate*2));
printf("Ваш недельный заработок: %f", total);
}
```

Если вы находите, что использование скобок сбивает вас с толку, разбейте уравнение на несколько отдельных частей. В Листинге6.5 приведен текст программы, которая выполняет тот же расчет недельного заработка, но проводит его в несколько последовательных этапов, определяя для этого дополнительные переменные. Результат каждого отдельного вычисления присваивается конкретной переменной и может быть выведен на экран, что делает получаемую информацию даже несколько более наглядной. Этот прием позволяет полностью контролировать весь процесс вычисления и с большей легкостью находить ошибки.

Листинг 6.5. Программа, осуществляющая вычисление в несколько этапов.

```
/*payroll2.c*/
main()
{
float rate, hours, total, regular, extra, d_time, overtime;
printf("Введите оплату одного часа работы: ");
scanf("%f", &rate);
printf("Введите количество отработанных часов: ");
scanf("%f", &hours);
regular = 40 * rate;
extra = hours - 40;
d_time = rate * 2;
overtime = extra * d_time;
total = regular + overtime;
printf("Ваш недельный заработок: %f", total);
}
```

Используемые алгоритмы обработки данных

Некоторые часто используемые алгоритмы обработки данных содержат арифметические операторы. Многие из них применяются настолько часто, что программисты даже не думают о них как об алгоритмах. Два наиболее важных из них называются «счетчиком» и «аккумулятором».

Счетчики

Счетчик — это переменная, которая увеличивает свое значение на единицу каждый раз, когда происходит определенное событие. Алгоритм счетчика таков:

variable = variable + 1

Учитель математики скажет вам, что эта формула бессмысленна, так как соответствующее уравнение не имеет решений, но в программировании на компьютере такая инструкция является вполне законной.

Рис. 6.9. Алгоритм счетчика

Компьютер сначала вычислит значение в правой части, а потом присвоит полученное значение переменной в левой части (рис.6.9). Таким образом, одна и та же переменная никогда не будет иметь два значения одновременно. С точки зрения компьютера, смысл выражения можно передать так:

Новое значение переменной равно старому значению плюс 1

Давайте проследим за работой алгоритма счетчика, так, как она показана на рис.6.10. У нас есть переменная count, которой присвоено начальное значение, равное нулю:
int count;
count=0;
Теперь вступает в действие алгоритм
count = count + 1;

Инструкция	Выполняется математическая операция	Значение переменной count после выполнения инструкции
count = 0;		0
count = count + 1;	count = 0 +1	1
count = count + 1;	count = 1 +1	2
count = count + 1;	count = 2 +1	3
count = count + 1;	count = 3 +1	4

Рис. 6.10. Выполнение алгоритма счетчика

Компьютер выполняет эту инструкцию так:
count = 0 + 1
К начальному значению переменной count, которое равно 0, добавлен литерал, имеющий значение 1. В результате вычислений получено значение 1, которое теперь присваивается переменной count. Значение переменной изменяется с 0 на1. Затем та же процедура повторяется снова:
count = count + 1;
Компьютер выполняет эту операцию как
count = 1 + 1
К текущему значению переменной count, равному 1, прибавляется литерал со значением 1. В результате они дают 2, и это новое значение присваивается переменной. С каждым новым выполнением этой операции, значение переменной count возрастает на единицу (*инкремент*).
Разумеется, можно присвоить переменной любое начальное значение и увеличивать его на любое отличное от единицы число. Если присвоить переменной count начальное значение, равное 1, при выполнении инструкции
count = count + 2;
значение переменной всегда будет нечетным числом: 1, 3, 5, 7 и так далее. Используя переменную count, можно считать пятерками, count = count + 5, или десятками, count = count + 10, или как угодно еще.
Чтобы считать в сторону уменьшения, достаточно слегка изменить алгоритм:
variable = variable - 1
Теперь при каждом выполнении операции значение переменной будет уменьшаться на единицу (*декремент*).

Операторы инкремента

Счетчики используют настолько часто, что в языке Си существуют специальные операторы инкремента и декремента переменной. Оператор ++variable увеличивает значение переменной на единицу еще до выполнения соответствующей инструкции. Оператор выполняет то же действие, что и инструкция
variable = variable + 1;
В качестве примера действия оператора инкремента, рассмотрим следующую программу:

```
/*count.c*/
main()
{
int count = 0;
printf("Первое значение переменной \
 count равно %d", count);
printf("Второе значение переменной \
 count равно %d", ++count);
printf("Последним значением переменной \
 count является %d", count);
}
```

Результат работы программы отображается в виде сообщений:
Первое значение переменной count равно 0
Второе значение переменной count равно 1
Последним значением переменной count является 1

Перед выполнением второй функции printf() компилятор увеличивает значение переменной count на 1. Тот же эффект был бы достигнут и при использовании инструкции

```
count = count + 1;
printf("Второе значение переменной
 count равно %d\n", count);
```

Использование оператора инкремента позволяет увеличить значение переменной без введения в текст программы отдельной инструкции присваивания.

Необходимо помнить, что оператор инкремента реально изменяет значение переменной. Проверьте, понимаете ли вы разницу между оператором инкремента ++count и выражением, приведенным в следующей строке:

```
printf("Второе значение переменной
                    count равно %d\n", count+1);
```

Выражение count+1 не изменяет значения, присвоенного переменной count. В результате выполнения этой инструкции значение переменной, увеличенное на единицу, только отображается на экране, но не заносится в память. Пример программы, в которой используются выражения вместо операторов инкремента, приведен в Листинге6.6.

Листинг 6.6. Использование выражений вместо операторов инкремента.

```
main()
{
int count = 0;
printf("Первое значение переменной \
 count равно %d", count);
printf("Второе значение переменной \
 count равно %d", count+1);
printf("Последним значением переменной \
 count является %d", count);
}
```

В результате работы программы мы увидим следующие сообщения:

```
Первое значение переменной count равно 0
Второе значение переменной count равно 1
Последним значением переменной count является 0
```

Значение выражения count+1 представлено на экране как 1, но полученный результат не был внесен в соответствующую область памяти. Поэтому третья функция printf() вывела на экран монитора исходное значение переменной, равное 0.

Оператор инкремента можно использовать так же, как и выражение, значение которого будет присвоено переменной, стоящей в левой части инструкции присваивания. Например, следующая инструкция увеличивает значение переменной count, а затем присваивает полученное значение переменной number:

```
number = ++count;
```

те же операции можно было выполнить и так:

```
count = count + 1;
number = count;
```

Оператор ++ можно использовать с именем переменной как инструкцию:

```
++number;
```

Если же знаки ++ помещены справа от имени переменной,

```
variable++;
```

то приращение значения переменной произойдет после завершения соответствующей инструкции. Посмотрите на слегка модифицированную программу из Листинга6.6:

```
main()
        {
        int count = 0;
        printf("Первое значение переменной \
 count равно %d", count);
        printf("Второе значение переменной \
 count равно %d", count++);
        printf("Последним значением переменной \
 count является %d", count);
        }
```

Оператор ++ увеличит значение переменной count после выполнения второй инструкции, использующей функцию printf(). Вторая функция printf() отобразит на экране исходное значение переменой, и сообщения, выведенные на экран, будут выглядеть так:

```
Первое значение переменной count равно 0
Второе значение переменной count равно 0
Последним значением переменной count является 1
```

Во время выполнения первой и второй функций printf(), значение переменной равно 0 и увеличивается на 1 только перед выполнением третьей функции printf().

Используя оператор инкремента, можно сохранить начальное значение переменной в другой переменной и одновременно увеличить его, как показано в программе:

```
main()
        {
        int number, storage;
        puts("Введите значение числа:");
        scanf("%d", &number);
        storage = number++;
        printf("Начальное значение числа: %d\n", storage);
        printf("Новое значение числа: %d", number);
        }
```

В инструкции

```
storage = number++;
```

мы, во-первых, присваиваем значение number переменной storage, а во-вторых, увеличиваем переменную number на единицу.

Оператор декремента работает аналогичным образом, но уменьшает значение переменной на единицу. Синтаксис использования оператора таков:

```
--variable
```

уменьшает значение переменной на 1 до выполнения инструкции

```
variable--
```

уменьшает значение переменной на 1 после выполнения инструкции

Аккумуляторы

Аккумулятор также увеличивает значение переменной. Но, в отличие от счетчика, который всегда увеличивает значение переменной на одну и ту же величину, аккумулятор может иметь произвольный шаг (и способ) изменения при каждой новой операции. В общем виде синтаксис аккумулятора таков:

```
variable = variable + other_variable;
```

Аккумулятор получил такое название оттого, что он накапливает значение переменной. Посмотрите на следующий пример:

```
int total, number;
total = 0;
scanf("%d", &number);
total = total + number;
```

Допустим, что переменной number присвоено значение 10. После выполнения инструкции

```
total=total+number;
```

переменная total приобретает значение 10, так как компьютер выполнил операцию сложения, используя следующие значения:

```
total=0+10;
```

Теперь предположим, что снова происходит ввод данных с помощью функции scanf() и выполняется новая операция суммирования, но на этот раз переменной number присвоено значение 15:

```
scanf("%d", &number);
total = total + number;
```

теперь компьютер использует в вычислениях следующие значения:

```
total = 10 + 15;
```

Произошло накопление значений переменной number.

В Листинге 6.7 приведен текст программы, в которой вводятся три значения и вычисляется их среднее арифметическое. Для расчета среднего арифметического пользуются простым математическим выражением:

```
average = (A + B + C) / 3
```

В программе мы использовали аккумулятор, чтобы подсчитать сумму трех чисел, и счетчик для определения количества введенных значений (чтобы продемонстрировать работу счетчика). В главе 9 будет показан более эффективный способ использования этих алгоритмов.

Листинг 6.7. Программа, использующая счетчик и аккумулятор для вычисления среднего арифметического трех чисел.

```
/*average1.c*/
main()
        {
        float number, total, count, average;
        total = 0.0;
        count = 0.0;
        printf("Введите первое число: ");
        scanf("%f", &number);
        total += number;
        ++count;
```

```
printf("Введите второе число: ");
scanf("%f", &number);
total += number;
++count;
printf("Введите третье число: ");
scanf("%f", &number);
total += number;
++count;
average = total / count;
printf("Среднее арифметическое равно %f", average);
}
```

Операторы присваивания

Приводимые ниже операторы присваивания являются сокращенной записью различных типов аккумуляторов.

Оператор	Пример	Эквивалент
+=	total += amount	total = total + amount
–=	total –= discount	total = total – discount
*=	total *= tax_rate	total = total * tax_rate
/=	total /= count	total = total / count
%=	total %=count	total = total % count

Каждый из них выполняет операции, используя в качестве общего элемента переменную, имеющую присвоенное начальное значение. Чтобы понять, как действуют эти операторы, посмотрите на рис.6.11. Оператор как бы копирует переменную и арифметический символ левой части уравнения в правую часть. Инструкция

total *= rate;

соответствует инструкции

total = total * rate;

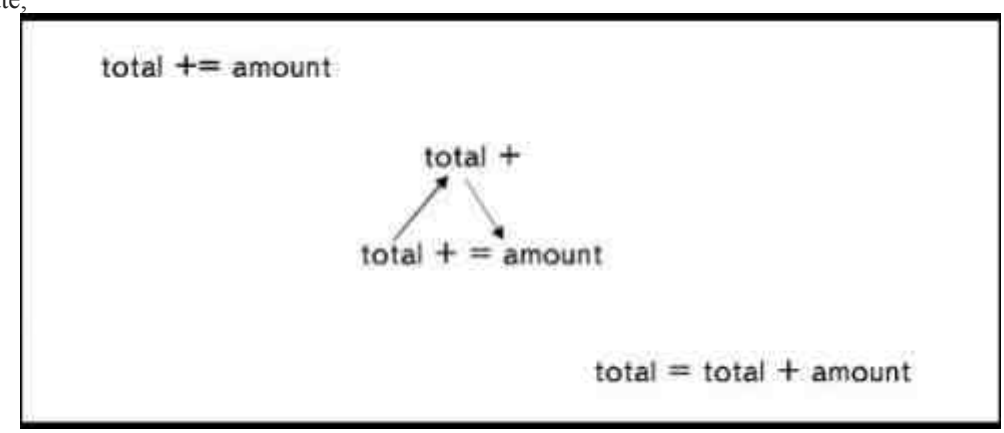

Рис. 6.11. Оператор присваивания

Присваивание начального значения

Очень важно, чтобы все счетчики и аккумуляторы имели присвоенное начальное значение. Вы помните, что оба алгоритма увеличивают (или уменьшают) текущее значение переменной на определенную величину. Если не присвоить начальное значение, то эта величина будет прибавлена к случайному содержимому области памяти, зарезервированной для переменной. Инициализация переменной очищает содержимое памяти, так же как нажатие клавиши Clear на калькуляторе.

В качестве примера обратимся к Листингу 6.7, содержащему программу расчета среднего арифметического значения трех чисел. Допустим, в области памяти, отведенной для переменной total, содержится случайное значение 1827. Если переменной не было присвоено начальное значение

total = 0;

то, когда мы вводим первое число 75, первый аккумулятор должен будет выполнить следующую математическую операцию:

total = 1827 + 75

Присваивание начального значения 0 дает уверенность в том, что математические расчеты будут выполнены правильно:

total = 0 + 75

Проектирование программы

Теперь, когда вы умеете выполнять математические операции, используя операторы, вы можете проектировать программы с более сложной логической структурой, а следовательно, возрастает вероятность появления ошибок. Вам потребуется дополнительное время, чтобы убедиться, что ваша программа работает именно так, как вы планировали, и выдает действительно правильные результаты. Рассмотрим несколько примеров, демонстрирующих «подводные камни», которые могут вам встретиться.

Остерегайтесь логических ошибок

Ранее в этой же главе мы приводили пример программы для расчета недельного заработка (см. Листинг6.4). Компиляция этой программы пройдет без ошибок, и вы можете пользоваться ею некоторое время, прежде чем обнаружится главный изъян, а именно, в программе подразумевается, что служащий работает, по меньшей мере, 40 часов в неделю, или, что то же самое, минимальной возможной оплатой является оплата сорокачасовой рабочей недели. Если это действительно так, то никаких проблем не возникнет. Но что, если работник работает меньше сорока часов в неделю и должен получать зарплату в соответствии с реально отработанным временем? Если в эту программу ввести значение отработанных часов меньшее, чем 40, все расчеты окажутся неверными. Проблема содержится в инструкции

```
total = (40 * rate) +((hours - 40) * (rate * 2));
```

Во-первых, даже если кто-то работал меньше 40 часов, все равно значение оплаты одного часа будет умножено на 40, как указано в действии 40 * rate. Во-вторых, результат операции hours - 40 окажется отрицательной величиной. Это отрицательное значение будет умножено на оплату часа работы, увеличенную вдвое, и полученный результат прибавлен к основной зарплате. Добавление отрицательной величины приведет к уменьшению исходного значения, и в результате работник потеряет из заработка оплату в двойном размере за каждый час, недостающий до40.

Решение этой проблемы приводится в следующих главах. Пока же поиск решения представляется менее важной задачей, чем ясное понимание сущности проблемы.

Ищите образцы

Когда вы пишете программу, ищите образцы. Отыскивайте инструкции, которые будут использоваться неоднократно, возможно, лишь с небольшими изменениями. Если вы увидите какие-то закономерности, это поможет лучше понять, как работает программа, и позволит впоследствии без труда написать сходную с ней.

В качестве примера возьмем знакомую программу, в которой мы использовали деление нацело, чтобы набрать определенную сумму денег (см. Листинг6.2). Следующую последовательность инструкций можете рассматривать как образец:

```
twenties = amount / 20;
r20 = amount % 20;
tens = r20 / 10;
r10 = r20 % 10;
fives = r10 / 5;
ones = r10 % 5;
```

Обратите внимание на то, что переменные справа от знака равенства используются дважды: один раз с оператором деления (/), второй раз с

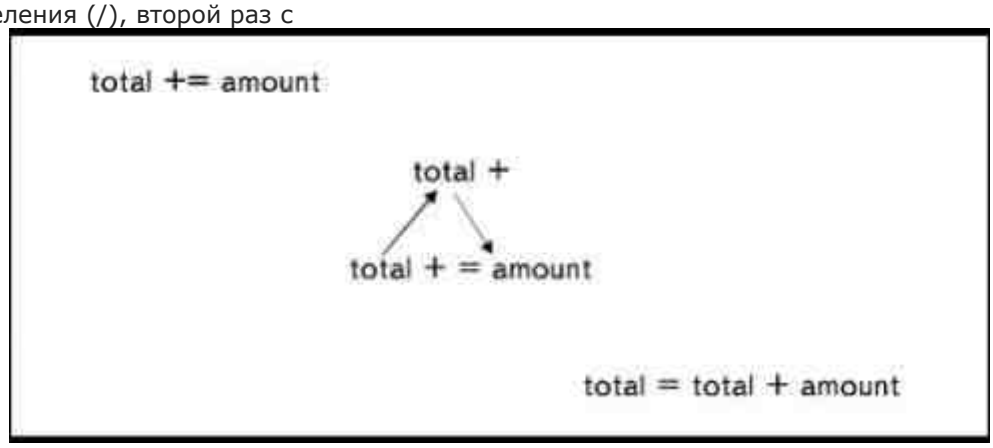

Рис. 6.12. Образец в программе из Листинга 6.2

оператором получения остатка от деления нацело (%). Заметьте также, что результат вычисления остатка от деления нацело используется в следующей за ним строке. На рис.6.12 проиллюстрирован приведенный образец.

Эту программу можно легко расширить распространением приведенного образца на другие уровни. Например, если вы хотите использовать банкноты достоинством 50 долларов, то можете модифицировать программу, добавив следующие строки:

```
fifties = amount / 50;
r50 = amount % 50;
twinties = r50 / 20;
r20 = r50 % 20;
```

В тот же образец мы добавили две новые инструкции и слегка видоизменили строку, в которой рассчитывается количество двадцатидолларовых банкнот.

Диагностические проблемы

Если вы не до конца уверены, что с результатами работы программы все обстоит благополучно, попробуйте ввести дополнительные функции printf(). Разделите весь процесс на такое количество дискретных операций, какое только возможно. После каждой операции, изменяющей значение переменной, помещайте функцию printf(), отображающую текущее значение переменной, даже если оно не интересует вас в качестве конечного результата.

При выполнении программы следите за тем, что выводят на экран эти дополнительные функции printf() и сравнивайте с тем, что вы ожидали увидеть. Появление первого неправильного значения укажет путь к исправлению ошибки. Например, если при работе программы расчета заработной платы появятся следующие сообщения:

значение regular равно 400
значение extra равно -2
значение d_time равно 40

отрицательное значение переменной extra, содержащей значение оплаты сверхурочных, должно немедленно насторожить вас. Можно предположить, что проблема кроется в неправильных инструкциях где-то перед функцией printf(), отображающей значение этой переменной. После исправления ошибки дополнительные инструкции вывода можно убрать из текста программы.

Вопросы

1. В чем заключается различие между операторами / и %?
2. Можно ли использовать разные типы данных в одной операции? Если да, то как это отразится на результатах операции?
3. Что такое выражение?
4. Где можно использовать выражение?
5. Опишите порядок приоритета арифметических операторов.
6. Зачем при записи операций используют круглые скобки?
7. Объясните разницу между инструкциями count=count+1 и count++.
8. Что такое аккумулятор?
9. Дайте описание оператора присваивания.
10. В чем разница между --count и count--?

Упражнения

1. Напишите программу, которая сообщает пользователю, сколько лет ему будет в 2000 году.
2. Напишите программу расчета квадрата и куба числа, введенного с клавиатуры.
3. Напишите программу перевода температуры из шкалы Фаренгейта (F) в шкалу Цельсия (C). Формула пересчета C=(5.0/9.0)Ф(F–32).
4. Модифицируйте программу из упражнения 3 так, чтобы она сообщала, на сколько градусов отстоит введенное значение температуры от точки замерзания по шкале Фаренгейта и по шкале Цельсия.
5. Объясните, почему следующая программа написана неверно:

```
6.  #define TAX_RATE 0.06
7.  main()
8.  {
9.  float cost, total;
10. printf("Введите стоимость единицы товара: ");
11. scanf("%f", &cost);
12. printf("Введите величину транспортных расходов: ");
13. scanf("%f", &shipping);
```

```
14. total = cost + cost * tax_rate + shipping;
15. printf("Общая стоимость составляет %f", total);
    }
```

ГЛАВА 7

ДЛЯ ЧЕГО НУЖНЫ ФУНКЦИИ

Если вы возьмете все свое имущество и свалите в большую кучу на полу, то поиск нужных вещей будет каждый раз отнимать массу времени. Чтобы облегчить себе жизнь, вы предусмотрительно сортируете предметы, раскладывая их по полкам, и как только вам понадобится какая-нибудь вещь, вы точно знаете, где ее можно найти (по крайней мере, так предполагается).

Ваша программа тоже может страдать от отсутствия порядка. Как только она начнет приобретать достаточно большие размеры, работать, оставаясь в пределах одной функции main(), будет все труднее. Программные коды, к сожалению, нельзя разложить по полкам, но их можно распределить по функциям.

Функцией называется выделенная последовательность инструкций, предназначенных для решения определенной задачи. Вы можете написать свои функции и затем использовать их точно так же, как функции из библиотек языка

Си или Си++. Вы вызываете функцию для выполнения определенной задачи, передавая ей аргументы и получая от нее результат.

Функции особенно полезны, когда требуется выполнить одну и ту же задачу несколько раз. Например, если при выполнении программы требуется вывести на экран монитора большой кусок текста, возникает необходимость приостанавливать вывод информации по мере заполнения экрана. Вместо того чтобы писать соответствующие инструкции каждый раз, когда надо это сделать, можно оформить их в виде отдельной функции. После этого, где бы ни

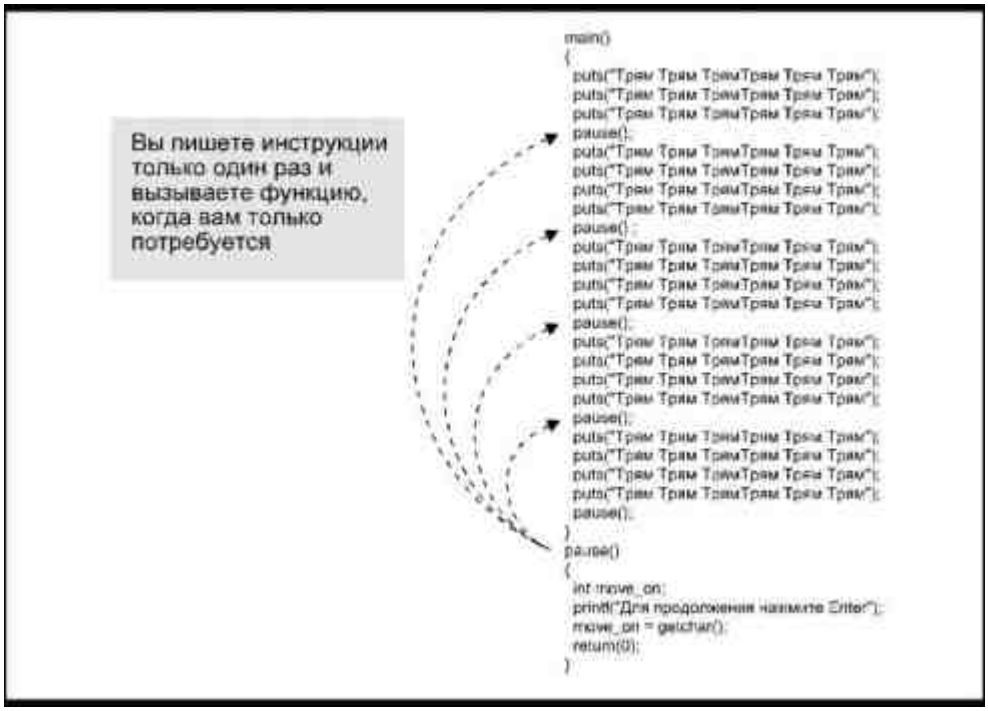

Рис. 7.1. Многократный вызов функции

потребовалось временно приостановить вывод информации на дисплей, достаточно просто вызвать данную функцию (рис.7.1).

Вы можете использовать функции для структурирования сложной программы. Попытайтесь поделить программу на блоки, каждый из которых выполняет какую-то одну вполне законченную задачу. Эти блоки, в свою очередь, при необходимости также можно разделить на меньшие. Продолжайте деление на все более мелкие части до тех пор, пока остается возможность написать инструкции для выполнения самостоятельной задачи. Каждый такой набор инструкций помещается в свою собственную функцию. Как только вы напишите все необходимые функции и объедините их функцией main()— программа готова!

Как использовать функции

Собственные функции помещаются после закрывающей фигурной скобки main().

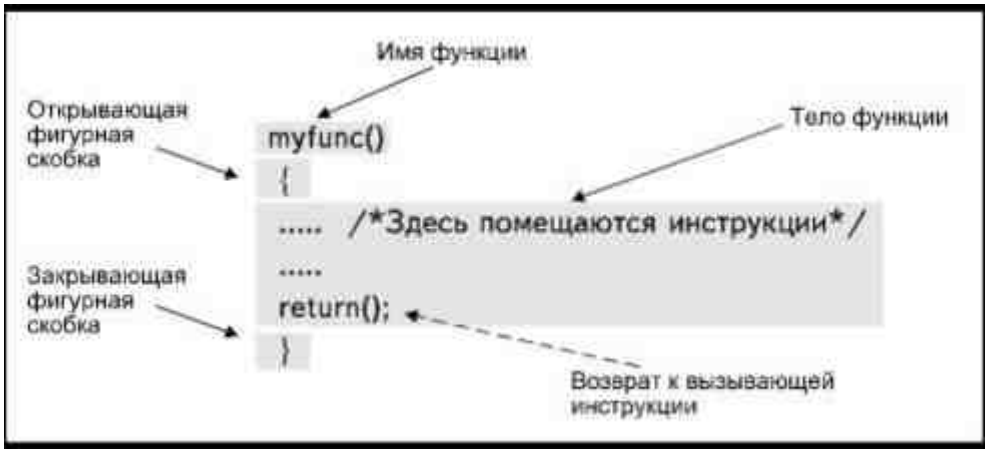

Рис. 7.2. Структура функции

Каждая функция имеет структуру, аналогичную структуре функции main() (рис.7.2): после имени функции ставится пара круглых скобок, точка с запятой в конце строки отсутствует, инструкции, составляющие тело функции, заключаются в пару фигурных скобок.

При вызове функции компьютер выполняет инструкции, записанные в теле функции, а затем управление возвращается строке, которая стоит непосредственно после инструкции вызова функции. В зависимости от компилятора использование инструкции return (0); в теле функции может быть необязательным. Си может обеспечить автоматический возврат после завершения всех инструкций.

На рис. 7.3 в качестве примера показана программа, которая выводит на дисплей вопрос и ответ на него. Цифры указывают порядок выполнения инструкций.

Инструкция

answer();

в main() вызывает функцию answer() так, как если бы она содержалась в библиотеке Си, с той разницей, что текст функции answer() находится не в

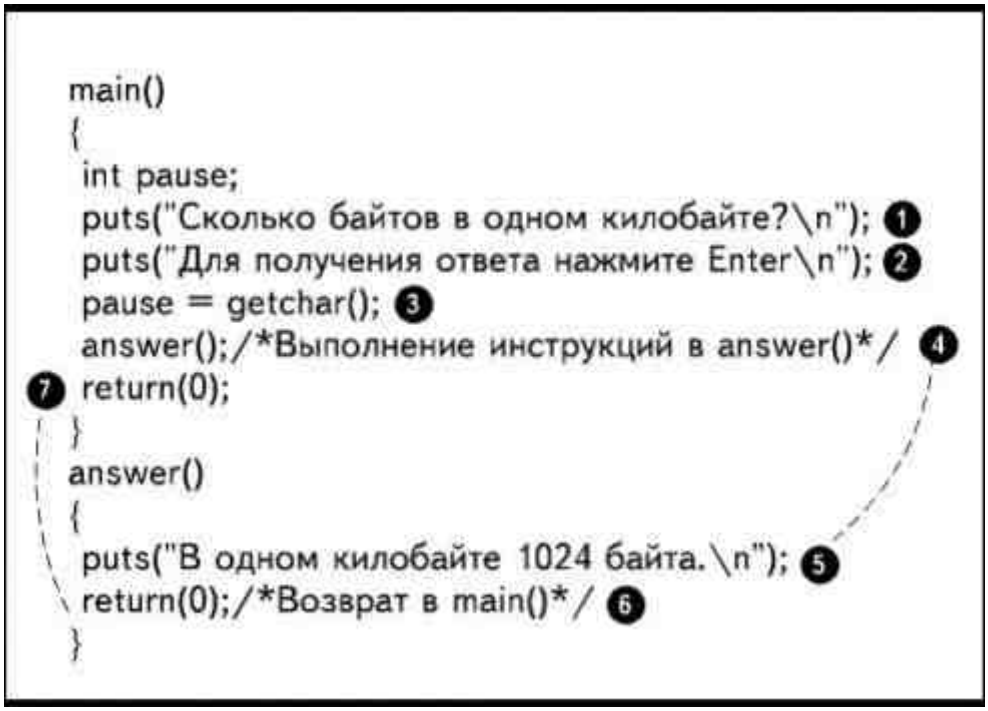

Рис. 7.3. Вызов функции

библиотеке, а включен в программный код. Вызов этой функции осуществляется после выполнения двух функций puts(). Функции answer() не передаются никакие параметры, так как она содержит всю информацию, необходимую для выполнения своей задачи. После выполнения функции puts() в answer() инструкция return(0); передает управление инструкции, помещенной в теле main() непосредственно за инструкцией вызова функции answer(). Инструкция return(0); в main() завершает выполнение программы.

Приведенную на рис. 7.3 программу компьютер выполнит в следующем порядке:

puts("Сколько байтов в одном килобайте?");

puts("Для получения ответа нажмите Enter");

```
pause = getchar();
answer();
puts("В одном килобайте 1024 байта.");
return(0);
return(0);
```

Функцию можно вызвать из любого места программы, в том числе, из другой функции. Инструкция return(0); в теле функции всегда передает управление инструкции, следующей за вызовом функции. В Листинге 7.1 приведен текст программы с двумя не библиотечными функциями. Первая из них называется

question() и вызывается из main(). Вторая значится под именем answer() и вызывается из функции question(). Инструкция return(0) в answer() возвращает управление назад в question(), из которой позже происходит возврат в main(). Ниже приведен результат работы этой программы, причем для каждого сообщения указана функция, выводящая его на экран.

Добро пожаловать в наш Опросник.	main()
Скажите название графического интерфейса фирмы Microsoft.	question()
Для получения правильного ответа нажмите Enter.	question()
Правильный ответ - Windows.	answer()
Благодарим за участие.	main()

Листинг 7.1. Программа, в которой осуществляется вызов двух функций.

```
/*quiz2.c*/
main()
        {
        puts("Добро пожаловать в наш Опросник.\n");
        question();
        puts("Благодарим за участие.\n");
        return(0);
        }
question()
        {
        int move_on;
        puts("Скажите название графического \
 интерфейса фирмы Microsoft.\n");
        puts("Для получения правильного ответа \
 нажмите Enter.\n");
        move_on = getchar();
        answer();
        return(0);
        }
answer()
        {
        puts("Правильный ответ - Windows.\n");
        return(0);
        }
```

Лучший способ использования функций— это разбивка программы на отдельные блоки. Функция main() выполняет при этом обязанности «мажордома», определяя начало программы и обеспечивая вызов других функций. Это хорошо продемонстрировано в Листинге7.2, являющемся вариантом программы Опросника. Здесь main() содержит только инструкции вызова функций. Настоящая «работа» программы обеспечивается в функциях, вызываемых main().

Листинг 7.2. Программа, использующая main() для вызова других функций.

```
/*quiz3.c*/
main()
        {
        welcome();
        question();
        answer();
        the_end();
        return(0);
        }
welcome()
        {
        puts("Добро пожаловать в наш Опросник.\n");
        return(0);
        }
question()
        {
        int move_on;
        puts("Скажите название графического
         интерфейса фирмы Microsoft.\n");
        puts("Для получения правильного
         ответа нажмите Enter.\n");
        move_on = getchar();
        return(0);
        }
answer()
        {
        puts("Правильный ответ - Windows.\n");
        return(0);
        }
the_end()
        {
        puts("Благодарим за участие.\n");
        return(0);
        }
```

Подобная структура упрощает процесс поиска ошибок. Например, если обнаружились проблемы с выводом на экран правильного ответа, ошибку следует искать в инструкциях функции answer().

Функции выполняются в порядке вызова независимо от их местоположения в тексте программы. Мы можем поместить функции в Листинге7.2, например, в таком порядке:

```
main()
```

```
answer()

welcome()

the_end()

question()
```

Результат работы программы от этого не изменится, так как не изменился порядок вызова функций в main().

Переменные в функциях

Если программа содержит другие функции, кроме функции main(), вам необходимо решить, где и как вы будете определять переменные. Си имеет несколько типов переменных. В этой главе мы рассмотрим автоматические, внешние и статические переменные.

Автоматические (локальные) переменные

Некоторые функции нуждаются в собственных переменных и константах. В качестве примера снова рассмотрим проблему временной остановки вывода информации на экран монитора. Можно приостановить вывод, используя следующие инструкции:

```
printf("Для продолжения нажмите любую клавишу");

move_on = getchar();
```

Как мы уже говорили, если вывод информации на экран во время выполнения программы надо приостанавливать несколько раз, выполняющие это действие инструкции помещают в отдельную функцию. Каждый раз, когда возникает необходимость притормозить вывод, достаточно просто вызвать соответствующую функцию. Так как getchar() возвращает значение, переменную, принимающую его, можно определить внутри функции pause():

```
pause()
        {
        int move_on;
        printf("Для продолжения нажмите любую клавишу");
        move_on = getchar();
        return(0);
        }
```

В данном случае переменная move_on является составной частью функции pause(). Переменную, которая определена внутри функции, принято называть локальной переменной для этой функции. Локальную переменную можно использовать только внутри той функции, где она была определена. В языке Си переменные такого типа обычно называют автоматическими.

Задумайтесь над этим. Если вы определили переменную внутри какой-либо функции, то эта переменная имеет смысл только для данной функции. Сказанное справедливо и для переменных, определенных в функции main(). Посмотрите на программу

```
main()
        {
        int move_on;
        puts("Сообщения заполняют экран\n");
/*Здесь будет серия сообщений, которые заполнят экран*/
        pause();
        }
        pause()
        {
        printf("Для продолжения нажмите любую клавишу");
        move_on = getchar();
        return(0);
        }
```

При компиляции этой программы будет выдано сообщение об ошибке, так как функция pause() не может использовать переменную move_on. Ведь она определена внутри main(), а поэтому является локальной для main() и только внутри этой функции имеет смысл и может быть использована. Для того чтобы переменная move_on могла быть использована в функции pause(), следует определить ее внутри именно этой функции.

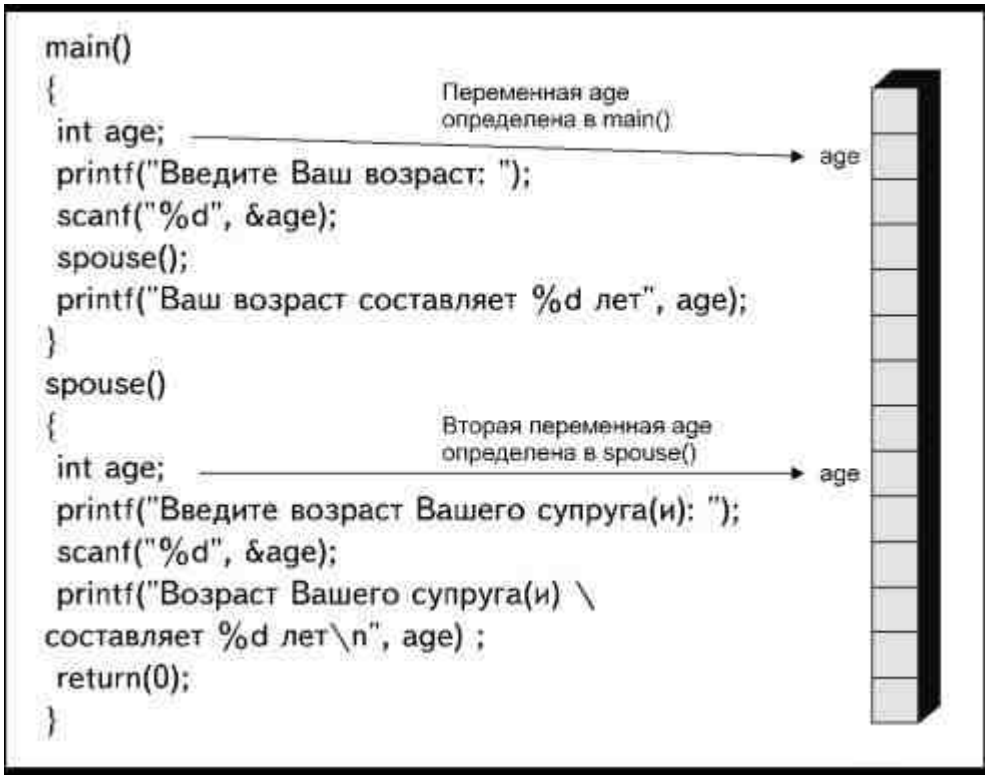

Рис. 7.4. Автоматические переменные могут иметьодно и то же имя в разных функциях

Вследствие того, что автоматические переменные имеют смысл только внутри собственной функции, можно использовать одно и то же имя переменной в разных функциях. На рис.7.4 продемонстрирована переменная с именем age, которая определена и в функции main(), и в функции spouse(). Каждая переменная является автоматической (или локальной) для той функции, где ее определили, так что на самом деле в программе присутствуют две переменные с одинаковым именем. Каждая из них имеет собственное значение, которое хранится в различных элементах памяти. Компилятор определяет, значение какой из переменных age должно быть отображено на экране, на основании того, в какой из функций помещена инструкция printf();. Выполняя функцию printf(), помещенную внутри spouse(), программа отобразит значение переменной age, определенной внутри этой функции. При выполнении инструкции printf(), помещенной внутри функции main(), она будет иметь дело с переменной age, определенной в main(). Таким образом, компилятор поддерживает одновременно две переменные с именем age.

Внешние (глобальные) переменные

Внешние переменные — это переменные, которые может использовать любая функция программы. В некоторых языках они носят название *глобальных переменных*. Для того чтобы создать внешнюю переменную, ее следует определить перед функцией main():

```
int temp;

main()
```

Здесь переменная temp определена как внешняя, так что с ней могут работать все функции, содержащиеся в программе. Посмотрите на текст программы, приведенный в Листинге7.3. Переменная temp является внешней и ее используют функции main(), convert(), freeze() и boil(). Значение переменной, введенное в main(), используется при расчетах другими функциями. В программе присутствует еще одна переменная— celsius. Поскольку значение переменной celsius используется только функцией convert(), переменная определена внутри этой функции, являясь, таким образом, локальной переменной, которая не может использоваться нигде кроме convert().

Листинг 7.3. Программа, содержащая внешнюю переменную.

```
/*f_to_c.c*/

int temp;

main()

        {
```

```
        printf("Введите значение температуры: ");
        scanf("%d", &temp);
        convert();
        freeze();
        boil();
        }
convert()
        {
float celsius;
        celsius = (5.0 / 9.0) * (temp - 32);
        printf("%d градусов по шкале Фаренгейта
                соответствует %6.2f градусам \
        по шкале Цельсия\n", temp, celsius);
        return(0);
        }
freeze()
        {
        printf("Это составляет %d градусов
        от точки замерзания воды\n", temp-32);
        return(0);
        }
boil()
        {
        printf("Это составляет %d градусов
         от точки кипения воды\n", 212-temp);
        return(0);
        }
```

Если в программе одновременно присутствуют внешняя и локальная переменные с одним и тем же именем, функция, в которой определена локальная переменная, будет иметь доступ только к ней и не сможет использовать внешнюю переменную.

Статические переменные

Автоматические (локальные) переменные существуют только во время выполнения функций, в которых они определены. В момент начала работы функции для такой переменной резервируется память. Когда выполнение функции заканчивается, зарезервированная область памяти освобождается и переменная прекращает свое существование.

Если функция вызывается неоднократно, локальные переменные создаются столько раз, сколько вызовов данной функции имеется в программе, поэтому адреса этих переменных могут все время меняться. Соответственно, и значения локальных переменных, хранящиеся в памяти, теряются каждый раз при завершении работы функции.

Тем не менее, значение, присвоенное переменной, можно сохранить в памяти и после окончания выполнения функции, если определить эту переменную как *статическую*. Это делается следующим образом:

```
myfunc()
        {
        static int count;
        }
```

Адреса хранения статических переменных остаются неизменными на протяжении всего времени выполнения программы. После прекращения работы функции память, закрепленная за переменной, не освобож-

дается, поэтому и записанное в ней значение переменной сохраняется. При следующем вызове этой же функции переменная будет иметь значение, оставшееся от предыдущего вызова. Запомните, что переменная при этом остается локальной и имеет смысл только внутри своей функции.

Для того чтобы пояснить, что такое статическая переменная, рассмотрим следующий пример. Посмотрите на программу, приведенную в Листинге7.4. Вэтом фрагменте статические переменные не используются. Функция doit() вызывается четырежды, и каждый раз при этом двум переменным присваивается начальное значение 0, которое отображается с помощью функции printf(). Впроцессе работы функции, значение каждой переменной увеличивается на единицу. Однако значение переменных теряется после очередного завершения работы функции, а при следующем вызове им снова присваивается начальное значение0, так что результат работы программы выглядит так:

значение autovar равно 0 значение statvar равно 0

значение autovar равно 0 значение statvar равно 0

значение autovar равно 0 значение statvar равно 0

значение autovar равно 0 значение statvar равно 0

Листинг 7.4. Программа, иллюстрирующая использование локальных переменных.

```
/*stat.c*/
main()
        {
        doit();
        doit();
        doit();
        doit();
        }
doit()
        {
        int autovar = 0;
        int statvar = 0;
        printf("значение autovar равно %dзначение statvar \
 равно %d\n", autovar, statvar);
        ++autovar;
        ++statvar;
        return(0);
        }
```

Давайте теперь слегка изменим определение переменных в функции doit(), с тем чтобы создать статическую переменную:

static int statvar = 0;

При выполнении программы со статической переменной мы получим следующий результат:

значение autovar равно 0 значение statvar равно 0

значение autovar равно 0 значение statvar равно 1

значение autovar равно 0 значение statvar равно 2

значение autovar равно 0 значение statvar равно 3

При первом выполнении функции переменной statvar присваивается начальное значение, равное 0. Однако statvar определена как статическая переменная, что указывает компилятору на необходимость сохранения в памяти присвоенного ей значения и после завершения работы функции. Поэтому после первого завершения работы функции переменная statvar будет иметь значение, равное 1, которое получается благодаря инструкции ++statvar;. При следующем вызове функции исходным значением переменной является сохраненное за ней значение1 и поэтому присваивания ей 0 в качестве начального значения не производится.

Каждый раз во время выполнения функции значение переменной увеличивается, и это новое значение сохраняется за переменной при очередном вызове функции.

Передача параметров

Существуют определенные задачи, которые можно выполнить только с помощью передачи функции параметров. Например, когда мы передаем параметр функции puts(), это означает, что мы записываем аргумент*, то есть строку, которую хотим отобразить на экране монитора, внутри круглых скобок. Функцию puts() вызывают с помощью инструкции, которая выглядит примерно так:

```
puts("Привет!");
```

Строка "Привет!" передается библиотечной функции puts() и сообщает ей, какую именно информацию следует вывести на экран.

Передача параметров вашим собственным функциям происходит аналогичным образом. Любые данные, которые вы хотите передать функции, должны быть заключены в круглые скобки. Совокупность переменных, передаваемых функции, называется списком аргументов**.

*** В литературе нет устоявшейся традиции в использовании терминов «аргумент» и «параметр». В том случае, когда используются оба термина, *параметром* чаще называют переменную из списка параметров, заданных в определении функции, а *аргументом* — конкретное значение, используемое при обращении к функции. В данной книге автор употребляет оба эти термина как синонимы. *(Прим.перев.)***

**** В дальнейшем этот список мы также будем называть списком *фактических аргументов*, или *фактических параметров*. *(Прим.перев.)***

Теперь посмотрим, что происходит с библиотечной функцией, получающей параметры. Например, функция puts() выполняет работу, которую можно выразить словами: «Вывести некую информацию на экран монитора». Таким образом, данная функция ожидает, что ей будет передан параметр, содержащий эту «некую информацию». Для этого в записи функции должно иметься место, где расположится получаемый аргумент (рис.7.5), иными словами, при записи

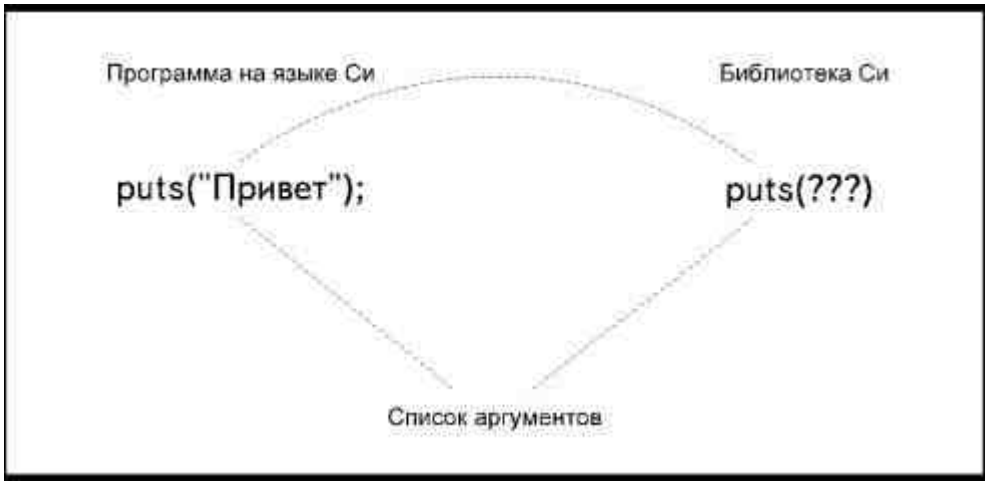

Рис. 7.5. Формальный аргумент необходим для полученияпереданного функции значения

функции, принимающей параметры, необходимо указывать список аргументов. Эти аргументы на самом деле являются переменными, которые будут использованы для хранения полученных данных*. Функции может быть передано сколько угодно аргументов, лишь бы их количество и типы данных в инструкции вызова функции соответствовали тем, которые она ожидает получить.

Разберемся теперь, как передать параметр нашей собственной функции. Посмотрите на следующую программу:

```
main()
        {
        int count;
        count = 5;
        doubles(count);
        }
```

*** Этот список также часто называют списком *формальных параметров*, или *формальных аргументов*.**

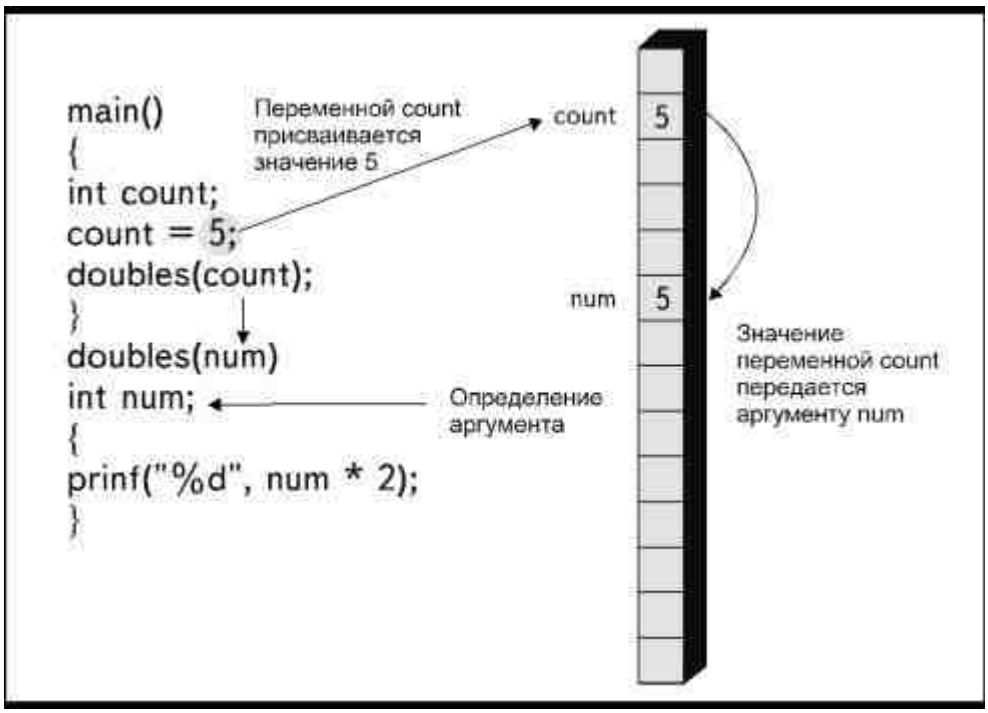

Рис. 7.6. Передача параметров

doubles(num)

int num;

{

printf("%d", num * 2);

return(0);

}

Инструкция

doubles(count);

в функции main() вызывает функцию и передает ей значение переменной count. Функция doubles() получает аргумент в качестве значения переменной num. Переменная num, таким образом, имеет то же значение, что и переменная count. Обратите внимание, строка, в которой определяется переменная num, стоит перед фигурной скобкой, открывающей тело функции doubles(). В записанных таким образом инструкциях производится определение списка аргументов. Определяя аргументы, мы указываем компилятору типы значений, которые будут переданы функции. Запомните, что вне фигурных скобок, ограничивающих тело функции, могут быть помещены только инструкции *определения списка аргументов*. Если же возникает необходимость определить другие переменные, это всегда следует делать внутри скобок.

Вот что происходит при работе функции doubles() в приведенной выше программе (можете следить по рис.7.6).

Замечания по Си++

В Си++ можно определять тип аргумента непосредственно в списке аргументов функции:doubles(int num)

1. Осуществляется вызов функции doudles(), которой передается значение переменной count.
2. Аргумент функции doubles() с именем num получает значение 5.
3. Функция удваивает полученное значение и отображает конечный результат с помощью функции printf().

Листинг 7.5 наглядно показывает, что список аргументов функции может содержать любое необходимое количество аргументов, относящихся к любым типам. Функция area() подсчитывает площадь помещений. Значения длины, ширины и номера этажа вводятся с клавиатуры в функции main(), а затем передаются функции area() при ее вызове:

area(length, width, fnum);

Листинг 7.5. Передача нескольких параметров.

```
/*area.c*/
main()
        {
        float length, width;
        int fnum;
        printf("Введите номер этажа: ");
        scanf("%d", &fnum);
        printf("Введите длину этажа: ");
        scanf("%f", &length);
        printf("Введите ширину этажа: ");
        scanf("%f", &width);
        area(length, width, fnum);
        }
area(size, wide, num)
float size, wide;
int num;
        {
        float area;
        area = size * wide;
        printf("Площадь %d этажа равна %f.2f", num, area);
        return(0);
        }
```

Здесь три аргумента были получены в том же порядке, в каком переданы. Значение переменной length присвоено переменной size, содержимое переменной width передано wide, а значение fnum— переменной num. Типы аргументов соответствуют друг другу: два вещественных значения получили две переменные типа float, а переменная типа int получила целое число.

Функции area() необходимо каким-то образом хранить результаты вычислений. С этой целью внутри тела функции area() нами определена переменная с именем area.

Если бы в инструкцию вызова функции вкралась ошибка и аргументы были перечислены нами в следующем порядке:

```
area(width, length, fnum);
```

то значение переменной width получила бы переменная size, а значение length— переменная wide. Поскольку типы переменных по-прежнему находятся в соответствии с получаемыми значениями, ошибки при компиляции не возникнет. Более того, результат работы программы будет совершенно правильным, ведь от изменения порядка мест сомножителей, произведение не меняется. Но предположим, что при вызове функции аргументы оказались расположенными в таком порядке:

```
area(fnum, width, length);
```

Теперь значение переменной fnum получит переменная wide, содержимое width перейдет в size, а переменной num будет присвоено значение length. Нетрудно заметить, что два аргумента имеют типы, не соответствующие получаемым значениям. Даже если компилятор не сообщит об ошибке, в результате работы программы мы все равно получим неверную информацию.

Рассмотрим еще один пример, приведенный в Листинге7.6. В этой программе вводятся значения двух переменных: стоимость единицы продукции (cost) и процент скидки (discount). Затем переменные cost и discount передаются функции price() через формальные аргументы amount и mrkdown. Переменные reduce и net определяются внутри функции price() и являются для нее автоматическими.

Листинг 7.6. Передача параметров.

```
/*discount.c*/
main()
```

```
        {
        float cost, discount;
        printf("Введите стоимость единицы товара: ");
        scanf("%f", &cost);
        printf("Введите процент скидки
         (в виде десятичной дроби): ");
        scanf("%f", &discount);
        price(cost, discount);
        }
price(amount, mrkdown)
float amount, mrkdown;
        {
        float reduced, net;
        reduced = amount * mrkdown;
        net = amount - reduced;
        printf("Стоимость с учетом скидки
         составляет %.2f долларов", net);
        return (0);
        }
```

Функция price() умножает цену товара на процент скидки, вычитает полученную сумму из цены и выводит на дисплей величину стоимости с учетом скидки. Врезультате выполнения программы на экране монитора появляются следующие сообщения:

Введите стоимость единицы товара: 100

ведите величину скидки (в виде десятичной дроби): 0.05

Стоимость с учетом скидки составляет 95 долларов

Предположим, что случайно вы изменили вызов функции следующим образом:

```
price(discount, cost);
```

Компилятор не сообщит об ошибке, так как значения двух переменных типа float передаются двум аргументам типа float. К несчастью, они передаются не тем аргументам, каким положено: значение discount будет передано amount, а значение cost— mrkdown.

Если теперь присвоить переменной cost значение 100, а переменной discount значение 0.05, функция переставит их и будет считать, что цена равна пяти центам, а размер скидки составляет 10000 процентов. В результате мы увидим, что товар имеет отрицательную стоимость в размере –4.95доллара, вместо 95.50.

Возвращаемые значения

Функция может как получать, так и возвращать значения. Для получения значения, возвращаемого функцией getchar(), нужно сделать такую запись:

```
key = getchar();
```

Приведенная инструкция вызывает функцию getchar(), которая вводит единичный символ с клавиатуры. После выполнения ввода символ присваивается в качестве значения переменной с именем key. Это и есть возврат значения.

Ваши собственные функции также могут возвращать значения в функции, из которых они были вызваны. Если вы хотите возвратить значение, необходимо добавить в тело функции несколько дополнительных элементов. Посмотрите на следующую программу:

```
main()
        {
        char letter;
        letter = getlet();
```

```
        putchar('\n');
        printf("Вы ввели символ %c", letter);
        }
char getlet()
        {
        printf("Введите символ: ");
        return(getchar());
        }
```

Инструкция

```
letter = getlet();
```

вызывает функцию getlet(). Эта функция принимает символ с клавиатуры и возвращает его переменной letter в функции main().

Для того чтобы возвратить значение переменной, необходимо указать компилятору тип возвращаемого значения. Это осуществляется определением типа перед именем функции

```
char getlet();
```

В данной инструкции мы указали компилятору, что значение, возвращаемое функцией getlet(), относится к символьному типу. Значение, которое будет возвращено, указывается в круглых скобках после ключевого слова return(). Инструкция, записанная в строке

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

Замечания по Си++

Если компилятор Си следует стандарту ANSI, вы должны указать тип функции даже в том случае, если функция не возвращает никаких значений. При этом тип определяется как void:

```
void myfunct()
```

Эта информация говорит компилятору, что функция не будет возвращать вызывающей функции никакого значения. В Си++ введение ключевого слова void является не обязательным, но желательным.

выполняет большую часть работы. Запомните, что функция getchar() в инструкциях применяется там, где возможно использование выражения или некоторого значения. В рассматриваемой программе функция getchar() вводит

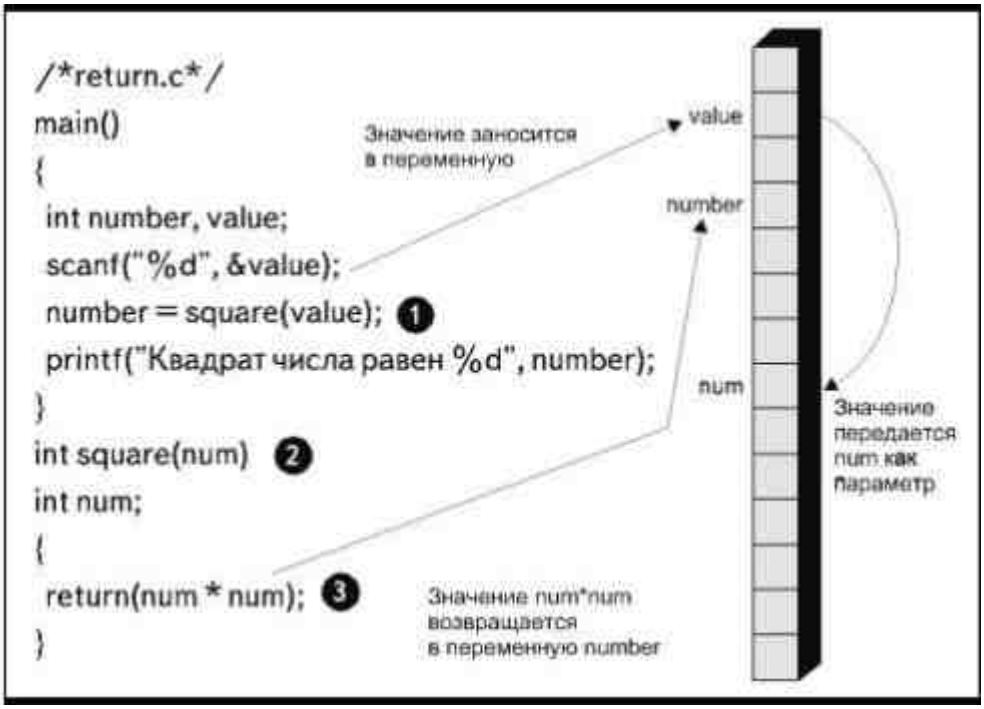

Рис. 7.7. Программа, иллюстрирующая возврат значения из функции

символ, затем инструкция return() после выполнения функции printf() передает символ назад, присваивая его переменной letter, и возвращает управление в main().

Обратите особенное внимание на работу ключевого слова return() в этом примере. Удостоверьтесь, что вы действительно поняли, как используются return() и функция getchar().

Давайте теперь рассмотрим программу, в которой значение и передается, и возвращается. Программа, изображенная на рис.7.7, вводит с клавиатуры значение в переменную, а затем выводит на экран квадрат этого значения.

Инструкция

number = square(value);

вызывает функцию square(), передавая ей значение переменной value. Определение функции как

Замечания по Си++

При работе с компиляторами языка Си++ (и некоторыми Си-компиляторами) желательно начинать программу с прототипов функций. Прототип— это строка определения функции, повторенная в начале программы перед main(). Выглядит она так:

void square(int num);

main()

Прототип сообщает компилятору типы и количество аргументов тех функций, которые будут использованы в программе.

int square(num)

сообщает компилятору Си, что square() возвращает целочисленное значение и что она получит аргумент в переменную num. В этом примере return() является только инструкцией функции. Строка

return(num * num)

возвращает значение выражения num * num, записывая его в переменную number, которой присвоен вызов функции square() в main().

Возврат значений типа float

В тех случаях, когда значения, возвращаемые функцией, относятся к типу целочисленных или символьных, определение типа перед именем функции не является строго обязательным. Си изначально построен таким образом, чтобы воспринимать только данные типа int или char, так что, если тип не указан, Си посчитает, что возвращаемое значение относится к типу int или char.

Замечания по Си++

Перегрузка— это процесс в Си++, позволяющий операторам и функциям работать одновременно с данными разных типов. Можно использовать одно и то же имя для нескольких функций, как показано в следующих прототипах:

int doubles(int num);

float doubles(float num);

В программе будут присутствовать две функции doubles(), одна из которых удваивает значения данных типа int, а другая— данных типа float.

```
/*float.c*/
float square();
main()
{
 float number, value;
 printf("Введите число, \
квадрат которого хотите вычислить:");
 scanf("%f", &value);
 number = square(value);
 printf("Квадрат числа равен %.2f", number);
}
float square(num)
float num;
{
 return(num * num);
}
```

Определяет тип функции как float

Определяет тип возвращаемого значения как float

Определяет функцию

Определяет тип принимаемого значения как float

Рис. 7.8. Определение функции типа float

Если возвращаемые данные относятся к числам с плавающей точкой, необходимо сделать две вещи:

1. Указать тип float перед именем функции.
2. Определить саму функцию.

Функция определяется перед main() так же, как внешняя переменная. На рис.7.8 показано, как это сделать. Инструкция

float square();

указывает компилятору, что он будет иметь дело с функцией типа float. За исключением определения функции и ввода данных другого типа, приведенная на рисунке программа идентична программе вычисления квадрата целых чисел.

Большинство компиляторов позволяет определять тип функции и внутри main():

```
main()
        {
        float number, value, square();
```

Если ваш компилятор не позволяет этого делать, определяйте функцию всегда перед main().

Использование return() в функции main()

Возможно, вы задумались над тем, что означает запись return(0) в функции main(). Обычно мы используем эту инструкцию, чтобы возвратить значение функции, но куда же мы передаем 0, когда программа заканчивается? Ответ прост: мы возвращаем его операционной системе.

При запуске программы на языке Си можно считать, что операционная система вызывает функцию main(). Когда программа заканчивает выполнение, инструкция return() возвращает управление в систему. Пара-

метр инструкции return() может, например, сообщать системе, имела ли место ошибка и что это была за ошибка. В этом случае запись return(0) сообщит, что ошибок не было. Некоторые программы могут возвращать и другие значения с тем, чтобы проинформировать операционную систему о возникших во время выполнения ошибках. В этом случае появится возможность выполнения дополнительных действий в зависимости от того, каким образом была прекращена работа программы*.

Использование макроопределений

Вы уже знаете, что, используя директиву #define, можно задавать константы. Например, если написать инструкцию #define PI 3.14, компилятор подставит значение3.14 на место всех встречающихся в программе констант PI.

Директива #define предписывает компилятору заменить имя константы на то, что следует за этим именем. Если после имени константы ввести какую-нибудь инструкцию, компилятор тоже произведет подстановку. Например, в следующей инструкции мы подставляем на место константы ENTER функцию printf():

```
#define ENTER printf("Пожалуйста, введите число: ")
```

Теперь, при необходимости отобразить сообщение, записанное в аргументе функции printf(), достаточно в соответствующем месте программы использовать инструкцию ENTER:

```
#define ENTER printf("Пожалуйста, введите число: ")
main()
        {
        int age, size;
        ENTER;
        scanf("%d", &age);
        ENTER;
        scanf("%d", &size);
        }
```

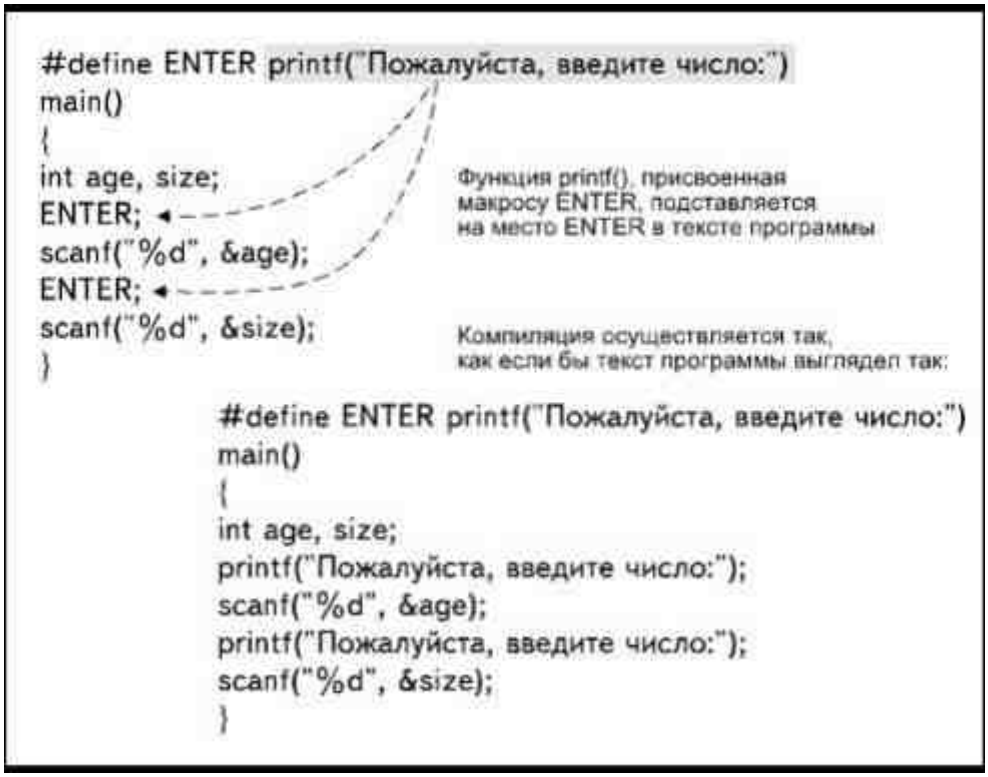

Рис. 7.9. Использование макроопределения

При выполнении программы сообщение «Пожалуйста, введите число:» будет появляться на экране точно так же, как если бы в main() была полностью написана инструкция, содержащая функцию printf() (рис.7.9).

Директивы, подобные той, которую мы только что рассмотрели, называются *макроопределениями или макросами**. Они являются очень мощным средством, позволяющим избежать необходимости вручную вводить одну и ту же инструкцию несколько раз в одной программе. Например, макрос ENTER можно использовать всякий раз, когда нужно подсказать пользователю, что он должен ввести данные. Еще более

мощным программным средством макроопределения делает то обстоятельство, что им можно передавать аргументы, так же, как функциям. В приведенной ниже программе, например, макрос CONVERT используется для перевода значения температуры из шкалы Фаренгейта в шкалу Цельсия:

```
#define ENTER printf("Пожалуйста,
            введите значение температуры: ")
#define CONVERT(temp) (5.0 / 9.0) * (temp - 32)
main()
        {
        float climate;
        ENTER;
        scanf("%f", &climate);
        printf("это соответствует %f по
                шкале Цельсия\n", CONVERT(climate));
        }
```

*** В литературе также используется термин *макроподстановка. (Прим.перев.)***

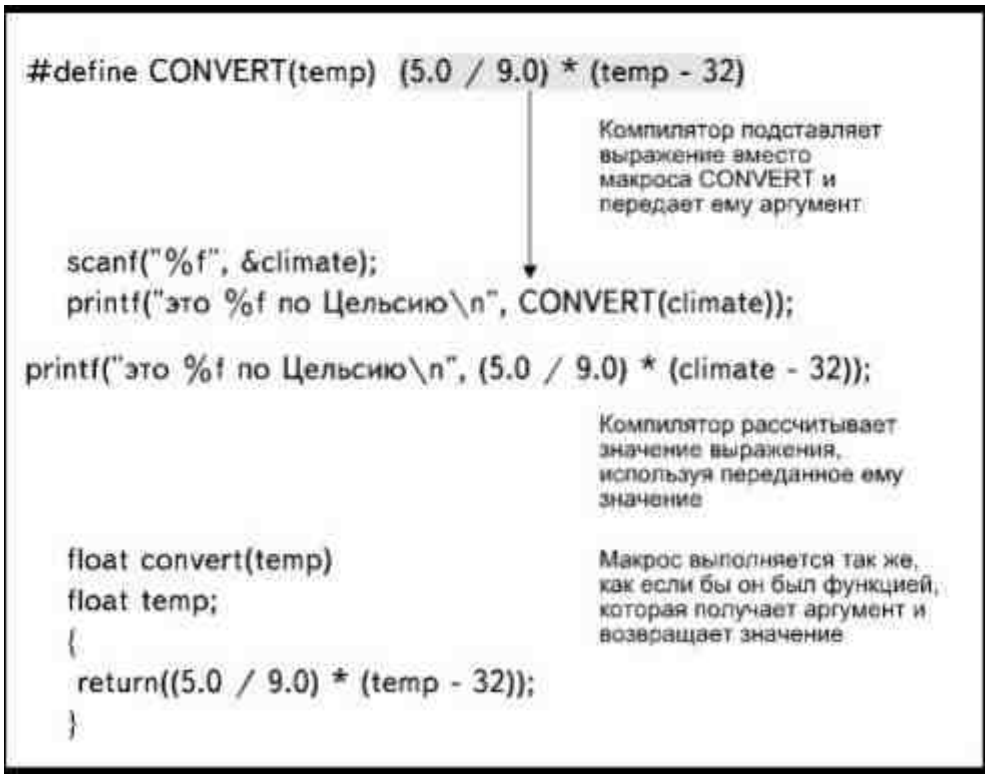

Рис. 7.10. Определение выражения как макроса

Во второй директиве #define определяется макрос CONVERT, который требует передачи ему одного значения. Аргумент, принимаемый макросом CONVERT, подставляется в выражение (5.0 / 9.0) * (temp - 32) на место слова temp. Вызов макроса осуществляется способом, аналогичным вызову функции, с использованием в качестве аргумента значения, которое мы хотим преобразовать (рис.7.10). Если в ответ на запрос введено значение212, происходит вызов CONVERT в функции printf() и расчет значения выражения (5.0 / 9.0) * (212 - 32).

Использование макросов имеет те же преимущества, что и использование констант. Например, если вы захотите заменить стандартный запрос ввода данных, это делается простым внесением изменений в макроопределение ENTER в начале программы. Если вы сделали ошибку в формуле перевода температуры, вам придется отредактировать только одну строку программы, а не все места, где встречается имя макроопределения.

Проектирование программы

В целом, не существует никаких строгих правил, указывающих, когда следует использовать функции, а когда можно поместить все инструкции в функцию main(). Со временем, когда вы приобретете опыт, решение будет возникать само собой. Если вы пишете короткую программу, где вводится одно-два числа, выполняются математические операции и результат выводится на экран, то можно ограничиться функци-

ей main(). В самом деле, было бы глупо делить программу, текст которой включает пять или десять строк, на функции без особой на то необходимости.

С другой стороны, если вы написали длинную программу, поместив все ее инструкции в main(), и оказалось, что где-то закралась ошибка, обнаружить ее будет весьма непросто. Если же программа разделена на функции, вы можете начать диагностику с вопроса: «Какая функция с наибольшей вероятностью является источником проблемы?» и искать ошибку в отдельных частях программы, начав с наиболее вероятного места ее нахождения.

Автоматические или внешние переменные?

Многие начинающие программисты очень часто ограничиваются определением только внешних переменных, надеясь избежать сложностей, связанных с передачей и получением аргументов. Например, в программе, приводившейся в Листинге 7.3, переменная temp используется четырьмя различными функциями, а когда переменная определена как внешняя, нет необходимости передавать ее значение в качестве аргумента.

Хотя использование внешних переменных кажется более простым, их применение может привести к трудно обнаруживаемым ошибкам выполнения и получению неправильных результатов. Поскольку внешняя переменная может использоваться всеми функциями программы, то любая из них может изменить значение этой переменной. Если в результате выполнения программы были получены неправильные значения, вам придется исследовать весь текст на предмет поиска ошибки. Безобидная на вид инструкция, затерявшаяся где-нибудь в редко используемой функции, может катастрофически изменить значение переменной.

Использование автоматических переменных и передача значений в качестве аргументов позволит вам более успешно управлять ходом программы. Значение автоматической переменной может изменить только функция, в которой она определена. Если результат работы программы оказался неправильным, вам достаточно ввести дополнительные функции printf() для отображения значения каждой локальной переменной. Это легко позволит вам определить местонахождение ошибочных инструкций.

Неправильный ввод

Программы, представленные в этой главе, были отобраны таким образом, чтобы продемонстрировать основные принципы использования функций. Эти программы не обязательно показывают лучший способ выполнения конкретной задачи и не имеют защиты от случайных ошибок.

Например, как бы хорошо вы ни составили текст программы, вы не можете контролировать ввод пользователем данных с клавиатуры. В программе, приведенной в Листинге 7.6, стоимость единицы товара и процент скидки вводятся с использованием функции scanf(). Расчет стоимости с учетом скидки основан на предположении, что процент скидки введен в виде десятичной дроби, например, 0.05 для 5 процентов. Чтобы гарантировать правильный ввод этого значения, в запрос ввода данных добавлено указание:

Введите размер скидки (в виде десятичной дроби):

Тем не менее, пользователь может ошибочно ввести число 5 вместо 0.05. Если он так и поступит, то, вместо того чтобы уменьшить стоимость на 5 процентов, программа уменьшит ее на 500 процентов и сообщит, что, оказывается, покупатель должен получить с продавца деньги за свою покупку. Это нехорошо.

В собственной программе никогда не полагайтесь на то, что ввод будет неизменно соответствовать указанному формату. Вы должны быть готовы к тому, что пользователь введет данные, не соответствующие тем, которые ожидаются программой. Как предотвратить некоторые из возможных ошибок, вы узнаете в главе 8.

Вопросы

1. В чем заключаются различия между библиотечными функциями языка Си и функциями, которые вы пишете сами?
2. Всегда ли вызов функции осуществляется из функции main()?
3. Что происходит, когда заканчивается выполнение функции?
4. Объясните разницу между автоматическими и внешними переменными.
5. В чем заключаются преимущества автоматических и в чем— внешних переменных?
6. В каких случаях используются статические переменные?

7. Как передать значение функции?
8. Как получить значение от функции?
9. Что такое макрос?
10. Как происходит возврат значения типа float?

Упражнения

1. Напишите программу Опросника, в котором задаются четыре вопроса; каждый вопрос и ответ оформите в виде отдельной функции.
2. Напишите программу, в которой вводится число, а затем вызывается функция для расчета и отображения четвертой степени этого числа.
3. Внесите изменения в программу из упражнения 2 так, чтобы функция вычисляла четвертую степень числа, а затем передавала результат в main() для вывода на дисплей.
4. Объясните, почему следующая программа написана неверно:

```
5.    dothis()
6.    {
7.    puts("Это первое");
8.    main()
9.    return;
10.   }
11.   main()
12.   {
13.   puts("Это второе");
14.   return(0);
      }
```

ГЛАВА 8

ПОЗВОЛЬТЕ КОМПЬЮТЕРУ ПРИНИМАТЬ РЕШЕНИЯ

Начиная с этого момента, ваше знакомство с основами программирования будет продвигаться вперед семимильными шагами. Сказанное совершенно не означает, что вопросы, освещаемые в этой и последующих главах, будут более сложными, чем те, в изучении которых вы уже преуспели. Вовсе нет. Но когда вы соедините то, что вам предстоит изучить теперь, с тем, что вы уже знаете, то подниметесь на новый уровень в постижении премудростей программирования.

Начиная с этой главы, мы будем придавать особое значение изучению логики построения программы. Так как вы уже достаточно хорошо знакомы с синтаксисом и структурой языка Си/Си++, то отметите эту смену акцентов. Теперь, вместо того чтобы тратить время на каждую точку с запятой или скобку, вам предлагается сконцентрировать внимание на алгоритмах и способах решения проблем. Вы узнаете, что обычно не существует «единственно верного» способа написать программу, решение одной и той же задачи может идти различными путями.

В этой главе рассказывается, как сделать так, чтобы компьютер мог принимать решения. Вместо того чтобы выполнять все инструкции в том порядке, в каком вы их написали, программа будет выбирать, какую из инструкций ей следует выполнить в том или ином случае, основываясь на введенных вами критериях. Фактически, вы сможете использовать приемы, которые будут описаны в этой главе для того, чтобы разрешить большинство логических проблем, с которыми столкнулись при чтении предыдущих глав.

If — маленькое слово с большими возможностями

Все эти чудеса выполняются с помощью ключевого слова if. Это короткое слово имеет, однако, большой вес. Инструкция if используется в тех случаях, когда необходимо решить, должна ли быть выполнена конкретная инструкция программы (if по-английски значит «если»). Структура if выглядит следующим образом:

```
if (condition)
instruction;
```

Этой записью мы говорим: «Если некоторое условие выполняется (является истинным), инструкция должна быть выполнена» (рис.8.1). То есть, компьютер, встретив ключевое слово if, выполняет инструкцию, следующую за ним, если условие в скобках является истинным.

Рис. 8.1. Структура инструкции if

Если условие не выполняется, компьютер пропускает инструкцию, записанную после if, и переходит к следующим строкам программы. Использование if не будет вызывать у вас затруднений, как только вы запомните основные моменты:

- условие заключается в круглые скобки;
- точку с запятой ставят не после условия, а только в конце инструкции;
- Си относится к языкам свободного формата, поэтому условие и инструкцию можно помещать в одной строке. Разделяя их, мы просто делаем программу более удобной для чтения.

Условия

Условием в инструкции if является сравнение значений: значение переменной или константы сравнивается с литералом, или со значением другой переменной или константы. Сравнение выполняется с помощью одного из следующих *операторов отношения*:

Оператор Значение Пример

==	равно	if (tax == 0.06)
>	больше	if (hours > 40)
<	меньше	if (hours < 40)
>=	больше или равно	if (salary >= 10000)
<=	меньше или равно	if (cost <= limit)
!=	не равно	if (count != 1)

Обратите внимание: когда вы хотите узнать, равны ли два значения друг другу, то должны использовать оператор отношения, состоящий из двух знаков равенства (==) подряд. Если поставить только один знак равенства, компилятор сгенерирует предупреждение, или, реже, ошибку. Единичный знак равенства (=) используется для обозначения присваивания значения переменной.

Простейшая инструкция с использованием if выглядит примерно так:

```
if (time > 11)
        puts("Уже поздно, ступайте домой.");
```

Здесь говорится: «Если значение переменной time больше 11, тогда следующее сообщение должно быть выведено на дисплей». Если значение переменной time окажется меньше 11, сообщение не появится.

С помощью условия if можно проверять значения числовых или символьных переменных, но не строк. Например, при компиляции следующего фрагмента программы компилятор не сообщит об ошибке, но и нужный результат тоже не будет получен:

```
gets(name);
if (name == "Адам")
        puts("Позвоните домой");
```

Строковые переменные являются темой отдельного разговора и подробно обсуждаются в главе 10.

В программе, приведенной в Листинге 8.1, используется инструкция if. Эта программа является вариантом программы, которую мы уже видели в предыдущей главе. В ней рассчитывалась общая стоимость наименования товара с учетом налога на продажи. Здесь добавлен расчет специального налога на предметы роскоши для товаров, цена которых превышает 40000 долларов. Расчет этого налога выполняется в инструкции:

```
if (cost > 40000.00)
        luxury = cost * 0.005;
```

Листинг 8.1. Программа расчета стоимости товаров с учетом налога на предметы роскоши.

```
/*luxury1.c*/
main()
        {
        float cost, tax, luxury, total;
        luxury = 0.0;
        printf("Введите цену товара: ");
        scanf("%f", &cost);
        tax = cost * 0.06;
        if (cost > 40000.00)
                luxury = cost * 0.005;
        total = cost + tax + luxury;
        printf("Стоимость единицы товара
        с учетом налогов составляет %.2f", total);
        }
```

В этих инструкциях говорится: «Если значение переменной cost больше 40000, то переменной luxury присваивается значение, равное значению cost, умноженному на 0.5 процента». Последние две строки инструкций в программе выполняются в любом случае, независимо от того, присутствует налог на предметы роскоши или нет, так как эти строки расположены после точки с запятой, завершающей инструкцию if.

Заметьте, что в начале программы переменной luxury присваивается начальное значение, равное 0, в отличие, например, от переменной tax. Причина в том, что переменной tax в любом случае будет присвоено значение, полученное в результате выполнения математических операций tax = cost * 0.06, а расчет и присваивание значения переменной luxury осуществляется только в том случае, если стоимость превышает 40тысяч долларов. Если условие не соблюдается, расчет не производится. В этом случае, если бы переменная luxury не была инициализирована, в выражение расчета общей стоимости было бы подставлено случайное значение, хранящееся в памяти. Поэтому представляется в высшей степени разумным, если вы всегда будете присваивать начальное значение переменным, которые используются в инструкции if, так же, как это делается при использовании счетчика и аккумулятора.

Используя операторы отношения больше и меньше, удостоверьтесь, что с их помощью вы составили именно те условия, которые необходимы. В приведенном примере налог на предметы роскоши добавляется только в том случае, если цена товара хотя бы на один цент превышает 40 тысяч долларов, то есть если вещь стоит 40 тысяч долларов 1 цент и больше. Если стоимость товара составляет ровно 40 тысяч долларов, налог на предметы роскоши не добавляется. Если бы налогом облагались товары, стоимостью 40 тысяч долларов и выше, следовало бы указать условие:

```
if (cost >= 40000.00)
```

Разница между операторами больше и больше или равно либо меньше и меньше или равно с виду кажется незначительной, но она может весьма существенно сказаться на результатах, получаемых от программы.

Составные инструкции

В общем случае if выполняет только одну инструкцию. Если возникает необходимость, чтобы при выполнении одного условия выполнялось несколько команд, следует использовать *составную инструкцию*. Составной инструкцией называется последовательность любых инструкций, заключенных в фигурные скобки. С точки зрения синтаксиса языка такая последовательность будет рассматриваться как единая инструкция.

В качестве примера рассмотрим программу, текст которой приведен в Листинге8.2. Здесь при истинности одного условия выполняются две инструкции. Открывающая фигурная скобка после условия указывает начало составной инструкции, а закрывающая фигурная скобка в конце второй инструкции обозначает ее конец. Все инструкции, помещенные внутри фигурных скобок, будут выполнены только при истинности соответствующего условия. Табуляцию мы использовали только для того, чтобы подчеркнуть, что обе инструкции являются частями инструкции if. Для компилятора же табуляция не содержит никакой информации.

Листинг 8.2. Составные инструкции.

```
/*luxury2.c*/
main()
        {
        float cost, tax, luxury, total;
        luxury = 0.0;
        printf("Введите цену товара: ");
        scanf("%f", &cost);
        tax = cost * 0.06;
        if (cost > 40000.00)
                        {
                        luxury=cost*0.005;
                        printf("Сумма налога на предметы
                роскоши составляет %.2f\n", luxury);
                        }
        total = cost + tax + luxury;
        printf("Стоимость единицы товара с учетом
                налогов составляет %.2f", total);
```

```
}
```

Конструкция if...else

Сама по себе инструкция if используется в тех случаях, когда важно выполнить некие действия при истинности определенного условия. Возможна ситуация, когда при истинности условия должен быть выполнен один набор инструкций, а в противном случае— другой. Допустим, мы хотим, чтобы в тех случаях, когда цена какого-либо товара ниже той, для которой установлен налог на предметы роскоши, программа генерировала сообщение: «Для данного наименования налог на предметы роскоши не установлен». Это можно сделать с помощью двух инструкций if:

```
if (cost > 40000.00)
        {
        luxury = cost * 0.005;
        printf("Размер налога на предметы роскоши для \
 данного наименования составляет %.2f", luxury);
        }
if (cost < 40000.00)
        puts("Для данного наименования налог \
 на предметы роскоши не установлен");
```

Но есть более эффективный способ. Можно объединить обе инструкции в одну, пользуясь тем, что есть только два возможных случая в использовании одной и той же переменной: либо цена товара больше 40 тысяч долларов, либо цена товара меньше или равна указанной сумме. Если одно из условий не выполняется, следовательно, выполняется второе условие, так что можно

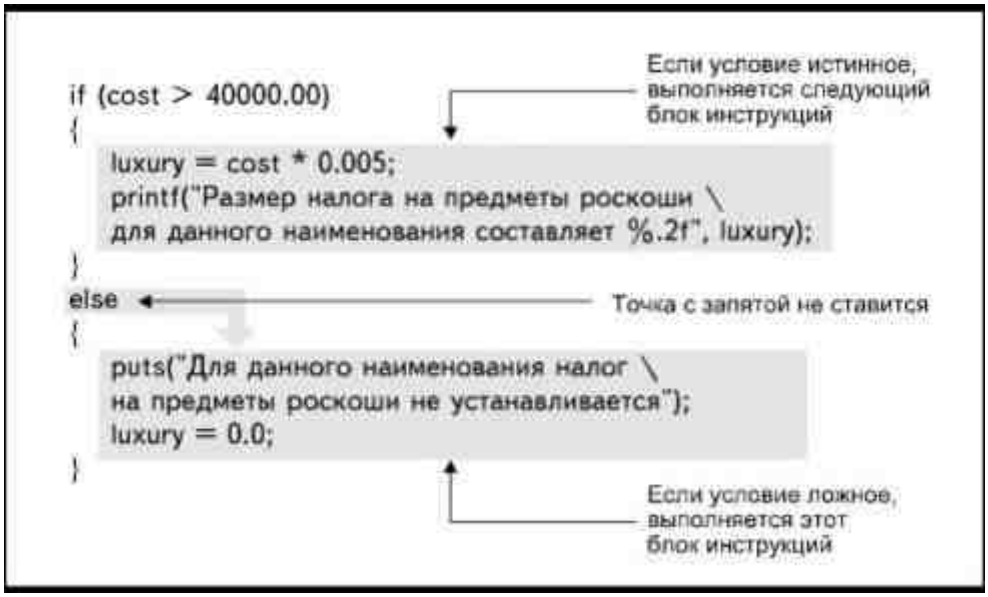

Рис. 8.2. Инструкции, модифицированные с использованием ключевого слова else

скомбинировать их, используя ключевое слово else (которое переводится как «иначе»):

```
if (condition)
        instruction;
else
        instruction;
```

Здесь сказано: «Если условие истинное, то должна быть выполнена команда, являющаяся частью инструкции if, иначе надо выполнить инструкцию, следующую за else». Инструкция, помещенная после ключевого слова else, выполняется только в том случае, если условие оказалось ложным. Если возникает необходимость выполнить в этом случае несколько инструкций, можно использовать составную инструкцию, заключив ее в фигурные скобки точно так же, как для if. Точка с запятой ставится в конце каждой инструкции и не ставится после ключевого слова else.

Для того чтобы вывести на экран сообщение об отсутствии налога на предметы роскоши, программу можно слегка изменить, как это показано на рис.8.2. Обратите внимание, что в этом случае нет необходимо-

сти непременно присваивать переменной luxury начальное значение, так как в инструкции if теперь учитываются все возможные варианты условия.

Дополненный Опросник

В главе 7 мы предложили в качестве примера несколько программ, которые выводили на экран монитора вопросы и ответы. Поскольку вы тогда еще не познакомились с инструкцией if, то не имели возможности вести подсчет очков за правильные ответы. Подсчет очков, как демонстрирует программа из Листинга8.3, представляет собой, по существу, сравнение правильного ответа, заложенного в тексте программы, и ответа, введенного с клавиатуры.

В этой программе используются функции, которые выводят вопрос на экран монитора, вводят с клавиатуры ответ пользователя, определяют правильность ответа и подсчитывают количество верных и ошибочных ответов. И вопрос, и правильный ответ передаются функции— вопрос в виде строки литералов, ответ в виде целого числа. Программа построена таким образом, чтобы в нее можно было при желании добавлять вопросы, дописывая инструкции вызова функции ask():

```
ask("9+5= ", 14);
```

Листинг 8.3. Опросник с подсчетом очков.

```
/*score*/
int correct, wrong;
main()
        {
        char question[15];
        int answer;
        correct = 0;
        wrong = 0;
        ask("4 + 5 = ", 9);
        ask("6 + 2 = ", 8);
        ask("5 + 5 = ", 10);
        ask("4 + 7 = ", 11);
        printf ("Количество верных ответов: %d.\n", correct);
        printf ("Количество неверных ответов: %d.\n", wrong);
        }
ask(quest, ans)
char quest[15];
int ans;
        {
        int guess;
        printf("%s", quest);
        scanf("%d", &guess);
        if (guess == ans)
                ++correct;
        else
                ++wrong;
        return(0);
        }
```

Логические операторы

Как вы уже могли заметить в приведенных выше примерах, инструкция if проверяет выполнение условия только для одной переменной и одного значения. Значит, в инструкции можно ввести только одно условие

с целью проверки его истинности. На самом деле часто возникает необходимость проверить в условии более одного значения.

Посмотрите на программу, приведенную в Листинге8.4. В ней предполагается, что не каждая единица продаваемого товара облагается налогом на продажи. Поэтому, вместо того чтобы автоматически добавлять сумму налога к стоимости каждого наименования товара, программа спрашивает, должен ли данный товар облагаться налогом, и если да— добавляет сумму в размере 6 процентов от стоимости товара.

Листинг 8.4. Работа программы основывается на указаниях пользователя.

```
/*iftax.c*/
main()
        {
        int taxable;
        float cost, tax;
        tax = 0.0;
        printf("Введите цену товара: ");
        scanf("%f", &cost);
        printf("Введите Y, если товар облагается
         налогом, N, если не облагается: ");
        taxable = getchar();
        if (taxable == 'Y')
                tax = cost * 0.06;
        printf("\nСтоимость товара с учетом
         налога составляет %f", cost + tax);
        }
```

Если на ваш взгляд программа написана вполне корректно, посмотрите ее текст еще раз более внимательно— там есть серьезное упущение. Программа написана так, что пользователь должен в ответ на запрос ввести прописную букву Y, если товар облагается налогом. Если пользователь вводит строчную букву y, программа будет считать, что налог в данном случае не взимается, ведь в инструкции if в качестве правильного условия рассматривается только прописная буква.

Замечания по использованию функции getchar()

Некоторые компиляторы помещают значения, введенные функцией getchar(), в *буфер*. Это означает, что введенный символ хранится в памяти компьютера до тех пор, пока пользователь не нажмет клавишу Enter. Проверьте документацию вашего компилятора, чтобы выяснить, использует ли он функции getch() или getche(). Как правило, введенные с их помощью значения не помещаются в буфер, так что можно ввести значение простым нажатием нужной клавиши, без нажатия **Enter**.

Кроме того, использование функции getchar() после ввода с помощью scanf() может создать дополнительную проблему. В главе 5 мы говорили, что данные, формат которых не соответствует ожидаемому (указанному в строке формата), функция scanf() игнорирует, и они остаются в буфере. Функция getchar() может прочитать один из этих символов еще до того, как будет введен нужный символ. Можно избежать подобной опасности, если вместо scanf() использовать функцию gets() либо, если ваш компилятор позволяет это, применить функции ввода символов без буферизации, такие как getch() или getche().

Другим возможным решением является очистка буфера перед каждой функцией getchar(). Добавьте в начало программы директиву #include , а затем вставьте функцию fflush(stdin) перед вызовом getchar(). Функция fflush() удаляет все символы, которые могли остаться в буфере стандартного устройства ввода.

В подобной ситуации правильнее было бы проверять оба возможных варианта ввода, то есть и строчную и прописную буквы Y. Можно сделать это с помощью двух инструкций if. А можно использовать логический оператор ИЛИ, который выглядит как две вертикальные черты:

```
if (taxable == 'Y' || taxable == 'y')
```

В данной инструкции сказано: «Если переменная taxable имеет значение Y ИЛИ y, то...» Таким образом, мы добьемся того, что товар будет рассматриваться как облагаемый налогом, если выполняется одно из этих двух условий. Если не выполняется ни одно из них, то есть пользователь ввел любой другой символ, товар будет считаться не облагаемым налогом. Условие должно быть целиком помещено в круглые скобки, причем имя переменной taxable следует повторить дважды. Запись условия как (taxable == 'Y' || 'y') приведет к ошибке компиляции.

Есть три логических оператора: ИЛИ (||), И (&&) и отрицания (!). Оператор ИЛИ означает, что для выполнения инструкции if достаточно истинности одного из двух (или обоих вместе) заданных условий. Оператор И указывает на то, что должны быть истинными оба условия одновременно. Оператор отрицания означает, что инструкция if выполняется, если некое условие оказалось ложным.

Операторы И и ИЛИ можно использовать не только для проверки равенства переменной одному из двух значений (как мы уже делали), но и для тестирования значений различных переменных. Например, вы пишете программу, в которую вводится размер годового дохода пользователя и количество иждивенцев (Листинг8.5).

Листинг 8.5. Проверка значений двух переменных.

```
/*twovars.c*/
main()
        {
        int depents;
        float income;
        puts("Укажите сумму Вашего годового дохода");
        scanf("%f", &income);
        puts("Пожалуйста, укажите количество иждивенцев");
        scanf("%d", &depents);
        if (income < 20000 && depents > 2)
                puts("Вы освобождены от уплаты
         подоходного налога");
        }
```

В условии if здесь проверяются значения двух переменных: income и depents. Для того чтобы сообщение, записанное в инструкции if, оказалось выведенным на экран, значение переменной income должно быть меньше 20 тысяч долларов, И одновременно значение переменной depents должно быть больше двух. Если хотя бы одно из перечисленных условий не выполняется, например, количество иждивенцев равно одному или доход составляет 20001 доллар, функция puts() не будет выводить на экран сообщение об освобождении от уплаты налога.

Будьте очень внимательны, когда используете оператор И, чтобы быть уверенным, что ваша программа работает нужным образом. Например, никогда не используйте оператор И для проверки двух альтернативных значений одной переменной. Условие

```
if (taxable == 'Y' && taxable == "y")
```

никогда не будет выполнено, так как одна переменная не может одновременно иметь два значения. Тем не менее, используя оператор И, можно проверить, находится ли значение одной и той же переменной в определенных границах допустимых значений.

Для примера предположим, что налогом на предметы роскоши облагаются товары, цена которых находится в пределах от 40 тысяч до 60 тысяч долларов. Условие проверяется следующим образом:

```
if (cost >= 40000.00 && cost <= 60000.00)
```

Для того чтобы выполнялась инструкция if, должны быть истинными оба условия одновременно, так как мы использовали оператор И. При этом цена товара должна быть в одно и то же время больше или равна 40000 и меньше или равна 60000 долларов, то есть находится в пределах определенных допустимых значений (рис.8.3).

Если в данном случае использовать оператор ИЛИ, то это приведет к ошибочному выполнению инструкции, так как любое значение переменной cost будет рассматриваться как удовлетворяющее условию. Напротив, если мы хотим

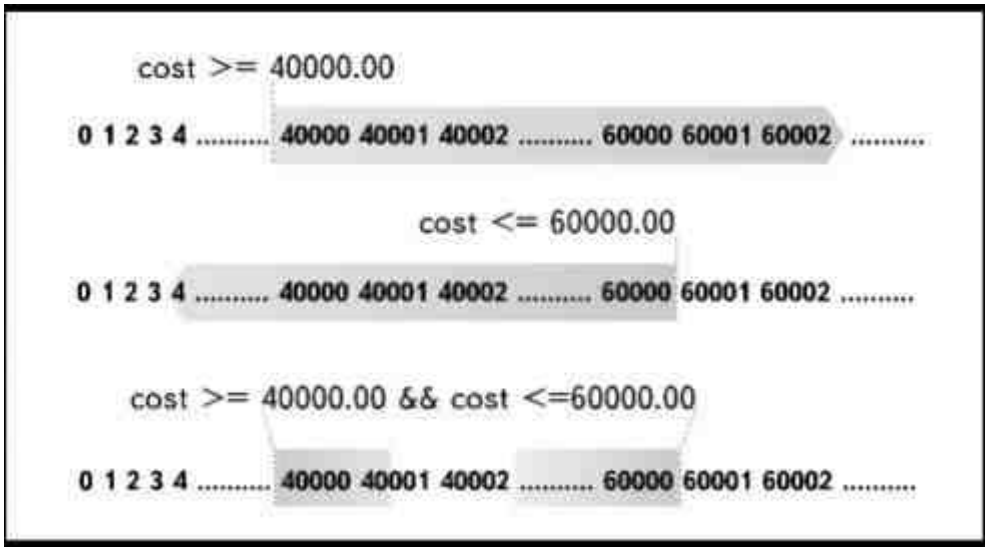

Рис. 8.3. Так можно узнать, находится ли число в границах определенных значений

определить, находится ли значение вне каких-то установленных границ значений, мы должны использовать именно оператор ИЛИ и поменять местами операторы «больше или равно» и «меньше или равно». При такой записи условие if будет проверять, выходит ли значение числа за указанные рамки, определяя, меньше это число или больше установленных ограничивающих значений:

```
if (income <= 20000.00 || income >= 60000.00)
        puts("Вы не относитесь к среднему классу");
```

Оператор отрицания называют *унарным* оператором, так как он работает только с одним объектом, а именно, с переменной или константой. Условие считается не выполненным (ложным) в том случае, когда значение выражения, стоящего в круглых скобках после if, равно0. При любом другом значении выражения, будь оно положительным или отрицательным, полагается, что условие выполнено. Например, следующие инструкции выводят на экран монитора сообщение: «Ошибка в расчете», так как переменная count имеет значение, равное0:

```
int count;
count = 0;
if (! count) puts("Ошибка в расчете");
```

Такая запись условия в точности соответствует строке

```
if (count == 0)
```

Аналогичным образом, следующая инструкция выводит на экран сообщение: «Правильный результат», так как переменная count имеет значение, отличное от нуля:

```
int count;
count = 1;
if (count) puts("Правильный результат");
```

Запись if (count) аналогична записи if (count != 0), которая определяет, отличается ли значение переменной от нуля. В последующих главах вы узнаете, как используются унарные операторы.

Вложенные инструкции if

Конструкция if...else может содержать инструкции любого типа. Они могут включать ввод и вывод значений, выполнение математических операций или вызов собственных функций. Но инструкция в условии может оказаться и другой инструкцией if. В этом случае она будет называться вложенной инструкцией. Ниже приведен пример, где одна инструкция if вложена в другую:

```
if (income > 100000)
        if (status == 'S')
                taxrate = 0.35;
```

Второе условие проверяется только в том случае, если выполнено первое, так что значение переменной taxrate присваивается только при выполнении обоих условий. Ту же самую логическую конструкцию можно было записать следующим образом:

```
if (income > 100000 && status == 'S')
        taxrate = 0.35;
```

Обе инструкции выполняют одну и ту же задачу, но второй способ записи, с использованием оператора &&, кажется более ясным, так как нет необходимости расшифровывать смысл второй инструкции if. Достаточно просто прочитать инструкцию, чтобы понять принцип действия: «Если значение переменной income больше чем 100000 И одновременно переменная status имеет значение "S", переменной taxrate присваивается значение 0.35».

Как правило, любые две последовательно вложенные инструкции if можно заменить одной инструкцией, использующей логический оператор И. Опять же, как правило, имеет смысл избегать вложенных инструкций if, так как они могут приводить к появлению запутанных ситуаций, логических ошибок и трудных для чтения фрагментов текста программы. Посмотрите на следующий пример:

```
main()
        {
        float income;
        scanf("%f", &income);
        if (income >= 20000.00)
                if (income <= 100000.00)
                        puts("Размер Вашего
 подоходного налога составляет 22%");
        else
                puts("Ваш доход меньше
 20000$ - подоходный налог равен 15%");
        }
```

В этом случае логика рассуждений автора программы была, по-видимому, такова: «Если значение переменной income больше 20000 и меньше 100000, следует вывести одно сообщение, а если значение income меньше 20000, должно быть выведено второе сообщение».

Компиляция программы пройдет без ошибок, но, к несчастью, работать она будет неправильно. Когда вы введете значение меньшее, чем 20000, не будет выполняться ни одна из функций puts(), а когда введете значение больше 100000, подоходный налог окажется определенным в размере 15процентов.

Причина ошибки кроется в том, что ключевое слово else связано с ближайшей к нему инструкцией if независимо от отступов, которые были сделаны в тексте программы. В этой программе использование табуляции создает впечатление, что else связано с первым условием if, но это не так. На самом деле else связано со второй инструкцией if, которая выполняется только в тех случаях, когда значение переменной income больше 20000 и меньше 100000.

Если вы хотите, чтобы программа работала в соответствии с вашей логикой, инструкции следует написать примерно таким образом:

```
if (income >= 20000.00)
        {
        if (income <= 100000.00)
                puts("Размер Вашего
 подоходного налога составляет 22%");
        }
else
        puts("Ваш доход не превышает 20 тысяч долларов, \
поэтому размер подоходного налога составляет 15%");
```

Фигурные скобки изолируют вложенную инструкцию, и теперь ключевое слово else действительно будет связано с первой инструкцией if.

Еще лучше написать эту же программу с использованием только одной инструкции if:

```
if (income >= 20000.00 && income <= 100000.00)
        puts("Размер Вашего подоходного
 налога составляет 22%");
else
        puts("Ваш доход не превышает 20 тысяч долларов, \
поэтому размер подоходного налога составляет 15%");
```

В этом варианте мы полностью исключаем путаницу между вложенными инструкциями if и использование дополнительных наборов фигурных скобок.

Даже теперь, после того как мы разобрались с логикой программы, она все еще имеет некоторое упущение. Все сообщения, представляемые программой, относятся к уровню дохода, не превышающему 100 тысяч долларов. В хорошо продуманной программе должны быть приняты во внимание все возможные ситуации. Если в программу, в том виде, в каком она существует сейчас, ввести значение переменной income, равное 150000, программа не выдаст никакого сообщения. Значит ли это, что и подоходный налог платить не обязательно?

Один из способов учета всех возможных вариантов приведен в Листинге8.6. В программе используются вложенные инструкции if...else. Постарайтесь убедиться, что вы действительно понимаете, как они сгруппированы. Если первое условие является истинным (income < 20000.00), выполняется первая функция puts(), а все остальные инструкции пропускаются. Первое ключевое слово else связано с первым if, так что проверка второго условия (income < 100000.00) осуществляется только в том случае, если первое условие оказалось ложным.

Листинг 8.6. Использование вложенных инструкций для учета всех возможных условий.

```
/*brackets*/
main()
        {
        float income;
        puts("Укажите размер Вашего годового дохода: ");
        scanf("%f", &income);
        if (income < 2000.00)
                puts("Размер Вашего
        подоходного налога составляет 15%");
        else
                if (income < 2000.00)
                        puts("Размер Вашего
         подоходного налога составляет 22%");
                else
                        puts("Размер Вашего
         подоходного налога составляет 35%");
        }
```

Обратите внимание на то, что хотя возможны три различных варианта значений переменной, используются только два условия if. При использовании последовательной комбинации if...else требуется написать на одно условие меньше, чем количество возможных вариантов. Действительно, если существует три различных условия, то при невыполнении первого и второго условия обязательно должно выполняться третье, так что нет необходимости вводить третью инструкцию if для проверки его истинности. Но если бы переменная имела четыре возможных варианта значений, следовало бы ввести уже три комбинации if...else.

Конструкция switch/case/default

Если в программе следует учесть больше трех возможных вариантов, конструкция с вложенными инструкциями if...else может оказаться очень запутанной. В таких случаях в качестве альтернативы используется переключатель switch. Переключатель switch представляет собой структуру, построенную по принципу меню, и содержит все возможные варианты условий и инструкции, которые следует выполнить в каждом конкретном случае. Пример подобной конструкции приведен в Листинге8.7.

Листинг 8.7. Программа, в которой используется инструкция switch.

```
/*switch.c*/
main()
        {
        int answer;
        puts("Си это: \n");
        puts("1. Язык, на котором
 говорят на юге Франции\n");
        puts("2. Язык, который используется только
 для написания \
больших компьютерных программ\n");
        puts("3. Компилирующий язык, легко
        совместимый с любыми системами\n");
        puts("4. Ничего из перечисленного\n");
        puts("Введите номер ответа\n");
        answer = getchar();
        putchar('\n');
        switch (answer)
                {
                case '1':
                        puts("К сожалению, Вы ошиблись, \
на юге Франции говорят на языке Паскаль\n");
                        break;
                case '2':
                        puts("Вы ошибаетесь,
 язык Си используют для написания программ\n");
                        puts("любых типов и размеров\n");
                        break;
                case '3':
                        puts("Очень хорошо, Вы
 дали правильный ответ\n");
                        puts("Си является
 компилирующим языком и может использоваться\n");
                        puts("с
         любыми компьютерными системами\n");
                        break;
                case '4':
                        puts("К сожалению,
 Вы ошибаетесь, правильный ответ - номер 3\n");
                        break;
                default:
                        puts("Вы ввели символ,
 не соответствующий \
```

ни одному из номеров ответа\n");

}

В круглых скобках после переключателя switch находится переменная типа int или char, следом расположен блок инструкций, заключенных в фигурные скобки, которые содержат ряд ветвей case. Каждая ветвь case выполняет инструкции, основываясь на возможном значении переменной. Это значение должно быть представлено в виде целого числа или символа, заключенного в одинарные кавычки, либо в виде имени целочисленной или символьной константы.

Ветвь case '1':, например, предписывает программе выполнить следующие ниже инструкции, если значение переменной переключателя switch соответствует символу '1'. Если значение переменной отличается от единицы, компилятор переходит к проверке условия второй ветви case.

В тех случаях, когда значение переменной удовлетворяет условию ветви case, выполняются инструкции, следующие за данным условием. Инструкция break в конце каждой ветви case передает управление в конец switch, так что, как только выполнены инструкции одной из ветвей case, остальные игнорируются и выполнение switch завершается.

Если значение переменной не удовлетворяет условиям ни одной из ветвей case, выполняется ветвь, помеченная инструкцией default. Это дает возможность учесть все возможные варианты ввода. После инструкции default нет необходимости ставить break, поскольку она всегда является последней в процедуре выполнения switch. Имеет смысл включать default даже тогда, когда вы полагаете, что учли все возможные условия и все возможные случаи ввода значений.

Если вы пропустите инструкцию break, компьютер выполнит все инструкции, помещенные в соответствующей ветви case, и далее, вплоть до первого встреченного в тексте break. Вы можете использовать эту особенность для выполнения определенного набора инструкций, при наличии нескольких равноценных вариантов ответа, например, так:

```
case 'Y':
case 'y': puts ("Вы ответили \"Да\"");
        break;
case 'N':
case 'n': puts ("Вы ответили \"Нет\"");
        break;
```

Проверка чисел с плавающей точкой и строк

Мы уже говорили, что в ветви case значение условия должно быть целым числом или символом. Поэтому нельзя написать инструкцию типа case 12.87: или case "Adam". Строки будут подробно рассматриваться в главе 10. Что касается значений типа float, то если вы хотите проверить такое значение, оно должно быть каким-то образом преобразовано в формат целого числа или символа.

В большинстве случаев для преобразования формата можно использовать конструкцию if...else. В Листинге8.8 приведен текст программы, в которой конструкция if...else присваивает целочисленное значение переменной level, основываясь на значении типа float, присвоенном переменной income.

Листинг 8.8. Использование вложенных инструкций if для преобразования чисел с плавающей точкой.

```
/*switch.c*/
main()
        {
        float income;
        char level;
        printf("Введите сумму Вашего годового дохода: ");
        scanf("%f", &income);
        if(income<= 20000.00)
                level = '1';
        else
                if(income<= 60000.00)
                        level = '2';
```

```
                else
        if(income<= 120000.00)
                                        level = '3';
                        else
                                        level = '4';
        switch(level)
                {
                case '1':
                        puts("Ставка налога = 15%");
                        break;
                case '2':
                        puts("Ставка налога = 28%");
                        break;
                case '3':
                        puts("Ставка налога = 32%");
                        break;
                case '4':
                        puts("Ставка налога = 36%");
                        break;
                default:
                        puts("Вы
неправильно ввели сумму дохода");
                }
        }
```

На первый взгляд, кажется неразумным использовать несколько конструкций if...else и переключатель switch, в то время как можно было поместить все инструкции в if...else. Действительно, в простых программах, таких, как эта, можно без труда так и поступить, но когда инструкции, выполняемые в каждом случае, становятся более сложными, имеющими собственные вложенные инструкции if, использование переключателя switch является, несомненно, более удобным. Задание ветвей case позволит легко определить, сколько именно условий следует проверить и какие именно действия должны быть выполнены в каждом случае. Оформление процедуры присваивания значения с помощью серии последовательных конструкций if...else стоит той дополнительной работы, которую придется ради него проделать.

Проектирование программы

Как вы могли убедиться, существует несколько способов создания одной и той же программы: использование нескольких инструкций if, вложенных инструкций if, логических операторов или конструкции switch. В Листинге8.9 в качестве примера показано, как текст программы Опросника (в котором раньше использовался переключатель switch) теперь переделан с использованием вложенных инструкций if...else. В данном случае выбор одного из вариантов определяется вашим личным вкусом. Никто не может сказать, что ваш способ является ошибочным, до тех пор, пока ваша программа делает все, что от нее требуется. Разумеется, если в результате работы программы мы получаем неправильные результаты, значит в ней есть ошибочные инструкции, но при этом совершенно не обязательно, что ошибка была вызвана именно неправильным выбором структуры программы.

Листинг 8.9. Программа Опросника, в которой используются вложенные инструкции if...else.

```
/*quiz4.c*/
main()
        {
        int answer;
        puts("Си это: \n");
        puts("1. Язык, на котором говорят на юге Франции\n");
```

```
puts("2. Язык, который
используется только для написания \
больших компьютерных программ\n");
puts("3. Компилирующий язык,
 легко совместимый с любыми системами\n");
puts("4. Ничего из перечисленного\n");
puts("Введите номер ответа\n");
answer = getchar();
putchar('\n');
if(answer == '1')
        {
        puts("К сожалению, Вы ошиблись.\n");
        puts("На юге Франции
говорят на языке Паскаль\n");
        }
else
        if(answer == '2')
                {
                puts("Вы ошибаетесь, язык Си
 используют для написания программ\n");
                puts("любых типов и размеров\n");
                }
        else
                if(answer == '3')
                        {
                        puts("Очень хорошо, Вы
 дали правильный ответ\n");
                        puts("Си является
компилирующим языком и может использоваться\n");
                        puts("с
 любыми компьютерными системами\n");
                        }
                else
                        if(answer == 4)
                        puts("К сожалению,
Вы ошибаетесь, правильный ответ - номер 3\n");
                        else
                        puts("Вы ввели символ, не
соответствующий номеру ответа\n");
}
```

Выбирайте тот метод, который кажется вам более удобным и позволяет написать более стройную программу без использования излишне усложненных конструкций. Имейте в виду, часто бывает так, что лучший метод требует использования большего количества инструкций, а самая короткая программа не всегда является самой лучшей. И не забывайте о том, что какой бы метод вы ни выбрали для создания программы, всегда следует проверять правильность ввода.

Проверка правильности ввода

Не очень надейтесь на то, что пользователь всегда будет вводить правильное число или символ, даже если можно использовать и прописные и строчные буквы. Например, в главе 7 мы приводили текст программы, в которой требовалось ввести процент скидки в виде десятичной дроби. Все расчеты могут оказаться неверными, если ошибочно ввести 5 вместо 0.05 для пятипроцентной скидки. Одним из способов решения этой проблемы является введение добавочной инструкции:

```
if (mrkdown >1)
        mrkdown = mrkdown / 100;
```

В главе 6 вы видели программу расчета оплаты труда. Если помните, там возникла ситуация, когда программа вычитала из заработной платы сотрудника оплату в двойном размере за каждый час, недостающий до 40-часовой рабочей недели. Программа, в которой этот недостаток исправлен, приведена в Листинге8.10. Инструкция if включает два набора инструкций: один для случая, когда количество отработанных за неделю часов было не меньше сорока, и второй для случая, когда количество отработанных часов было меньше 40. Значение любой переменной, которая будет отображаться на экране, присваивается в результате выполнения того или иного набора инструкций, так что никаких случайных величин на экране не появится.

Листинг 8.10. Исправленная программа расчета заработной платы.

```
/*allhours.c*/
main()
        {
        float rate, hours, total,
 regular, extra, d_time, overtime;
        printf("Введите оплату одного часа работы: ");
        scanf("%f",&rate);
        printf("Введите число отработанных часов: ");
        scanf("%f", &hours);
        d_time = rate * 2;
        if(hours <= 40)
                {
                regular = hours * rate;
        extra = 0.0;
                overtime = 0.0;
                total = regular;
                }
        else
                {
                regular = 40 * rate;
                extra = hours -40;
                overtime = extra * d_time;
                total = regular + overtime;
                }
        printf("Нормальный
 недельный заработок: %.2f\n", regular);
        printf("Отработано
 сверхурочных часов : %.2f\n", extra);
        printf("Средняя часовая оплата
 сверхурочных: %.2f\n", d_time);
```

```
        printf("Заработок
за сверхурочные часы: %f.2\n", overtime);
        printf("Общая сумма
недельного заработка: %f.2\n", total);
        }
```

Значение оплаты сверхурочных (удвоенное значение оплаты одного часа) присваивается переменной d_time. Расчет этого значения производится перед инструкцией if, так как каждый работник имеет определенную ставку оплаты сверхурочных часов, даже если он реально и не работал больше нормы за истекший период. Соответственно, поскольку сам по себе расчет не зависит от выполнения какого-либо условия, для него не используется инструкция if.

Однако и теперь в программе не разрешены все проблемы, которые могут возникнуть. Например, вы можете ввести отрицательное значение почасовой оплаты или указать какое-нибудь несуразное количество отработанных часов (скажем, 2500 в неделю). Как окончательно решить все проблемы, возникающие из-за ошибочного ввода, вы узнаете в следующей главе.

Вопросы

1. В чем различие между символами = и ==?
2. Как выполнить несколько инструкций при соблюдении одного определенного условия?
3. Каково назначение else?
4. Как вы определите, находится ли число в границах допустимых значений?
5. Объясните использование «вложенных» инструкций if.
6. Чем отличается переключатель switch от инструкции if?
7. Как можно использовать числа с плавающей точкой в конструкциях switch?
8. Объясните, как можно использовать инструкцию if для проверки правильности ввода?

Упражнения

1. Напишите программу, в которой вводится числовое значение, а затем выдается сообщение, четное или нечетное число было введено.
2. Напишите программу, в которой вводится число и затем выдается сообщение, находится ли значение числа в пределах от 1 до 100.
3. Напишите программу, в которой вводится целое число, а затем выдается сообщение, в каком интервале находится значение числа: меньше 0, от 0 до 50, от 51 до 100, от 101 до 150, больше 150.
4. Напишите программу, которая просит пользователя ввести числовые значения в переменные lownum и highnum. Значение lownum должно быть меньше чем highnum. Если числа введены не в соответствии с этим условием, программа должна поменять значения, поместив меньшее число в lownum, а большее— в highnum. Значения переменных должны быть выведены на экран.
5. Объясните, почему следующая программа написана неверно:
6. main()
7. {
8. int age;

```
9.      printf(Укажите Ваш возраст);
10.     scanf("%f", &age);
11.     if age < 18 then
12.                     puts("Вы не
13.   можете участвовать в выборах");
14.     else
15.                     if age > 18 then
16.                                     puts("Вы
17.      можете участвовать в выборах");
        }
```

ГЛАВА 9
ЦИКЛЫ

Программа начинает и заканчивает свою работу, но это не значит, что каждая инструкция в программе должна выполняться только однажды. Вы можете решить провести несколько циклов обработки данных или выполнить несколько различных вычислений. Вы можете также попросить пользователя вводить данные до тех пор, пока он не сделает это надлежащим образом.

Язык Си и Си++ имеет три структуры, известные под названием *циклов*, которые используются для управления повторами:

цикл for;

цикл do...while;

цикл while.

Любой из этих циклов может быть применен для повторения инструкции, группы инструкций или даже целой программы.

Использование цикла for

Цикл for используется в том случае, когда известно точное количество повторов, которое нужно выполнить. Структура такого цикла показана на рис.9.1. Обратите внимание на то, что точка с запятой ставится только после инструкции, а не после параметров for. Однако три параметра внутри круглых скобок отделяются друг от друга точкой с запятой.

В приведенной ниже программе цикл for используется для того, чтобы вывести на экран монитора числа от 1 до 10, расположенные друг под другом.

```
main()
        {
        int repeat;
        for (repeat = 1; repeat <= 10; repeat++)
                printf("%d\n", repeat);
        }
```

Этот цикл управляется переменной repeat, которая называется *индексом**. Индексу можно присвоить любое имя, но значение переменной обязательно должно быть целым числом. Выражение в круглых скобках после for делится на три составляющие:

repeat=1	инициализация переменной repeat путем присваивания ей начального значения
repeat <= 10	задает условие повтора цикла до тех пор, пока значение переменной repeat остается меньше или равно 10
repeat++	приращение значения переменной repeat после каждого повтора цикла

*** Иногда используется термин управляющая переменная цикла. *(Прим.перев.)***

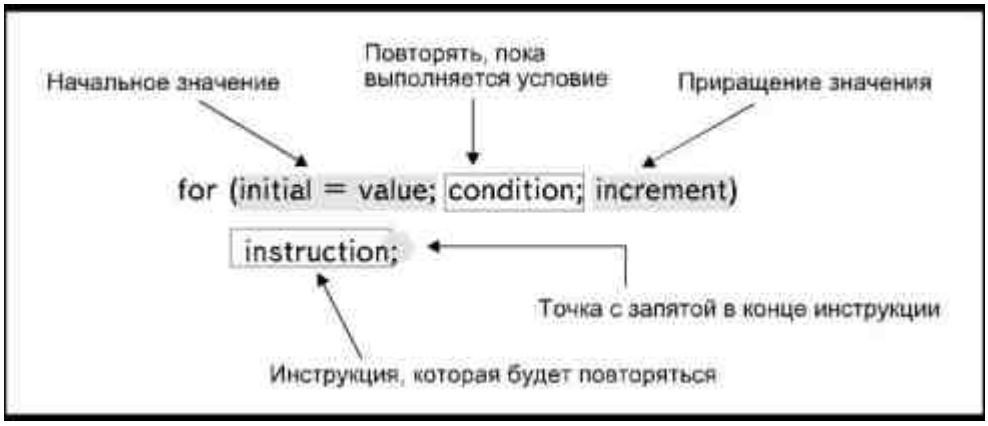

Рис. 9.1. Структура цикла for

Замечания по Си++

В языке Си++ можно определить также и тип индекса внутри круглых скобок цикла for:

for (int repeat=1; repeat <= 10; repeat++)

При каждом новом повторе цикла программа выводит на экран текущее значение переменной repeat.

Когда программа начнет выполнение цикла, она присвоит переменной repeat начальное значение, равное 1. Затем будет проверено, является ли истинным условие, что значение переменной меньше или равно 10. Если условие истинное, начнется выполнение инструкции, связанной с циклом, то есть вывод на экран значения переменной.

После выполнения инструкции произойдет увеличение значения переменной на единицу и снова будет проведена проверка истинности условия (рис.9.2). Так как условие все еще является истинным, цикл будет выполнен во второй раз, отображая на дисплее текущее значение переменной. Этот процесс будет

Рис. 9.2. Условие проверяется перед каждым повтором цикла

повторяться до тех пор, пока значение переменной не вырастет до 11. Как только это произойдет, условие repeat <= 10 уже не будет истинным, так что выполнение инструкции прекратится и цикл завершится.

В предыдущем примере значение индекса использовалось непосредственно в инструкции вывода. В то же время можно написать инструкции следующим образом:

```
main()
        {
        int repeat;
        char letter;
        puts("Введите 10 символов");
        for(repeat = 1; repeat <= 10; repeat++)
                letter = getchar();
        }
```

В этой программе функция getchar() выполняется 10 раз, по количеству повторов цикла, пока значение переменной repeat не увеличится с 1 до 11. Индекс в данном случае используется только для определения количества повторов. С тем же результатом можно было записать инструкции следующим образом:

```
        for (repeat = 101; repeat <= 110; repeat++)
                letter = getchar();
        }
```

Здесь также вводится 10 символов, но теперь значение индекса изменяется от 101 до 110. Точное значение индекса приобретает значение только в том случае, когда оно само по себе используется в цикле.

Создание паузы в программе

Цикл for можно использовать и без инструкций, с целью создания задержки в программе:

for (delay = 1; delay <= 1000; delay++);

Хотя инструкции, связанные с циклом, отсутствуют, тем не менее, цикл будет повторен 1000 раз, пока выполняется указанное условие, то есть пока значение переменной delay не возрастет с 1 до 1001. Выполнение повторов цикла приостановит переход программы к выполнению следующих инструкций.

Одним из возможных применений такого цикла является временная остановка вывода на экран сообщений, с тем, чтобы дать пользователю время прочитать инструкции или подсказки. Этот способ можно использовать наряду с тем, где пользователю предлагается нажать Enter для продолжения вывода сообщений.

Вы можете провести эксперимент, чтобы выяснить, как долго будет выполнять тысячу повторов ваша система. Длительность паузы зависит от быстродействия компьютера. Выполнение цикла может длиться 1 секунду на быстром компьютере и 2 секунды на медленном*. Чтобы установить желательную длительность паузы, можно увеличивать или уменьшать количество повторов, заданное в условии.

*** Автор несколько утрирует. Для получения паузы в 1 секунду даже на очень медленном компьютере цикл потребуется повторить несколько тысяч раз. *(Прим.перев.)***

Составные инструкции

В одном цикле можно выполнить несколько инструкций. Для этого всю последовательность инструкций, относящихся к циклу, следует заключить в фигурные скобки. В качестве примера посмотрите программу перевода 101 последовательного значения температур (от 32 до 132) в значения по шкале Цельсия:

```
main()
        {
        int temp;
        float celsius;
                puts("Шкала Фаренгейта\tШкала Цельсия\n");
                for (temp = 32; temp <= 132; temp++)
        {
                        celsius = (5.0 / 9.0) * (temp - 32);
                        printf("%d\t\t%6.2f\n",
 temp, celsius);
                        }
        }
```

Функция printf() форматирует вывод и отображает значения на экране в колонках. Два символа табуляции (\t\t) позволяют согласовать вывод значения температуры с заголовком колонки. Указатель формата (%6.2) отображает значение с двумя знаками после точки, определяя значение ширины поля, равное шести символам.

При каждом повторе цикла выполняются две инструкции. Значение индекса определяет как количество повторов, так и значения, которые следует перевести в шкалу Цельсия. Сравните только что просмотренную вами программу со следующей:

```
main()
        {
        int temp, repeat;
        float celsius;
        puts("Шкала Фаренгейта\tШкала Цельсия\n");
        temp = 10;
        for (repeat = 1; repeat <= 10; repeat++)
                {
                celsius = (5.0 / 9.0) * (temp - 32);
                printf("%d\t\t%6.2f\n",
```

```
temp, celsius);
                    temp += 10;
                    }
          }
```

Теперь индекс определяет только количество повторов. Значения температуры, которые следует преобразовать, определяет переменная temp. Эта переменная увеличивает свое значение на 10 при каждом повторе: с 10 до 20, 30 и так далее, вплоть до 100.

Использование переменных

Если во время составления программы вы не знаете, сколько повторов потребуется, вы все же можете использовать цикл for, при условии, что количество повторов будет указано в момент запуска программы на выполнение. Можно ввести значение в переменную и использовать ее в условии. Например, следующая программа просит пользователя указать пределы значений температуры, которые требуется преобразовать, то есть, по сути, пользователь сам должен определить количество повторов цикла:

```
main()
          {
          int temp, start, end;
          float celsius;
                    printf("Введите начальное
значение температуры: ");
                    scanf("%d", &start);
                    printf("Введите конечное
значение температуры: ");
                    scanf("%d", &end);
                    puts("Шкала Фаренгейта\tШкала Цельсия\n");
                    for (temp = start; temp <= end; temp++)
                              {
                              celsius = (5.0 / 9.0) * (temp - 32);
                              printf("%d\t\t%6.2f\n",
temp, celsius);
                              }
          }
```

Здесь требуется ввести начальное и конечное значения температур, которые мы хотим привести к шкале Цельсия. Переменные start и end используются в цикле for для установки начального значения индекса и для проверки условия. Цикл завершится, когда значение переменной temp превысит величину переменной end. Таким образом, если вы введете числа 20 и 43, программа преобразует значения температур от 20 до 43 градусов по Фаренгейту в соответствующие значения по шкале Цельсия. Цикл будет повторен 24 раза, затем завершится.

Вложенные циклы

Если один цикл for выполняется внутри другого, принято говорить, что мы имеем дело с вложенным циклом. Внутренний цикл целиком выполняется во время каждого повторения внешнего цикла. Вложенные циклы for можно представить себе как двухмерные, а единичный— как одномерный.

В качестве иллюстрации рассмотрим следующую программу:

```
main()
          {
          int row, column;
          for (row = 1; row <= 10; row++)
                    {
                    for (column = 1; column <= 10; column++)
```

```
                    printf("*");
            putchar('\n');          /*вне внутреннего
цикла, но внутри внешнего*/
            }
        }
```

Программа выводит на экран монитора 10 рядов, состоящих из 10 звездочек. Здесь используются две целочисленные переменные row и column. Во внешнем цикле переменная row увеличивает свое значение на единицу при каждом повторе. Кроме того, при каждом повторе внешнего цикла, внутренний цикл повторяется 10 раз, увеличивая значение переменной column и выводя на экран

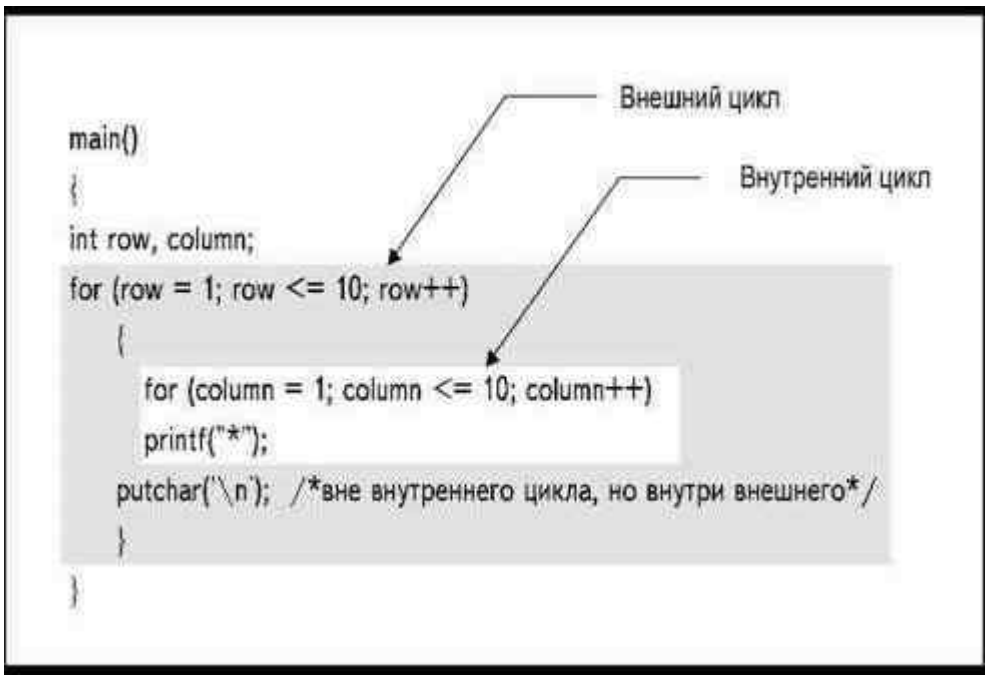

Рис. 9.3. Внешний и внутренний циклы

ряд из десяти звездочек (обратите внимание, что имена переменным даны с таким расчетом, чтобы пояснить логику программы: row по-английски значит «строка», а column— «столбец» или «колонка»). На рис.9.3 продемонстрирована работа этих вложенных циклов. К инструкциям внутреннего цикла относится

```
for (column = 1; column <= 10; column++)
printf("*");
```

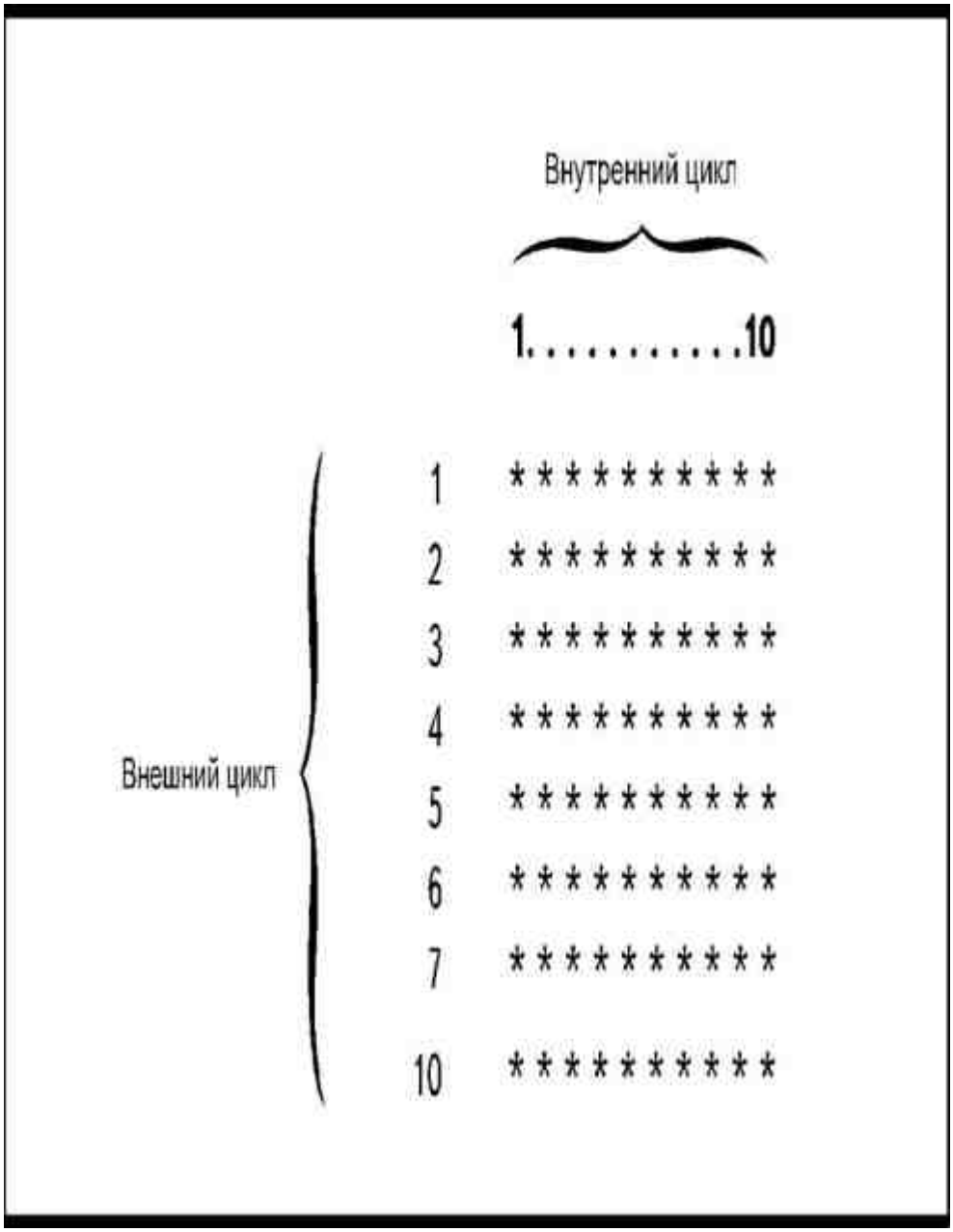

Рис. 9.4. Значения, которые имеют переменные во время каждого повтора цикла

В результате работы программы на экране появятся 100 звездочек: 10 внутренних циклов сформируют 10 колонок, а 10 внешних циклов— 10 рядов. Значения, которые переменные приобретают при каждом повторе цикла, показаны на рис.9.4.

Обратите внимание на положение инструкции putchar('\n');. Она не относится к внутреннему циклу, но в то же время находится внутри фигурных скобок, ограничивающих внешний цикл. Эта инструкция выполняется десять раз, по одному на каждый повтор внешнего цикла, вставляя код «новая строка» после каждого ряда звездочек.

В Листинге 9.1 приведен другой пример использования вложенных циклов. Десять внешних и десять внутренних циклов здесь используются для того, чтобы создать таблицу умножения. Вместо того чтобы просто выводить на экран ряды звездочек, программа выводит результаты произведения значения переменной row на значение переменной column.

Листинг 9.1. Программа создания таблицы умножения.

```
/*timestab.c*/
main()
        {
        int row, column;
        puts("\t\tТаблица Пифагора\n\n");
        for(row = 1; row <= 10; row++)
```

```
        {
        for(column = 1; column <= 10; column++)
                printf("%6d", row * column);
        putchar('\n');
        }
}
```

Таблица Пифагора

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	16	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Рис. 9.5. Результат работы программы, создающей таблицу умножения

Результат работы программы, названной нами TIMESTAB.C, изображен на рис.9.5.

Наконец, рассмотрим следующую программу:

```
main()
        {
        int row, column;
        for (row = 1; row <= 10; row++)
                {
                for (column = 1; column <= row; column++)
                        printf("*");
                putchar('\n');
                }
        }
```

Она выводит на экран последовательность звездочек, показанную на рис.9.6.

```
*
**
***
****
*****
******
*******
********
*********
**********
```

Рис. 9.6. Количество повторов внутреннего циклаопределяется номером повтора внешнего цикла

Мы видим, что каждый ряд звездочек имеет разную длину. Это обусловлено тем, что количество повторов внутреннего цикла не одинаково, а возрастает с каждым следующим выполнением внешнего цикла: одна звездочка в первом ряду, две звездочки во втором ряду, три в третьем и так далее. Количество колонок совпадает с номером ряда. Мы добились такого эффекта, используя переменную row в качестве условия внутреннего цикла.

При первом выполнении внешнего цикла внутренний цикл выполняется только один раз, выводя на экран одну звездочку. При втором повторе внешнего цикла внутренний цикл выполняется два раза, выводя две звездочки. В результате продолжения этого процесса получается узор из звездочек.

Будьте внимательны, когда пишете программу, содержащую два и больше вложенных циклов for. Если в программе указано 100 повторов внешнего и 100 повторов внутреннего цикла, это означает, что потребуется выполнить 10 тысяч повторов!

Использование цикла do...while

Цикл do...while используется в тех случаях, когда вы не знаете точного количества повторов, но в то же время вам известно, что цикл необходимо выполнить, по меньшей мере, один раз. Структура цикла do...while приведена на рис.9.7.

Выполнение инструкций, заключенных в фигурные скобки, повторяется до тех пор, пока является истинным условие, указанное в while. Правильность условия проверяется только в конце цикла, так что, даже если условие с самого начала не было истинным, цикл будет выполнен, по меньшей мере, один раз.

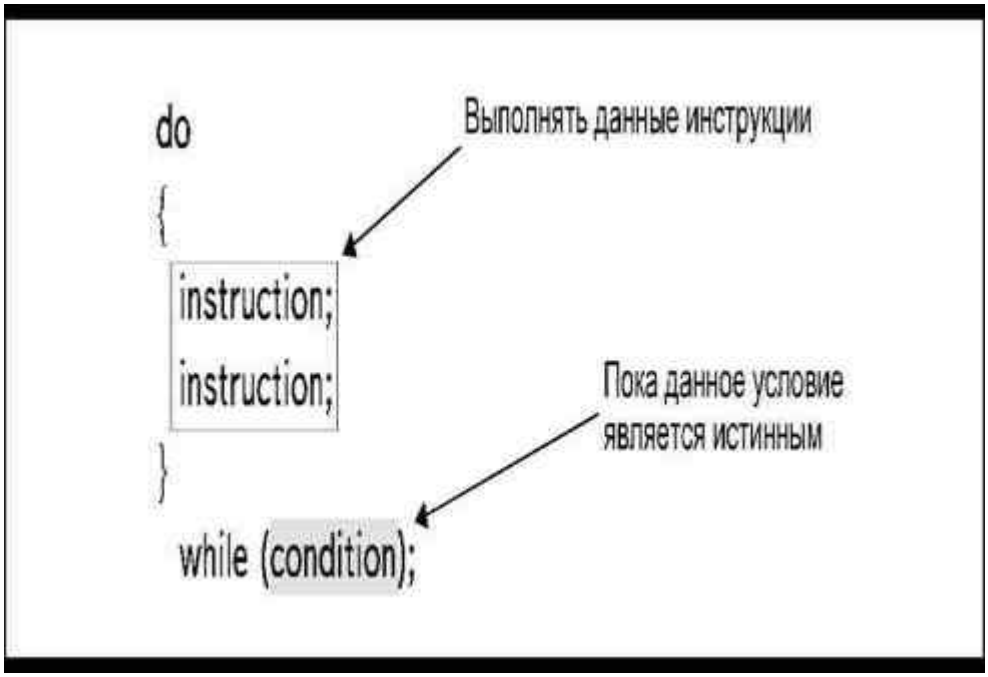

Рис. 9.7. Структура цикла do...while

Используя цикл do...while, следует указывать условие так, чтобы выполнение его не оказалось бесконечным. Ведь если условие все время будет выполняться, остановить повторение цикла можно будет только перезагрузкой системы с помощью кнопки **Reset** или комбинации клавиш **Ctrl+Alt+Del**.

Цикл do...while часто используется для того, чтобы повторять программу до тех пор, пока пользователь не решит закончить ввод:

```
main()
        {
        int temp;
        float celsius;
        char repeat;
        do
                {
                printf("Введите значение температуры: ");
                scanf("%d", &temp);
                celsius = (5.0 / 9.0) * (temp - 32);
                printf("%d градусов по
Фаренгейту соответствует %6.2f по Цельсию\n",
temp, celsius);
                printf("Желаете ввести еще значение?");
                repeat = getchar();
                putchar('\n');
                }
        while (repeat == 'y' || repeat == 'Y');
        }
```

Чтобы избежать проблем, связанных с использованием функции getchar() после функции scanf(), применяйте вместо getchar() функции getch() или getche(), либо вызывайте функцию fflush(stdin) для очистки буфера. Вопрос подробно рассматривался в главе 8.

В этом примере практически весь текст программы, исключая определение переменных, входит в большой блок do...while. Цикл будет повторяться до тех пор, пока в ответ на запрос будет вводиться символ Y или y. Обратите внимание, что инструкция, выводящая на экран запрос о продолжении работы, является последней инструкцией в блоке.

Цикл do...while также используют и для того, чтобы обеспечить правильность ввода данных. Например, у нас есть программа, в которой требуется ввести процент скидки стоимости товара в виде десятичной дроби. Если пользователь вводит значение, не соответствующее формату, например, меньше 0 либо больше или равно 1, можно, с помощью цикла do...while, предложить ему повторить ввод, написав инструкции примерно следующего содержания:

```
do
        {
        printf("Введите размер скидки:");
        scanf("%f", &discount);
        }
while (discount < 0 || discount >= 1);
```

Условие, записанное в while с использованием логического оператора ИЛИ, проверяет, находится ли введенное число в определенных допустимых границах. Цикл будет повторяться до тех пор, пока пользователь не введет данные надлежащим образом. Но тут есть одна тонкость.

Описанная процедура может дать хорошие результаты только в том случае, если пользователь в конце концов поймет, что от него требуется, и введет правильное число. Если же вы имеете дело с человеком, который не имеет представления о том, что такое десятичная дробь, цикл может повторяться до бесконечности. Одним из способов справиться с такой ситуацией является использование алгоритма счетчика. Например, можно написать следующую программу:

```
main()
        {
        int count;
        float discount;
        count=0;
        do
                {
                printf("Введите размер скидки: ");
                scanf("%f", &discount);
                count++;
                }
        while ((discount < 0 || discount >= 1)
                                        && count < 20);
        if (count == 20)
                puts ("Дурак");
        }
```

Теперь у пользователя есть 20 попыток, чтобы ввести правильные данные. Значение переменной count увеличивается на единицу после каждой неправильной попытки.

Для того чтобы выполнить только одну инструкцию в цикле do...while, нет необходимости использовать фигурные скобки. Например, следующая программа позволяет вводить символы, пока не будет нажата клавиша **Y**:

```
main()
        {
        int a;
        do
                a = getchar();
        while (a != 'y' && a != 'Y');
        }
```

```
/*nest*/
main()
{
char repeat;
int temp;
float celsius;
do
{
do
{
printf("Вводите значение температуры правильно:");
scanf("%d", &temp);
}
while (temp < 0 || temp > 100);
celsius = (5.0 / 9.0) * (temp - 32);
printf("%d градусов по Фаренгейту \
соответствует %6.2f по Цельсию\n",temp, celsius);
printf("Желаете ввести еще значение?");
repeat = getchar();
putchar('\n');
}
while (repeat == 'y' || repeat == 'Y');
}
```

Повторение внешнего цикла — это повторение всей программы

Внутренний цикл проверяет правильность ввода данных

Рис. 9.8. Вложенные циклы do...while

Вложенные циклы do

Вложенные циклы do...while можно использовать для того, чтобы обеспечить несколько уровней повторов. Можно, например, использовать внешний цикл, чтобы повторять всю программу, пока пользователь не решит заняться чем-нибудь другим, и одновременно внутренний цикл для проверки правильности ввода, как это показано на рис.9.8. Внутренний цикл используют для ввода чисел, значения которых должны изменяться в промежутке от 0 до 100. Если ввод данных соответствует этому условию, внутренний цикл завершается и выполняются инструкции внешнего цикла. Выполнение внешнего цикла повторяется, пока является истинным условие

```
while (repeat == 'y' || repeat == 'Y');
```

Как только пользователь вводит какой-нибудь другой символ, внешний цикл завершается, приводя тем самым к завершению всей программы.

Использование цикла while

Цикл while используется в том случае, когда не известно точное число повторов и при этом нет необходимости, чтобы цикл непременно был выполнен хотя бы один раз. Структура цикла while такова:

```
while (condition)
instruction;
```

Синтаксис показан на рис.9.9. Составные инструкции записываются следующим образом:

```
while (condition)
        {
        instructions;
        }
```

Рис. 9.9. Структура цикла while

Так же, как и цикл do, цикл while выполняется до тех пор, пока является истинным условие, но в отличие от конструкции do...while, условие проверяется до начала выполнения цикла, даже если цикл выполняется первый раз. Если условие окажется ложным, цикл не будет выполнен ни разу.

Используя цикл для проверки правильности ввода, помещайте первую инструкцию ввода вне цикла. После этого можно продолжать запрос о вводе данных до тех пор, пока не будет введено правильное значение. Например, таким образом:Так же, как и цикл do, цикл while выполняется до тех пор, пока является истинным условие, но в отличие от конструкции do...while, условие проверяется до начала выполнения цикла, даже если цикл выполняется первый раз. Если условие окажется ложным, цикл не будет выполнен ни разу.

```
printf("Введите размер скидки: ");
scanf("%f", &discount);
while (discount < 0 || discount >= 1)
        {
        printf("Вы ввели неправильное значение");
        scanf("%f", &discount);
        }
```

Комбинирование циклов разных типов

В программе можно использовать любые комбинации вложенных циклов всех типов: while, for и do...while, если этого требует логика построения программы. В приведенной ниже программе, например, используется цикл while, вложенный внутрь цикла for. В программе осуществляется преобразование десяти значений температуры, находящихся в пределах от 0 до 100. Значения вводятся с клавиатуры.

```
/*mixed.c*/
main()
        {
        int temp, count;
        float celsius;
        for (count = 1; count <= 10; count++)
                {
                printf("Введите значение
температуры в пределах от 1 до 100: ");
                scanf("%d", &temp);
                while (temp < 0 || temp > 100)
                        {
                        printf("Ошибка,
повторите ввод: ");
                        scanf("%d", &temp);
                        }
                celsius = (5.0 / 9.0) * (temp - 32);
                printf("%d градусов по Фаренгейту
соответствует %6.2f по Цельсию\n",      temp, celsius);
```

```
                }
        }
```

Внешний цикл for повторяется ровно 10 раз. Внутренний цикл будет повторяться до тех пор, пока пользователь не введет правильные данные.

Проектирование программы

Использование циклов добавляет еще один уровень сложности в проектирование программы. Вам придется тщательно разрабатывать и тестировать алгоритмы программы, чтобы быть уверенным в том, что все они работают правильно.

Во-первых, следует выбрать, какой цикл— for, do...while или while— вы будете использовать. Начните с вопроса:

- Знаю ли я, сколько раз нужно повторить цикл, а если нет, то буду ли знать к моменту запуска программы на выполнение?

Если вы отвечаете на этот вопрос утвердительно, то используйте цикл for. Если ответ отрицательный, то ответьте на следующий вопрос:

- Нужно ли мне, чтобы цикл был обязательно выполнен, по меньшей мере, один раз?

Если на этот вопрос вы ответили положительно, вам следует использовать цикл do...while, если отрицательно— можно использовать цикл while.

Допустим для примера, что вы хотите рассчитать среднее арифметическое значение последовательности чисел. Если неизвестно точное количество чисел в последовательности, вам следует использовать циклы do...while или while, но не цикл for. Возможно, вы хотите предоставить пользователю возможность прекратить выполнение программы сразу после ее запуска, без того, чтобы цикл был выполнен хотя бы один раз. Следовательно, надо использовать цикл while.

Но как прекратить повторение циклов? Необходимо, чтобы пользователь мог как-то сообщить программе, что он уже закончил ввод данных и хочет завершить процедуру ввода. Одним из способов, позволяющих сделать это, является вывод на экран запроса, в котором после каждого повтора цикла пользователю предлагается продолжить или закончить ввод. Но такая процедура требует двух вводов на каждый повтор цикла, а именно, числа, которое добавляется в данные, и символа Y (y), указывающего на желание продолжить ввод.

Возможно, более удобным вариантом является использование некоторого значения, которое, будучи введенным, укажет программе на необходимость завершения процедуры. Пример программы, в которой используется такой способ, приведен в Листинге9.2. Выполнение этой программы прекращается после ввода отрицательного числа. Ввод первого значения находится за пределами цикла, так что программу можно остановить сразу же после запуска, путем ввода первого отрицательного значения. При вводе каждого положительного числа происходит увеличение счетчика на единицу, а число добавляется к сумме введенных значений. Обратите внимание, каждое последующее число вводится в самом конце цикла непосредственно перед тем, как while проверяет условие до начала следующего повторения.

Отметьте также, что условие

```
if (count > 0)
```

проверяется до того, как вычисляется среднее арифметическое. Это сделано для того, чтобы избежать деления на нуль, которое вызывает ошибку выполнения в большинстве компьютерных систем.

Листинг 9.2. Программа вычисления среднего арифметического значения.

```
/*average.c*/
main()
        {
        float number, total;
        int count;
        total = 0.0;
        count = 0;
        printf("Введите число. Отрицательное
 значение заканчивает ввод: ");
        scanf("%f", &number);
        while (number >= 0 )
```

```
                {
                count++;
                total += number;
                printf("Введите число. Отрицательное
значение заканчивает ввод: ");
                scanf("%f", &number);
                }
        if (count > 0)
                {
                number = total / count;
                printf("Сумма = %.2f Количество = %d
Среднее = %.2f", number, count, total);
                }
        }
```

Теперь предположим, что мы написали программу, в которой все значения вводятся внутри цикла:

```
while (a >= 0)
        {
        scanf("%f", &a);
        count++;
        total += a;
        }
```

Если теперь ввести отрицательное значение для прекращения выполнения цикла, оно ошибочно будет расценено как ввод и добавлено к общей сумме прежде, чем while проверит выполнение условия.

Если в программе есть вложенные циклы, следует быть очень внимательным при использовании счетчиков и аккумуляторов. Допустим, например, что мы решили повторять программу расчета среднего арифметического, включив ее в цикл do...while, как это продемонстрировано в Листинге9.3. Эта программа будет работать неверно из-за неправильного размещения инструкций

```
total = 0.0;
count = 0;
```

Листинг 9.3. Неправильное размещение операций присваивания.

```
/*ave_bad.c*/
main()
        {
        char repeat;
        float number, total;
        int count;
        total = 0.0;
        count = 0;
        do
        {
        printf("Введите число. Отрицательное
значение заканчивает ввод: ");
        scanf("%f", &number);
        while (number >=0)
                {
```

```
                    count++;
                    total += number;
                    printf("Введите число. Отрицательное
значение заканчивает ввод: ");
                    scanf("%f", &number);
                    }
          if(count > 0)
                    {
                    number = total / count;
                    printf("Сумма = %.2f Количество = %d
Среднее = %.2f", number, count, total);
                    printf("Желаете ввести другую
последовательность чисел?");
                    repeat = getchar();
                    putchar('\n');
                    }
          }
          while ((repeat == 'y') || (repeat == 'Y'));
          }
```

Из-за того, что операции присваивания расположены вне основного цикла, они будут выполнены только один раз, при запуске программы на выполнение. Потом, при каждом повторе основного цикла, в котором вводятся новые последовательности чисел, счетчик и аккумулятор будут сохранять свои прежние значения. В результате будет происходить расчет среднего, общего для всех введенных во всех циклах чисел. Эти две инструкции присваивания следовало бы поместить внутри внешнего цикла. Если написать текст следующим образом:

```
do
          {
          total = 0;
          count = 0;
```

значения счетчика и аккумулятора будут обновляться при начале ввода каждой новой серии чисел. Обратите внимание, что в программе используется цикл while, вложенный в цикл do...while.

Использование флагов

Флагом (flag) называется алгоритм, который сообщает программе о том, что выполнено некоторое условие или произошло какое-то событие, так же как настоящий флаг, оставленный астронавтами на Луне, свидетельствует о том, что на лунную поверхность ступала нога человека. Переменной-флагу присваивается значение в начале выполнения программы или во внешнем цикле, а затем ей присваивается другое значение, указывающее на то, что произошло некоторое событие или было выполнено определенное условие.

Например, программа перевода значений температур использует цикл do...while для ввода значений:

```
do
          {
          printf("Введите значение температуры
в пределах от 0 до 100: ");
          scanf("%d", &temp);
          }
```

Одна и та же подсказка появляется во всех случаях как при первом вводе, так и в случае ввода ошибочного значения. Возможно, было бы неплохо вывести на экран другое сообщение, указывающее пользова-

телю, что он ввел неправильное значение. Текст программы, в которой именно так и сделано, приведен в Листинге9.4.

Листинг 9.4. Программа, в которой используется флаг для отображения альтернативных сообщений.

```
/*flag.c*/
main()
        {
        int temp;
        float celsius;
        char repeat;
        char flag;
        do
                {
                flag = 'n';
                do
                        {
                        if (flag == 'n');
                                printf("Введите значение
температуры от 0 до 100: ");
                        else
                                printf("Вводи значение
правильно, дурак: ");
                        scanf("%d", &temp);
                        flag = 'y';
                        }
                while (temp < 0 || temp > 100)
                        celsius = (5.0 / 9.0) * (temp - 32);
                        printf("%d градусов по Фаренгейту
соответствует %6.2f по Цельсию\n",      temp, celsius);
                        printf("Желаете продолжить ввод?");
                        repeat = getchar();
                        putchar('\n');
                        }
        while (repeat == 'y' || repeat == 'Y');
        }
```

Переменной с именем flag в начале каждого внешнего цикла присваивается значение 'n'. В начале каждого повтора внутреннего цикла значение флага проверяется. Если оно равно 'n', то программа выводит на экран одно сообщение, при любом другом значении флага на экран выводится второе сообщение.

При первом выполнении цикла флаг имеет значение 'n', поэтому на экран выводится первое сообщение. Когда пользователь вводит число, значение переменной меняется на 'y'. Однако если пользователь ввел неправильное значение, внутренний цикл снова повторяется, но в этом случае условие (flag == 'n') не выполнится, так что на экран будет выведено второе сообщение.

Когда пользователь вводит правильное значение температуры, оно преобразуется в значение по шкале Цельсия, и внешний цикл повторяется. При следующем выполнении внешнего цикла флаг переустанавливается заново и пользователю предоставляется новый шанс ввести правильное значение температуры. Обратите внимание, что значение флага переустанавливается при каждом выполнении внешнего цикла, так же как значения счетчика и аккумулятора.

Хотя флаг и может быть определен с любым типом данных, рекомендуется все же выбирать для него тип int или char. Значения, присваиваемые флагу, также целиком определяет автор программы. В нашем примере использовалось значение 'n', чтобы отобразить правильный ввод данных, и 'y', чтобы отобразить неправильный ввод. Вы можете использовать любые другие значения, какие подскажет ваша фантазия.

Использование инструкции break

Использование флага несколько перегружает программу, так как требуется определение дополнительной переменной, введение нескольких строк для присваивания ей значения и лишняя конструкция if...else. Можно избежать этих сложностей и выполнить ту же работу, если применить цикл while. Пример такой программы приведен в Листинге9.5.

Листинг 9.5. Использование цикла while и инструкции break.

```
/*break*/
main()
        {
        int temp;
        float celsius;
        printf("Введите значение температуры
 в пределах от 0 до 100.
 Ввод значения 555 прекращает работу программы: ");
        scanf("%d", &temp);
        while (temp != 555)
                {
                while ((temp < 0 || temp > 100)
                                        && temp != 555)
                        {
                        printf("Ошибка,
 повторите ввод: ");
                        scanf("%d", &temp);
                        }
                if (temp == 555)
                        break;
        celsius=(5.0/9.0)*(temp-32);
        printf("%d градусов по Фаренгейту
 соответствует %6.2f по Цельсию\n",      temp, celsius);
        printf("Значение введено, для
 прекращения работы наберите 555: ");
        scanf("%d", &temp);
                }
        }
```

В этой программе на экран тоже выводятся два сообщения: одно предлагает ввести число, а второе появляется при ошибочном вводе. Обратите внимание, что в этой программе отсутствует запрос о продолжении работы после каждого повтора цикла. Вместо этого, для окончания работы пользователю предлагается ввести число 555.

Считается, что при правильном вводе данных значения чисел лежат в промежутке от 0 до 100 или равны 555, причем ввод последнего немедленно прекращает работу программы, так как в ней записаны инструкции:

```
if (temp == 555)
        break;
```

Инструкция break завершает цикл, в который она помещена, так же, как если бы условие while или условие цикла for перестало выполняться.

Все программы перевода значений температур из одной шкалы в другую, приведенные в этой главе, являются «правильными» программами. Во всех этих примерах мы применяли различный подход и использовали разные алгоритмы для выполнения одной и той же задачи. Если в программе присутствует запрос о необходимости прекращения работы, пользователю не надо помнить, какое именно специальное значение он должен ввести для прекращения работы, но зато при каждом повторе цикла он должен отвечать на дополнительный вопрос помимо ввода собственно значения температуры. Использование специальных значений, вроде числа 555 в нашем примере, избавляет от необходимости лишний раз нажимать на клавиши. Однако неудобство этого способа состоит в том, что нельзя использовать некоторые значения, которые могут оказаться корректными данными. Например, если прекращение работы программы происходит при вводе отрицательного значения, как тогда быть, если мы хотим преобразовать значения температуры ниже нуля?

Критерии «правильности» программы определяются тем, выполняется ли она без ошибок и работает ли она столько времени, сколько требуется.

Вопросы

1. Какими критериями вы будете руководствоваться при выборе цикла for, do или while?
2. Какие функции выполняют параметры for?
3. В каком случае прекращается выполнение цикла for?
4. Что такое вложенный цикл?
5. Как можно использовать флаг?
6. Каково назначение инструкции break?

Упражнения

1. Отредактируйте текст программы из Листинга 8.10 (глава 8) таким образом, чтобы она повторялась до тех пор, пока пользователь не пожелает прекратить ввод данных.
2. Напишите программу, которая рассчитывает сумму 6-процентного налога на продажи для товаров, имеющих стоимость в пределах от 1 до 50 долларов, и выводит информацию на экран монитора в виде таблицы
3. Цена Налог Итого
4. 1 .06 1.06
   2 .12 2.12
5. Напишите программу, которая вводит десять чисел в пределах от 0 до 25.
6. Напишите программу, которая выводит на дисплей следующий график:
7. * * * * *
8. * * * *
9. * * *
10. * *
11. *
12. * *
13. * * *

```
14.     * * * *
        * * * * *
```

15. Объясните, почему следующая программа написана неправильно:

```
16.     main()
17.                 {
18.                 float row, column;
19.                 puts("\t\tТаблица Пифагора\n\n");
20.                 for (row = 1; row <= 10; row++)
21.                             {
22.                             for (column = 1;
23.      column <= 10; column+)
24.                 printf("%d", row * column);
25.                             }
26.                             putchar('\n');
                    }
```

ГЛАВА 10
МАССИВЫ И СТРОКИ

Переменные, с которыми вы имели дело до сих пор, могут хранить только одно значение в каждый момент времени. То есть, если вы, например, хотите узнать среднее арифметическое для 31 значения температуры, вам придется 31 раз ввести значение переменной, по одному за цикл, используя аккумулятор для подсчета суммы. При этом, когда вводится второе значение, первое теряется, когда вводится третье— утрачивается второе, и так далее. Если в программе есть соответствующие инструкции, вы будете знать сумму этих чисел и их среднее арифметическое, но исходные значения будут потеряны.

Если же вы захотите сохранить все значения для дальнейшего использования в программе, вам потребуется определить 31 переменную и написать 31 инструкцию ввода данных.

В этой главе рассказывается о том, как внести значения в специальную переменную, которая называется *массивом* (array). Кроме того, здесь подробно описывается работа со строковыми переменными.

Массивы

Представьте себе группу людей, стоящих в очереди, например, перед входом в кинотеатр. Единственная вещь, которая объединяет этих людей,— это то, что они стоят в одной очереди. Тот факт, что кто-то из них занимает первую позицию, не говорит нам ничего об этом человеке. У нас нет оснований думать, что он проворнее, выше, или богаче, чем любой другой из стоящих за ним.

В то же время, хотя все эти люди являются отдельными индивидами, не связанными никакими иерархическими отношениями, все они входят в одно множество (в данном случае— составляют одну очередь). Поскольку они стоят вместе, в определенном смысле их можно рассматривать как группу. Например, если капельдинер попросит очередь сдвинуться немного влево, чтобы не загораживать автомат с попкорном, очередь сдвинется, продолжая составлять единое целое.

Массив можно представить себе в виде некоего подобия такой очереди. Массив— это множество значений, которыми можно оперировать как группой. Каждый элемент или значение в массиве является отдельной переменной, и с ней можно обращаться, как с обычной переменной, но так как эти переменные собраны в массив, их также можно трактовать как общность. И так же, как место в очереди не дает нам никакой дополнительной информации о человеке, который его занимает, положение элемента массива не имеет никакого отношения к присвоенному ему значению.

Вы уже создавали и использовали массивы в форме строк. Строка и в самом деле является массивом, состоящим из отдельных символов. При отображении строки сообщения на дисплее функция puts() обращается с массивом как с группой, но к каждому символу можно обратиться и в отдельности.

Работа с массивом осуществляется поэтапно. Во-первых, необходимо определить массив как переменную, во-вторых, в массив надо ввести значения, и, наконец, можно использовать массив в соответствии с требованиями логики программы.

Определение массива

Для определения массива необходимо указать типы значений, которые он содержит, и максимальное количество элементов, которое может быть в него записано. Синтаксис определения массива показан на рис.10.1. Для того чтобы создать массив, содержащий, например, 31 целое число, надо написать определение:

```
int temps[31];
```

Запомните, что в этом случае используются именно квадратные скобки, а не круглые или фигурные. Массив определяют так же, как и другие переменные, перед main(), если хотят создать внешний массив (доступный для всех функций), и внутри main(), когда хотят определить автоматический массив.

При определении массива одновременно определяются и его отдельные элементы, как это показано на рис.10.2. К первому элементу массива temps обращаются как к temps[0], ко второму— как к temps[1], к третьему— как к temps[2] и так далее. Заметьте, что нумерация элементов массива начинается с 0, так что массив, состоящий из пяти элементов, показан на рисунке как массив, содержащий элементы с номерами от 0 до 4. Элемент с номером temp[5] не существует. Число, заключенное в квадратные скобки, называется *индексом* (subscript), и про элементы массива можно говорить как про «нулевой элемент массива temps», «первый элемент массива temps» и так далее.

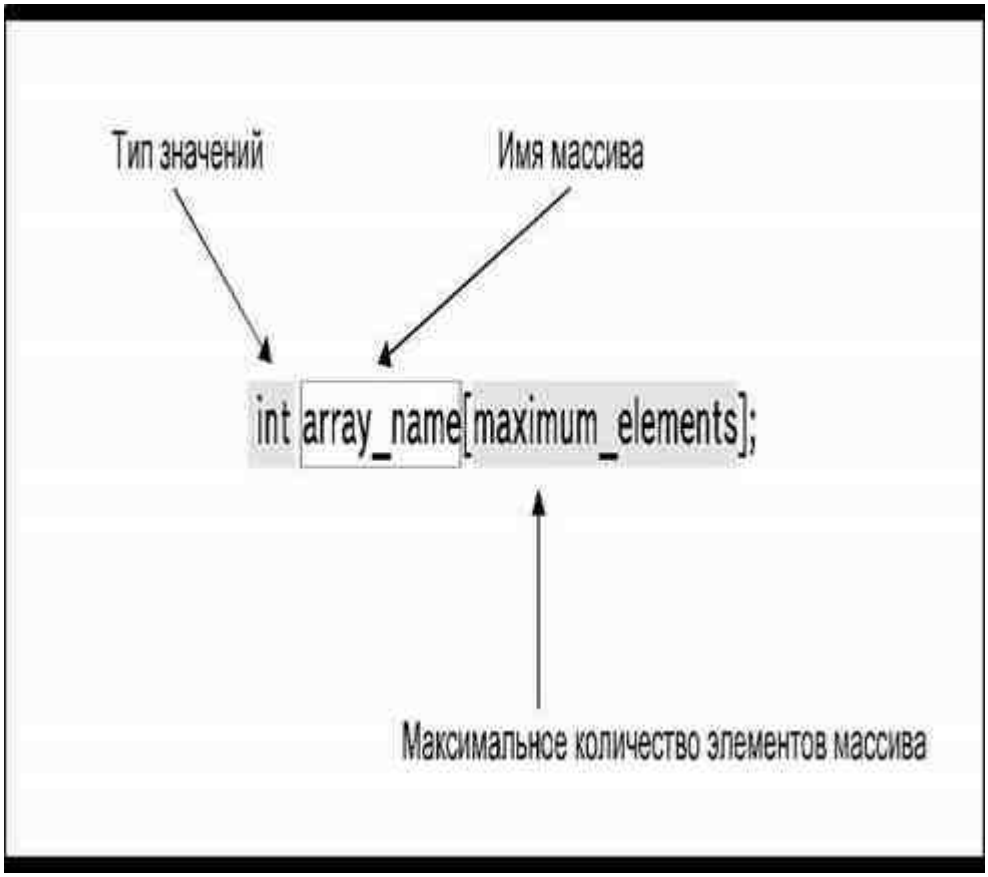

Рис. 10.1. Синтаксис определения массива

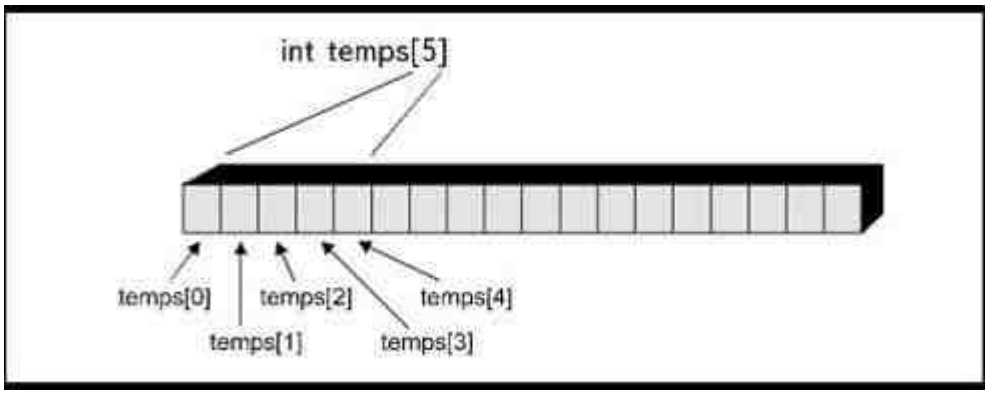

Рис. 10.2. При определении массива также определяютсяи составляющие его элементы

Нельзя использовать индекс, имеющий значение больше указанного максимального количества элементов массива минус один. Если определен массив, имеющий 10элементов, значит максимальным возможным индексом является число9. Компилятор не будет генерировать ошибку, если существует проблема с индексом, но при запуске программы произойдет ошибка выполнения или программа выдаст результаты, далекие от истинных.

Определение строковой переменной на самом деле является определением массива символьных элементов:

char name[5];

С помощью функций gets() и puts() можно вводить и выводить весь массив символьных данных как единое целое, но при работе с другими типами данных эти операции выполняются с каждым элементом массива отдельно.

Ввод значений в массив

После того как вы определили массив, в него можно вводить информацию. Начальные значения элементам массива можно присвоить при его определении:

int temps[5] = {45, 56, 12, 98, 12};

Таким образом, мы создали пять элементов, имеющих следующие значения:

temps[0] 45

temps[1] 56

temps[2] 12

temps[3] 98

temps[4] 12

Для инициализации подобного массива используется соответствующая инструкция перед main(). Подобная инициализация возможна и внутри main(), или другой функции, но в этом случае массив должен быть описан как статический:

```
int temps[5] = {45, 56, 12, 98, 12};
main()
        {
        static float prices[3] = {23.45, 34.56, 12.34};
```

Отдельным элементам массива можно присваивать значения внутри main() или другой функции, как в инструкции

```
temps[0] = 45;
```

Если вы хотите присвоить значение каждому элементу массива, вы можете использовать цикл для ускорения процедуры. В том случае, когда число элементов известно заранее, используйте цикл for:

```
main()
        {
        int temps[31];
        int index;
        for (index = 0; index 31; index++)
                {
                printf("Введите значение
 температуры #%d: ", index);
                scanf("%d", &temp[index]);
                }
        }
```

Переменная index использована здесь для того, чтобы определить количество повторений цикла. В нашем случае цикл будет выполнен 31 раз, по одному на каждый элемент массива. При каждом повторе цикла на экране будет появляться подсказка:

Введите значение температуры #N

где N отображает индекс элемента, которому присваивается значение.

Обратите внимание, что количество повторов определяется увеличением значения переменной index от 0 до 30, а не до 31, поэтому данное значение может использоваться одновременно и как индекс для определения номера текущего элемента массива (которые пронумерованы от 0 до 30). Когда переменная index имеет значение 0, элемент temps[index] на самом деле является элементом temps[0]. Таким образом, значение, которое мы вводим в элемент temps[index] с помощью функции scanf(), присваивается первому элементу с именем temps[0]. С очередным выполнением цикла for значение переменной index возрастает, и при следующих повторах функция scanf() вводит значение в новые элементы массива. Этот процесс проиллюстрирован в табл.10.1.

Таблица 10.1. Использование цикла для увеличения индекса массива.

Повторение	Значение индекса	Элемент, которому присваивается значение
1	0	temps[0]
2	1	temps[1]
3	2	temps[2]
4	3	temps[3]
5	4	temps[4]
6	5	temps[5]
.	.	.
.	.	.
.	.	.
26	25	temps[25]
27	26	temps[26]
28	27	temps[27]
29	28	temps[28]
30	29	temps[29]
31	30	temps[30]

Очень важно, чтобы вы поняли различия между значением индекса переменной, записанным в квадратных скобках, и значением элемента массива как таковым. Индекс только указывает на позицию, которую элемент занимает в массиве, но не имеет никакого отношения к значению самого элемента. Посмотрите на

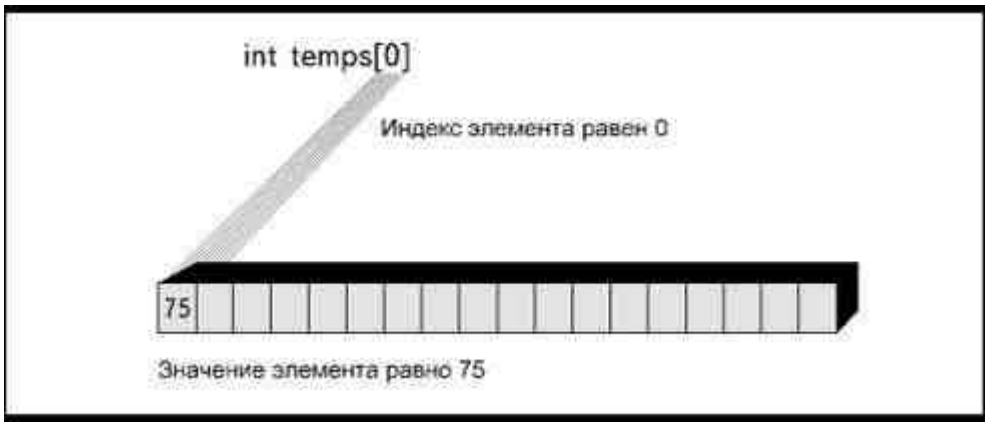

Рис. 10.3. Разница между индексом и значением элемента массива

рис.10.3. Индекс элемента имеет значение 0, а переменная temps[0] равна75. Как мы уже поясняли это на примере очереди перед кинотеатром, значение индекса переменной не связано со значением переменной. Содержимое следующей переменной temps[1] может быть больше, меньше или равно значению переменной temps[0].

В том случае, когда значение элементу массива не было присвоено, он так же, как и любая переменная, будет иметь своим значением некий «мусор». Можно быстро инициализировать целочисленный массив, если в цикле присвоить его элементам нулевые начальные значения:

```
for (index = 0; index < 31; index++)
        temps[index] = 0;
```

Используя массив, можно ввести любое количество элементов, внеся соответствующие изменения в определение массива и условие цикла.

Работа с массивами

Элемент массива можно использовать в любых инструкциях, где используется переменная: в процедурах ввода, вывода и в выражениях. К элементу массива всегда обращаются через его индекс.

Значения, хранящиеся в массиве, можно использовать везде, где это может потребоваться в программе. Например, в Листинге10.1 приведен текст программы, в которой массив используется при выполнении двух различных задач. Во-первых, по элементам массива вычисляется среднее арифметическое значение, во-вторых, массив используется для перевода значений температуры из шкалы Фаренгейта в шкалу Цель-

сия. Каждый раз, когда массив участвует в цикле for, значение переменной index возрастает от 0 до индекса последнего элемента массива.

Листинг 10.1. Использование массива для выполнения двух задач.

```
/*array1.c*/
main()
        {
        int temps[31];
        int index, total;
        float avarage, celsius;
        total = 0.0;
        /*загрузка значений в массив*/
        for (index = 0; index < 31; index++)
                {
                printf("Введите значение
 температуры #%d: ", index);
                scanf("%d", &temp[index]);
                }
        /*подсчет среднего арифметического*/
         for (index = 0; index < 31; index++)
                total += temps[index];
        average = total / 31.0;
        printf("Среднее значение
 температуры составляет: %f\n\n", average);
        puts("Шкала Фаренгейта\t\tШкала Цельсия\n");
        /*перевод значений в градусы Цельсия*/
        for (index = 0; index < 31; index++)
                {
                celsius = (5.0/9.0)*(temps[index]-32);
                printf("%d\t\t%6.2f\n",
 temps[index], celsius);
                }
        }
```

Однако, как вы, может быть, заметили, в программе подразумевается, что пользователь введет значения во все элементы массива, а их у нас 31. Если, например, мы имеем дело с наблюдениями за температурой в ноябре или феврале, полученные результаты не будут правильными, поскольку мы не используем все элементы массива, как это предполагает логика программы.

В программе, приведенной в Листинге 10.1, мы можем использовать любое количество элементов массива, вплоть до 31. Вместо цикла for предыдущей программы теперь используется цикл do...while, и инструкция

```
index = 0;
```

выполняется в начале каждого цикла, чтобы переменная index, использованная в качестве индекса массива, всегда указывала на первый элемент. Ввод подобной инструкции в цикле for не являлся необходимым, так как в этом случае начальное значение индекса устанавливается в самой инструкции for.

Листинг 10.2. Использование цикла do...while для загрузки массива.

```
/*array2.c*/
main()
        {
```

```
int temps[31];
int index, total;
float avarage, celsius, count;
total = 0.0;
        /*загрузка значений в массив*/
index = 0;
do
        {
        printf("Введите значение температуры #%d, \
для прекращения введите 555: ", index);
        scanf("%d", &temp[index]);
        }
while (index < 31 && temps[index-1] != 555);
        /*подсчет среднего арифметического*/
 index = 0;
do
        {
        total += temps[index];
        index++;
        }
while (index < 31 && temps[index-1] != 555);
count = index;
average = total / count;
printf("Среднее значение
 температуры составляет: %f\n\n", average);
puts("Шкала Фаренгейта\t\tШкала Цельсия\n");
        /*перевод значений в градусы Цельсия*/
index = 0;
do
{
        celsius = (5.0 / 9.0) * (temps[index] - 32);
        printf("%d\t\t%6.2f\n",
 temps[index], celsius);
        index++;
        }
while (index < 31 && temps[index-1] != 555);
}
```

В этой программе процедура ввода прекратится, когда окажутся введены значения всех элементов массива, или раньше, если вы введете значение флага 555, который проверяется в условии

```
while (index < 31 && temps[index-1] != 555);
```

Заметьте, что из значения переменной index в условии вычитается единица. Это сделано потому, что значение индекса увеличивается после ввода каждого числа. Если вы не хотите вводить значения во все элементы массива, введите после очередного значения температуры число 555, которое будет присвоено следующему элементу массива и прекратит выполнение цикла.

Число 555 используется как флаг. При выполнении программа будет ожидать появления этого значения, чтобы определить, не хочет ли пользователь прекратить ввод данных раньше, чем все элементы массива окажутся заполнены.

Когда для такого массива вычисляется среднее арифметическое значение, цикл do...while суммирует все значения, введенные в массив, пока не доберется до его конца или не получит значение 555. Для того чтобы переменную index, которая содержит количество введенных значений температур, можно было использовать в качестве индекса массива, она должна быть определена как целочисленная. А чтобы в результате операции деления получить число типа float, значение переменной необходимо определить как float. Таким образом, в программе должна быть определена float-переменная count, в которую перед выполнением математических операций записывается количество введенных значений температур:

```
count = index;

average = total / count;
```

Так как в последней строке инструкций цикла происходит увеличение переменной index, ее значение после выполнения цикла становится на единицу больше, чем индекс последнего элемента, содержащего правильное значение температуры. Например, если значения введены в элементы массива с номерами от 0 до4, переменная index будет равна пяти. Но поскольку это значение отражает количество заполненных элементов, то его можно использовать как счетчик при вычислении среднего арифметического.

Если вы внимательно присмотритесь к логике программы, то увидите, что в ней присутствует потенциальная проблема. Источником возможной ошибки являются второе и третье условия while. Если введено действительно 31 значение температуры (от 0 до 30), переменная index, которая используется в условии temps[31], возрастает до 31, что на единицу больше максимального допустимого

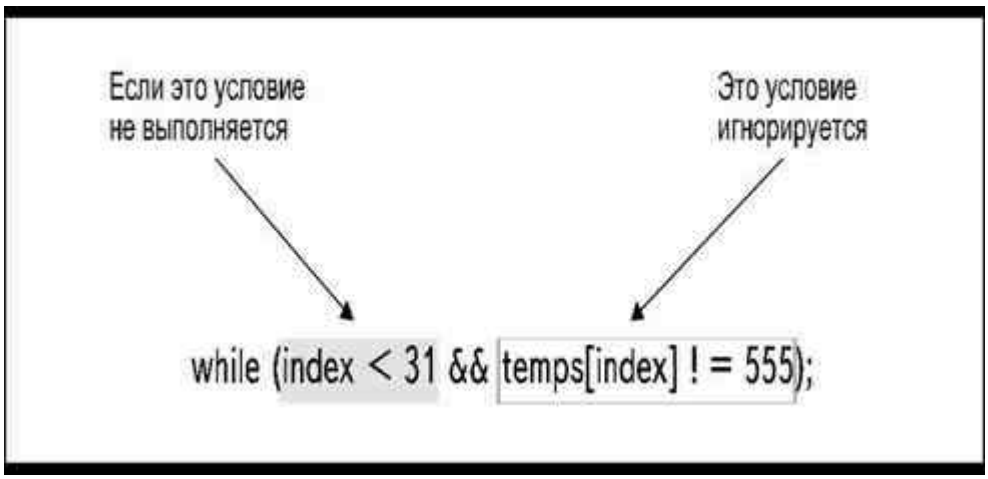

Рис. 10.4. Сокращенная схема проверки условия

значения индекса. Однако ошибки не произойдет, так как условия записаны в следующем порядке:

```
while (index < 31 && temps[index] != 555);
```

В целях ускорения выполнения программы большинство компиляторов, проверяя условие И, автоматически прекращают оценивать истинность дальнейших условий, как только встретят первое ложное условие (рис.10.4). Так как оператор И требует выполнения всех указанных условий одновременно, нет необходимости просматривать второе, если первое не выполняется.

После того как введены значения 31 элемента массива, первое условие в выражении (index < 31) перестает выполняться. Как только это происходит, программа больше не будет тратить время на проверку второго условия и, следовательно, никогда не заметит, что индекс выходит за допустимые пределы.

Вы можете легко проверить справедливость сказанного, если поменяете порядок условий в инструкции:

```
while (temps[index] != 555 && index < 31);
```

Теперь, когда вы запустите программу и введете 31 значение, вы получите ошибку выполнения, так как компьютер попытается проверить элемент массива с индексом, превышающим максимально допустимый.

Некоторые компиляторы не имеют возможности прерывания операции проверки условия. Работая с этими компиляторами, вы получите ошибку выполнения, обусловленную логикой программы, в любом случае. Чтобы выйти из этого положения, можно определить максимальное количество элементов 32, но использовать только31.

Просмотр массива

Используя циклы, вы можете просматривать элементы массива для сравнения входящих в него значений или для поиска определенного значения. Рассмотрим процедуру:

```
/*highlow*/
```

```
...
high = temps[0];
low = temps[0];
index = 1;
while (index < 31 && temps[index] != 555)
        {
        if (temps[index] > high)
                high = temps[index];
        if (temps[index] > low)
                low = temps[index];
        index++;
        }
printf("Минимальное значение равно %d\n", low);
printf("максимальное значение равно %d\n", high);
...
```

Предполагается, что переменные high и low определены как целочисленные. Здесь описана процедура просмотра каждого элемента массива с целью поиска наибольшего и наименьшего из введенных значений. Ключевым моментом является то, что изначально значение первого элемента массива присваивается и переменной high, и переменной low. Действительно, когда имеется только один элемент, его значение является одновременно и максимальным и минимальным из рассматриваемых. Таким образом, мы даем переменной некое начальное

Элемент массива	Значение элемента	Значение переменной high	Значение переменной low	Инструкции изменения значений
temps[0]	65	65	65	high = temps[0] low = temps[0]
temps[1]	95	95	65	high = temps[1]
temps[2]	75	95	65	
temps[3]	34	95	34	low = temps[3]

Рис. 10.5. Значения переменных в процессе выполнения цикла

значение, которое затем будет сравниваться со всеми другими значениями элементов массива. Если значение какого-либо элемента превысит текущее значение переменной high, оно станет новым максимальным значением, если оно окажется меньше текущего значения переменной low, оно станет новым минимальным значением. На рис.10.5 продемонстрировано несколько первых прохождений такого цикла.

Поиск в массиве

Часто возникает необходимость найти в массиве определенное значение. Для этого нужно просмотреть каждый элемент массива и проверить его значение на предмет соответствия некоторой заданной величине. Если требуется только узнать, встречается ли в массиве некоторое значение, то можно прекратить поиск, как только будет обнаружен первый элемент, значение которого представляет искомую величину. То есть, если массив состоит из тысячи элементов и мы нашли нужное значение в первом же элементе, нет необходимости проверять остальные 999 элементов.

Ниже написана процедура поиска в нашем учебном массиве значений температур:

```
/*found.c*/
...
printf("Укажите значение, которое желаете найти: ");
scanf("%d", &num);
index = 0;
found = 0;
while (! found && index < 31 && temps[index] != 555)
        {
        if (temps[index] == num)
                found = 1;
        else
                index++;
        }
if (! found)
        puts("Указанного значения нет в массиве");
else
        printf("Указанное значение
 содержится в элементе %d\n", index);
...
```

Здесь предполагается, что была определена целочисленная переменная found. Эта переменная используется как флаг, то есть указывает на то, что искомое значение найдено и можно прекратить дальнейший просмотр значений массива.

Переменная found используется с унарным оператором отрицания. Выражение

```
while (! found)
```

означает: «Пока значение found не является истинным». Эта инструкция выполняется, пока переменная found имеет нулевое значение. Так как начальное значение переменной было присвоено за пределами цикла, при первом прохождении условие окажется истинным, и программа проверит первый элемент массива (имеющий индекс0).

Как только элемент, имеющий искомое значение, обнаружен, условие, записанное в инструкции if, становится истинным и переменной found присваивается значение1. Это означает, что условие «Пока значение found не является истинным» больше не будет выполняться, и цикл завершается. Значение, присвоенное переменной index во время последнего прохождения цикла, соответствует номеру элемента массива, который содержит искомое значение.

Заключительные инструкции if выводят на экран одно из двух сообщений в зависимости от того, найдено ли искомое значение в данном массиве.

Обратите внимание, что инструкция while содержит три условия, объединенные логическим оператором И. Цикл будет повторяться до тех пор, пока не произойдет одно из трех событий: найдено искомое значение, проверены все элементы массива или обнаружен элемент, имеющий значение555.

Передача массива функции

Массив может быть передан целиком из одной функции в другую, хотя в языках Си и Си++ процедура передачи массива довольно сильно отличается от передачи переменных других типов. При передаче переменной, не являющейся массивом, Си копирует данные, помещая их в область памяти, отведенную для принимающей переменной. Таким образом, имеются два набора данных, и если вызванная функция изменит значение переданной ей переменной, содержимое исходной переменной затронуто не будет.

При передаче массива происходит передача только лишь его адреса. Си не делает копии данных, а как бы дает исходному массиву (занимаемой им области памяти) другое имя. В Листинге10.3 продемонстрировано, как передавать массив в качестве аргумента. Обратите внимание, что для вызова аргумента используется только имя массива без индекса:

```
convert(temps)
```

Листинг 10.3. Передача массива.

```
/*arr_pass.c*/
#define COUNT 31
main()
        {
        int temps[COUNT];
        int index;
        float celsius;
        /*загрузка значений в массив*/
        for (index = 0; index < COUNT; index++)
                {
                printf("Введите значение
 температуры #%d: ", index);
                scanf("%d", &temp[index]);
        }
        /*передача массива функции convert()*/
        convert(temps);
        }
        /*функция преобразования значений*/
convert(heat)
int heat[];
        {
        int index;
        float celsius;
        for (index = 0; index < COUNT; index++)
                {
                celsius = (5.0 / 9.0) * (temps[index] - 32);
                printf("%d\t\t%6.2f\n",
 heat[index], celsius);
                }
        }
```

Передача массивов в Си стандарта K&R

Так отмечается дополнительная информация, в основном касающаяся особенностей языка Си++, которую при первом чтении можно пропустить.Исходный K&R-стандарт Си не позволяет передавать массив, используя только его имя, как это сделано в примере. В этом случае для передачи массива необходимо использовать указатель на переменную, с которым вы познакомитесь в главе 11. Однако компиляторы ANSI-Си и Си++ позволяют передавать массив с использованием синтаксиса, показанного в этой главе.

Замечания по Си++

Запомните, что Си++ позволяет определять массив непосредственно в

аргументе, например, следующим образом:

```
convert(heat[])
```

Так как в действительности был передан адрес массива, нам нет необходимости указывать количество элементов в формальном аргументе. Поэтому в записи используются только пустые квадратные скобки, указывающие на то, что имеется в виду именно массив:

```
convert(heat);
int heat[];
```

Передача параметра посредством передачи адреса вместо копирования данных экономит память, но этот способ может давать побочные эффекты. Изменение

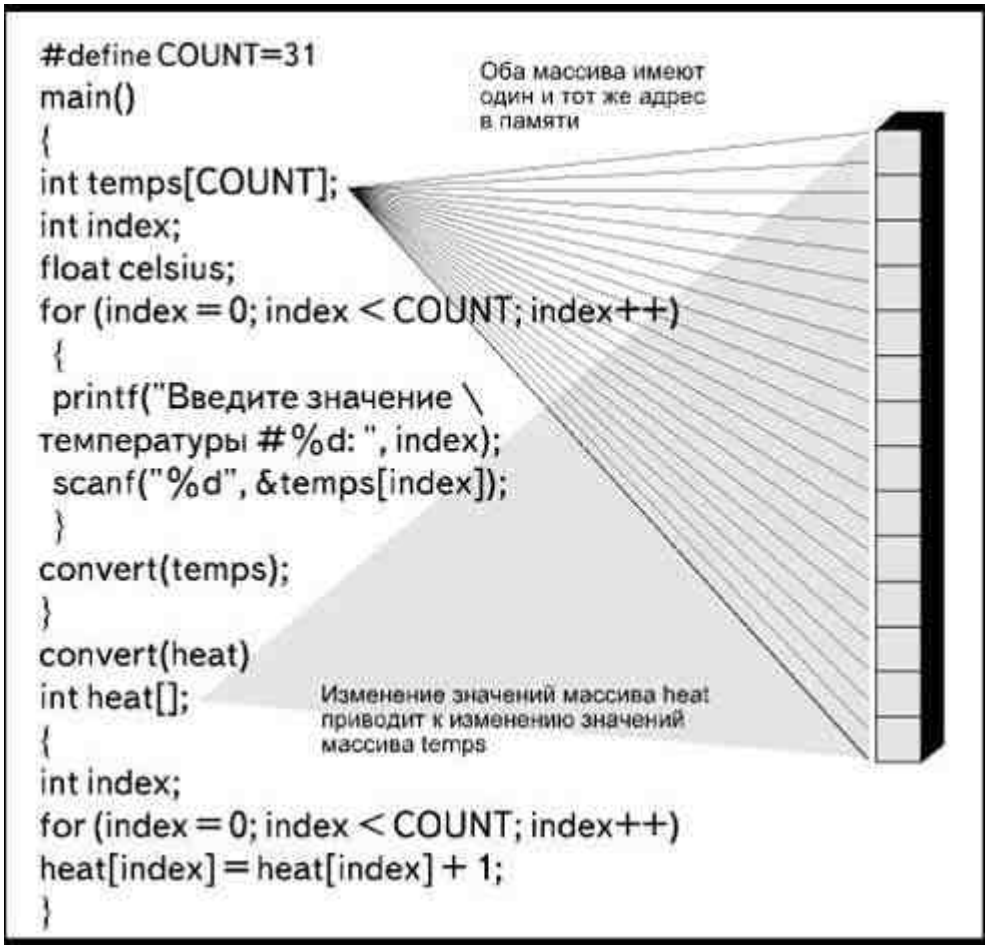

Рис. 10.6. Изменение значения массива функцией

массива в вызываемой функции приводит к изменению значений переданного ей массива, то есть если функция convert() присвоит новые значения массиву heat, эти же значения будут присвоены и массиву temps. Рис.10.6 демонстрирует программу, в которой функция convert() добавляет к значению каждого элемента массива heat единицу. Массив heat имеет тот же адрес в памяти, что и массив temps, поэтому значение каждого элемента массива temps также увеличится на единицу.

Использование массивов

Представьте себе, что вы являетесь менеджером в отеле, и в вашем ведении находятся помещения для проведения заседаний. У вас есть десять комнат для совещаний, каждая из которых рассчитана на определенное количество человек. Вам нужна программа, которая будет выполнять следующие три функции:

- составлять таблицу, содержащую сведения о номере и максимальной вместимости каждой комнаты;
- предоставлять информацию о максимальной вместимости определенной комнаты;
- выводить список комнат, имеющих определенную вместимость.

Есть несколько способов создания такой программы. Вы можете использовать массив, состоящий из 20 элементов, содержащих как номер комнат, так и количество мест, на которое каждая из них рассчитана. Массив будет выглядеть примерно так:

int room[20] = {102, 12, 107, 43...

то есть нечетные элементы отображают номера комнат, а четные— вместимость. Комната 102 рассчитана на 12 человек, комната 107 может вместить 43 человека и так далее. Определив элемент, содержащий номер комнаты, вы знаете, что значение максимальной вместимости этой комнаты представлено следующим элементом массива.

Другой вариант связан с использованием двухмерного массива. Такие массивы мы рассмотрим позже, когда будем обсуждать строки.

Третьей возможностью является использование двух параллельных массивов. Параллельные массивы— это два независимых массива, которые можно соотнести друг с другом. Как это выглядит? Представьте себе две линии выпускников колледжа в актовый день. Когда две колонны достигают входа, ученики располагаются друг напротив друга. Первый человек в правом ряду войдет в зал в паре с первым человеком из левого ряда и так далее. Если расположить учеников произвольно, то никаких отношений между правым и левым рядом не будет. Но можно составить ряды в таком порядке, чтобы при встрече рядов образовались определенные пары.

Когда вы используете параллельные массивы, то создаете два независимых массива, порядок расположения элементов в которых, тем не менее, находится в соответствии. То есть первый элемент одного массива каким-то образом соотносится с первым элементом другого массива и так далее.

В Листинге 10.4 приведен текст программы, в которой проблемы с распределением комнат решаются путем использования параллельных массивов. Массив room содержит список номеров комнат, которые находятся в вашем распоряжении, а в массиве max записаны данные о количестве мест, на которые рассчитана каждая комната. Если мы знаем, что номер комнаты хранится в пятом элементе массива room, то нам известно, что вместимость этой комнаты находится также в пятом элементе, только массива max.

Листинг 10.4. Использование двух параллельных массивов.

```
/*rooms.c*/
int room[10] = {102, 107, 109, 112,
 115, 116, 123, 125, 127, 130};
int max[10] = {12, 43, 23, 12, 20, 15, 16,
 23, 12, 15};
main()
        {
        int index, choice, num, rooms, flag, found;
        rooms = 10;
        puts("1. Показать вместимость всех комнат\n");
        puts("2. Определить вместимость
 определенной комнаты\n");
        puts("3. Найти комнату определенной вместимости\n");
        printf("Введите Ваш выбор от 1 до 3: ");
        scanf("%d", &choice);
        if (choice == 1)
                {
                for (index = 0; index < rooms; index++)
                printf("Комната #%d рассчитана на
 %d мест\n", room[index], max[index]);
                }
        if (choice == 2)
                {
                printf("введите номер комнаты: ");
                scanf("%d", &num);
                index = 1;
```

```
                found = 0;
                while (! found && index < rooms)
                        if (room[index] == num)
                                found = 1;
                        else
                                index++;
                if (! found)
                        puts("Комнаты с
таким номером нет в списке\n");
                else
                        printf("Комната #%d
рассчитана на %d мест\n", room[index], max[index]);
                }
        if (choice == 3)
                {
                flag = 0;
                printf("Введите минимальное
желательное количество мест:");
                scanf("%d", &num);
                for (index = 0; index < rooms; index++)
                        if (max[index] >= num)
                                {
                                flag = 1;
                                printf("Комната #%d
рассчитана на %d мест\n", room[index], max[index]);
                                }
        if (flag == 0)
                        puts("Комнат с
таким количеством мест нет\n");
                }
        }
```

В этой программе максимальное значение индекса массива присваивается переменной rooms, а затем на экран выводится меню. Выбор одного из пунктов меню определяет, какая из функций будет выполняться. Это осуществляется с помощью последовательности из трех инструкций if. Аналогичную программу можно было написать с использованием инструкции switch или вложенных инструкций if...else.

Первая процедура в цикле for выводит список всех комнат и их вместимость. Здесь необходима только одна индексная переменная, которая используется в качестве индекса как для массива room, так и для массива max.

Вторая процедура просматривает массив в поисках элемента, имеющего значение, соответствующее номеру комнаты, введенному пользователем.

В третьей процедуре выполняется поиск номеров комнат, которые могут обеспечить определенную вместимость, указанную пользователем. Но после обнаружения первого подходящего варианта поиск не останавливается. Просмотр массива происходит до конца с целью составления списка всех комнат, удовлетворяющих указанному условию. Переменная flag используется для индикации того, что не обнаружено ни одной подходящей комнаты.

Строки

Строкой называется массив символов. При определении строки вы присваиваете ей имя и указываете максимальное количество символов, которое может в ней содержаться. Однако не забывайте о том, что один элемент массива всегда резервируется для нулевого символа (\0), вследствие чего при определении строки следует указывать количество элементов на единицу больше действительно предполагаемой максимальной длины строки.

Си и Си++ позволяют осуществлять ввод и вывод массива символов в виде некой целостности— строки. В то же время, каждый символ является отдельным независимым элементом массива (рис.10.7). Например, с помощью следующей программы можно ввести строку, а затем отобразить составляющие ее отдельные символы:

```
main()
        {
        char name[20];
        int index;
        printf("Введите Ваше имя: ");
        scanf("%s", name);
        for (index = 0; index < 20; index++)
                printf("%c\n", name[index]);
        }
```

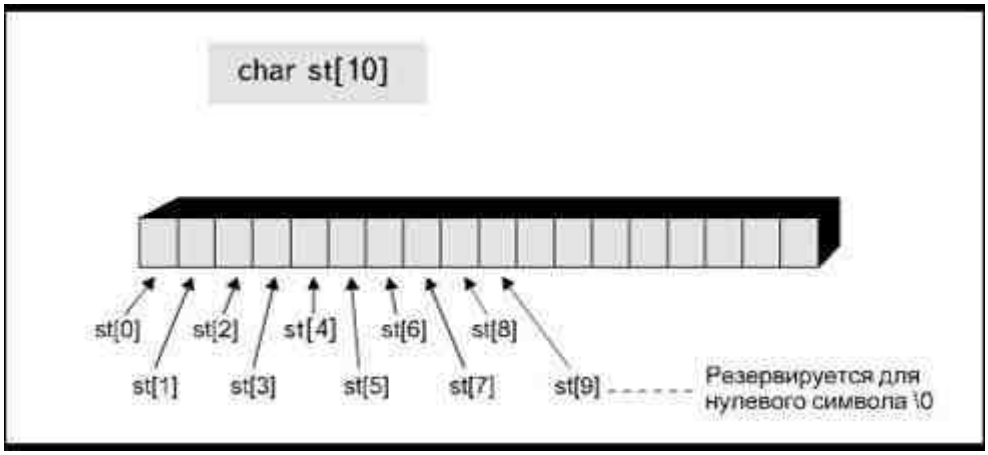

Рис. 10.7. Каждый из символов, составляющих строку, в действительности является отдельным элементом массива

Если в действительности было введено меньше двадцати символов, элементы массива, следующие за нулевым символом, содержат случайные величины.

Если бы в языке Си отсутствовала функция gets(), то ввод строки можно было осуществить путем последовательного ввода символов и присваивания их в качестве значения отдельным элементам массива, как это продемонстрировано в Листинге10.5.

Листинг 10.5. Имитация функции gets().

```
/*getstr.c*/
main()
        {
        char name[10], letter;
        int index;
        index = 0;
        puts("Введите имя, по окончании нажмите Enter\n");
        do
                {
                letter = getchar();
                name[index] = letter;
                index++;
```

```
            }
while (letter !='\r' && index < 9);
name[index] = '\0';
putchar('\n');
puts(name);
}
```

К счастью, язык Си расценивает строки как особую разновидность массивов, позволяя осуществлять ввод и вывод строк как единого целого. Однако какие-либо дополнительные приемы обращения со строками в языке Си отсутствуют. В то же время, использование строк настолько распространено в программировании, что большинство Си и Си++ компиляторов имеют специальные функции для работы со строками. Конечно, для этих целей можно написать и собственные функции, но использование стандартных библиотек представляется более эффективным. В табл.10.2 приведены типичные функции работы со строками, которые содержатся в библиотеках Си и Си++.

Таблица 10.2. Функции работы со строками.

Повторение	Значение индекса	Элемент, которому присваивается значение
1	0	temps[0]
2	1	temps[1]
3	2	temps[2]
4	3	temps[3]
5	4	temps[4]
6	5	temps[5]
.	.	.
.	.	.
.	.	.
26	25	temps[25]
27	26	temps[26]
28	27	temps[27]
29	28	temps[28]
30	29	temps[29]
31	30	temps[30]

В качестве примера мы обсудим несколько функций работы со строками и посмотрим, какие инструкции следует написать, чтобы выполнить те же действия без использования библиотечных функций.

Сравнение двух строк

В Си и Си++ нельзя непосредственно сравнить значение двух строк с помощью, например, такого условия:

```
if (string1 == string2)
```

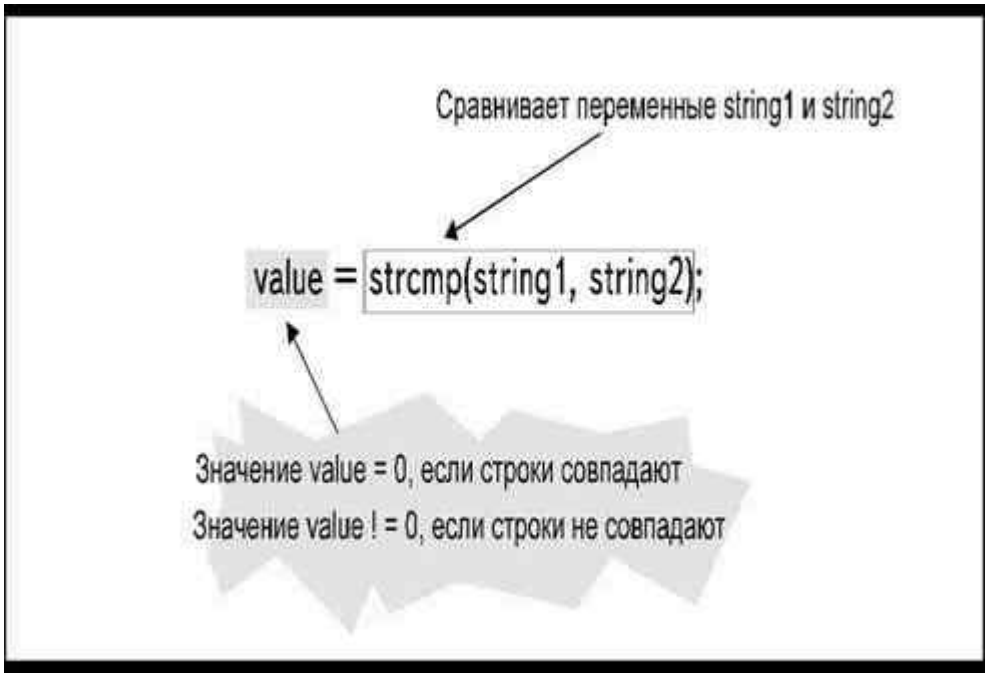

Рис. 10.8. Синтаксис функции strcmp()

Однако большинство библиотек содержит функцию strcmp(), которая возвращает нулевое значение в том случае, если строки одинаковы, либо значение, отличное от нуля, если строки не совпадают. Синтаксис функции strcmp() показан на рис.10.8. Эта функция используется в программе следующим образом:

```
if (strcmp(name1, name2) == 0)
        puts("Имена совпадают");
else
        puts("Имена не совпадают");
```

Некоторые компиляторы возвращают отрицательное число, если первая строка оказывается с точки зрения алфавита «меньше» второй, и положительное число, если «больше».

Если компилятор не имеет функции strcmp(), можно написать собственную функцию, которая сравнивала бы две сроки элемент за элементом как параллельные массивы, и останавливалась при обнаружении пары несовпадающих значений:

```
main()
        {
        int index, flag;
        char name[10], name1[10];
        gets(name);
        gets(name1);
        flag = 0;
        for (index = 0; index < 10; index++)
                if (name[index] != name1[index])
                        {
                        flag = 1;
                        break;
                        }
                if (flag == 1)
                        puts("Строки не совпадают");
                else
                        puts("Строки совпадают");
        }
```

Определение длины строки

Длина строки не обязательно должна совпадать с длиной массива. Например, вы можете определить массив name, который содержит 20элементов, но ввести в него имя, состоящее из меньшего числа символов. Большинство компиляторов Си и Си++ имеют функцию strlen(), которая позволяет определить количество символов, действительно содержащееся в строке:

```
gets(name);

count = strlen(name);

printf("Строка %s содержит %d символов", name, count);
```

Функция присваивает значение, отражающее количество символов во введенной строке (в нашем случае она называется name), целочисленной переменной count. Без использования функции strlen() ту же процедуру можно выполнить с помощью следующих инструкций:

```
main()
        {
        int index;
        char name[10];
        gets(name);
        for (index = 0; index < 10; index++)
                {
                if (name[index] == '\0')
                        break;
                }
        printf("%d", index);
        }
```

В данной программе осуществляется просмотр массива в поисках нулевого символа. Дело в том, что позиция элемента, содержащего нулевой символ, соответствует количеству символов в строке, то есть ее длине.

В качестве примера того, как можно использовать функцию определения длины строки, ниже приведена программа, отображающая строку в обратном порядке:

```
main()
        {
        char name[10];
        int index, count;
        gets(name);
        count = strlen(name);
        for (index = count; index > 0; index--)
                putchar(name[index-1]);
        putchar('\n');
        }
```

Здесь значение длины строки используется в качестве индекса в условии цикла for, причем значение его уменьшается при каждом повторе цикла. Если строка содержит 5символов, цикл будет повторен 5раз, изменяя значение переменной index от 5 до 1. Так как элементы такого массива имеют номера от 0 до 4, то, чтобы получить номер элемента, из значения переменной index вычитается единица.

Присваивание строк

Язык Си не позволяет непосредственно присваивать символы строке, используя инструкции вроде

```
name = "Сэм";
```

Для этой цели можно вызвать функцию strcpy(), с которой позволяют работать большинство компиляторов. Синтаксис функции следующий:

```
strcpy(name, "Сэм");
strcpy(name, name1);
```

В первом примере символы "Сэм" присваиваются строковой переменной с именем name. Во втором примере символы, которые уже были присвоены переменной name1, копируются в переменную name.

Для того чтобы выполнить эти же действия с помощью собственных инструкций, вам придется последовательно присваивать символьные значения каждому элементу массива:

```
char name[] = "Алан";
main()
        {
        char person[10];
        int count, index;
        count = strlen(name);
        for (index = 0; index <= count; index++)
                person[index] = name[index];
        puts(person);
        }
```

В этой программе происходит присваивание символьных значений одного массива соответствующим элементам другого массива.

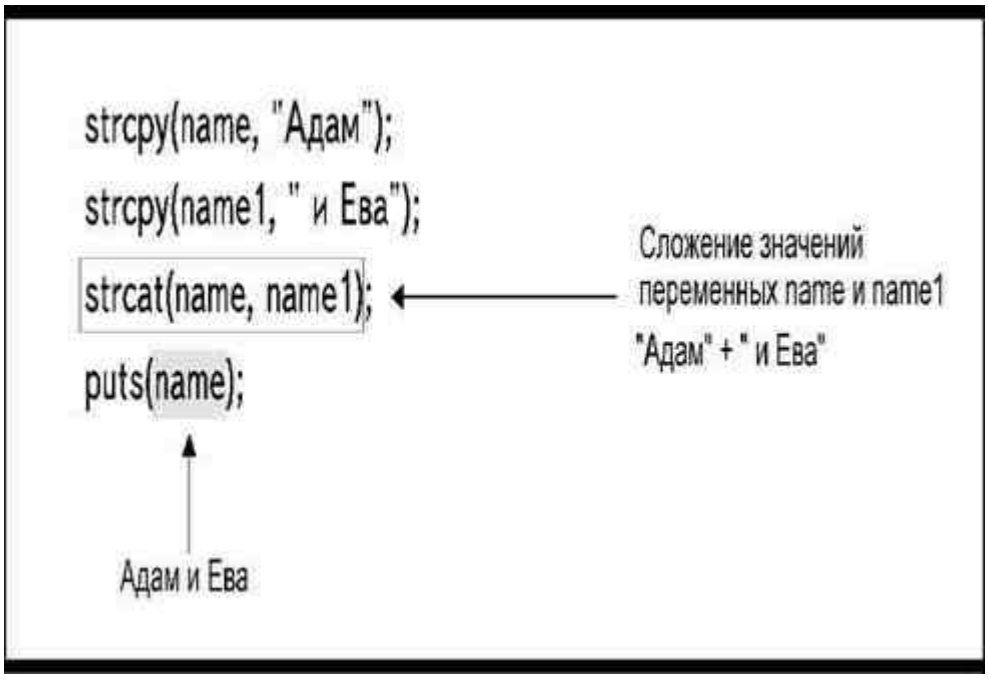

Рис. 10.9. Синтаксис функции strcat()

Замечания по Си++

Благодаря перегрузке язык Си++ допускает использование оператора + для сложения двух строк с помощью инструкции:

```
new_string = string1 + string2;
```

Слияние строк

Процедура слияния двух строк состоит в том, что символы, содержащиеся в одной строке, добавляются в конец другой, сдвигая нулевой символ. Этот процесс называется конкатенацией (concatenation). В стандарте языка Си K&R определена функция strcat(), работа которой продемонстрирована на рис.10.9. Символы из строки, которая передается функции в качестве второго параметра, добавляются в конец строки, передаваемой в качестве первого параметра.

Массивы строк

Можно организовать массив строк точно так же, как массив данных любого другого типа. Но массив строк, по сути, будет являться уже массивом массивов символов. Массив, элементы которого сами являются массивами, называется двухмерным массивом.

Двухмерный массив можно представить себе как таблицу, имеющую ряды и колонки. Такой массив следует определять с двумя индексами, один из которых определяет количество рядов таблицы, а второй устанавливает количество колонок. Ниже приведены инструкции, определяющие массив, имеющий 10 рядов и 20 колонок, то есть содержащий 200 целочисленных переменных:

int table[10][20];

Представим себе каждый элемент как целое число, занимающее собственную

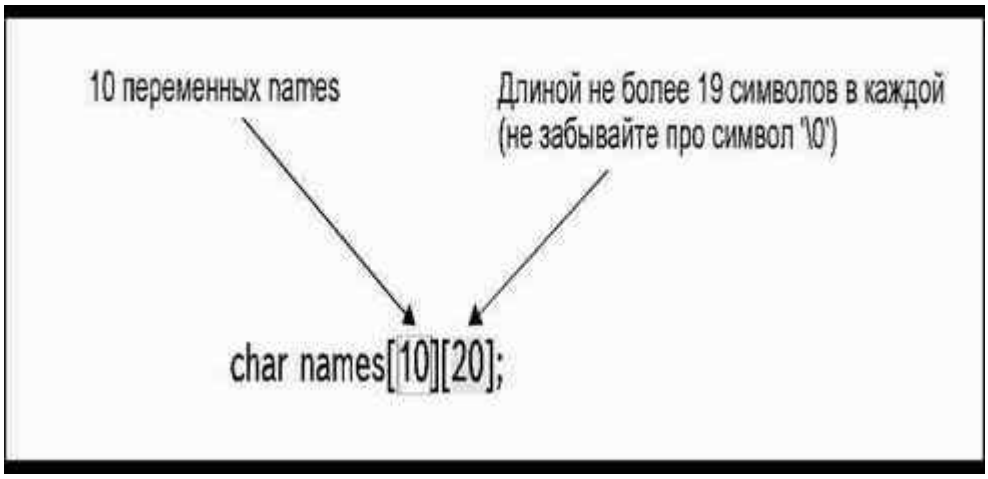

Рис. 10.10. Определение массива строк

клеточку в таблице 10Ф20. Элемент table[0][0] находится в левом верхнем углу таблицы, а элемент table[0][1] занимает соседнюю клетку справа в том же ряду.

Определяя массив строк, также необходимо использовать два индекса. Первый определяет максимальное количество строк в массиве, а второй указывает максимальную длину каждой строки. Таким образом, определение

char names[10][20];

задает десять строковых переменных names длиной не больше 19 символов в каждой (рис.10.10).

Если вы хотите задавать строки путем ввода значений отдельных символов, следует использовать вложенные циклы. Внешний цикл будет повторяться 10 раз, по одному на каждую строку, а внутренний должен иметь 19 повторов для ввода значений одной строки. На рис.10.11 приведена программа, в которой вводятся значения 10 имен (строковых переменных) в массив, а затем все эти 10 строк с помощью цикла for последовательно выводятся на экран.

Для того чтобы ввести символ в строку, необходимо использовать оба индекса:

name[index][index2] = letter;

Первый индекс указывает номер нужной строки внутри массива, а второй определяет позицию символа внутри строки. Например, на рис.10.12 элемент name[0][3] является четвертым символом (буква м) первой строки ("Адам") в

```
/*sarray.c*/
main()
{
char name[10][20], letter;
int index, index2;
for (index = 0; index < 10; index++)
{
index2 = 0;
puts("Введите имя, для завершения \
нажмите Enter\n");
do
{
letter = getchar();
name[index][index2] = letter;
index2++;
}
while (letter != '\n' && index2 < 19);
name[index][index2] = '\0';
putchar('\n');
}
for (index = 0; index < 10; index++)
puts(name[index]);
putchar('\n');
}
```

Во внешнем цикле вводятся 10 имен

Во внутреннем цикле вводятся символы (не больше 19) в каждое имя

Рис. 10.11. Программирование массива строк

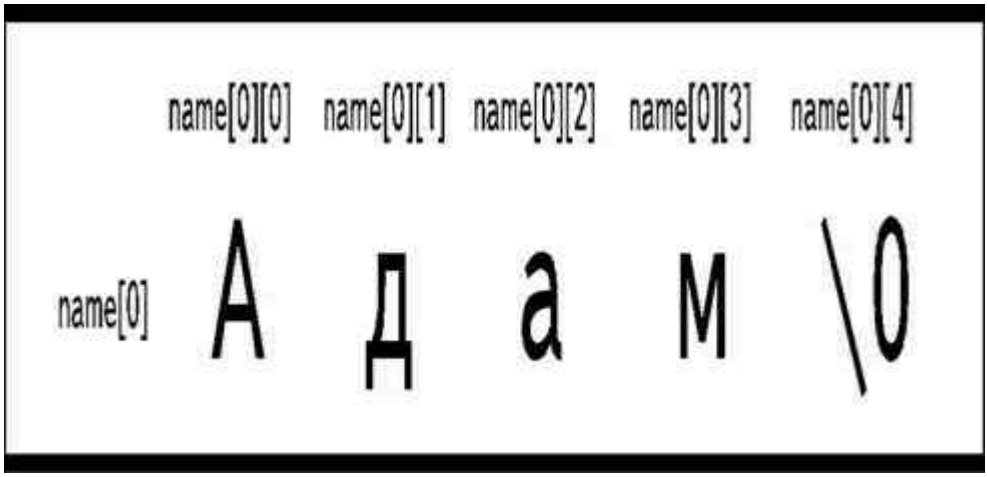

Рис. 10.12. Строки и элементы массива строк

массиве имен. Чтобы вывести на экран строку целиком, нужен только один индекс, указывающий номер строки:

```
puts(name[index]);
```

К счастью, Си и Си++ позволяют ввести всю строку как единое целое, используя только один цикл, повторяющийся по одному разу на каждую строку массива:

```
main()
        {
        char name[10][20];
        int index;
```

```
for (index = 0; index < 10; index++)
        gets(name[index]);
for (index = 0; index < 10; index++)
        puts(name[index]);
}
```

Проектирование программы

Так как определение максимального количества элементов массива должно производиться в самом начале программы или функции, решение о том, какого размера должен быть массив, является одним из первых решений, которое вам придется принять при планировании программы. Если при определении массива его максимальный размер окажется меньше, чем потребуется при выполнении программы, это приведет к ошибке выполнения или получению ни с чем несообразных результатов.

С другой стороны, следует избегать определения массивов, имеющих чересчур большое количество элементов, с целью действовать наверняка. Массивы, особенно состоящие из значений типа float или строк,— это настоящие пожиратели памяти. Когда ваша программа становится достаточно большой и сложной, определение массивов, имеющих значительно большие размеры, чем это действительно необходимо, может привести к ошибке выполнения, связанной с нехваткой памяти.

Таким образом, вы оказываетесь между Сциллой и Харибдой. Поэтому планируйте размеры массива с особенной тщательностью. Старайтесь оставлять допуск не слишком большой, но достаточный для того, чтобы быть уверенным, что номер индекса никогда не превысит максимального количества элементов массива.

Вопросы

1. Что такое массив?
2. Может ли массив содержать переменные нескольких типов?
3. Как вы будете определять массив?
4. Что такое двухмерный массив?
5. Как вы будете определять двухмерный массив?
6. Какова взаимосвязь между значением индекса и значением элемента массива?
7. Как сравнить две строки?
8. Как присвоить значение строковой переменной?

Упражнения

1. Напишите программу, в которой массивы используются для хранения имен, адресов и номеров телефонов 20 человек.
2. Внесите в программу из упражнения 1 изменения так, чтобы иметь возможность ввода имени и последующего просмотра массива в поисках номера телефона соответствующего человека.
3. Объясните, почему следующая программа написана неправильно:

```
4.   main()
5.           {
6.           int temps(31);
7.           int index, total;
8.           for (index = 0; index < 31; index++)
9.                   {
```

```
10.                         printf("Введите значение
11.  температуры #%d: ", index);
12.                         scanf("%d", &temps(index));
13.                         }
14.                 high = temps(0);
15.                 low = temps(0);
16.                 index = 1;
17.                 while (index < 31)
18.                         {
19.                         if (temps(index) > high)
20.                                 high = temps(index);
21.                         else
22.                                 low = temps(index);
23.                         index++;
24.                         }
25.                 printf("Минимальное значение
26.  температуры равно %d\n", low);
27.                 printf("Максимальное значение
28.  температуры равно %d\n", high);
                    }
```

ГЛАВА 11
СТРУКТУРЫ И УКАЗАТЕЛИ

Все переменные, которые мы использовали в этой книге до настоящего момента, относились к одному определенному типу, они были или символьными, или строковыми, или целочисленными, или с плавающей точкой. Даже массивы независимо от их размерности содержали переменные только одного типа. Все встречавшиеся нам переменные были простыми, то есть такими переменными, значения которым присваиваются путем простого использования имени переменной в инструкциях ввода данных или присваивания.

В этой главе будет рассказано о двух классах переменных, которые являются более многоликими: структурах и указателях. Структурой называется переменная, представляющая собой множество других переменных, которые могут относиться к различным типам. Указателем называется переменная, с помощью которой можно обращаться к области памяти, отведенной под другую переменную.

Структуры и указатели являются основой, обеспечивающей возможность для развития сложных приложений.

Использование структур

В программе зачастую требуется отразить реальную ситуацию, когда некую информацию нужно организовать и обрабатывать систематически. Такая необходимость возникает в бухгалтерских системах, при решении инженерных проблем, в обучающих системах и во многих других приложениях.

В программах мы обычно представляем каждый элемент данных в виде переменной, определенной с типом float, int, char или как строка. Но в реальной жизни нам часто приходится иметь дело с объектами, которые содержат не один, а несколько типов данных. Предположим, что вы составляете каталог своих компакт-дисков. Скажем, у вас есть набор карточек, каждая из которых содержит название, описание, цену, отношение к музыкальному стилю и номер, под которым диск хранится на стеллаже (рис.11.1).

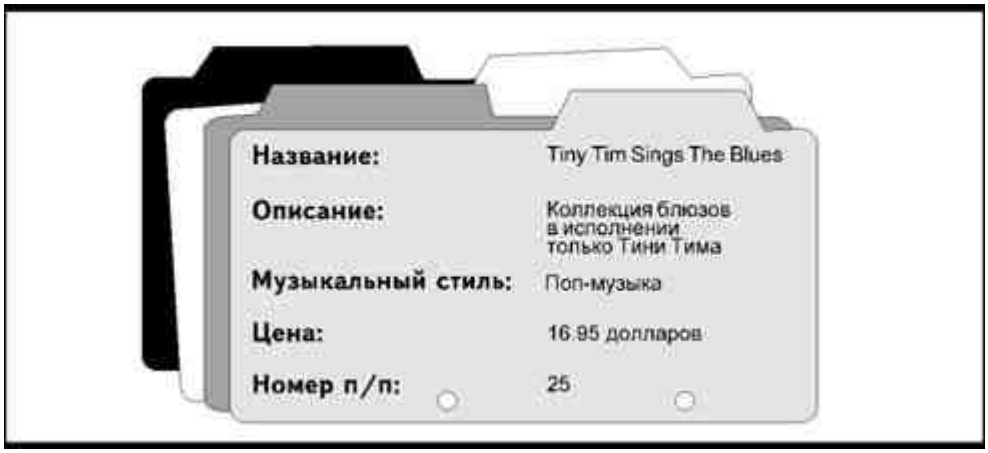

Рис. 11.1. Не компьютеризированная картотека

Вы сможете легко компьютеризировать свою картотеку, если присвоите каждый из элементов соответствующей переменной:

```
char name[20], description[40], category[12];

float cost;

int number;
```

Однако в реальной жизни всеми этими сведениями мы пользуемся не по отдельности, а заносим их вместе на одну регистрационную карточку, ведь несмотря на то, что каждый бит информации на карточке представляет собой

Замечания по Си++

В Си++ структуры часто используют для определения классов. Классом называется структура, которая может содержать как переменные, так и функции. Определяя функцию внутри класса, ее можно связать со структурой, как, например, в этом примере:

```
struct square {
```

```
float number;
void assignumber(double);
float squareit(void);
}              amount;
void square :: assignumber(float num)
{
number = num;
}
void square :: squareit(void)
{
float toreturn;
toreturn = number * number;
return(toreturn);
}
square.assignumber(25.0);
cout << "" << square.squareit();
```

Символ :: в определении функции называется оператором области видимости (по-английски— scoping) и связывает функцию с классом.

отдельный и независимый элемент, все вместе они составляют целостность, а именно, описывают один из объектов коллекции.

Если вы хотите иметь программу, которая адекватно отражала бы эту ситуацию, вам придется каким-то способом объединить данные разных типов во множество. К счастью, в языке Си и Си++ действительно существует возможность объединения данных разных типов в единое целое, представляющее собой как бы компьютерный вариант регистрационной карточки. Такая совокупность данных называется *структурой*.

Определение структуры

Первым шагом в создании взаимосвязанного множества переменных является определение структуры. Определяя структуру, вы тем самым определяете собственный тип данных. Вы даете структуре имя и указываете компилятору имя и тип каждого элемента данных, которые должны содержаться в структуре (рис.11.2).

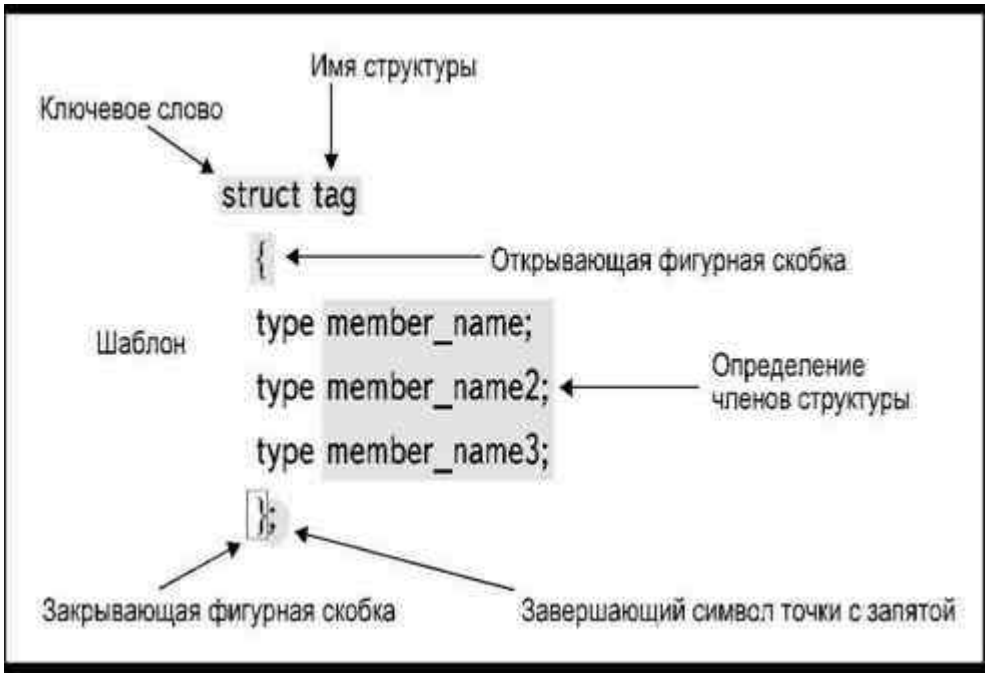

Рис. 11.2. Элементы структуры

Определение структуры должно начинаться с ключевого слова struct, за которым следует имя структуры, называемое *типом записи* (tag). Элементы данных, которые составляют структуру, называются *членами структуры* и помещаются между открывающей и закрывающей фигурными скобками.

Синтаксис определения членов структуры аналогичен синтаксису определения переменной. Необходимо указать типы данных каждого члена и размер всех строк и массивов. Определение каждого члена структуры заканчивается точкой с запятой. Точка с запятой отмечает также и конец определения структуры.

Список членов структуры носит название *шаблона* (template). Структура на самом деле не определяет никаких переменных. Члены структуры сами по себе не являются переменными, они представляют собой компоненты одной или нескольких переменных. Такие переменные называются *структурными переменными* и должны быть определены как имеющие тип соответствующей структуры. Шаблон определяет эти компоненты и говорит компилятору, сколько памяти следует зарезервировать для каждой структурной переменной.

Вы можете определить вашу картотеку компакт-дисков с помощью следующей структуры:

```
struct CD
        {
        char name[20];
        char description[40];
        char category[12];
        float cost;
        int number;
        };
```

Таким образом, вы создаете новый тип данных с именем CD, который содержит пять элементов информации: три строки, одно значение типа float и одно целочисленное значение. Размер области памяти, зарезервированной для структуры, составит 78 элементов (рис.11.3).

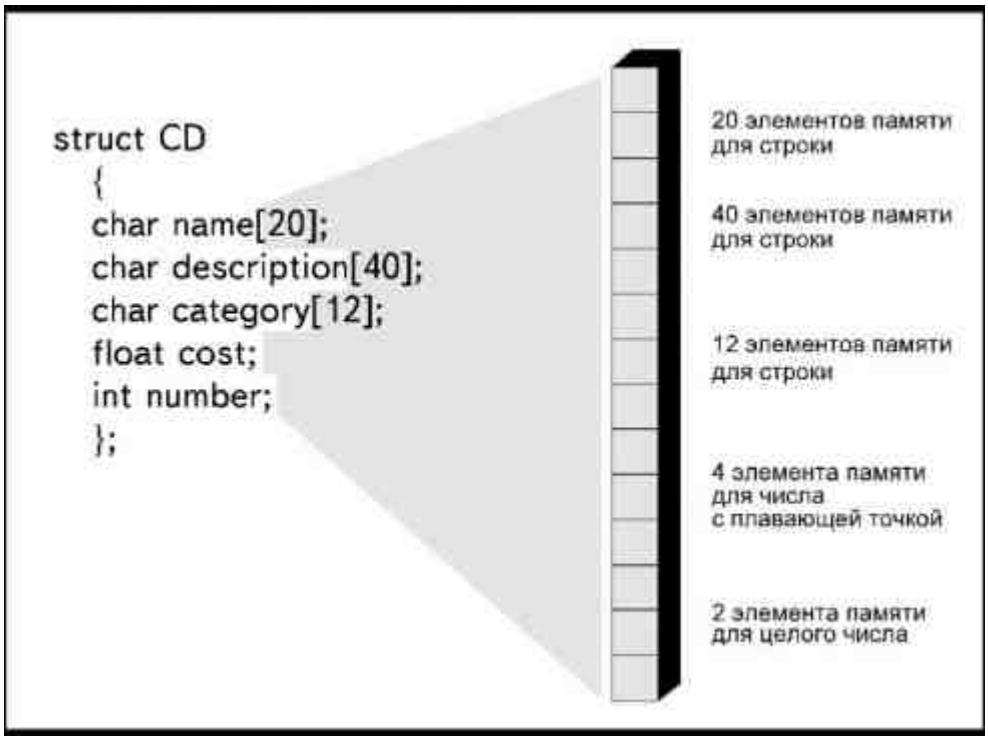

Рис. 11.3. Структура резервирует область памяти для своих членов

Определение структурных переменных

Определение структуры еще не отводит память для нее. Помните, прежде чем использовать любой тип данных, необходимо определить соответствующую переменную. Нельзя использовать данные типа float, например, до того, как определена float-переменная. Аналогично, нельзя использовать структуру до того, как определена переменная структурного типа. Синтаксис определения структурной переменной приведен на рис.11.4.

Ключевое слово struct указывает компилятору на то, что он имеет дело со структурой, а тип записи определяет шаблон, к которому относится переменная. Следом за типом записи идет имя переменной, которое

будет использоваться в программе. Например, чтобы получить доступ к картотеке CD, следует определить переменную

struct CD disc;

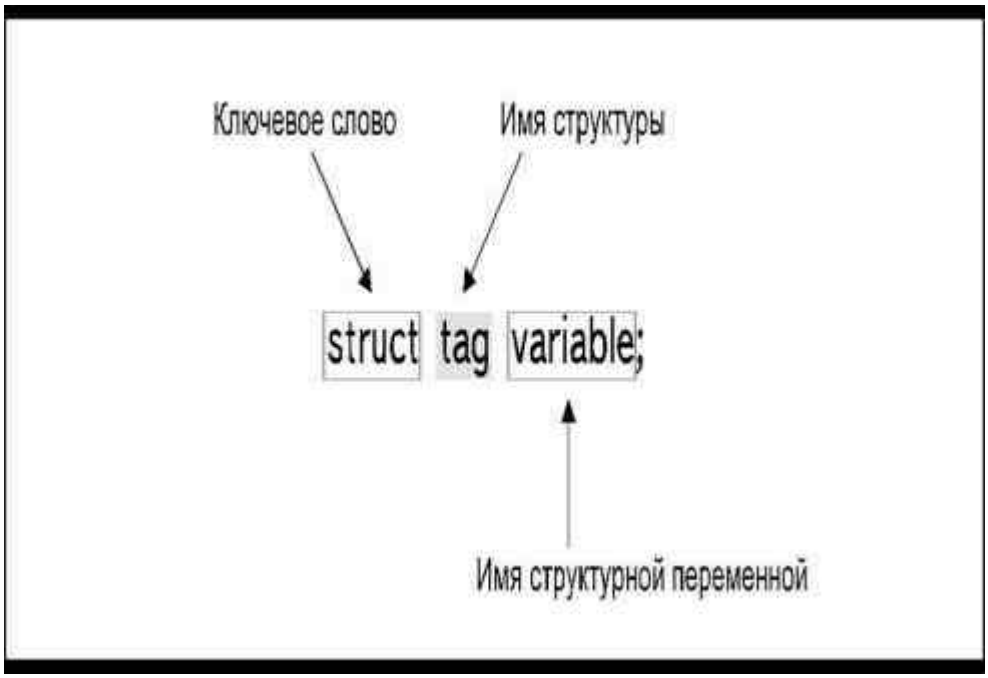

Рис. 11.4. Определение структурной переменной

Теперь у нас есть переменная disc, которая является множеством из пяти элементов. Как и для любой другой переменной, для переменной disc выделяется область памяти. Но так как disc является структурной переменной типа CD, для нее отводится область памяти размером в 78 элементов, в которой будут содержаться значения трех строк и двух чисел: float и int (рис.11.5).

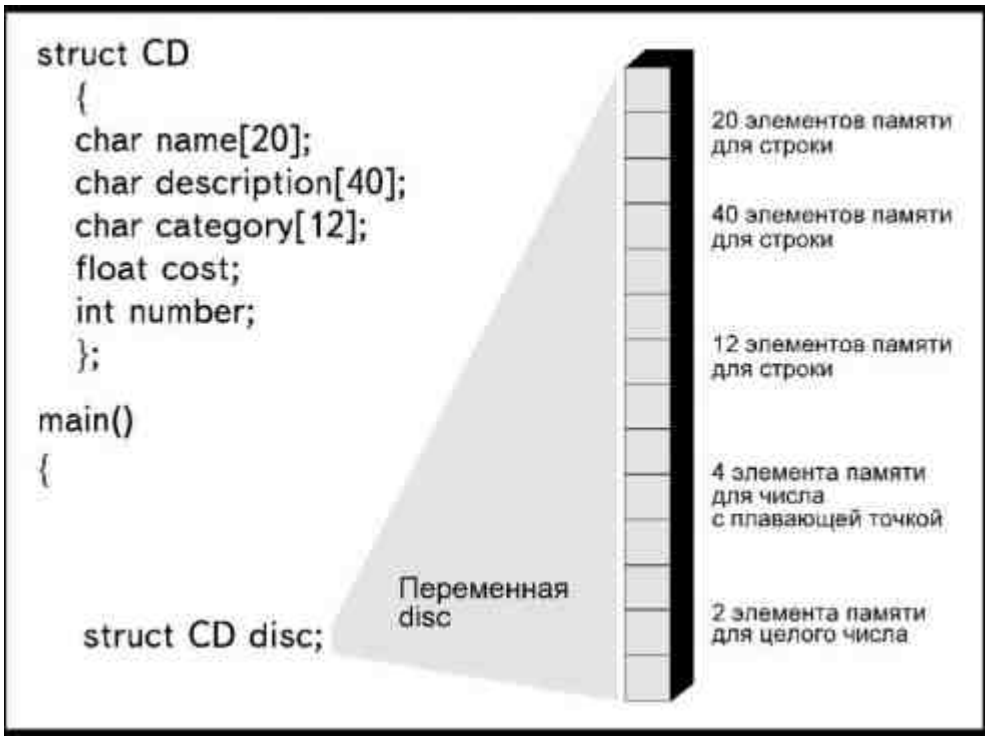

Рис. 11.5. Определение структурной переменной

Если в программе нужно иметь несколько переменных одного структурного типа (например, CD), можно определить их в одной инструкции:

struct CD disc, cdrom;

Здесь определяются две переменные, disc и cdrom, относящиеся к одному структурному типу CD. Однако если вы хотите определить структурные переменные нескольких типов, их следует определять по отдельности. Например, если бы у вас были определены две структуры с именами CD и VIDEO, инструкции определения соответствующих переменных следовало бы писать так:

```
struct CD disc, cdrom;

struct VIDEO movies, vacation;
```

В принципе, можно создать структуру и определить переменную в один прием. Для этого следует поместить имя переменной (или их имена) между закрывающей фигурной скобкой определения структуры и точкой с запятой:

```
struct CD

        {

        char name[20];

        char description[40];

        char category[12];

        float cost;

        int number;

        } disc;
```

Для того чтобы разобраться со всеми терминологическими тонкостями, давайте посмотрим на каждый элемент в определении:

- CD является *типом записи структуры*; это имя нового типа данных— множества элементов, называемого структурой;
- элементы, составляющие структуру, называются *членами структуры*; при их определении используется тот же синтаксис, что и при определении переменной;
- disc является именем переменной, которая будет использоваться в программе; это переменная типа CD, и она содержит все члены, определенные как части структуры; иногда такую переменную называют *структурной переменной*.

Присвоение начального значения

В том случае, если начальные значения членов структуры известны, можно присвоить их при определении переменной. Если вы создаете только одну переменную структурного типа, можно инициализировать ее как часть определения структуры:

```
struct CD

        {

        char name[20];

        char description[40];

        char category[12];

        float cost;

        int number;

        } disc = {"Лучшие песни", "Тини Тим",

 "поп-музыка", 12.50, 12};
```

В этих инструкциях мы создали структуру CD, определили переменную disc и присвоили начальные значения всем пяти членам структуры (рис.11.6).

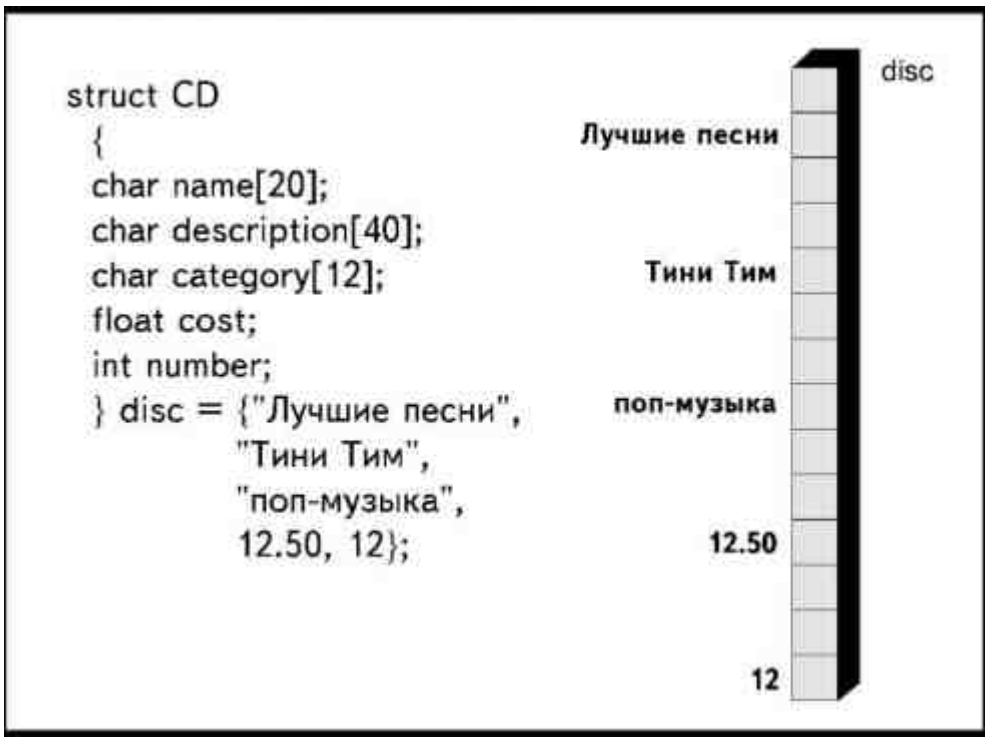

Рис. 11.6. Инициализация структурной переменной

Как другой возможный вариант, можно инициализировать члены структуры при определении переменной:

```
struct CD disc = {"Моя жизнь", "Биография Б. Гейтса",
                  "книга на диске", 24.99, 213};
```

Поскольку Си является языком свободного формата, можно записать процедуру присваивания в несколько строк, однако символы внутри кавычек при этом переносить не следует.

Структуру можно определить как глобальную, перед функцией main(), или как локальную, записав определение внутри main() или другой функции. Но! Если вы хотите присвоить начальное значение структуре, которая содержит строки, ее обязательно следует определять перед main(), или как статическую переменную:

```
static struct CD;
```

Использование структуры

Член структуры всегда является частью структуры. Нельзя обратиться к структурному элементу как к таковому. Например, если вы попытаетесь присвоить значение члену cost структуры CD, используя следующий синтаксис

```
cost = 23.13;
```

компилятор сгенерирует ошибку, вызванную тем, что переменная cost не была определена. Для переменной с таким именем не была зарезервирована никакая область памяти. Переменная, которая вас интересует, называется disc, и именно она содержит элемент с именем cost. Для того чтобы обратиться к члену структуры, необходимо указать его вместе с именем переменной, используя следующий формат:

```
structure_variable.member_name
```

Такая запись означает, что необходимо указать имя структурной переменной, поставить точку, а затем указать имя члена структуры, как это сделано в инструкциях:

```
gets(disc.name);
gets(disc.description);
gets(disc.category);
disc.cost = 16.95;
disc.number = 5;
```

Эти инструкции говорят компилятору, в какую именно область памяти следует поместить введенное значение, например, в член name структурной переменной disc.

Получение значений членов структуры осуществляется аналогичным образом, с точкой между именем структурной переменной и именем члена. В Листинге11.1 приведен текст программы, в которой вводится,

а затем отображается на экране информация о картотеке компакт-дисков. В тексте программы нет ни одного случая, когда бы имя члена структуры использовалось без указания имени структурной переменной.

Вводить и выводить значения элементов структуры можно в любом порядке, не обязательно придерживаться именно того, который указан в описании структуры.

Листинг 11.1. Использование структуры при вводе и выводе.

```
/*CD1.c*/
struct CD
        {
        char name[20];
        char description[40];char category[12];
        float cost;
        int number;
        } disc;
main()
        {
        puts("Введите сведения о диске\n\n");
        printf("Введите название: ");
        gets(disc.name);
        printf("Введите описание: ");
        gets(disc.description);
        printf("Введите категорию: ");
        gets(disc.category);
        printf("Введите цену: ");
        scanf("%f", &dics.cost);
        printf("Введите номер ячейки: ");
        scanf("%d", &disc.number);
        puts("Введена следующая информация о диске:\n\n");
        printf("Название:      %s\n", disc.name);
        printf("Описание:      %s\n", disc.description);
        printf("Категория:     %s\n", disc.category);
        printf("Цена:                  %6.2f\n", disc.cost);
        printf("Номер п/п:     %d\n", disc.number);
        }
```

Массивы структур

До сих пор мы работали с эквивалентом только одной регистрационной карточки. Но мы определили структуру с тем, чтобы хранить информацию обо всей коллекции записей. Для этого надо использовать не одиночную переменную, а массив. Создавая массив структур, мы создаем массив шаблонов. Такой массив можно определить, используя индекс с именем структурной переменной:

```
struct CD disc[10];
```

Таким образом, мы резервируем десять областей памяти, каждая из которых обладает достаточным объемом, чтобы хранить целую структуру (рис.11.7). Обращение к элементу массива происходит путем указания соответствующего индекса после имени самой структурной переменной, а не члена структуры:

```
gets(disc[0].name);
gets(disc[1].name);
```

В Листинге 11.2 приведен текст программы, использующей массив структур для выполнения задачи, которая может встретиться в реальной жизни, а именно, для заполнения регистрационных карточек, составляющих картотеку коллекции компакт-дисков.

Листинг 11.2. Использование массива структур.

```
/*CD2.c*/
struct CD
        {
        char name[20];
        har description[40];
        har category[12];
        float cost;
        int number;
        } disc[10];
main()
        {
        int index, repeat;
        har flag;
        flag = 'Y';
        index =0;
        do
                {
                puts("Введите сведения о диске
 #%d\n", index);
                printf("Введите название: ");
                gets(disc[index].name);
                printf("Введите описание: ");
                gets(disc[index].description);
                printf("Введите категорию: ");
                gets(disc[index].category);
                printf("Введите цену: ");
                scanf("%f", &dics[index].cost);
                printf("Введите номер ячейки: ");
                scanf("%d", &disc[index].number);
                index++;
                if(index < 10)
                        {
                        printf("Желаете ввести
 данные о следующем диске? Да -Y, нет -N");
                        scanf("%C", &flag);
                        }
                }
        while (index < 10 && (flag == 'Y' || flag == 'y'));
        puts("Название                    Номер п/п");
        for (repeat = 0; repeat < index; repeat++)
```

```
                    printf("%s %d\n", disc[repeat].name,
disc[repeat].number);
            }
```

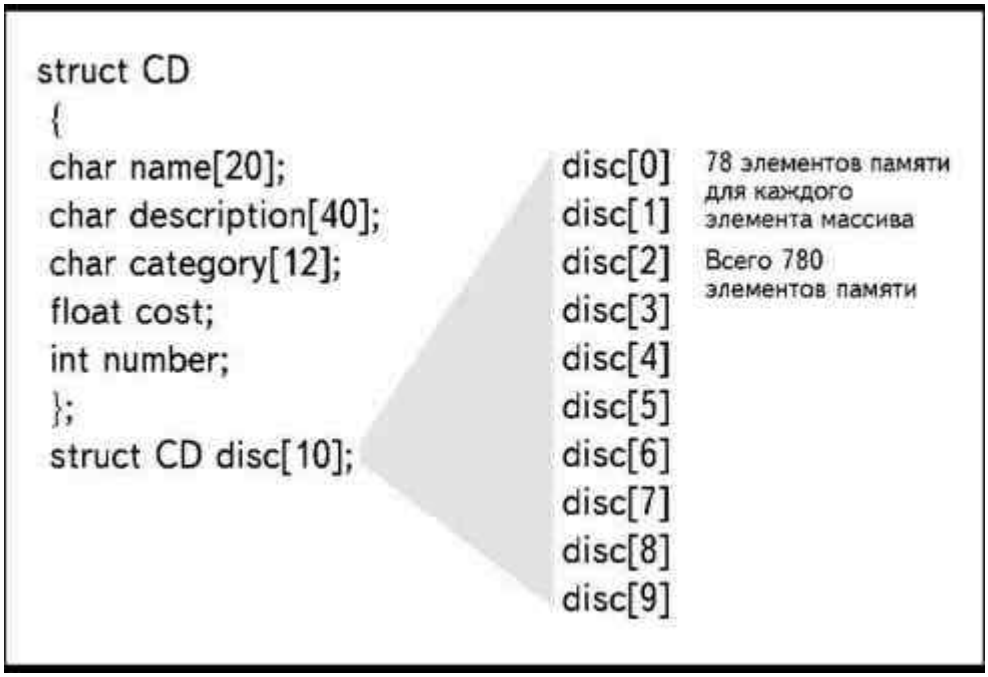

Рис. 11.7. Массив структур

Определение массива, включающего 10 элементов disc, является частью определения структуры:

```
} disc[10];
```

Переменная flag ведена для того, чтобы пользователь мог указать, хочет ли он продолжить ввод информации, если заполнено меньше 10«карточек». Цикл do обеспечивает ввод элементов массива: пять пунктов (членов структуры) для каждого элемента массива (регистрационной карточки в картотеке коллекции). Переменная index используется как индекс в процессе создания картотеки.

После ввода значений каждого элемента массива, программа спрашивает пользователя, хочет ли он заполнить следующую карточку. После ввода значений всех 10элементов массива запрос больше не предъявляется.

В цикле do используется следующее условие:

```
while (index < 10 && (flag == 'Y' || flag == 'y'));
```

Логический оператор И указывает на то, что оба условия должны выполняться одновременно: значение переменной index должно быть меньше десяти, и переменная flag должна иметь значение 'Y' или 'y'. Если бы в условии не были поставлены внутренние скобки, компилятор Си скомбинировал бы первые два условия и трактовал всю инструкцию как условие ИЛИ (рис.11.8). И если в

Рис. 11.8. Интерпретация условия при отсутствии внутренних круглых скобок

этом случае переменная flag примет значение 'y', цикл будет повторяться и после ввода значений всех элементов.

Когда будут введены данные последнего элемента массива (карточки), значение переменной index окажется на единицу больше количества элементов массива. Эта переменная в дальнейшем используется в

цикле for, в котором на дисплей выводится информация о содержании компакт-дисков и их местонахождении.

Обратите внимание на то, что все обращения к членам структуры включают имя переменной с указанием индекса массива.

Структуры и функции

В исходном K&R-стандарте языка Си использование структур ограничено. Структуры могут передаваться в качестве аргументов только с использованием указателей, о которых мы будем говорить дальше в этой главе. Кроме того, отсутствует возможность прямого присваивания одной структуры другой, например, следующим образом:

```
cdrom = disc;
```

В современных компиляторах Си и Си++ такая возможность существует. Теперь можно непосредственно присваивать одну структурную переменную другой.

Большинство компиляторов Си++ и компиляторы Си, поддерживающие стандарт ANSI, позволяют передавать и возвращать структуру целиком. В Листинге11.3 продемонстрировано, как можно передать структуру функции. Значения членов структуры вводятся в main(), затем вся структура передается функции putdisc() для вывода. При вызове функции используется структурная переменная disc:

```
putdisc(disc);
```

Таким образом, структура целиком передается функции. Функция содержит переменную (формальный аргумент), которая будет получать передаваемую структуру. Получающая переменная определяется с тем же типом структуры:

```
putdisc(disk)
struct CD disk;
```

Теперь функция может использовать отдельную структурную переменную с именем disk, которая содержит передаваемые члены структуры disc. Внутри функции обращение к членам структуры происходит с использованием структурной переменной disk, то есть имени получающей переменной.

Листинг 11.3. Передача структуры.

```
/*CD3.c*/
struct CD
        {
        char name[20];
        char description[40];
        char category[12];
        float cost;
        int number;
        } disc;
main()
        {
        puts("Введите сведения о диске\n\n");
        printf("Введите название: ");
        gets(disc.name);
        printf("Введите описание: ");
        gets(disc.description);
        printf("Введите категорию: ");
        gets(disc.category);
        printf("Введите цену: ");
        scanf("%f", &dics.cost);
        printf("Введите номер ячейки: ");
        scanf("%d", &disc.number);
```

```
        putdisc(disc);
        }
putdisc(disk)
struct CD disk;
        {
        puts("Введена следующая информация о диске:\n\n");
        printf("Название:     %s\n", disc.name);
        printf("Описание:     %s\n", disc.description);
        printf("Категория:    %s\n", disc.category);
        printf("Цена:                         %6.2f\n", disc.cost);
        printf("Номер п/п:    %d\n", disc.number);
        }
```

В Листинге 11.4 демонстрируется, как функция возвращает структуру. Члены структуры вводятся в функции getdisc(), а затем передаются назад в main() с помощью инструкции

```
return(inputdisc);
```

Обратите внимание, что тип записи структуры (CD) используется и в определении функции, и в определении переменных функции getdisc(). Это вызвано тем, что при возврате переменных, отличных от типов int или char, тип переменной следует указать в определении функции. Типом возвращаемой переменной в данном случае является структура CD, так что функция определяется следующим образом:

```
struct CD getdisc();
```

Структура, используемая в функции, также относится к типу CD, поэтому она определяется как:

```
struct CD inputdisc;
```

Обратите также внимание на то, что функция getdisc() определяется как глобальная, вместе со структурной переменной disc. Такой тип определения требуется в тех случаях, когда функция возвращает структуру.

Листинг 11.4. Возвращение структуры.

```
/*CD4.c*/
struct CD
        {
        char name[20];
        char description[40];
        char category[12];
        float cost;
        int number;
        } disc, getdisc();
main()
        {
        disc = getdisc();
        puts("Введена следующая информация о диске:\n\n");
        printf("Название:     %s\n", disc.name);
        printf("Описание:     %s\n", disc.description);
        printf("Категория:    %s\n", disc.category);
        printf("Цена:                         %6.2f\n", disc.cost);
        printf("Номер п/п:    %d\n", disc.number);
        }
struct CD getdisc()
```

```
{
struct CD inputdisc;
puts("Введите сведения о диске\n\n");
printf("Введите название: ");
gets(inputdisc.name);
printf("Введите описание: ");
gets(inputdisc.description);
printf("Введите категорию: ");
gets(inputdisc.category);
printf("Введите цену: ");
scanf("%f", &inputdiscdics.cost);
printf("Введите номер ячейки: ");
scanf("%d", &inputdisc.number);
return(inputdisc);
}
```

Указатели

В то время как функции можно передавать столько аргументов, сколько требуется (в том числе и структуры), с помощью инструкции return() возвращается только один параметр. Иными словами, функция может вернуть вызывающей процедуре только одно значение. В качестве альтернативы можно использовать глобальные переменные, однако в этом случае зачастую теряется контроль над программой и затрудняется поиск ошибок. Когда возникает необходимость передать назад больше одного значения, можно использовать указатели.

К переменной можно обратиться одним из двух способов. Используя имя переменной, вы обращаетесь к значению переменной, которое хранится в памяти. Используя оператор получения адреса (&), вы обращаетесь к тому адресу в памяти, где хранится значение переменной.

Когда вы используете оператор присваивания

```
tax = 35;
```

вы, тем самым, вносите в элементы памяти некое *значение*. По адресу, соответствующему области памяти, отведенной для переменной tax, будет содержаться значение 35. Можно вывести на дисплей *адрес элемента памяти* переменной, если использовать оператор получения адреса. Инструкция

```
printf("%d", &tax);
```

отобразит адрес, по которому хранится значение переменной tax, но не само значение, присвоенное переменной и хранящееся в элементах памяти по этому адресу. Однако оператор получения адреса вместе с именем переменной (&tax) может быть использован только в выражении. Он не является переменной, и поэтому инструкция типа

```
&tax = 25;
```

будет ошибочной.

Указателем называется переменная, которая содержит значение адреса элемента памяти, где хранится значение другой переменной. Если значение переменной tax хранится в элементе, расположенном по адресу 21260, то и указатель на переменную tax будет иметь значение 21260*.

Чтобы создать указатель, используйте синтаксис, показанный на рис.11.9. Начать следует с определения типа данных, которые хранятся в элементах памяти, определяемых указателем. Символ «звездочка» говорит компилятору, что вы создаете указатель. В конце указывается имя переменной. Например, инструкция

```
int *taxptr;
```

создаст переменную типа указатель (об этом говорит звездочка перед именем) с именем taxptr, которая будет содержать адрес некой целочисленной переменной. Инструкция

```
float *net;
```

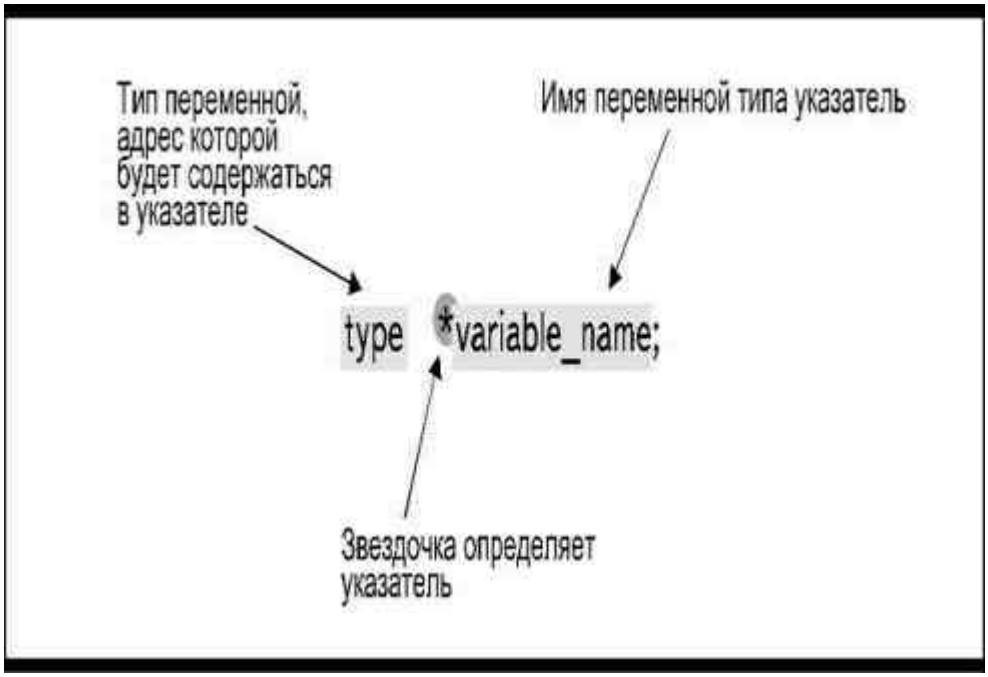

Рис. 11.9. Определение указателя

создаст указатель с именем net, который будет содержать адрес переменной типа float.

Если переменная типа указатель определена, она должна на что-то указывать. Для этого следует присвоить указателю значение в виде адреса соответствующей переменной:

taxptr = &tax;

В этих инструкциях мы помещаем адрес, по которому содержится значение переменной tax, в переменную taxptr (рис.11.10). Если значение переменной tax хранится по адресу 21260, переменная taxptr получит значение 21260.

*** Здесь имеется в виду только смещение переменной относительно начала сегмента данных (при работе в больших моделях памяти). Чтобы отобразить на экране полный адрес, нужно воспользоваться указателем формата "%p", а не "%d", как в приведенном примере. *(Прим.перев.)***

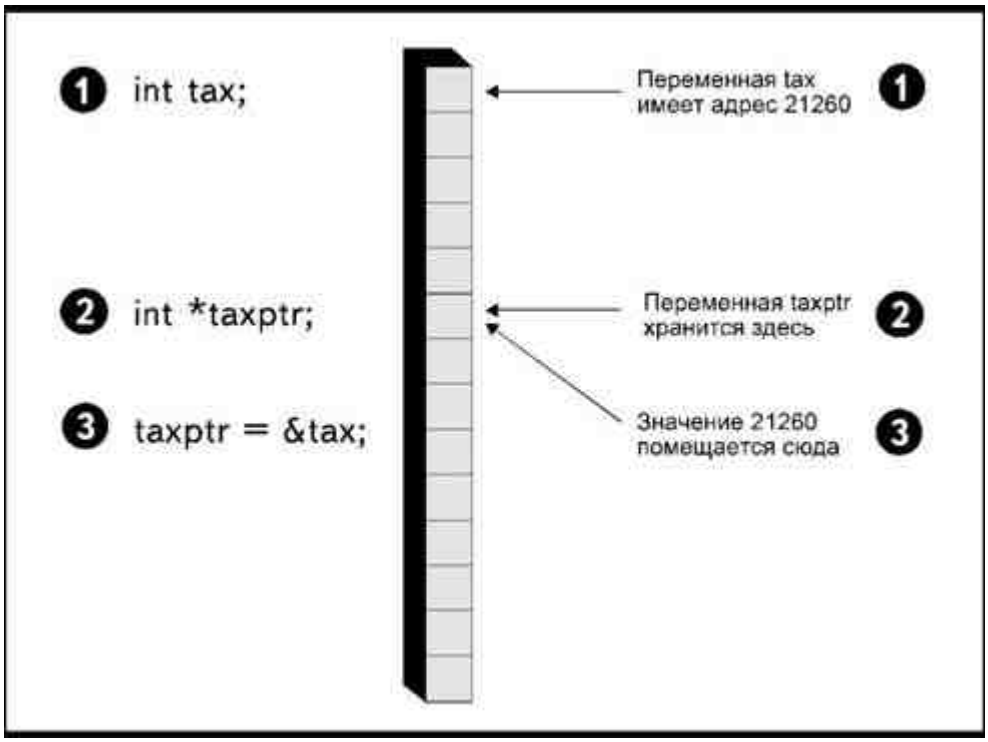

Рис. 11.10. Присваивание адреса указателю

В качестве примера рассмотрим программу:

main()

```
{
int *taxptr;
int tax;
taxptr = &tax;
tax = 35;
printf("Значение переменной tax равно %d\n", tax);
printf("Указатель переменной tax
имеет значение %d\n", taxptr);
}
```

Эта программа выведет на дисплей сообщения:

Значение переменной tax равно 35

Указатель переменной tax имеет значение 21260

Возможно, вы спросите: «Ну и что из этого? В конце концов, того же самого результата можно было добиться, присвоив значение адреса переменной tax обычной целочисленной переменной, а не какому-то указателю». Преимущество использования указателя состоит в том, что к нему можно обращаться как к

*taxptr. Звездочка в данном случае будет сообщать компилятору, что мы интересуемся *содержимым области памяти, адрес которой является значением указателя*. То есть нас интересует не значение 21260, а содержимое памяти по этому адресу. В то время как значение переменной taxptr равно 21260, значение *taxptr равно 35.

Значением, которое можно присвоить переменной taxptr, является только адрес. Попытка написать инструкцию

taxptr = 21260;

может привести к ошибке компиляции, так как здесь мы пытаемся присвоить целочисленное значение переменной, определенной как указатель. Единственное значение, которое может быть присвоено переменной taxptr, это адрес другой переменной, определенный с помощью оператора получения адреса:

taxptr = &tax;

Однако можно присвоить значение указателю *taxptr, написав инструкцию

*taxptr = 35;

Эта инструкция означает: «Поместить значение 35 в элемент памяти, на который указывает переменная taxptr». Поскольку указатель содержит значение 21260,

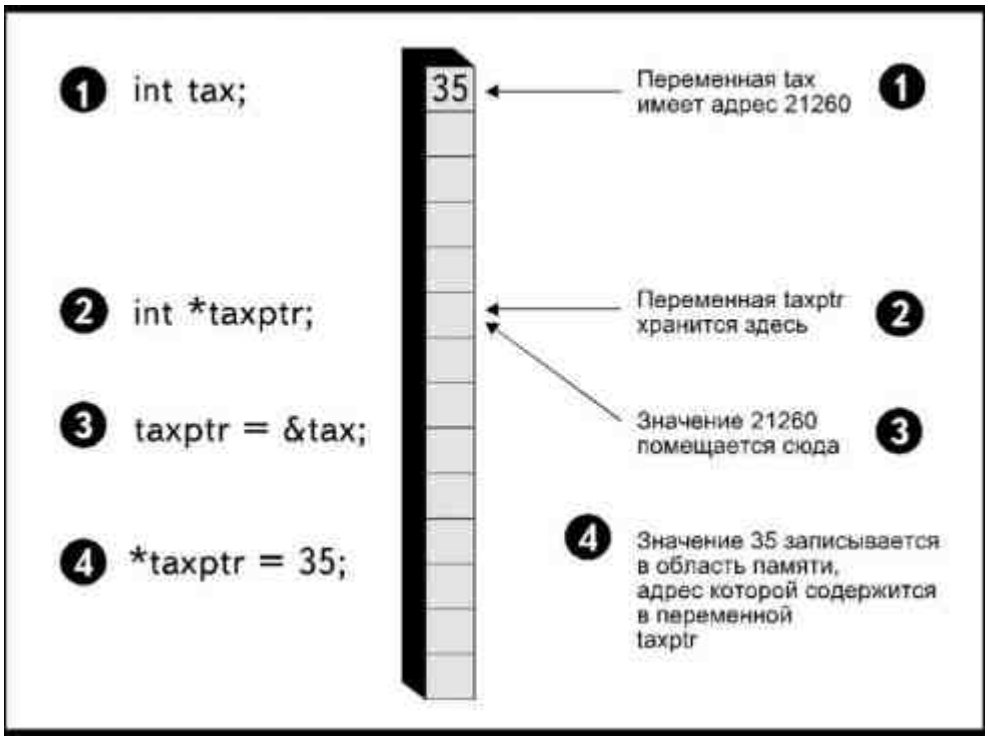

Рис. 11.11. Присваивание значения указателю

число 35 будет размещено в памяти именно по адресу 21260, а так как эту область памяти занимает переменная tax, она приобретет значение 35 (рис.11.11).

Разумеется, можно возразить, что такой способ изменения значения переменной является слишком длинным. Намного легче просто присвоить переменной tax новое значение. Все это так, но дело в том, что по-настоящему оценить эффективность применения указателей можно, например, когда они используются в качестве аргументов функций.

Указатели и функции

Без указателя мы должны передавать аргумент функции по значению. Это значит, что значение вызывающего параметра передается получающему параметру. Копируемые значения занимают отдельные области памяти, и изменение значения получающего параметра не приводит к изменению значения передающего параметра.

При передаче параметра по его адресу с помощью указателя мы не делаем дублирующих копий. Напротив, мы создаем вторую переменную, которая указывает на ту же область памяти. Таким образом, если мы изменяем значение переменной по указателю, то изменяем также и значение передающей переменной.

В программе, приведенной в Листинге 11.5, вводится значение строки и символа. Затем подсчитывается, сколько раз символ встречается в строке, и определяется позиция первого появления символа в строке.

Листинг 11.5. Программа, в которой используется указатель для возврата значений.

```
/*letcount.c*/
main()
        {
        char name[20], letter;
        int number, start;
        puts("Введите имя\n");
        gets(name);
        printf("Введите символ");
        letter = getchar();
        countlet(name, letter, &number, &start);
        printf("\nСимвол %c встречается в имени
 %s %d раз\n", letter, name, number);
        printf("Первый раз символ встречается
 в %d позиции", start);
        }
countlet(ndplume, alpha, count, first)
char ndplume[], alpha;
int *count, *first;
        {
        int index, flag;
        *count = 0;
        index = 0;
        flag = 0;
        *first = 0;
        while (ndplume[index != '\0')
                {
                if (ndplume[index] == alpha)
                        {
                        *count = *count+1;
                        if (flag == 0)
```

```
                    {
                    *first = index+1;
                    flag = 1;
                    }
                }
            index++;
            }
        }
```

После ввода строки и символа мы передаем функции countlet() четыре переменных: строку, символ и адреса переменных count и start, определенных с помощью оператора получения адреса.

Функция присваивает адреса получающим указателям: адрес переменной number хранится в *count, а адрес переменной start содержится в *first (рис.11.12).

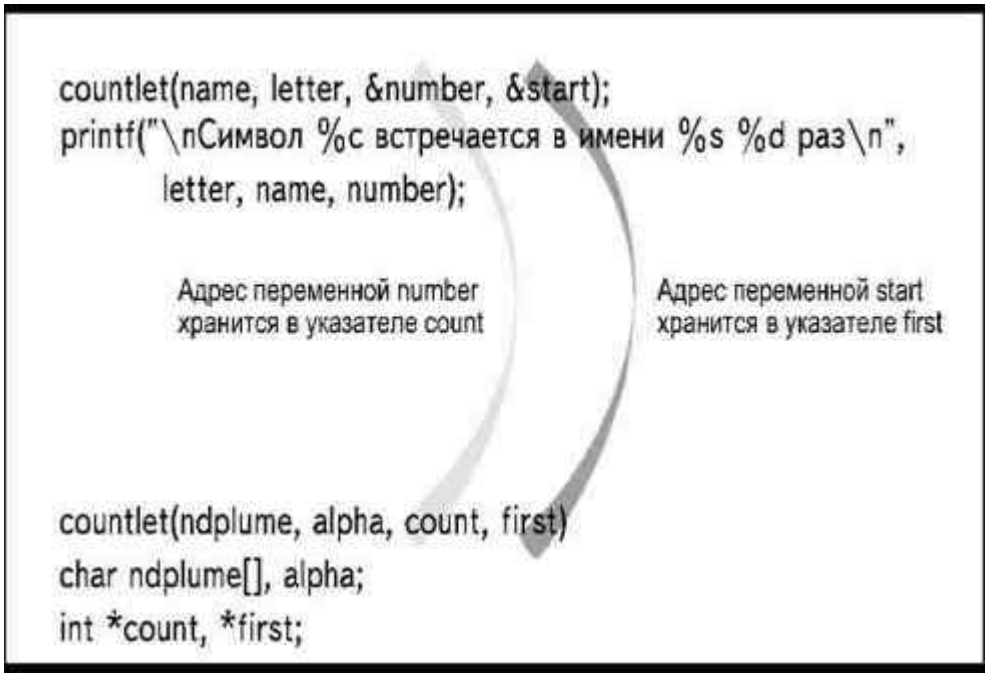

Рис. 11.12. Передача адресов функции

```
countlet(ndplume, alpha, count, first)
char ndplume[], alpha;
int *count, *first;
{
   int index, flag;
   *count = 0;
   index = 0;
   flag = 0;
   *first = 0;
while (ndplume[index] != '\0')
{
   if (ndplume[index] == alpha)
   {
     *count = *count+1;
   if (flag == 0)
   {
     *first = index+1;
     flag = 1;
   }
  }
   index++;
}
}
```

Если обнаружен искомый символ, значение счетчика увеличивается

Определяет первый случай присутствия искомого символа в строке

Инструкция index++; выполняется при каждом новом выполнении цикла

Рис. 11.13. Фигурные скобки отмечают блоки инструкций

Переменные инициализируются путем присваивания нулевого значения, а затем начинает выполняться цикл while:

while (ndplum[index] != '\0')

Цикл проверяет каждый символ в строке и продолжает проверку до тех пор, пока не встретит нулевой символ. Как только выясняется, что очередной символ является нулевым, выполнение цикла прекращается.

Положение фигурных скобок внутри цикла while является весьма существенным, так как одна инструкция if целиком вложена в другую (рис.11.13). Первое из условий if проверяет, является ли очередной встреченный символ тем, количество повторений которого мы подсчитываем. Если это так, значение счетчика, который в данном случае является указателем, увеличивается на единицу:

*count = *count + 1;

Это приводит к тому, что происходит увеличение на единицу значения, содержащегося в области памяти, адрес которой является значением указателя. Так как указатель содержит адрес переменной number, значение number увеличивается на 1, хотя эта переменная является локальной, определенной внутри функции main(). Обратите внимание, что в данном случае нельзя использовать синтаксис

*count++;

так как это приведет к изменению значения адреса, реально являющегося значением указателя, а не содержимого соответствующего элемента памяти*.

Второе условие if проверяет значение переменной flag. Если оно равно нулю, значит указанный символ встречен в строке впервые. При этом указателю *first присваивается значение, соответствующее номеру

символа в строке плюс единица. Добавляя единицу к индексу, мы указываем положение символа в строке, начиная отсчет с 1 (такой способ является наиболее привычным для большинства людей). После этого значение переменной flag меняется на 1, так что номер позиции первого встреченного в строке символа уже не будет меняться при обнаружении следующих символов.

После выполнения условия if значение индекса увеличивается, в результате чего при следующем прохождении цикла будет проверяться следующий символ строки. Весьма существенно, чтобы инструкция index++; помещалась после закрывающей фигурной скобки внешней инструкции if, но перед двумя заключительными закрывающими скобками, одна из которых отмечает конец инструкции while, а вторая завершает функцию. Если указанная инструкция будет помещена в каком-нибудь другом месте, функция не сможет успешно работать.

После завершения выполнения функции countlet() управление передается назад в main(), где осуществляется вывод информации, включающий вывод строки.

*** В этом случае можно использовать инструкцию (*count)++;.*(Прим.перев.)***

символа, количества появлений символа в строке и первого случая появления символа в строке.

Обратите внимание на то, что для возврата значения функция не использует инструкцию return. Вместо этого значения переменных number и start присваиваются через указатели. Другими словами, в программе возврат значений осуществляется путем использования указателей в качестве аргументов функции.

Можно написать ту же программу без использования указателей, определив переменные number и start как глобальные. Тогда и функция main(), и функция countlet() смогут обращаться к ним без передачи аргумента, однако использование указателей расширяет возможности контроля за работой программы. Переменные остаются локальными, но к ним можно обращаться и менять их значения путем передачи их адресов.

Использование указателей совершенно необходимо при работе с дисковыми файлами и выводе информации на принтер, что и будет являться предметом нашего разговора в главе 12.

Вопросы

1. Где может быть использован структурный тип данных?
2. Как определить структуру?
3. В чем заключается различие между типом записи структуры и структурной переменной?
4. Как обратиться к элементу структуры?
5. Может ли структура содержать элементы одного типа?
6. Что такое указатель?
7. Если в программе присутствует определение типа float *num, в чем будут выражаться различия между num и *num?
8. Для чего в языке Си используют указатели?
9. Каким образом указатели передаются функции?

Упражнения

1. Напишите программу, в которой структура используется для составления инвентарной описи. Информация включает в себя название продукта, цену, количество, имя поставщика.Внесите изменения в программу из упражнения 1 с тем, чтобы можно было вводить информацию в массив структур, состоящий из 20 элементов.
2. Внесите изменения в программу из упражнения 2 так, чтобы выводить на экран общую стоимость включенных в опись товаров.
3. Напишите программу, в которой две переменные типа float определяются в main() как локальные, а затем используются в функции,

вычисляющей квадраты обоих чисел.

4. Объясните, почему следующая программа написана неверно:

```
main()
  {
  struct CD
            {
            char description[40];
            char category[12];
            char name[20];
            float cost;
            int number;
            } disc;
  puts("Введите сведения о диске");
  printf("Введите название: ");
  gets(name);
  printf("Введите описание: ");
  gets(description);
  printf("Введите категорию: ");
  gets(category);
  printf("Введите цену: ");
  scanf("%f", &cost);
  printf("Введите номер ячейки: ");
  scanf("%d", &number);
  puts("Введена следующая информация о диске: ");
  printf("Название:     %s\n", name);
  printf("Описание:     %s\n", description);
  printf("Категория:    %s\n", category);
  printf("Цена:                          %6.2f\n", cost);
  printf("Номер п/п:    %d\n", number);
  }
```

ГЛАВА 12
ВЫВОД НА ДИСК И ПРИНТЕР

Простое отображение информации на дисплее используется практически в каждой программе, но возможности его несколько ограничены. Даже использование временной остановки выполнения программы, чтобы дать пользователю возможность ознакомиться со всеми сообщениями, не решает проблемы полностью: как только сообщение уходит за пределы экрана, его уже невозможно прочитать без повторного запуска программы.

Более того, значения, присвоенные переменным, сохраняются только на время выполнения программы. Как только работа программы завершена, вся введенная информация теряется. Это означает, что если вы, например, ввели сведения о своей коллекции компакт-дисков в массив структурных переменных, они утрачиваются после завершения работы программы, и когда вы в следующий раз обратитесь к компьютеру, все данные придется вводить заново.

Для того чтобы сохранить информацию для себя или ознакомить других людей с результатами работы своей программы, нужно распечатать эти результаты на бумаге. А чтобы иметь возможность в любой момент обратиться к однажды введенным данным, необходимо сохранить информацию в файле на диске.

Что такое файловая структура

Выводимые данные отправляются на диск или печатающее устройство в зависимости от соответствующих инструкций вывода не сразу. Вместо этого они прежде поступают в область памяти, предназначенную для временного хранения информации, которая называется *буфером*. И только когда буфер заполняется, данные переправляются на диск или принтер (рис.12.1). Вводимые с диска данные также сначала поступают в буфер, откуда могут быть выведены на экран или присвоены в качестве значения переменной.

Для того чтобы направить данные в буфер или получить их из буфера, необходимо некоторое связующее звено между вашей программой и операционной системой компьютера. Этим звеном является файловая структура.

Когда программа открывает файл для работы, она тем самым создает специальную структуру в памяти. Эта структура содержит сведения,

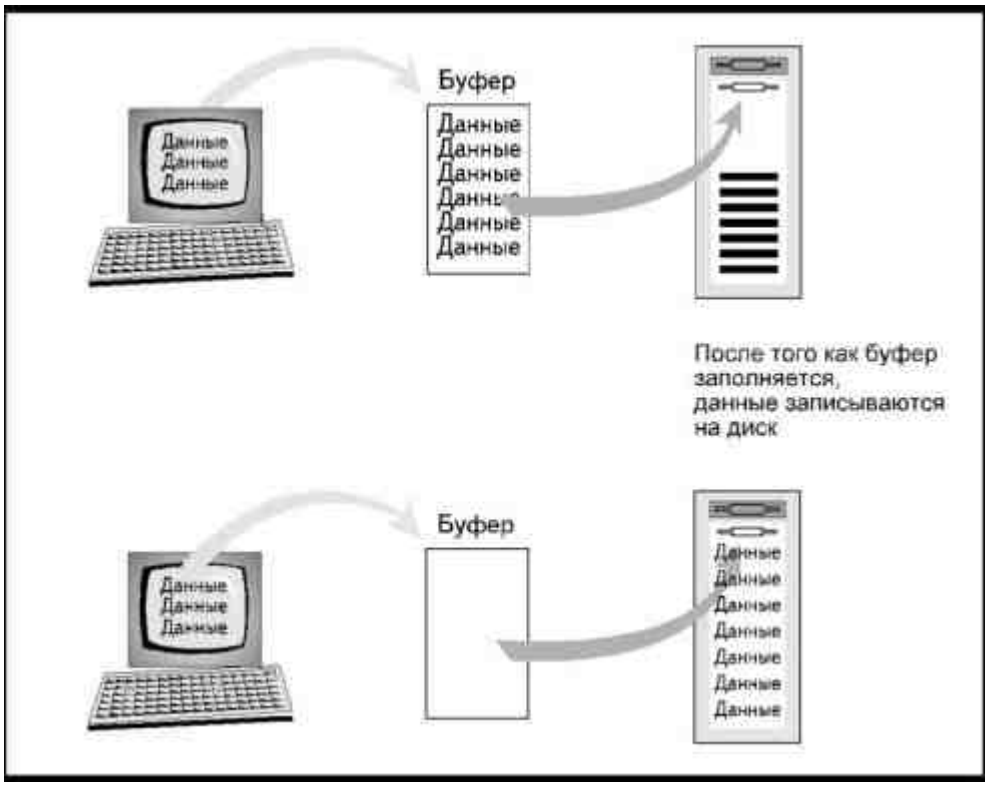

Рис. 12.1. Некоторое время данные хранятся в буфере

необходимые вашей программе и компьютеру для осуществления вывода данных в файл и ввода из файла, а также для печати информации на принтере.

Например, структура содержит адрес буфера файла, чтобы компьютер знал, где искать информацию, которую вы хотите вывести на диск, или куда поместить данные, которые вы хотите считать с диска. Кроме того, эта структура хранит сведения о количестве символов, остающихся в буфере, а также о позиции следующего символа как выводимого из буфера, так и поступающего в него (рис.12.2).

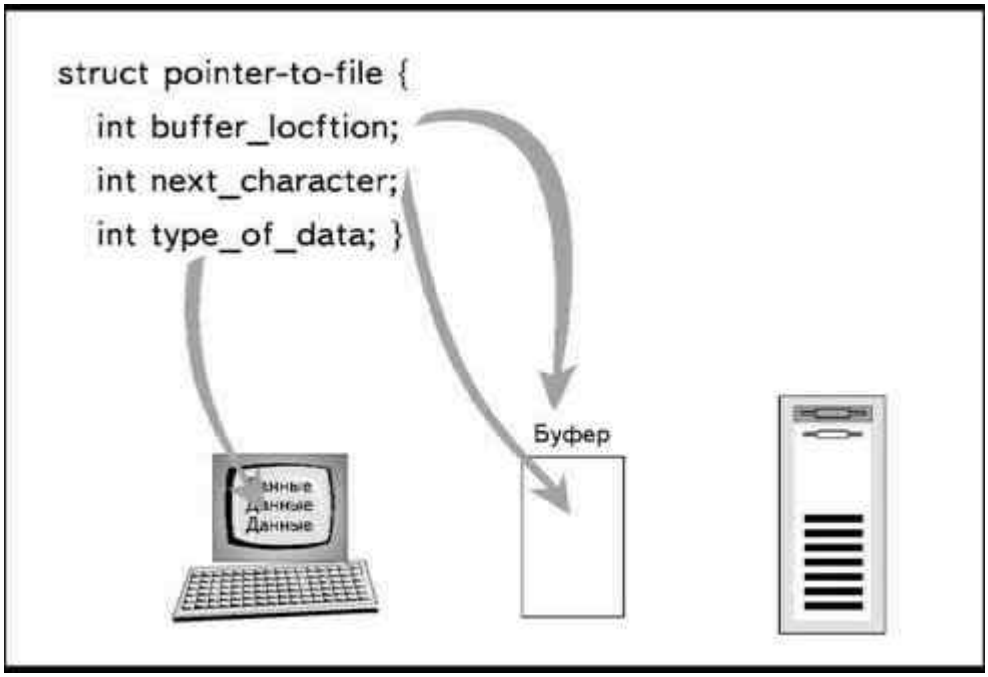

Рис. 12.2. Файловая структура хранит информацию,необходимую для нормального выполнения файловых операций

Почти все компиляторы Си и Си++ хранят информацию, необходимую для работы с файлами, в файле заголовков STDIO.H. Этот файл содержит определения констант, которые нужны для операций с файлами. Кроме того, он может содержать описание файловой структуры. Для того чтобы воспользоваться функциями работы с файлами, программу следует начинать с инструкции

```
#include
```

которая сделает файловые константы и описание файловой структуры доступными в процессе компиляции и компоновки программы.

При вводе данных из дискового файла происходит их копирование в память компьютера, информация, остающаяся на диске, не изменяется во время работы программы. По этой причине программисты называют такой ввод *чтением*

Замечания по Си++

Многие компиляторы Си++ имеют дополнительные файлы заголовков, содержащие специальные функции для выполнения файловых операций. Эти файлы могут называться IOSTREAM.H, FSTREAM.H и так далее, в зависимости от функций, которые в них содержатся, и конкретного компилятора. Проверьте документацию вашего компилятора.

данных из файла. При выводе данных на диск в файл помещается копия данных, хранящихся в памяти. Эта процедура называется *записью* на диск.

Указатель на файл

Ввод или вывод информации в файлы обеспечивается с помощью так называемого указателя на файл, который является указателем на файловую структуру в памяти. При записи информации в файл или при чтении из файла программа получает необходимую информацию из структуры. Указатель на файл определяется следующим образом:

```
FILE *file_pointer;
```

Имя структуры FILE говорит программе о том, что определяемая переменная является указателем именно на файловую структуру. Звездочка предписывает создать указатель с соответствующим именем переменной.

Если вы собираетесь использовать одновременно несколько файлов, вам нужны указатели для каждого из них. Например, если вы пишете программу, в которой содержимое одного файла копируется в другой, вам необходимы два указателя на файлы. Два указателя требуются и в том случае, если вы хотите прочитать информацию с диска и распечатать ее на принтере:

```
FILE *infile, *outfile;
```

Как открыть файл

Связь между программой и файлом устанавливается при помощи функции fopen(), синтаксис которой показан на рис.12.3.

Эта функция присваивает адрес структуры указателю. Первым параметром этой функции является имя файла, которое должно быть указано в соответствии с

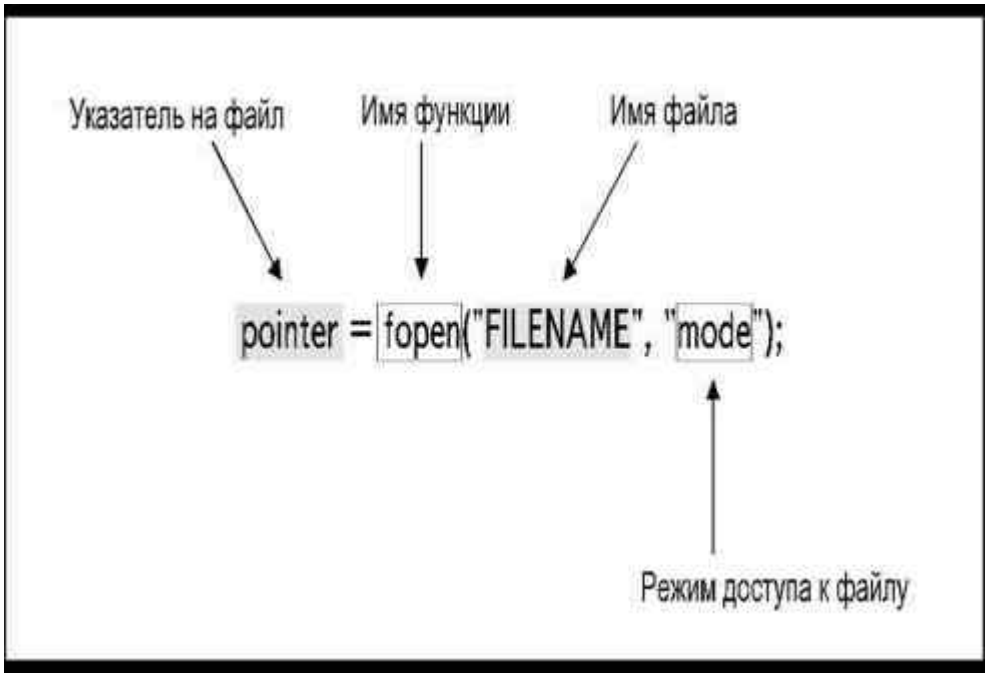

Рис. 12.3. Синтаксис функции fopen()

определенными правилами. Например, в операционной системе MS-DOS имя файла может состоять максимум из восьми символов, плюс расширение имени, состоящее не более чем из трех символов (расширение не является обязательным элементом). Если вы хотите вывести информацию на печатающее устройство, а не в дисковый файл, в качестве имени файла в кавычках указывается "PRN". При этом автоматически осуществляется вывод данных на принтер.

В качестве второго параметра функции передается режим доступа к файлу, то есть сообщение о том, какие операции пользователь намерен производить с файлом. В Си и Си++ параметр, определяющий режим доступа, также заключается в кавычки. Возможны следующие варианты:

r Указывает на то, что будет выполняться *чтение* информации из файла в память компьютера. Если файл к этому моменту не существует на диске, программа сообщит об ошибке выполнения.

w Указывает на то, что будет выполняться *запись* данных на диск или вывод на принтер. Если файл к этому моменту не существует, операционная система создаст его. Если файл уже существует на диске, вся записанная в нем на данный момент информация будет уничтожена.

a Указывает на то, что следует *добавить* информацию в конец файла. В случае отсутствия файла, операционная система создаст его. Если он существует, выводимые новые данные будут добавлены в конец файла без уничтожения текущего содержимого.

Компиляторы Си++ и многие компиляторы Си стандарта ANSI позволяют открывать файл одновременно и для чтения, и для записи. Для этого в аргументе функции указываются режимы доступа "r+", "w+" или "a+".

Например, если вы хотите создать файл с именем CD.DAT для хранения картотеки коллекции компакт-дисков, вы должны использовать следующие инструкции:

```
FILE *cdfile;

cdfile = fopen("CD.DAT", "w");
```

Если в программе требуется осуществить чтение из файла, а не запись в него, используйте следующую запись:

```
FILE *cdfile;

cdfile = fopen("CD.DAT", "r");
```

Обратите внимание, что и имя файла, и символ, определяющий режим доступа, заключены в двойные кавычки. Это обусловлено тем, что они передаются функции fopen() как строки. Имя файла можно ввести с клавиатуры, как значение строковой переменной, а затем использовать имя этой переменной в качестве аргумента, без кавычек.

Если вы хотите распечатать информацию о вашей коллекции на принтере, используйте следующую последовательность инструкций:

```
FILE *cdfile;

cdfile = fopen("PRN", "w");
```

Учтите, что вывод информации на принтер возможен только с режимом доступа "w".

Как Си/Си++ работает с файлами

Си сохраняет сведения о текущей позиции чтения и записи в файле, используя специальный указатель.

При чтении информации из файла, указатель определяет следующие данные, которые должны быть считаны с диска. Когда файл открывается впервые с использованием режима доступа "r", указатель помещается на первый символ файла. При выполнении очередной операции чтения, указатель перемещается к следующей порции данных, которые должны быть прочитаны. Величина шага перемещения при этом зависит от количества информации, которая считывается за один прием (рис.12.4). Если за один раз считывается только один символ, указатель передвинется на следующий символ, если читается целая структура, указатель перейдет на следующую структуру. Как только вся информация прочитана из файла, указатель попадает на специальный код, называемый *символом конца файла**. Попытка продолжения чтения после достижения конца файла приведет к ошибке выполнения.

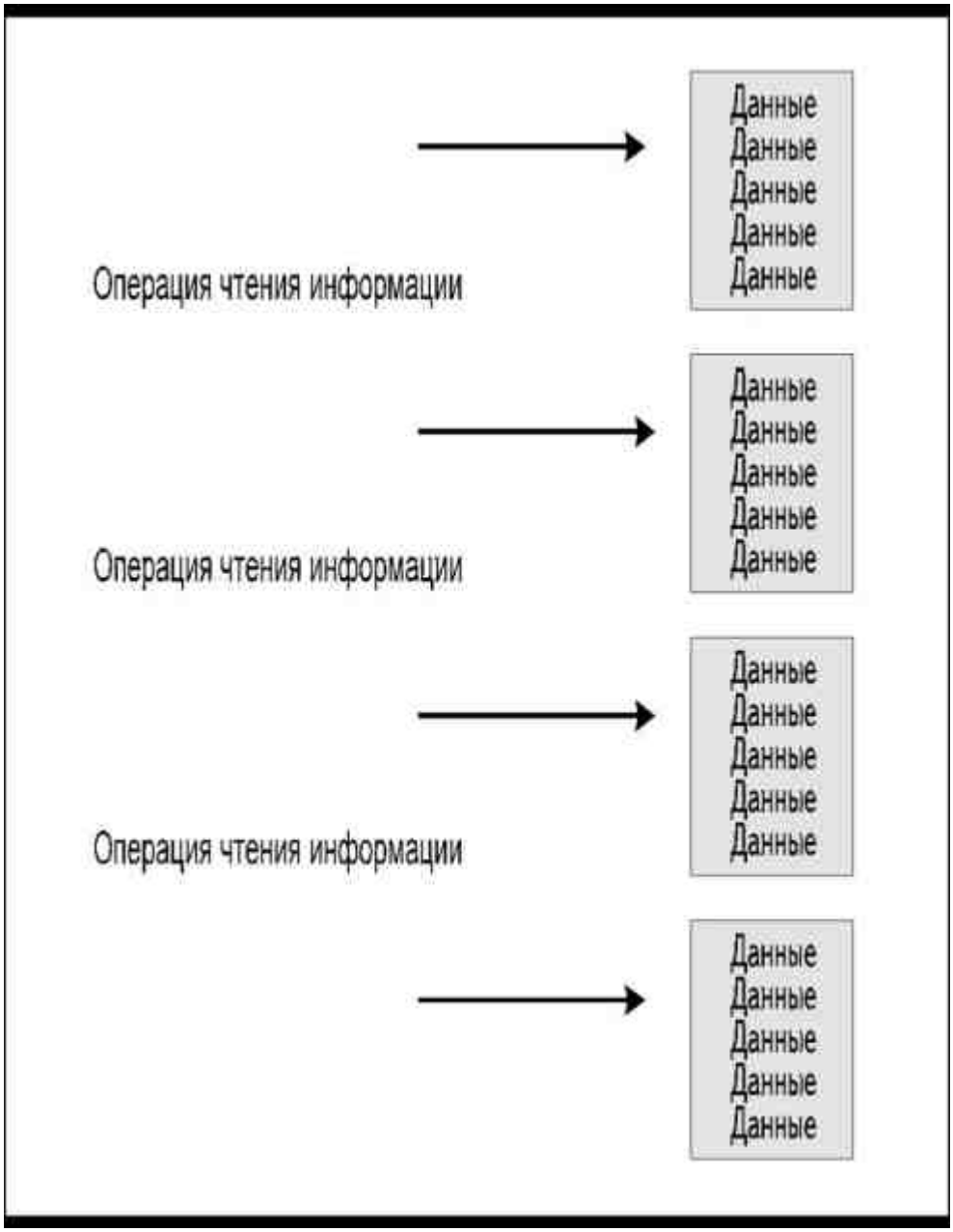

Рис. 12.4. Указатель сохраняет сведения о текущей позиции в файле

Если файл открывается с режимом доступа "w", указатель также помещается в начало файла, так что первые введенные данные будут помещены в начало файла. При закрытии файла после введенного массива данных будет добавлен символ конца файла. Если файл к моменту его открытия с использованием режима доступа "w" уже существует, все содержащиеся в нем данные затираются и «поверх» них записывается новая информация, введенная с помощью процедуры записи. Любые данные, которые могут остаться не уничтоженными, располагаются после нового символа конца файла, так что к ним уже нельзя будет обратиться при следующем чтении данных из файла. Таким образом, любая попытка записи данных в существующий файл с использованием режима доступа "w" приведет к уничтожению хранящейся в нем на данный момент информации. Это произойдет даже в том случае, если файл будет просто открыт и закрыт, без записи каких-либо данных.

Если файл открывается с использованием режима доступа "a", указатель помещается на символ конца файла. Новые данные, которые записываются в файл, размещаются после уже существующих данных, а затем добавляется символ конца файла.

*** Наличие символа конца файла на самом деле вовсе не является обязательным.*(Прим.ред.)***

Как избежать ошибок выполнения

Иногда возникает ситуация, в которой операционная система не может открыть файл, указанный в функции fopen(). Причиной этому может быть отсутствие места на диске. Не исключена и возможность того, что указанный файл просто не существует. Возможна также ситуация, когда вы пытаетесь распечатать данные на принтере, а принтер не включен, или в нем отсутствует бумага.

Если вы попытаетесь использовать файл, который нельзя открыть, программа остановится в результате ошибки выполнения. Чтобы избежать подобного аварийного выхода из программы, можно использовать инструкцию if, которая будет останавливать программу в случае невозможности открытия файла. С помощью следующих инструкций, использующих if, можно открыть файл и проверить, действительно ли он открыт:

```
if ((cdfile = fopen("CD.DAT", "w")) == NULL)
        {
        puts("Невозможно открыть файл");
        exit();
        }
```

NULL— это специальное значение, определенное в файле заголовков STDIO.H. Если произошла ошибка и открыть файл невозможно, система вернет значение NULL вместо адреса файловой структуры, и выполнение программы прекратится. Аналогичное условие можно внести и в инструкции печати информации на принтере: if ((cdfile = fopen("prn", "w")) == NULL) { puts("Включите принтер и запустите программу снова"); exit(); }

Замечания по Си++

В Си++ наряду со стандартными потоками ввода cin и вывода cout существует возможность создания потоков для работы с файлами. С их помощью можно осуществлять чтение и запись в файл, используя операторы >> и <<. Например, некоторые компиляторы позволяют создать поток для чтения файла, используя следующий синтаксис:

```
ifstream file_pointer(file_name)
```

Для записи в файл используется следующий синтаксис:

```
ofstream file_pointer(file_name)
```

Для выполнения этих операций может оказаться необходимым использование специальных файлов заголовков. Проверьте документацию вашего компилятора, чтобы узнать имя файла заголовков и способ его использования.

Если принтер не включен, на экране появится сообщение о том, что надо включить принтер, а затем запустить программу еще раз. Можно использовать и цикл while, с тем чтобы дать пользователю возможность исправить ошибку:

```
while ((cdfile = fopen("prn", "w")) == NULL)
        {
        puts("Включите принтер и нажмите Enter");
        flag = getchar();
        }
```

Заметьте, что в некоторых системах нет необходимости в подобных сообщениях. Если принтер не готов к печати, операционная система выведет на дисплей собственное сообщение, предлагающее пользователю либо остановить выполнение операции (abort), либо попробовать повторить ее еще раз (retry). При появлении такого сообщения следует включить принтер и нажать клавишу R, чтобы повторить попытку вывода на печать.

Как закрыть файл

После окончания записи в файл или чтения из файла необходимо его закрыть, то есть прервать связь между файлом и программой. Это осуществляется с помощью инструкции

```
fclose(file_pointer);
```

Закрывая файл, мы получаем гарантию, что вся информация, имевшаяся в буфере, действительно записана в файл. Если выполнение программы заканчивается до закрытия файла, какая-то не попавшая на диск часть информации может остаться в буфере, в результате чего она будет утрачена. Кроме того, не будет надлежащим образом записан символ конца файла, и в следующий раз программа не сможет получить доступ к файлу.

Следует добавить, что закрытие файла освобождает указатель, после чего он может быть использован с другим файлом или для выполнения других операций с тем же файлом. В качестве примера предположим, что вы хотите создать файл, записать в него данные, а затем убедиться, что информация записана правильно. Для этого в программе можно использовать структуру, приведенную в Листинге12.1.

Листинг 12.1. Использование одного указателя файла в двух операциях.

```
FILE *cdfile;
if((cdfile = fopen("CD.DAT", "w")) == NULL)
        {
        puts("Невозможно открыть файл");
        exit();
        }
/* Здесь должны располагаться инструкции записи в файл */

fclose(cdfile);
if((cdfile = fopen("CD.DAT", "r")) == NULL)
        {
        puts("Невозможно открыть файл");
        exit();
        }

/* В этом месте должны быть записаны
 инструкции чтения из файла */

fclose(cdfile);
```

Здесь файл сначала открывается с использованием режима доступа "w", затем в него записывают данные. Во второй раз файл открывается с использованием режима доступа "r", что позволяет прочитать данные и вывести их на экран.

Некоторые компиляторы позволяют обеспечить запись всех данных в файл путем очистки буфера с помощью функции

flush()

Эта функция позволяет без закрытия файла очистить буфер и записать все имеющиеся в нем данные на диск или направить их на принтер.

Функции ввода и вывода

Существует несколько способов передачи данных в файл и получения их из файла в зависимости от используемой функции:

- посимвольная запись данных в файл или вывод их на принтер с использованием функции putc() или fputc();
- посимвольное чтение данных из файла с использованием функции getc() или fgetc();
- построчная запись данных в файл или вывод их на принтер с использованием функции fputs();
- построчное чтение данных из файла с использованием функции fgets();
- форматированный вывод символов, строк или чисел на диск или на принтер с помощью функции fprintf();
- форматированный ввод символов, строк или чисел из файла с помощью функции fscanf();
- запись целой структуры с использованием функции fwrite();
- чтение целой структуры с использованием функции fread().

Работа с символами

Посимвольная передача данных является самой основной формой файловых операций. Хоть она и не принадлежит к числу широко распространенных на практике способов обращения с информацией, тем не менее, она хорошо иллюстрирует основные принципы работы с файлами. В приведенной ниже программе происходит посимвольная запись данных в файл, которая продолжается до тех пор, пока не нажата клавиша **Enter**:

```
/*fputc.c*/
#include
main()
        {
        FILE *fp;
        char letter;
        if((fp = fopen("MYFILE","w"))==NULL)
                {
                puts("Невозможно открыть файл");
                exit();
                }
        do
                {
                letter=getchar();
                fputc(letter, fp);
                }
        while(letter != '\r');
                fclose(fp);
        }
```

Файл открывается с режимом доступа "w". Если файл с именем MYFILE не существует к моменту выполнения программы, он будет создан. В цикле do, с помощью функции getchar(), осуществляется ввод последовательности символов,

Буферизированный ввод функции getchar()

Если в вашем компиляторе функция getchar() используется для буферизированного ввода, то для ввода отдельных символов, предназначенных для записи в дисковый файл, вы можете применить вместо нее функцию getche(). В этом случае цикл do...while можно заменить циклом while:

```
while ((letter = getche())) != '\r')
```

которые затем записываются в файл с помощью функции putc(). Синтаксис записи putc() таков:

```
putc(char_variable, file_pointer);
```

С теми же аргументами может использоваться и функция fputc().

Цикл выполняется до тех пор, пока не нажата клавиша **Enter**, которая вводит код «возврат каретки» (\r), после чего файл закрывается.

Посимвольное чтение из файла

Для посимвольного чтения из файла используется функция getc() или fgetc(). Синтаксис записи следующий:

```
char_variable = getc(file_pointer);
```

Например, для того чтобы получить символ из файла, связанного с указателем на файл fp, надо написать инструкцию:

```
letter = getc(fp);
```

Ниже приведена программа, которая читает файл, созданный в предыдущем примере:

```
/*fgets.c*/
#include
main()
        {
        FILE *fp;
        int letter;
        if((fp = fopen("MYFILE","r"))==NULL)
                {
                puts("Невозможно открыть файл");
                exit();
                }
        while((letter = fgetc(fp)) != EOF)
                printf("%c", letter);
        fclose(fp);
        }
```

Здесь файл открывается с режимом доступа "r", после чего в цикле while осуществляется посимвольное чтение. Большая часть работы программы выполняется внутри условия цикла while. Выражение

```
letter = fgetc(fp)
```

присваивает символ, полученный из файла, переменной letter. Цикл выполняется до тех пор, пока указатель не достигнет конца файла и значение переменной не станет равно значению специальной константы EOF* (она определена в файле заголовков STDIO.H), что, собственно, и является признаком достижения конца файла.

В приведенных образцах программ использовался один и тот же указатель на файл и имя переменной как для записи, так и для чтения данных из файла. В реальной программе такого быть не должно. Даже в том случае, если в обеих операциях используется одно и то же имя файла, запись и чтение следует осуществлять только с использованием разных указателей и имен переменных.

Работа со строками

Вместо того чтобы работать с отдельными символами, можно читать из файла и записывать в него целые строки текста. Построчная запись и чтение осуществляются с использованием функций fputs() и fgets().

Функция fputs() имеет следующий синтаксис:

```
fputs(string_variable, file_pointer);
```

Эта функция выполняет построчную запись данных в файл или вывод на принтер, но не добавляет код «новая строка». Для того чтобы каждая строка записывалась на диск (или печаталась на принтере) действительно как отдельная строка, необходимо вводить код «новая строка» вручную. Например, в приведенной ниже программе создается файл имен:

```
/*fputc.c*/
#include
main()
        {
        FILE *fp;
        char flag;
        char name[20];
        if((fp = fopen("MYFILE","w"))==NULL)
                {
                puts("Невозможно открыть файл");
```

```
                exit();
                }
        flag = 'y';
        while(flag != 'n')
                {
                puts("Введите имя");
                gets(name);
                fputs(name, fp);
                fputs("\n",fp);
                printf("Желаете ввести другое имя?");
                flag=getchar();
                putchar('\n');
        }
        fclose(fp);
        }
```

*** Имя константы является аббревиатурой выражения end-of-file. *(Прим.перев.)***

Выполнение цикла while продолжается до тех пор, пока в ответ на подсказку не будет введен символ n. В этом цикле осуществляется ввод имени с клавиатуры с помощью функции gets(), после чего имя записывается на диск с помощью функции fputs(). Далее в файл записывается код «новая строка», и, наконец, программа спрашивает пользователя, желает ли он продолжить ввод имен.

Если ваш компилятор может использовать функцию strlen(), можно несколько упростить процедуру ввода, используя следующие инструкции:

```
printf("Пожалуйста, введите имя: ");
gets(name);
while(strlen(name) > 0)
        {
        fputs(name, fp);
        fputs("\n", fp);
        printf("Пожалуйста, введите имя: ");
        gets(name);
        }
```

Символы, которые вы набираете на клавиатуре, присваиваются строковой переменной name, а затем проверяется, не оказалась ли длина строки равной 0. Если на запрос сразу же нажать клавишу Enter, строка будет иметь нулевую длину и выполнение цикла прекратится. Если до нажатия **Enter** ввести хотя бы один символ, строка и код «новая строка» будут записаны на диск.

Некоторые компиляторы позволяют еще более упростить алгоритм ввода строки, например, так:

```
printf("Пожалуйста, введите имя: ");
while(strlen(gets(name)) > 0)
        {
        fputs(name, fp);
        fputs("\n", fp);
        printf("Пожалуйста, введите имя: ");
        }
```

где ввод строки выполняется внутри условия while.

Для того чтобы напечатать строку на принтере, вместо записи ее на диск используется имя файла "prn". Чтобы открыть файл, требуется указать:

```
if ((fp = fopen("prn", "w")) == NULL)
```

Для создания программы печати длина строки определяется равной 81 символу, чтобы строка могла уместиться во всю ширину экрана, прежде чем будет нажата клавиша **Enter**. В Листинге12.2 приводится текст программы, которая демонстрирует, как можно написать простой текстовый процессор. Строка не посылается на принтер до тех пор, пока не нажата клавиша **Enter**, что позволяет с помощью клавиши **Backspace** корректировать ошибки ввода строки.

Листинг 12.2. Программа вывода строки на печатающее устройство.

```
/*wp.c*/
#include "stdio.h"
main()
        {
        FILE *fp;
        char line[81];
        if ((fp = fopen("prn", "w")) == NULL)
                {
                puts("Принтер не готов к работе");
                exit();
                }
        puts("Введите текст, после ввода
 каждой строки нажимайте Enter\n");
        puts("Для прекращения ввода нажмите
 Enter в начале новой строки\n");
        gets(line);
        while (strlen(line) > 0)
                {
                fputs(line, fp);
                fputs("\n", fp);
                gets(line);
                }
        fclose(fp);
        }
```

Чтение строк

Чтение строк из файла осуществляется с помощью функции fgets(). Синтаксис функции:

```
fgets(string_variable, lenght, file_pointer);
```

Функция вводит строку целиком до символа новой строки, если ее длина не превышает значения, указанного в параметре lenght минус один символ. Параметр lenght является целым числом либо целочисленной константой или переменной, указывающей максимально возможное количество символов в строке.

Ниже приведена программа, в которой осуществляется чтение имен из файла, созданного в предыдущем примере:

```
/*fgets.c"/
#include "stdio.h"
main()
        {
        FILE *fp;
        char name[12];
        if ((fp = fopen("MYFILE", "r")) == NULL)
```

```
        {
        puts("Невозможно открыть файл");
        exit();
        }
while(fgets(name, 12, fp) != NULL)
        {
        printf(name);
        }
fclose(fp);
}
```

Ввод выполняется внутри цикла while до тех пор, пока значение читаемого символа не равно NULL. Как только указатель достигнет конца файла, строковой переменной присваивается значение NULL. При построчном чтении из файла для указания конца файла всегда используется NULL, а EOF используют при посимвольном чтении.

Если вы пишете программу, предназначенную для чтения любого текстового файла, указывайте значение аргумента lenght равным 80.

Кстати, обратите внимание, что функция printf() используется в этом примере для вывода содержимого строковой переменной без указателей формата. Каждая строка, читаемая из файла, включает код «новая строка», который был записан в файл в инструкции fputs("\n", fp);, и никаких дополнительных кодов «новая строка» в параметры функции printf() включать не требуется.

Форматированный ввод и вывод

Функции, работающие с символами и строками, предназначены только для чтения и записи текста. Если требуется записать или прочитать из файла, либо отпечатать на принтере данные, содержащие числовые значения, вы должны использовать функции fprintf() и fscanf(). Для записи инструкций с этими функциями используется тот же синтаксис, что и для функций printf() и scanf(), но в этом случае требуется еще включить указатель на файл, чтобы определить, куда следует записать или откуда читать данные:

```
fprintf(file_pointer, control_string, data_list);
fscanf(file_pointer, control_string, data_list);
```

Программа, приведенная в Листинге 12.3, предназначена для ввода и записи в файл данных инвентарной описи. Первый ввод осуществляется с помощью функции gets() перед началом цикла while. Цикл повторяется, пока длина каждого вводимого имени составляет как минимум один символ. Выполнение цикла прекращается, когда вместо ввода значения имени происходит нажатие клавиши **Enter**.

Листинг 12.3. Форматированный вывод.

```
/*fprintf.c*/
#include "stdio.h"
main()
        {
        FILE *fp;
        char name[20];
        int quantity;
        float cost;
        if ((fp = fopen("MYFILE", "w")) == NULL)
                {
                puts("Невозможно открыть файл");
                exit();
                }
        printf("Введите наименование товара: ");
        gets(name);
```

```
while (strlen(name) > 0)
{
printf("Введите цену товара: ");
scanf("%f", &cost);
printf("Введите количество
единиц товара: ");
scanf("%d", &quantity);
fprintf(fp, "%s %f %d\n", name,
cost, quantity);
printf("Введите наименование товара: ");
gets(name);
}
fclose(fp);
}
```

Обратите внимание, что в последней строке цикла происходит ввод следующего имени. Это позволяет прекратить повторение цикла простым нажатием клавиши **Enter**. Некоторые начинающие программисты, вероятно, написали бы этот цикл таким образом:

```
do
{
printf("Введите наименование товара: ");
gets(name);
printf("Введите цену: ");
scanf("%f", &cost);
printf("Введите количество единиц товара: ");
scanf("%d", &quantity);
fprintf(fp, "%s %f %d\n", name, cost, quantity);
}
while (strlen(name) > 0);
```

и эта программа работала бы столь же успешно, не считая того, что для окончания цикла требовалось бы нажать клавишу **Enter** трижды: первый раз при вводе названия и еще два раза в ответ на просьбу ввести цену и количество товара.

Внутри цикла while данные о цене и количестве каждого наименования товара вводятся с использованием функции scanf(), а затем записываются на диск с помощью инструкции

```
fprintf(fp, "%s %f %d\n", name, cost, quantity);
```

Обратите внимание, что код «новая строка» записывается в файл в конце каждой строки. Если просмотреть содержимое файла с помощью команды TYPE операционной системы MS-DOS, то каждая строка инвентарной описи и на экране будет начинаться с новой строки:

```
дискеты 1.120000 100
лента 7.340000 150
картридж 75.000000 3
```

Если бы код «новая строка» не был записан на диск, текст выводился бы подряд, в одну строку на экране, и выглядел примерно так:

```
дискеты 1.120000 100лента 7.340000 150картридж 75.000000 3
```

Заметьте, что при этом отсутствует пробел между числом, показывающим количество единиц одного товара, и наименованием следующего. Даже при таком способе записи можно без проблем осуществлять чтение из этого файла, так как компилятор в состоянии различить конец числового значения и начало строки, но что произойдет, если последним значением для каждого наименования товара окажется строка с названием фирмы-производителя? Информация в файле будет выглядеть примерно таким образом:

дискеты 1.120000 Memoryехлента

7.340000 Okaydataкартридж 75.000000 HP

и тогда при чтении данных из файла программа присоединит начало данных о следующем товаре к концу описания предыдущего. Например, данные о первом наименовании товара при этом выглядели бы так:

дискеты 1.120000 Memoryехлента

Все выведенные на диск данные, даже значения типа int или float, хранятся в виде текстовых символов. Об этом мы будем говорить чуть позже.

Чтение форматированных файлов

Ввод информации из форматированного файла осуществляется с использованием функции fscanf(). Она работает аналогично функции scanf() за исключением того, что fscanf() осуществляет ввод данных не с клавиатуры, а из

файла. К несчастью, использование fscanf() имеет те же ограничения, что и применение scanf(). Эта функция не может читать строки, содержащие пробелы, и если данные во входном потоке не будут соответствовать указателям в строке формата, она вернет ошибочные значения.

В Листинге 12.4 приводится текст программы, которая читает данные из форматированного файла, содержащего сведения об инвентарной описи. Заметьте, что цикл while повторяется до тех пор, пока функция fscanf() не возвратит значение EOF, которое будет свидетельствовать о достижении конца файла.

Листинг 12.4. Чтение форматированного текста из файла.

```
/*fscanf.c*/
#include "stdio.h"
main()
        {
        FILE *fp;
        char name[20];
        int quantity;
        float cost;
        if ((fp = fopen("MYFILE", "r")) == NULL)
                {
                puts("Невозможно открыть файл");
                exit();
                }
        while (fscanf(fp, "%s%f%d",
 name, &cost, &quantity) != EOF)
                {
                printf("Наименование товара:
        %s\n", name);
                printf("Цена:
        %.2f\n", cost);
                printf("Количество единиц:
        %d\n", quantity);
                }
        fclose(fp);
        }
```

Работа со структурами

Одним из способов преодолеть ограничения функции scanf() является объединение элементов данных в структуру с тем, чтобы впоследствии осуществлять ввод и вывод структур целиком. Структуру можно записать на диск с помощью функции fwrite() и прочитать из файла с помощью функции fread().

Синтаксис функции fwrite() такой:

fwrite(&structure_variable, structure_size,

number_of_structures, file_pointer);

На первый взгляд, эта инструкция выглядит несколько устрашающей, но на самом деле использовать ее очень легко:

- &structure_variable— имя структурной переменной с оператором получения адреса, сообщающим компилятору стартовый адрес информации, которую мы хотим записать на диск;
- structure_size— это количество символов в структуре; не обязательно подсчитывать его самому, для этого можно использовать библиотечную функцию sizeof(), записанную следующим образом:

  sizeof(structure_variable)

  которая автоматически определит размер указанной структуры;
- number_of_structures— это целое число, определяющее количество структур, которые мы хотим записать в один прием; здесь всегда следует указывать число 1, если только вы не собираетесь создать массив структур и записать его одним большим блоком;
- file_pointer— указатель на файл.

В качестве примера предположим, что вы хотите записать на диск сведения о своей коллекции компакт-дисков. Используя структуру CD, которую мы подробно разбирали в главе 11, пишем инструкцию:

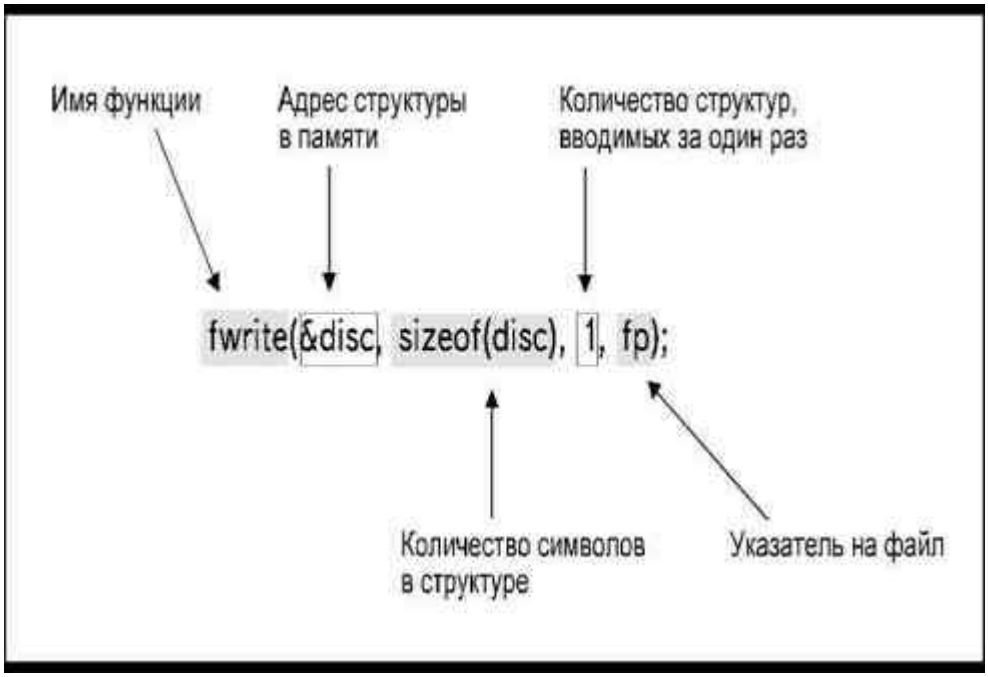

Рис. 12.5. Синтаксис функции fwrite() в инструкции записи структуры CD

fwrite(&disc, sizeof(disc), 1, fp);

Выполнение этой инструкции иллюстрирует рис.12.5.

Текст программы, которая вводит данные в структуру CD, а затем сохраняет ее на диске, приведен в Листинге12.5. Для ввода имени создаваемого файла используется функция gets(). Переменная, в которой хранится имя файла, используется функцией fopen() для того, чтобы открыть файл.

Информация о каждой структуре CD вводится с клавиатуры, после чего структура целиком записывается на диск.

Листинг 12.5. Запись структуры CD.

```
/*fwrite.c*/
#include "stdio.h"
main()
        {
        FILE *fp;
        struct CD
```

```
                {
                char name[20];
                char description[40];
                char category[12];
                float cost;
                int number;
                } disc;
        char filename[25];
        printf("Введите имя файла, который
вы желаете создать: ");
        gets(filename);
        if ((fp = fopen(filename, "w")) == NULL)
                {
                printf("Невозможно открыть файл
%s\n", filename);
                exit();
                }
        puts("Введите сведения о диске\n");
        printf("Введите название диска: ");
        gets(disc.name);
        while (strlen(disc.name) > 0)
                {
                printf("Введите описание: ");
                gets(disc.description);
                printf("Введите категорию: ");
                gets(disc.category);
                printf("Введите цену: ");
                scanf("%f", &disc.cost);
                printf("Введите номер ячейки: ");
                scanf("%d", &disc.number);
                fwrite(&disc, sizeof(disc), 1, fp);
                printf("Введите название: ");
        gets(disc.name);
                }
        fclose(fp);
        }
```

Чтение структур

Для того чтобы прочитать структуру целиком, используется функция fread(). Она имеет следующий синтаксис:

```
fread(&structure_variable, structure_size,
 number_of_structures, file_pointer);
```

За исключением имени функции эта инструкция полностью совпадает с записью функции fwrite(). Программа, в которой из файла считывается структура CD, приведена в Листинге12.6. Для чтения данных используется цикл while:

while (fread(&disc, sizeof(disc), 1, fp) == 1)

Функция fread() возвращает значение, соответствующее количеству успешно прочитанных структур. Так как в аргументе функции мы указали, что читать следует по одной структуре, функция возвращает значение 1. Цикл while будет выполняться до тех пор, пока считывание структур с диска проходит успешно. Если чтение структуры становится невозможным, например потому, что достигнут конец файла, функция возвращает значение 0, и выполнение цикла прекращается.

Листинг 12.6. Чтение структуры CD с диска.

```
/*fread.c*/
#include "stdio.h"
main()
        {
        FILE *fp;
        struct CD
                {
                char name[20];
                char description[40];
                char category[12];
                float cost;
                int number;
                } disc;
        char filename[25];
        printf("Введите имя файла, который
 желаете открыть: ");
        gets(filename);
        if ((fp = fopen(filename, "r")) == NULL)
                {
                printf("Невозможно открыть файл
 %s\n", filename);
                exit();
        }
        while (fread(&disc, sizeof(disc), 1, fp) == 1)
                {
                puts(disc.name);
                putchar('\n');
                puts(disc.description);
                putchar('\n');
                puts(disc.category);
                putchar('\n');
                printf("%f", disc.cost);
                putchar('\n');
                printf("%d", disc.number);
                }
        fclose(fp);
        }
```

В табл. 12.1 собраны все описанные способы ввода и вывода данных и показаны значения, которые возвращает каждая функция при невозможности продолжения чтения или записи данных.

Таблица 12.1. Функции ввода в файл и вывода из файла.

Тип данных	Функции вывода	Функции ввода	Возвращаемое значение
символы	putc(), fputc()	getc(), fgetc()	EOF
строки	fputs()	fgets()	NULL
форматированные данные	fprintf()	fscanf()	EOF
структуры	fwrite()	fread()	0

Чтение в массив

Во всех программах, приведенных до настоящего момента в качестве примера, выполнялось чтение данных из файла и отображение вводимой информации на экране. Однако если считывать данные в переменные, с ними можно выполнять любые операции, например, использовать их для записи в массив.

В Листинге 12.7 приведен текст программы, осуществляющей чтение информации из файла, содержащего данные о коллекции компакт-дисков, в массив структур CD (предполагается, что их количество не превышает20). Индекс используется для того, чтобы каждая считанная из файла структура сохранялась в отдельном элементе массива disc. После того как очередная структура прочитана и выведена на экран, стоимость очередного диска добавляется к сумме, отражающей общую стоимость коллекции, а значение индекса и счетчика увеличивается за счет выполнения следующих инструкций:

```
total = total + disc[index].cost;

index++;

count++;
```

Если бы нас интересовала только информация об общей стоимости и количестве экземпляров коллекции, можно было бы читать данные в структурную переменную, не используя массив, и просто подсчитывать значения переменных total и count. Однако если данные будут считаны в массив, вы сможете произвольным образом обращаться к структурам и печатать любую информацию.

Заметьте, что запрос о вводе имени файла в программе повторяется до тех пор, пока не будет введено имя файла, который действительно можно открыть.

Листинг 12.7. Чтение структуры в массив.

```
/*rarray.c*/
#include "stdio.h"
main()
        {
        FILE *fp;
        struct CD
                {
                char name[20];
                char description[40];
                char category[12];
                float cost;
                int number;
                } disc[20];
        int index, count;
        float total;
        count = 0;
        total = 0;
```

```
char filename[25];
printf("Введите имя файла данных: ");
gets(filename);
while ((fp = fopen(filename, "r")) == NULL)
        {
        printf("Невозможно открыть файл
%s\n", filename);
        printf("Введите имя файла данных: ");
        gets(filename);
        }
index = 0;
while (fread(&disc[index],
sizeof(disc[index]), 1, fp) == 1)
        {
        puts(disc[index].name);
        putchar('\n');
        puts(disc[index].description);
        putchar('\n');
        puts(disc[index].category);
        putchar('\n');
        printf("%f", disc[index].cost);
        putchar('\n');
        printf("%d", disc[index].number);
        total = total + disc[index].cost;
        index++;
        count++;
        }
fclose(fp);
printf("Общая стоимость коллекции
составляет %.2f\n", total);
printf("Коллекция содержит %.d дисков\n", count);
}
```

Замечания по Си++

Компиляторы Си++ позволяют читать из файла и записывать в файл данные, используя потоки (ifstream и ofstream), которые открываются с помощью операторов << и >>. Синтаксис чтения из файла:

file_pointer >> variable;

Синтаксис записи в файл:

file_pointer << variable;

Дополнение файла новыми данными

Если файл, который уже существует на диске, открыть с режимом доступа "w", вся информация, имевшаяся в нем на текущий момент, будет уничтожена. Для того чтобы добавить данные в уже существующий на

диске файл, следует открывать его с режимом доступа "a". Фактически, большинство компиляторов позволяет в одной программе и создавать файл, и добавлять в него данные. Если режим доступа "a" используется для файла, которого нет на диске, он будет создан, а если файл существует, в него будет добавлена новая информация.

Когда вы добавляете данные в файл, вы сами должны позаботиться о том, чтобы новые данные соответствовали формату уже имеющейся в файле информации. Например, вы можете открыть файл, содержащий некую последовательность символов, и записать туда структуру. Она действительно будет записана в файл, так как предполагается, что вы знаете, что делаете, но при следующих попытках прочитать информацию из этого файла вы получите ошибку выполнения или какие-нибудь непонятного вида данные.

Текстовый и двоичный форматы

Функции putc(), fputc() и fputs() служат для вывода текста. Если вы просмотрите дисковый файл, используя команду TYPE, то увидите точно такие же символы, какие ввели. Файл можно создать, использовав любую из этих функций, а затем прочитать из него с помощью функций getc(), fgetc() или fgets(). При этом функции, осуществляющие посимвольное чтение, будут вводить по одному символу, даже если изначально в файл была записана строка с помощью функции fputs(). Аналогично, функции, предназначенные для построчного чтения из файла, будут вводить данные целыми строками, даже если информация в файл была записана посимвольно.

В Листинге 12.8 приведен текст программы, которая копирует содержимое одного файла в другой, одновременно отображая его на экране. Программа будет работать с любым файлом, безотносительно того, как именно он был создан. В программе определены два указателя на файлы, так как мы обращаемся к двум файлам одновременно.

Листинг 12.8. Программа копирования содержимого файлов.

```
/*filecopy.c*/
#include "stdio.h"
main()
        {
        FILE *fp1, *fp2;
        char infile[25], outfile[25];
        int letter;
        printf("Введите имя файла для чтения: ");
        gets(infile);
        if ((fp1 = fopen(infile, "r")) == NULL)
                {
                printf("Невозможно открыть
 файл %s", infile);
                exit();
                }
        printf("Введите имя файла для записи: ");
        gets(outfile);
        if ((fp2 = fopen(infile, "w")) == NULL)
                {
                printf("Невозможно открыть
 файл %s", outfile);
                fclose(fp1);
                exit();
        }
        while ((letter = fgetc(fp1)) != EOF)
                {
                putchar(letter);
```

```
            fputc(letter, fp2);
            }
    fclose(fp1);
    fclose(fp2);
    }
```

Первый файл открывается с режимом доступа "r", чтобы можно было прочитать из него данные. Если файл невозможно открыть, программа завершается. Второй файл открывается с режимом доступа "w", что позволяет записывать в

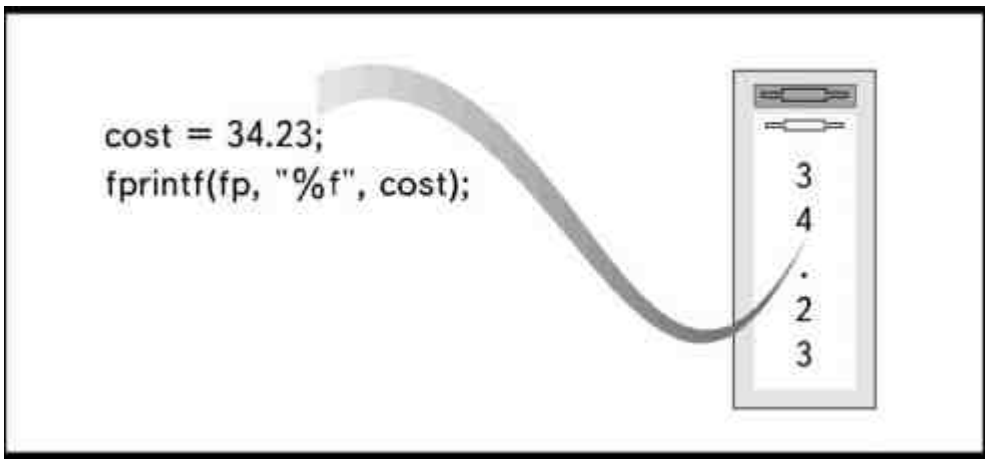

Рис. 12.6. Функция fprintf() записывает числовые значенияв виде текстовых символов

него данные. Если второй файл невозможно открыть, то перед завершением программы сначала закрывается первый файл. Это дает нам гарантию того, что первый файл, если он был успешно открыт, не окажется поврежден в момент выхода из программы.

Функция fprintf() записывает все данные в виде текста. Например, если использовать fprintf() для записи числа 34.23 на диск, пять символов будут записаны так, как это показано на рис.12.6. Если в дальнейшем для чтения данных из файла используется функция fscanf(), символы будут преобразованы в числовое значение и в таком виде записаны в переменную.

Вследствие того, что функция fprintf() записывает данные в виде текста, чтение из файла можно осуществлять и с помощью функций getc(), fgetc() или fgets(). Однако эти функции будут читать информацию в виде «печатных» символов. Например, если использовать функцию fgets(), числа будут считываться в виде символов, являющихся частью строки. При отображении на экране или печати на принтере данных, прочитанных с использованием функции fgets() или fgetc(), вы будете лишены возможности выполнения арифметических операций над отдельными элементами данных.

Двоичный формат

Для сохранения числовых переменных в двоичном формате используется функция fwrite(). Записанные таким образом данные на диске займут столько

Компиляторы Си++ и многие компиляторы, поддерживающие стандарт ANSI языка Си, позволяют создавать форматированные двоичные файлы для хранения числовых данных. Файл создается с режимом доступа "wb" и читается с режимом доступа "rb". Символ b указывает на двоичный формат. Если открыть файл с таким режимом доступа, можно записывать в файл целочисленные значения с помощью функции putw() и читать их из файла с помощью функции getw().

же места, сколько и в памяти. Если просмотреть содержимое такого файла с помощью команды TYPE, мы увидим на месте числовых значений бессмысленные буквы и значки. Это ASCII-символы, эквивалентные записанным в файл значениям.

Для чтения файла, записанного с помощью fwrite(), следует использовать функцию fread(). Вводить данные следует в структуру, имеющую строение, соответствующее сохраненным ранее данным. Структура может иметь другое имя, имена членов структуры тоже могут отличаться, но порядок, типы и размеры членов обеих структур должны совпадать.

Печать данных

С технической точки зрения вывести данные на принтер можно с помощью любой функции вывода: посимвольно, построчно, форматированными строками или структурами. Единственное, что необходимо,— это указать имя файла "prn" и режим доступа "w".

Однако «поструктурная» печать с помощью функции fwrite() практически не используется, так как числовые данные при этом будут напечатаны в двоичном формате в виде загадочных символов. Вместо этого для печати структур используется функция fprintf(), как это показано в Листинге12.9. В этой программе открываются два файла: дисковый файл открывается для чтения, а файл принтера— для вывода.

Листинг 12.9. Чтение и печать содержимого дискового файла.

```
/*fread1.c*/
#include "stdio.h"
main()
        {
        FILE *fp;
        struct CD
                {
                char name[20];
                char description[40];
                char category[12];
                float cost;
                int number;
                } disc;
        char filename[25];
        printf("Введите имя файла: ");
        gets(filename);
        if((fp = fopen(filename, "r")) == NULL)
                {
                printf("Невозможно открыть файл
 %s\n", filename);
                exit();
                }
        if((ptr = fopen("PRN", "w")) == NULL)
                {
                printf("Принтер не готов к
 работе\n", filename);
                fclose(fp);
                exit();
                }
        while (fread(&disc, sizeof(disc), 1, fp) == 1)
                {
                fprintf(ptr, "Название диска
        %s\n", disc.name);
                fprintf(ptr, "Описание:
        %s\n", disc.description);
                fprintf(ptr, "Категория:
        %s\n", disc.category);
```

```
            fprintf(ptr, "Стоимость:
%6.2f\n", disc.cost);
            fprintf(ptr, "Номер п/п:
%d\n", disc.number);
            fprintf(ptr, "\n\n");
            }
fclose(ptr);
fclose(fp);
}
```

Каждая структура целиком вводится функцией fread(), после чего отдельные члены структуры печатаются с использованием функции fprintf(). Функция fread() может читать строки, включающие пробелы, поэтому ее применение предпочтительнее, чем использование функции fscanf().

Инструкции

```
fprintf(ptr, "\n\n");
```

выводят по две пустые строки между отдельными структурами CD.

Проектирование программы

Знание того, как осуществляется запись в дисковый файл и чтение из него, открывает перед вами возможность создания сложных приложений. В этой главе все программы, которые демонстрировали ввод данных из дискового файла, читали его целиком. Но можно представить себе ситуацию, когда вы захотите поступить с данными каким-либо другим образом.

Например, вам может понадобиться просмотреть дисковый файл в поисках определенной записи. В этом случае следует открыть файл с режимом доступа "r", а потом использовать цикл для постепенного ввода данных, структура за структурой или строка за строкой в зависимости от того, к какому типу относится информация, записанная в файл. Во время каждого прохождения цикла

Функции произвольного доступа

Компиляторы Си++ и многие компиляторы Си, поддерживающие стандарт ANSI, имеют функции произвольного доступа к файлу. *Произвольный доступ* означает, что вы можете перейти непосредственно к определенному месту файла для чтения или внесения изменений в данные, расположенные именно в этом месте. Перемещение указателя файла в заданную позицию выполняет функция fseek(). Если за ней следует, например, функция fread(), то она будет читать данные, начиная с позиции, отмеченной файловым указателем. Кроме того, функция ftell() сообщает текущее положение указателя, а функция rewind() переносит указатель в начало файла.

значения вводимых данных сравниваются с искомыми. Для проверки значений строк используйте функцию strcmp(), конечно, если ваш компилятор это позволяет. Как только искомые данные найдены, они выводятся на экран, после чего файл закрывается.

Функции работы с файлами, описанные в этой главе, выполняют *последовательные операции*. Это означает, что они выполняют чтение файла от его начала. Вы можете использовать режим доступа "a" для того, чтобы добавить данные в конец файла, но вы не можете перейти прямо к определенной позиции в файле, чтобы изменить хранящуюся там информацию.

Это не означает, что информацию, хранящуюся в файле, вообще нельзя изменить. Существует алгоритм, позволяющий обращаться к содержимому файла последовательно. Вы познакомитесь с этим и другими алгоритмами в многочисленных примерах, которые приведены в следующей главе.

Вопросы

1. Что такое файловый буфер?
2. Как используется файловая структура?
3. Для чего в программах используют указатель на файл?
4. Опишите, в чем заключаются различия между режимами доступа "r", "w" и "a"?
5. Почему необходимо закрыть файл перед завершением работы программы?
6. Как вы будете выводить числовые данные?
7. Как напечатать данные на принтере?
8. В чем заключается различие между функциями fprintf() и fwrite()?
9. Как осуществить печать структур?
10. Для чего служит функция sizeof()?

Упражнения

1. Напишите программу, в которой функция fputs() используется для создания файла, содержащего названия кинофильмов.
2. Напишите программу, которая читает названия кинофильмов (упражнение1) в массив строк.
3. Напишите программу, в которой функция fprintf() используется для создания файла инвентарной описи, содержащей сведения о наименовании товара, его цене и количестве единиц, имеющихся в наличии.
4. Напишите программу, которая читает файл инвентарной описи, созданный в упражнении 3.
5. Отредактируйте программы из упражнений 3 и 4 так, чтобы они читали данные как структуры.
6. Объясните, почему следующая программа написана неверно:

```
7.    #include "stdio.c"
8.    main()
9.              {
10.             FILE fp;
11.             char letter;
12.   if ((fp = fopen("MYFILE", "w")) == NULL)
13.                     {
14.                     puts("Невозможно открыть файл");
15.                     exit();
16.                     }
17.             do
18.                     {
19.                     letter = getchar();
20.                     fputc(letter, fp);
21.                     }
22.             while(letter != '\n');
23.             fclose(fp);
```

}

ГЛАВА 13
КАК СОБРАТЬ ВСЕ ВМЕСТЕ

Теперь, когда вы познакомились с основами языка Си и Си++, можно приступить к созданию программ. В этой книге были описаны элементы, являющиеся как бы строительными блоками программы. Для того чтобы создать работающее приложение, необходимо расположить эти элементы в нужном порядке, составляя программу из последовательности инструкций, блоков и функций.

В этой заключительной главе мы проследим за созданием законченной прикладной программы, и вы увидите, что в завершенном виде она содержит функции, операторы и инструкции, которые вам уже хорошо известны.

Прикладная программа

В предыдущих главах мы познакомились с работой со структурами и дисковыми файлами на примере создания каталога компакт-дисков. Вы узнали, как создать и каким образом читать файл, содержащий информацию о компакт-дисках. Однако приведенные ранее образцы программ демонстрировали нам только принципы использования конкретных функций. В них не были реализованы никакие дополнительные возможности, необходимые при работе. Например, в них отсутствовала возможность удаления сведений о диске из картотеки или просмотра содержания в поисках информации об определенном диске.

Программа, которую мы создадим в этой главе, является законченным приложением, позволяющим выполнять следующие задачи:

- добавлять информацию о диске в файл;
- удалять информацию о диске из файла;
- изменять название диска или другие данные;
- изменять номер ячейки, в которой хранится диск;
- искать информацию об определенном диске;
- изменять порядок расположения записей в файле в соответствии с номерами ячеек хранения дисков;
- выводить на печать сведения о коллекции.

Записи и структуры

Хотя речь идет о каталоге компакт-дисков, в этой главе будет постоянно встречаться слово «запись» (record). *Запись* — это термин, который используется при организации баз данных, а наша программа, по существу, представляет собой элементарную базу данных.

Базой данных называют совокупность взаимосвязанных элементов, используемых несколькими приложениями под управлением системы управления базой данных. В нашем случае совокупностью элементов является множество структур CD, но настоящая база данных может использоваться в качестве картотеки клиентов фирмы, инвентарной описи или каталогов коллекций самых различных типов.

Базу данных можно представить себе в виде компьютерной версии каталога регистрационных карточек. Каждая «карточка» в этом случае называется *записью* и содержит все сведения об одном из элементов, составляющих множество. То есть каждая запись в программе, содержащей, например, картотеку компакт-дисков, хранит информацию об одном диске.

Если в программе используются структуры, каждая запись соответствует *отдельной структуре целиком*. Каждый раз, когда программа читает структуру с диска, она считывает одну запись. Аналогично, каждый раз, когда программа записывает структуру на диск, она сохраняет одну запись.

Нашу программу мы будем конструировать как последовательный ряд инструкций. Перед изучением каждого раздела этой главы, внимательно прочитайте его название, а затем напишите для себя инструкции, которые, по вашему мнению, необходимы для выполнения данной задачи. Например, начните чтение следующего раздела с выписывания глобальных определений, которые могут вам понадобиться: констант, указателей или структур. Потом прочтите текст раздела, чтобы познакомиться с приведенным в нем вари-

антом. Помните, что обычно существует несколько способов написания одной и той же программы, поэтому ваш метод совершенно не обязательно будет ошибочным, если инструкции в вашей версии не совпадут с приведенными в книге.

Приложение II в конце книги содержит полный текст программы. Вам будет полезно после прочтения каждого раздела данной главы заглядывать в Приложение II с тем, чтобы посмотреть, каким именно образом в программе реализован описанный алгоритм.

Глобальные определения

Первой задачей при создании программы является определение необходимых глобальных переменных. Так как мы будем работать с дисковыми файлами, необходимо включить файл заголовков STDIO.H:

```
#include "stdio.h"
```

Как вы вскоре убедитесь, мы будем использовать три указателя на файлы: один— на файл принтера и два— на дисковые файлы. Один дисковый файл будет содержать данные о коллекции, а второй мы используем в качестве временного хранилища информации. Имена дисковых файлов у нас будут определяться как константы, поэтому обращаться к ним можно, используя имена этих констант:

```
#define FILENAME "Cdfile"

#define TEMPFILE "Temp"
```

Для собственных дисковых файлов можно использовать любые имена, причем они могут включать полное указание пути доступа, например:

```
#define FILENAME "C:\DATA\CD.DAT"
```

Отметим, что в этой программе подразумевается, что в одном контейнере может храниться не более 20 дисков, каждый из которых занимает отдельную ячейку. Поэтому стоит определить дополнительную константу, чтобы исключить возможность ввода номера ячейки больше 20:

```
#define MAX 20
```

Разумеется, при желании можно увеличить это число.

Все три указателя на файлы определяются в одной инструкции:

```
FILE *fp, *tp, *printer;
```

Следующим этапом является определение структуры CD. Для него мы используем инструкцию, приведенную в главе 11:

```
struct CD
        {
        char name[20];
        char description[40];
        char category[12];
        float cost;
        int number;
        } disc;
```

Последнее, что необходимо сделать, это принять меры для исключения возможности размещения в каждой ячейке больше одного компакт-диска. В этой программе мы используем целочисленный массив для хранения сведений о номерах ячеек, которые уже заняты, и целочисленную переменную, определяющую общее количество компакт-дисков в коллекции:

```
int slots[MAX];

int count;
```

Функция main()

Все основные задачи в программе выполняются нашими собственными функциями. Функция main() позволяет выбрать задачу, которую предстоит выполнить, и вызвать соответствующую функцию. Наиболее удобным способом выбора задачи является указание одного из пунктов представленного на экране меню. Таким образом, функция main() должна выполнять следующие операции:

- предлагать меню до тех пор, пока пользователь не выберет пункт, прекращающий работу программы;
- выводить на экран список задач;

- принимать вводимое с клавиатуры значение выбранного пункта меню;
- вызывать функцию, выполняющую выбранную задачу;
- завершать программу, если пользователь сделал соответствующий выбор.

Для получения выбора из меню в функции main() нужно определить специально предназначенную для этого локальную переменную. При выборе одного из пунктов меню (путем нажатия определенной клавиши) main() будет присваивать соответствующее значение этой переменной и в дальнейшем использовать его для определения функции, которую следует вызвать. Следовательно, функцию main() следует начать с определения

```
char select;
```

Далее, прежде чем предоставлять пользователю меню, мы должны знать, какие ячейки в контейнере уже заняты. Чтобы определить это, вызываем функцию getslots(). Эта функция открывает файл, читает все записи и присваивает каждое считанное значение переменной disc.number элементам массива slots[], а также подсчитывает общее количество записей, внесенных в файл:

```
getslots()
        {
        int index;
        index = 0;
        count = 0;
        if ((fp = fopen(FILENAME, "r")) != NULL)
                {
                while (fread(&disc, sizeof(disc),
 1, fp) == 1)
                        {
                        slots[index] = disc.number;
                        index++;
                        count++;
                        }
                fclose(fp);
                }
        }
```

Вывод меню на экран необходимо повторять каждый раз после выполнения любой функции, вызываемой из меню. Если этого не сделать, выполнение программы завершится после выполнения одной функции. В этом случае, если вы захотите выполнить две задачи, например, добавить новую «карточку» и вывести на печать обновленный список, вам придется запустить программу дважды. Так как нам не известно точное количество повторов меню, мы используем цикл do.

Меню будет иметь восемь пунктов, по одному на каждую основную задачу, плюс восьмой для завершения программы. Для вывода меню на экран используются функции puts() или printf(), а ввод ответа, означающего выбор одного из пунктов, осуществляется с помощью функции getchar(). Использование getchar() позволяет избежать необходимости нажатия клавиши Enter при выборе каждой задачи. Если для выбора использовать цифры, а не буквы, мы тем самым избавляем себя от необходимости учитывать регистр вводимого символа. Итак, функция main() будет выглядеть примерно следующим образом:

```
main()
        {
        char select;
        do
                {
                puts("      Моя коллекция компакт-дисков\n");
                puts("1 Добавить карточку\n");
                puts("2 Удалить карточку\n");
```

```
                puts("3 Редактировать
содержимое карточки\n");
                puts("4 Изменить номер п/п\n");
                puts("5 Сортировка карточек\n");
                puts("6 Найти карточку\n");
                puts("7 Вывести на печать\n");
                puts("8 Выйти из программы\n");
                printf("Пожалуйста, введите номер
выбранного пункта: ");
                select = getchar();
                putchar('\n');
```

После того как выбор пункта меню сделан, main() должна вызвать соответствующую функцию. Восемь вариантов выбора требуют использования семи инструкций if. Для того чтобы работу программы было легче контролировать, а текст ее было легче читать, имеет смысл использовать переключатель switch, который будет проверять введенное значение переменной. Помните о необходимости включить ветвь default на случай, если пользователь введет цифру или букву, не предусмотренную в меню. Дополняем функцию main() следующими инструкциями:

```
switch(select)
        {
        case '1' :
                addcd();
                break;
        case '2' :
                delcd();
                break;
        case '3' :
                chcd();
                break;
        case '4' :
                chloc();
                break;
        case '5' :
                sort();
                break;
        case '6' :
                locate();
                break;
        case '7' :
                plist();
                break;
        case '8' :
                break;
        default:
                puts("Ошибка, повторите ввод\n\n");
                }
```

```
        }
        while (select != '8');
        return(0);
}
```

Цикл do...while повторяет меню (в данном случае инструкции switch) до тех пор, пока в ответ на предложение сделать выбор пользователь не введет цифру8, завершающую работу программы. После того как пользователь сделает выбор и выбранная задача будет завершена, меню снова появится на экране, и можно будет попросить программу выполнить другую задачу.

Добавление записей: функция addcd()

Функция, которая добавляет информацию о компакт-диске, построена по шаблону, аналогичному тому, который использовался в главе 12 в программе добавления данных в файл.

Между примером, показанным в главе 12, и функцией, использующейся в данной программе, есть, тем не менее, три отличия. В нашей программе пользователь не может добавить информацию о новом диске, если в контейнере нет места, куда бы можно было поместить этот диск. Функция getslots() подсчитывает количество дисков, уже имеющихся в контейнере. Когда их общее число достигает максимального, на экране появляется соответствующее сообщение и программа возвращается в меню:

```
int pause;
if (count >= MAX)
        {
        puts("К сожалению, свободных ячеек нет\n");
        pause = getchar();
        return;
        }
```

Далее, в нашей программе файл открывается с режимом доступа "a", поэтому файл будет создан даже в том случае, если он не существовал к моменту запуска программы:

```
if ((fp = fopen(FILENAME, "a")) == NULL)
        {
        printf("Невозможно открыть файл %s\n", FILENAME);
        exit();
        }
```

Если файл успешно открыт или вновь создан, данные будут добавлены в его конец.

Следующая функция просит пользователя ввести номер ячейки от1 до20. Так как ввод этой информации необходим при выполнении нескольких задач в программе, мы выделили его в отдельную функцию, которая называется getslot() и может быть вызвана при возникновении в этом необходимости:

```
getslot()
{
int index, flag, pause;
        do
        {
        flag = 0;
        printf("Введите номер ячейки: ");
        scanf("%d", &disc.number);
        for (index = 0; index < count; index++)
                {
                if (slots[index] == disc.number)
                        {
                        printf("К сожалению, ячейка
```

```
уже занята, попробуйте другую\n");
                                        flag = 1;
                                        }
                                }
                        }
                        while (disc.number < 1 || disc.number >
MAX || flag == 1);
                        count++;
                        slots[count] = disc.number;
return;
}
```

Функция getslot() не позволяет вводить номер уже занятой ячейки. После записи в файл информации о новом компакт-диске происходит вызов функции getslots(), которая обновляет массив номеров ячеек.

Удаление записи: функция delcd()

Функция, удаляющая информацию о диске, использует стандартный алгоритм для работы с последовательными файлами. Используя последовательный доступ, мы не можем непосредственно перейти к интересующему нас месту в файле, чтобы изменить содержащуюся там запись. Если открыть файл с режимом доступа "w", его содержимое будет уничтожено. При использовании режима доступа "a" мы сможем только добавить данные в конец файла.

Решение этой проблемы лежит в использовании двух файлов. Функция delcd() открывает файл с данными, используя режим доступа "r", и временный файл с режимом доступа "w":

```
if (fp = fopen(FILENAME, "r")) == NULL)
                {
                printf("Невозможно открыть файл %s\n", FILENAME);
                exit();
                }
if ((tp = fopen(tempfile, "w")) == NULL)
                {
                printf("Невозможно открыть файл %s\n", tempfile);
                fclose(fp);
                exit();
                }
```

После этого мы вводим название диска, который следует удалить:

```
printf("Введите название диска: ");
gets(delname);
```

Цикл while осуществляет чтение каждой структуры (записи) из исходного файла:

```
while (fread(&disc, sizeof(disc), 1, fp) == 1)
                {
```

Если запись не является той, которую пользователь хочет удалить, функция delcd() заносит ее во временный файл:

```
if (strcmp(disc.name, delname) != 0)
fwrite(&disc, sizeof(disc), 1, tp);
```

В нашей программе функция strcmp() используется в нескольких местах для определения того, прочитана ли из файла нужная запись. Функция strcmp() не входит в K&R-стандарт Си, но она есть во всех библиотеках ANSI Си и Си++.

Встретив нужную запись, функция delcd() не копирует ее во временный файл, а вместо этого устанавливает флаг, чтобы впоследствии узнать, была ли структура удалена:

```
else
        fflag = 'y';
        }
```

Цикл while продолжает копирование из одного файла в другой до тех пор, пока не достигнет конца файла, после чего оба файла закрываются:

```
fclose(fp);
fclose(tp);
```

Если функция delcd() не нашла записи, которую пользователь пожелал удалить, она выведет на экран соответствующее сообщение, после чего завершит работу:

```
puts("\nДиск с таким названием не найден");
pause = getchar();
```

Произведя удаление записи, вы получите два разных файла. Исходный файл содержит все данные, включая и информацию, которую вы хотите удалить. Временный файл содержит только данные, которые требовалось сохранить, так как функция delcd() не записала в него структуру, подлежащую удалению. Но мы не можем оставить данные в таком виде, программа способна вносить дополнительные данные и выполнять другие функции лишь с исходным файлом, поэтому мы должны привести его содержимое в соответствие с полученным результатом.

Некоторые компиляторы Си++ поставляются с библиотечными функциями, позволяющими измененять имена файлов, но в некоторых компиляторах они отсутствуют. В общем виде мы должны снова открыть оба файла, теперь уже с обратным порядком доступа, то есть исходный файл открывается с режимом доступа "w", после чего данные из него будут удалены, а временный файл открывается с режимом доступа "r". Для этого используется функция openwr(), вызываемая функцией delcd() и другими функциями:

```
if (fp = fopen(FILENAME, "w")) == NULL)
        {
        printf("Невозможно открыть файл %s\n", FILENAME);
        exit();
        }
if ((tp = fopen(TEMPFILE, "r")) == NULL)
        {
        printf("Невозможно открыть файл %s\n", tempfile);
        fclose(fp);
        exit();
        }
```

После того как файлы открыты, функция delcd() считывает записи из временного файла и переносит их в исходный файл. После закрытия файлов, исходный файл содержит уже исправленную информацию, не включающую данные, которые мы хотели удалить:

```
while (fread(&disc, sizeof(disc), 1, tp) == 1)
        fwrite(&disc, sizeof(disc), 1, fp);
fclose(fp);
fclose(tp);
```

Перед завершением функции delcd() программа вызывает функцию getslots(), обновляющую массив номеров ячеек.

Итак, теперь у нас есть два файла, которые содержат одинаковые данные. Единственный недостаток этой процедуры состоит в том, что диск компьютера должен иметь достаточно свободного места для хранения двух файлов с данными. Временный файл можно рассматривать как резервную копию. Если с исходным файлом произойдет что-то нежелательное, останется временный файл, содержащий данные в том виде, в каком они были до выполнения последней операции. Если вы по каким-либо причинам не хотите хранить временный файл на диске, то можете открыть его с режимом доступа "w", а потом немедленно закрыть:

```
tp = fopen(tempfile, "w");
fclose(tp);
```

В результате файл останется на диске, но совершенно пустой, так что он займет очень мало дискового пространства.

Программа продолжает проверку картотеки даже после того, как искомая запись удалена. Таким образом, вы можете удалить несколько одноименных записей.

Часто используемые функции

Большое количество описанных здесь процедур выполняется несколькими основными функциями программы. Вместо того чтобы каждый раз писать соответствующие инструкции, эти процедуры можно выделить в отдельные функции и вызывать по мере необходимости. Такими функциями являются:

- openwr(): открывает файл с данными для чтения, а временный файл для записи;
- openrw(): открывает файл с данными для записи, а временный файл для чтения;
- nofind(): выводит на экран соответствующее сообщение, если указанная пользователем запись не найдена.

Можно несколько усилить процедуру, если выводить на экран указанную запись и просить пользователя дать подтверждение, что эту запись действительно необходимо удалить. Если пользователь решит не удалять запись, программа должна скопировать ее во временный файл.

Редактирование данных: функция chcd()

Для внесения изменений в картотеку используется тот же основной алгоритм, что и для удаления записи: вся обновленная информация записывается во временный файл, а затем копируется обратно в файл данных. В этом случае, вместо того, чтобы игнорировать записи, которые мы хотим изменить, в них записывается новая информация, а уже затем они сохраняются во временном файле.

Прежде всего, функция chcd() запрашивает название компакт-диска, в который вы хотите внести изменения, затем выполняет цикл while, в котором считывается каждая структура:

```
openrw();
puts("Введите название диска: ");
gets(chname);
while (fread(&disc, sizeof(disc), 1, fp) == 1)
{
```

Затем, если очередную структуру редактировать не следует, функция записывает ее во временный файл:

```
if (strcmp(disc.name, chname) != 0)
fwrite(&disc, sizeof(disc), 1, tp);
```

Когда функция встречает искомую запись, она выводит на дисплей текущие данные, а затем дает подсказку для ввода новой информации:

```
else
        {
        fflag = 'y';
        puts("Текущие данные\n");
        showdisc();
        puts("Новые данные\n");
        printf("Введите название диска: ");
        gets(disc.name);
        printf("Введите описание: ");
        gets(disc.description);
        printf("Введите категорию: ");
```

```
gets(disc.category);
printf("Введите цену: ");
scanf("%f", &disc.cost);
if (count >= MAX)
        {
        puts("Невозможно ввести номер ячейки\n");
        pause = getchar();
        }
else
        {
        getslot();
        }
```

Инструкция if (count >= MAX) не позволяет пользователю ввести новый номер ячейки, если файл содержит максимально допустимое количество записей. В этом случае функция getslot() не выполнится и исходный номер ячейки будет прочитан с диска и занесен в конец отредактированной записи.

Функция chcd() написана таким образом, что пользователь вынужден набирать всю информацию о диске заново, даже если он хочет изменить всего один пункт. Вы можете самостоятельно сделать редактирование информации более удобным для пользователя, если измените ее так, чтобы он мог ограничиться простым нажатием клавиши Enter в том случае, если необходимо сохранить текущее содержимое какого-нибудь пункта. Например, для ввода нового названия можно использовать следующие инструкции:

```
printf("Введите название диска: ");
gets(disc.name);
if (strlen(name) > 0)
        strcpy(disc.name, name);
```

Если пользователь вводит новое название, оно присваивается переменной disc.name, после чего заносится в файл вместе с соответствующей записью. Если пользователь нажимает **Enter**, не печатая нового названия, содержимое disc.name остается без изменений и в записи сохраняется прежнее наименование диска.

Поскольку нам необходимо выводить информацию на экран несколько раз во время работы программы, процедуру вывода можно выделить в самостоятельную функцию, названную нами showdisc(), и вызывать ее по мере надобности:

```
showdisc()
        {
        printf("Название        %s\n", disc.name);
        printf("Описание        %s\n", disc.description);
        printf("Категория       %s\n", disc.category);
        printf("Цена                    %6.2f\n", disc.cost);
        printf("Номер п/п       %d\n", disc.number);
        puts("\n\n");
        return;
        }
```

После чтения новых данных функция chcd() записывает структуру во временный файл:

```
fwrite(&disc, sizeof(disc), 1, tp);
```

После завершения чтения файла fp функция chcd() закрывает оба файла. Если ни одна запись не была отредактирована (например, потому что не был найден указанный диск), функция выводит на экран соответствующее сообщение и останавливается:

```
fclose(fp);
fclose(tp);
```

```
if (fflag == 'n')
nofind();
```

Если изменения были внесены, функция chcd() снова открывает файлы с обратным порядком доступа, переписывает данные в исходный файл данных и обновляет массив номеров ячеек:

```
else
        {
        openwr();
        while (fread(&disc, sizeof(disc), 1, tp) == 1)
        fwrite(&disc, sizeof(disc), 1, fp);
        fclose(fp);
        fclose(tp);
        }
        getslots();
return;
}
```

Изменение номера ячейки: функция chloc()

Функция, которая используется для изменения номера ячейки в карточке компакт-диска, в сущности, не отличается от функции редактирования записи, за исключением процедуры ввода номера ячейки:

```
puts("Текущая информация\n");
showdisc();
puts("\nНовый номер\n");
getslot();
```

Однако в начало функции добавлена инструкция if, чтобы избежать изменения номера ячейки в том случае, если в контейнере нет свободных ячеек.

Вывод записи на экран: функция locate()

Для того чтобы вывести на экран определенную запись, следует ввести название диска, открыть файл для чтения и считывать записи до тех пор, пока не будет найдена нужная:

```
while (fread(&disc, sizeof(disc), 1, fp) == 1)
        {
        if (strcmp(disc.name, name) == 0)
                {
```

Функция locate() вызывает функцию showdisc(), выводящую информацию на экран, затем временно останавливает выполнение программы с тем, чтобы дать пользователю время прочитать представленные данные:

```
                fflag = 'y';
                showdisc();
                printf("Для продолжения работы
 нажмите Enter");
                pause = getchar();
                putchar('\n');
                }
        }
```

Функция showdisc() читает файл целиком, поэтому на экран будут выведены все записи с одинаковыми наименованиями.

Печать записей: функция plist()

Для того чтобы вывести на печать каталог коллекции компакт-дисков, программа должна открыть два файла: файл данных должен быть открыт для чтения, а

Комбинирование функций

В нашем приложении каждая задача представлена в виде отдельной функции. Но так как функции почти идентичны, их можно свести в одну. После того как запись найдена и выведена на экран монитора, в исправленной таким образом программе будет использоваться оператор if для выбора действий, которые необходимо выполнить:

```
if (select = '3')
{
puts("Новые данные\n");
printf("Введите название диска: ");
gets(disc.name);
printf("Введите описание: ");
gets(disc.description);
printf("Введите категорию: ");
gets(disc.category);
printf("Введите цену: ");
scanf("%f", &disc.cost);
}
getslot();
```

Если выбран пункт меню под номером 3, на экране появятся запросы для ввода названия диска, его описания, категории и цены. После выполнения инструкции if программа вызывает функцию getslot() для ввода нового номера ячейки. Если выбран пункт 4, изменяющий только номер ячейки, то и введен будет лишь новый номер ячейки.

файл принтера, для которого используется стандартное имя системы MS-DOS "prn", должен быть открыт для записи:

```
if ((fp = fopen(filename, "r")) == NULL)
        {
        printf("Невозможно открыть файл %s\n", filename);
        exit();
        }
if (printer = fopen("prn", "w")) == NULL)
        {
        printf("Печатающее устройство не готово
 к работе\n", filename);
        fclose(fp);
        exit();
        }
```

Функция plist() читает каждую запись и затем печатает данные с помощью функции fprintf():

```
while (fread(&disc, sizeof(disc), 1, fp) == 1)
        {
        fprintf(printer, "Название          %s\n",
 disc.name);
```

```
        fprintf(printer, "Описание          %s\n",
disc.description);
        fprintf(printer, "Категория         %s\n",
disc.category);
        fprintf(printer, "Цена              %6.2f\n",
disc.cost);
        fprintf(printer, "Номер п/п         %d\n",
disc.number);
        fprintf(printer, "\n\n");
        }
```

По окончании печати функция закрывает оба файла и возвращает управление в main():

```
        fclose(printer);
        fclose(fp);
        return;
}
```

Сортировка записей: функция sort()

Добавляя сведения о новом диске в картотеку, вы можете поместить его в ячейку, которая не обязательно имеет номер, следующий по порядку за номером ячейки предыдущего компакт-диска. Может случиться, что вы захотите оставить несколько ячеек пустыми, предназначая их для определенных дисков, которые еще не куплены. Иными словами, добавляя записи в файл, вы можете вводить их в порядке, который не соответствует их расположению в контейнере.

При выводе на принтер карточки дисков будут напечатаны в том порядке, в котором вы их вводили в файл, и для того чтобы найти диск по его номеру ячейки, придется просмотреть весь список.

Сортировка записей в данном случае означает, что карточки дисков будут переписаны таким образом, чтобы они хранились в порядке возрастания номеров ячеек. Диск, помещенный в первую ячейку, станет первым в списке, диск из второй ячейки— вторым и так далее.

Существуют буквально десятки способов сортировки содержимого файла. Некоторые алгоритмы включают чтение данных в массив, сортировку элементов массива в памяти и запись массива обратно на диск. Другие алгоритмы используют два или больше файлов, в которые данные заносятся блоками, а потом опять переписываются в один файл, но уже в другом порядке.

В приведенной здесь функции sort() применен алгоритм, который требует участия только одного файла, но зато использует массив структур. Для работы функции необходимо определить массив структур и несколько переменных:

```
sort()
        {
        struct CD temp[MAX];
        int index, loop1, loop2, endloop;
        loop1 = 0;
        loop2 = 0;
        endloop = 0;
        index = 0;
```

Функция sort() открывает файл данных для чтения, считывает данные в массив и закрывает файл:

```
while (fread(&disc, sizeof(disc), 1, fp) == 1)
        {
        temp[index] = disc;
        index++;
        }
fclose(fp);
```

Переменная index используется в качестве индекса массива.

Далее, функция sort() снова открывает файл, уже для записи, и заносит в него данные, используя для этого два цикла for:

```
if ((fp = fopen(filename, "w")) == NULL)
        {
        printf("Невозможно открыть файл %s\n", filename);
        exit();
        }
for (loop1 = 1; loop1 < MAX+1; loop1++)
        {
        for (loop2 = 0; loop2 < count; loop2++)
```

Внешний цикл for увеличивает значение переменной loop1 от 1 до максимального допустимого количества дисков. Во время первого прохождения внешнего цикла, внутренний цикл for просматривает массив в поисках диска, занимающего ячейку с номером 1. Как только этот диск найден, описывающая его структура записывается в файл:

```
if (temp[loop2].number == loop1)
        {
        fwrite(&temp[loop2], sizeof(temp[loop2]), 1, fp);
        endloop++;
        }
```

и внутренний цикл завершается. После этого повторяется выполнение внешнего цикла, и внутренний цикл снова просматривает массив, теперь уже в поисках диска в ячейке с номером 2. Как только соответствующий диск находится, данные о нем тут же записываются в файл. Этот процесс повторяется для всех имеющихся в наличии номеров ячеек.

Как только данные обо всех дисках оказываются записанными обратно в файл, необходимость дальнейшего поиска номеров ячеек отпадает. Мы использовали переменную endloop для того, чтобы запоминать количество структур, записанных в файл. После записи очередной структуры, функция sort() увеличивает значение endloop на единицу. Когда значение переменной endloop становится равным максимальному допустимому количеству компакт-дисков в коллекции, это означает, что все структуры уже записаны в файл и повторение циклов может быть прекращено:

```
if (endloop == count)
        break;
```

После чего файл закрывается и выполнение функции sort() завершается.

Не пожалейте времени и просмотрите полный текст программы снова. Подумайте, какие улучшения на ваш взгляд можно внести в работу программы и какие изменения в тексте вы могли бы для этого сделать.

ПРИЛОЖЕНИЕ I

Ответы и решения

Глава 1

1.Составьте детальный план программы для расчета заработной платы и оплаты сверхурочных на основе количества отработанных в неделю часов.

ВВОД

Ввести количество часов в рабочей неделе.

Ввести количество часов, отработанных сверхурочно.

Ввести величину оплаты за 1 час работы.

ОБРАБОТКА

Умножить количество часов в рабочей неделе на

оплату за 1 час работы.

Умножить количество отработанных сверхурочно

часов на оплату 1 часа, умноженную на 1.5.

Сложить оба результата.

ВЫВОД

Вывести результат последней операции на экран.

2.Составьте детальный план программы, которая определяет, имеет ли право данная персона уйти на пенсию (пенсионный возраст— 65 лет).

ВВОД

Ввести возраст указанного лица.

ОБРАБОТКА

Сравнить введенное значение с числом 65.

ВЫВОД

Если введенное значение больше или равно 65,

вывести на экран сообщение, что данное лицо

имеет право на пенсию.

Если введенное значение меньше 65, вывести

сообщение, что данное лицо не имеет права

на пенсию.

Глава 2

1.Напишите программу, которая выводит на экран монитора следующее сообщение:

Добро пожаловать в мой мир.Командовать

парадом буду я.

```
main
{
puts("Добро пожаловать в мой мир.\n");
puts("Командовать парадом буду я.\n");
}
```

2.Напишите программу, которая выводит в центре экрана ваше имя, адрес и номер телефона.

main

```
{
puts(" А. Аардварк\n");
puts(" 111 Атлантик Авеню\n");
puts(" Анкоридж, Аляска 09987\n");
puts(" 123-555-1234\n");
}
```

3.Объясните, почему данная программа написана неверно:

```
main()
        (
        puts("Меня зовут Алвин");
        }
```

Тело функции main() открывает круглая скобка

вместо фигурной.

Глава 3

1.Решите, какие типы данных вам необходимы, и напишите их определения для программы, которая рассчитывает недельную заработную плату сотрудника, получающего двойную оплату за сверхурочные часы (рабочая неделя— 40 часов).

```
float payrate, reg_hours, o_hours, total;
```

Если вы хотите посчитать заработок, включая

оплату за неполные часы, используйте переменные

reg_hours и o_hours с типом float, в противном

случае, переменные могут иметь тип int.

2.Решите, какие типы данных вам необходимы, и напишите их определения для программы, которая рассчитывает сумму и среднее арифметическое значение четырех чисел.

Для вещественных чисел:

```
float number_1, number_2, number_3, number_4, sum, average;
```

Для целых чисел:

```
int number_1, number_2, number_3, number_4, sum;
float average;
```

3.Объясните, какие ошибки имеются в следующих инструкциях:

```
char client[3]="Аякс";
main()
float tax_due;
char name(10);
int count(5);
```

Значение, присвоенное строковой переменной client,

включает 4 символа, в то время как максимальное

количество символов, указанное в определении,

равно 2 (не забывайте про символ \0 !).

В определении строковой переменной name использованы

круглые скобки, вместо квадратных.

В определении строковой переменной count использован

тип int, вместо char, и круглые скобки,

вместо квадратных.

Глава 4

1.Напишите программу вывода вашего имени и адреса на экран с использованием функции puts().

```
main
{
puts("А. Аардварк");
puts("111 Атлантик Авеню");
puts("Анкоридж, Аляска 09987");
}
```

2.Напишите программу вывода вашего имени и адреса на экран с использованием функции printf().

```
main
{
printf("А. Аардварк \n");
printf("111 Атлантик Авеню \n");
printf("Анкоридж, Аляска 09987\n");
}
```

3.Напишите функцию puts(), которая выводит слово "Заглавие" в середине экрана. Ширина экрана 80 символов.

```
puts("     Заглавие\n") /* 34 пробела */
```

4.Напишите функцию printf(), которая выводит слово "Страница" с правой стороны экрана.

```
printf("%80s","Страница");
```

5.Напишите функцию printf(), которая выводит значения следующих переменных:

```
float length, width, height, volume;
```

float length, width, height, volume;

printf("%f %f %f %f", length, width, height, volume);

6.Программа должна отображать имя и возраст субъектов. Напишите функцию printf(), которая выводила бы значения переменных:

```
char name[12];
int age;
```

char name[12];

int age;

printf("%s is %d years old", name, age);

7.Программа содержит следующие переменные:

```
char item[] = "Дискеты";
float cost = 3.55;
float markup = 0.75;
```

Напишите функцию printf(), которая выводит на экран следующие сообщения:

Наименование товара: Гибкий диск

Цена за 1 упаковку: 3.55

Наценка: 0.75

Обратите внимание на выравнивание.

printf("Наименование товара: %17s\n", item);

printf("Цена за 1 упаковку: %17.2f\n",cost);

printf("Наценка: %17.2f\n",markup);

8.Программа содержит следующую переменную:

int count = 30;

Используя значение переменной count для вывода числа в последней строке, напишите программу, которая подает звуковой сигнал и выводит на экран следующее сообщение:

Внимание! Внимание! Внимание! Внимание!

Нежелательное отклонение параметров среды.

У вас есть 30 секунд, чтобы покинуть помещение.

putchar('\007');

printf("Внимание! Внимание! Внимание! Внимание!\n");

printf("Нежелательное отклонение параметров среды.\n");

*printf("У вас есть %d секунд, чтобы покинуть *

помещение.", count);

Глава 5

1.Напишите программу, в которой вводится, а затем отображается на экране монитора в одной строке ваше имя и номер телефона.

```
main()
        {
        char name[25];
        char telephone[12];
        printf("Пожалуйста, введите Ваше имя: ");
        gets(name);
        printf("Введите свой номер телефона: ");
        gets(telephone);
        printf("Имя: %s Номер телефона: %s", name, \
 telephone);
        }
```

2.Напишите программу, в которой вводится число, а затем на экран выводится адрес области памяти, куда было записано это число.

```
main()
        {
        int number;
        printf("Пожалуйста, введите целое число: ");
        scanf("%d", &number);
        putchar('\n');
        printf("Число %d хранится в памяти по адресу %d", \
 number, &number);
        }
```

3.Напишите программу, в которой вводятся три числа, а затем эти числа отображаются на экране в порядке, обратном тому, в котором их вводили.

```
main()
        {
        int num_1, num_2, num_3;
        printf("Введите первое целое число: ");
        scanf("%d", &num_1);
        printf("Введите второе целое число: ");
        scanf("%d", &num_2);
        printf("Введите третье целое число: ");
        scanf("%d", &num_3);
        printf("3: %d 2: %d 1: %d", num_3, num_2, num_1);
        }
```

4.Напишите программу, в которой используются функции getchar(), gets() и scanf().

```
main()
        {
        char first[10], last[15];
        char initial;
        int age;
        printf("Введите Ваше имя: ");
        gets(first);
        printf("Введите Ваше отчество в виде инициала: ");
        initial = getchar();
        putchar('\n');
        printf("Введите Вашу фамилию: ");
        gets(last);
        printf("Введите Ваш возраст: ");
        scanf("%d", &age);
        printf("Имя: %s %c %s\n", first, initial, last);
        printf("Возраст: %d", age);
        }
```

5.Объясните, почему следующая программа написана неправильно:

```
main()
        {
        char initial;
        initial = gets();
        puts(initial);
        }
```

Переменная initial определена как символьная.

Поэтому символ должен вводиться с помощью функции

getchar() и выводиться на экран с помощью функции

putchar() или printf().

Глава 6

1.Напишите программу, которая сообщает пользователю, сколько лет ему будет в 2000 году.

```
main()
```

```
{
int year, age, togo;
printf("Введите год: ");
scanf("%d", &year);
printf("Укажите Ваш возраст: ");
scanf("%d", &age);
togo = 2000 - year + age;
printf("В 2000 году Вам будет %d лет", togo);
}
```

2.Напишите программу расчета квадрата и куба числа, введенного с клавиатуры.

```
main()
{
int number, square, cube;
printf("Введите целое число: ");
scanf("%d", &number);
square = number * number;
cube = number * number * number;
printf("Введено число %d\n", number);
printf("Квадрат числа равен %d\n", square);
printf("Куб числа равен %d\n", cube);
}
```

3.Напишите программу перевода температуры из шкалы Фаренгейта (F) в шкалу Цельсия (C). Формула пересчета C=(5.0/9.0)Ф(F–32).

```
main()
{
int temp;
float celsius;
printf("Введите значение температуры в виде \
 целого числа: ");
scanf("%d", &temp);
celsius = (5.0/9.0)*(temp-32);
printf("По шкале Фаренгейта: %d По шкале \
 Цельсия: %f", temp, celsius);
}
```

4.Модифицируйте программу из упражнения 3 так, чтобы она сообщала, на сколько градусов отстоит введенное значение температуры от точки замерзания по шкале Фаренгейта и по шкале Цельсия.

```
main()
{
int temp, ffreeze;
float celsius;
printf("Введите значение температуры в виде \
 целого числа: ");
scanf("%d", &temp);
celsius = (5.0/9.0)*(temp-32);
```

```
        ffreeze = temp - 32;
        printf("По Фаренгейту: %d От точки \
 замерзания: %d\n", temp, ffreeze);
        printf("По Цельсию: %f От точки замерзания: \
 %f", celsius, celsius);
        }
```

5.Объясните, почему следующая программа написана неверно:

```
#define TAX_RATE 0.06
main()
        {
        float cost, total;
        printf("Введите стоимость единицы товара: ");
        scanf("%f", &cost);
        printf("Введите величину транспортных расходов: ");
        scanf("%f", &shipping)'
        total = cost + cost * tax_rate + shipping;
        printf("Общая стоимость составляет %f", total);        }
```

В программе используется переменная shipping, которая не была определена. Вторая инструкция scanf() оканчивается апострофом, вместо точки с запятой.

Глава 7

1.Напишите программу Опросника, в котором задаются четыре вопроса; каждый вопрос и ответ оформите в виде отдельной функции.

```
char pause;
main()
        {
        quest1();
        quest2();
        quest3();
        quest4();
        }
quest1()
        {
        puts("Что является центром живой клетки?\n");
        puts("Для получения правильного ответа
                                        нажмите Enter\n");
        pause=getchar();
        puts("Клеточное ядро\n");
        return;
        }
quest2()
```

```
        {
        puts("Как можно заразиться солитером?\n");
        puts("Для получения правильного ответа
                                        нажмите Enter\n");
        pause=getchar();
        puts("Если есть недожаренное мясо\n");
        return;
        }
quest3()
        {
        puts("Назовите основные группы крови.\n");
        puts("Для получения правильного ответа
                                        нажмите Enter\n");
        pause=getchar();
        puts("Основных групп крови четыре A, B, AB и O\n");
        return;
        }
quest4()
        {
        puts("Как будет по-французски море?\n");
        puts("Для получения правильного ответа
                                        нажмите Enter\n");
        pause=getchar();
        puts("По-французски море называется la mer\n");
        return;
        }
```

2.Напишите программу, в которой вводится число, а затем вызывается функция для расчета и отображения четвертой степени этого числа.

```
main()
        {
        int number;
        printf("Введите целое число: ");
        scanf("%d", &number);
        tothefourth(number);
        }
tothefourth(value)
int value;
        {
        int power;
        power=value*value*value*value;
        printf("Четвертая степень числа %d равна %d",
                                        value, power);
        return;
```

```
        }
```

3.Внесите изменения в программу из упражнения 2 так, чтобы функция вычисляла четвертую степень числа, а затем передавала результат в main() для вывода на дисплей.

```
main()
        {
        int number, fourth;
        printf("Введите целое число: ");
        scanf("%d", &number);
        fourth = tothefourth(number);
        printf("Четвертая степень числа %d равна %d",
                                        number, fourth);
        }
int tothefourth(value)
int value;
        {
        return(value*value*value*value);
        }
```

4.Объясните, почему следующая программа написана неверно:

```
dothis()
        {
        puts("Это первое");
        main();
        return(0);
        }
main()
        {
        puts("Это второе");
        return();
        }
```

Собственная функция может помещаться только после функции main(), а не перед ней.

Собственная функция не может вызывать main().

Программа должна быть отредактирована следующим образом:

```
main()
        {
        puts("Это первое");
        dothis();
        return(0);
        }
dothis()
        {
        puts("Это второе");
```

```
return(0);
}
```

Глава 8

1.Напишите программу, в которой вводится числовое значение, а затем выдается сообщение, четное или нечетное число было введено.

```
main()
{
int number, remain;
printf("Введите число: ");
scanf("%d", &number);
remain=number % 2;
if(remain==0)
        puts("Введено четное число");
else
puts("Введено нечетное число");
}
```

2.Напишите программу, в которой вводится число и затем выдается сообщение, находится ли значение числа в пределах от 1 до 100.

```
main()
{
int number;
printf("Введите число: ");
scanf("%d", &number);
if(number > 0 && number <= 100)
        puts("Значение числа находится в пределах
                                        от 1 до 100");
}
```

3.Напишите программу, в которой вводится целое число, а затем выдается сообщение, в каком интервале находится значение числа: меньше 0, от 0 до 50, от 51 до 100, от 101 до 150, больше 150.

```
main()
{
int number;
printf("Введите число: ");
scanf("%d", &number);
if(number < 0)
        puts("Введено отрицательное число");
else
 if(number > 0 && number < 51)
         puts("Число находится в пределах
                                от 0 до 50");
 else
         if(number > 50 && number <= 101)
                 puts("Число находится в
                 пределах от 51 до 100");
```

```
                else
                        if(number > 101 && number < 151)
                                puts("Число находится в
                                пределах от 101 до 150");
                        else
                                puts("Введенное
                                число больше 150");
        }
```

4.Напишите программу, которая просит пользователя ввести числовые значения в переменные lownum и highnum. Значение lownum должно быть меньше чем highnum. Если числа введены не в соответствии с этим условием, программа должна поменять значения, поместив меньшее число в lownum, а большее— в highnum. Значения переменных должны быть выведены на экран.

```
main()
        {
        int lownum, highnum, temp;
        puts("Введите два числа. Первое число \n");
        puts("должно быть меньше, чем второе.\n");
        printf("Введите меньшее число: ");
        scanf("%d", &lownum);
        printf("Введите большее число: ");
        scanf("%d", &highnum);
        putchar('\n');
        if(lownum < highnum)
                puts("Вы ввели числа в правильном
                        порядке.\n");
        else
                {
                temp = highnum;
                highnum = lownum;
                lownum = temp;
                puts("Вы ввели числа неправильно.\n");
                puts("Придется исправить Вашу ошибку.\n");
                }
        printf("Значение переменной lownum
                равно %d\n", lownum);
        printf("Значение переменной highnum
                равно %d\n", highnum);
        }
```

5.Объясните, почему следующая программа написана неверно:

```
main()
        {
        int age;
        printf(Укажите Ваш возраст);
        scanf("%f", &age);
```

```
if age < 18 then
 puts("Вы не можете участвовать в выборах");
else
 if age > 18 then
  puts("Вы можете участвовать в выборах");
```

В первой инструкции printf() литерал не заключен в кавычки. Переменная age, определенная с типом int, вводится с помощью функции scanf() как float. Условие в инструкции if не заключено в круглые скобки. Слово then лишнее. Программа не выведет никакого сообщения в том случае, если будет указан возраст ровно 18 лет.

Глава 9

1.Отредактируйте текст программы из Листинга 8.10 (глава 8) таким образом, чтобы она повторялась до тех пор, пока пользователь не пожелает прекратить ввод данных.

```
main()
        {
        float rate, hours, total, regular, extra,
                                d_time, overtime;
        int moredata;
        do
                {
                printf("Введите оплату часа работы: ");
                scanf("%f", &rate);
                printf("Введите количество отработанных
                                часов: ");
                scanf("%f", &hours);
                d_time=rate * 2;
                if (hours <= 40)
                                {
                                regular = hours * rate;
                                extra = 0.0;
                                overtime = 0.0;
                                total = regular;
                                }
                else
                                {
                                regular = 40 * rate;
                                extra = hours - 40;
                                overtime = extra * d_time;
```

```
                    total = regular + overtime;
                    }
            putchar('\n');
            printf("Ваш обычный недельный заработок
                    равен %.2f\n", regular);
            printf("Вы отработали %.2f часов
                    сверхурочно\n", extra);
            printf("Оплата 1 часа сверхурочных равна
                    $%.2f\n", d_time);
            printf("Заработок за сверхурочные часы
                    равен %.2f\n", overtime);
            printf("Итого, Ваш недельный заработок
                    составляет %.2f\n", total);
            printf("Желаете продолжить расчет?
                    Y or N: ");
            moredata = getchar();
            putchar('\n');
            }
    while(moredata=='y' || moredata=='Y');
    }
```

2.Напишите программу, которая рассчитывает сумму 6-процентного налога на продажи для товаров, имеющих стоимость в пределах от 1 до 50 долларов, и выводит информацию на экран монитора в виде таблицы

Цена	Налог	Итого
1	$.06	$1.06
2	$.12	$2.12

```
main()
    {
    int cost;
    float shipping, total;
    puts("Цена\tНалог\tИтого\n");
    for(cost=1;cost<51;cost++)
            {
            shipping = cost * 0.06;
            total = cost + shipping;
            printf("$%d\t$%.2f\t\t$%6.2f\n",cost,
                    shipping, total);
            }
    }
```

3.Напишите программу, которая вводит десять чисел в пределах от 0 до 25.

```
main()
    {
    int count, number;
```

```
for(count=1;count<11;count++)
        {
        printf("Введите значение N%d", count);
        putchar('\n');
        do
                {
                printf("Вводите числа в пределах
                 от 0 до 25: ");
                scanf("%d",&number);
                }
        while(number<0 || number > 25);
                putchar('\n');
        }
}
```

4.Напишите программу, которая выводит на дисплей следующий график:

```
* * * * *
* * * *
* * *
* *
*
* *
* * *
* * * *
* * * * *
```

```
main()
        {
        int outer,inner;
        for(outer=5;outer>0;outer--)
                {
        for(inner=1;inner<=outer;inner++)
                        printf("*");
                putchar('\n');
                }
        for(outer=2;outer<6;outer++)
                {
                for(inner=1;inner<=outer;inner++)
                        printf("*");
                putchar('\n');
                }
        }
```

5.Объясните, почему следующая программа написана неправильно:

```
main()
```

```
{
float row, column;
puts("\t\tТаблица Пифагора\n\n");
for (row = 1; row <= 10; row++
        {
        for (column = 1; column <= 10; column+
                printf("%d", row * column);
        }
        putchar('\n');
}
```

Переменные row и column определены с типом float, но если вы хотите использовать их в цикле, они должны относиться к типу int. Инструкция putchar('\n') помещена таким образом, что она будет выполнена только один раз. Таблица, таким образом, будет выведена в одну линию. Поместите инструкцию на одну строку выше, перед закрывающей фигурной скобкой.

Глава 10

1.Напишите программу, в которой массивы используются для хранения имен, адресов и номеров телефонов 20 человек.

```
main()
{
char names[20][20], street[20][20],
        city[20][20];
char state[20][3], zip[20][6],
        phone[20][13], lookfor[20];
int count;
for(count=0;count<20;count++)
{
        puts("Введите имя");
        gets(names[count]);
        puts("Введите название улицы");
        gets(street[count]);
        puts("Введите город");
        gets(city[count]);
        puts("Введите штат");
        gets(state[count]);
        puts("Введите индекс");
        gets(zip[count]);
        puts("Введите номер телефона");
```

```
                gets(phone[count]);
                }
        }
```

2.Внесите в программу из упражнения 1 изменения так, чтобы иметь возможность ввода имени и последующего просмотра массива в поисках номера телефона соответствующего человека.

```
main()
        {
        char names[20][20], street[20][20],
                 city[20][20];
        char state[20][3], zip[20][6],
                 phone[20][13], lookfor[20];
        int count;
        for(count=0;count<20;count++)
                {
                puts("Введите имя: ");
                gets(names[count]);
                puts("Введите название улицы: ");
                gets(street[count]);
                puts("Введите город: ");
                gets(city[count]);
                puts("Введите штат: ");
                gets(state[count]);
                puts("Введите индекс: ");
                gets(zip[count]);
                puts("Введите номер телефона: ");
                gets(phone[count]);
                }
        puts("Введите имя: ");
        gets(lookfor);
        for(count=0; count<20;count++)
                {
                if(strcmp(names[count],lookfor)==0)
                        printf("%s %s\n", names[count],
                 phone[count]);
                }
        }
```

3.Объясните, почему следующая программа написана неправильно:

```
main()
        {
        int temps(31);
        int index, total;
        for (index = 0; index < 31; index++)
                {
                printf("Введите значение
```

```
                    температуры #%d: ", index);
            scanf("%d", &temps(index));
            }
    high = temps(0);
    low = temps(0);         index = 1;
    while (index < 31)
            {
            if (temps(index) > high)
                    high = temps(index);
            else
                    low = temps(index);
            index++;
            }
    printf("Минимальное значение
                    температуры равно %d\n", low);
    printf("Максимальное значение
                    температуры равно %d\n", high);
    }
```

Индексы всех элементов массива temps заключены в круглые скобки, вместо квадратных. Переменные high и low не определены. Переменная total определена, но не используется в программе. Конструкция if...else написана с ошибкой: если рассматриваемое значение не превышает текущее значение переменной high, это не значит, что оно непременно меньше текущего значения переменной low.

Глава 11

1.Напишите программу, в которой структура используется для составления инвентарной описи. Информация включает в себя название продукта, цену, количество, имя поставщика.

```
struct product
        {
        char name[20];
        float cost;
        int quantity;
        char vendor[20];
        } item;
main()
        {
        puts("Введите сведения о товаре\n\n");
        printf("Введите наименование:");
```

```
        gets(item.name);
        printf("Введите цену:");
        scanf("%f", &item.cost);
        printf("Введите количество единиц:");
        scanf("%d", &item.quantity);
        printf("Введите имя поставщика:");
        gets(item.vendor);
        }
```

2.Внесите изменения в программу из упражнения 1 с тем, чтобы можно было вводить информацию в массив структур, состоящий из 20 элементов.

```
struct product
        {
        char name[20];
        float cost;
        int quantity;
        char vendor[20];
        } item[20];
main()
        {
        int count;
        for(count=0;count<20;count++)
                {
                puts("\nВведите сведения о товаре\n\n");
                printf("Введите наименование:");
                gets(item[count].name);
                printf("Введите цену:");
                scanf("%f", &item[count].cost);
                printf("Введите количество единиц:");
                scanf("%d", &item[count].quantity);
                printf("Введите имя поставщика:");
                gets(item[count].vendor);
                }
        }
```

3.Внесите изменения в программу из упражнения 2 так, чтобы выводить на экран общую стоимость включенных в опись товаров.

```
struct product
        {
        char name[20];
        float cost;
        int quantity;
        char vendor[20];
        } item[20];
main()
        {
```

```
        float total;
        int count;
        total=0;
        for(count=0;count<20;count++)
                {
                puts("\nВведите сведения о товаре\n\n");
                printf("Введите наименование: ");
                gets(item[count].name);
                printf("Введите цену: ");
                scanf("%f", &item[count].cost);
                printf("Введите количество единиц: ");
                scanf("%d", &item[count].quantity);
                printf("Введите имя поставщика: ");
                gets(item[count].vendor);
                total = total + (item[count].cost *
                        item[count].quantity);
                }
        printf("Общая стоимость включенных в опись \
 товаров составляет %8.2f", total);
        }
```

4.Напишите программу, в которой две переменные типа float определяются в main() как локальные, а затем используются в функции, вычисляющей квадраты обоих чисел.

```
main()
        {
        float num1, num2;
        puts("Введите первое число");
        scanf("%f",&num1);
        puts("Введите второе число");
        scanf("%f",&num2);
        doubleit(&num1, &num2);
        }
doubleit(dcount1, dcount2)
float *dcount1, *dcount2;
        {
        float sq1, sq2;
        sq1 = *dcount1 * *dcount1;
        sq2 = *dcount2 * *dcount2;
        printf("Квадрат %f равен %f\n",*dcount1, sq1);
        printf("Квадрат %f равен %f\n",*dcount2, sq2);
        }
```

Для того чтобы вернуть значения вызывающей функции, используйте следующую программу:

```
main()
        {
        float num1, num2;
```

```
        puts("Введите первое число");
        scanf("%f",&num1);
        puts("Введите второе число");
        scanf("%f",&num2);
        doubleit(&num1, &num2);
        printf("Квадрат первого числа равен %f\n", num1);
        printf("Квадрат второго числа равен %f", num2);
        }
doubleit(dcount1, dcount2)
float *dcount1, *dcount2;
        {
        float sq1, sq2;
        *dcount1 = *dcount1 * *dcount1;
        *dcount2 = *dcount2 * *dcount2;
        }
```

5.Объясните, почему следующая программа написана неверно:

```
main()
        {
        struct CD
                {
                char description[40];
                char category[12];
                char name[20];
                float cost;
                int number;
                } disc;
        puts("Введите сведения о диске");
        printf("Введите название: ");
        gets(name);
        printf("Введите описание: ");
        gets(description);
        printf("Введите категорию: ");
        gets(category);
        printf("Введите цену: ");
        scanf("%f", &cost);
        printf("Введите номер ячейки: ");
        scanf("%d", &number);
        puts("Введена следующая информация о диске: ");
        printf("Название:       %s\n", name);
        printf("Описание:       %s\n", description);
        printf("Категория:      %s\n", category);
        printf("Цена:                           %6.2f\n", cost);
        printf("Номер п/п:      %d\n", number);
```

```
}
```

Ко всем переменным, используемым в инструкциях ввода и вывода, обращение происходит без использования имени структурной переменной. К переменным следует обращаться как к disc.name, disc.description и так далее.

Глава 12

1.Напишите программу, в которой функция fputs() используется для создания файла, содержащего названия кинофильмов.

```
#include "stdio.h"
main()
        {
        FILE *fp;
        char flag;
        char title[20];
        if((fp = fopen("MOVIES","w"))==NULL)
                {
                puts("Невозможно открыть файл");
                exit();
                }
        flag = 'y';
        while(flag!='n')
                {
                puts("Введите название кинофильма: ");
                gets(title);
                fputs(title, fp);
                fputs("\n",fp);
                printf("Желаете продолжить ввод?");
                flag=getchar();
                putchar('\n');
                }
        fclose(fp);
        }
```

2.Напишите программу, которая читает названия кинофильмов (упражнение1) в массив строк.

```
#include "stdio.h"
main()
        {
        FILE *fp;
        int index;
        char titles[80][12];
        index = 0;
        if((fp = fopen("MOVIES","r"))==NULL)
```

```
                {
                puts("Невозможно открыть файл");
                exit();
                }
        while(fgets(titles[index],12,fp)!= NULL)
                {
                puts(titles[index]);
                index++;
                if(index>80)
                        {
                        puts("К сожалению, вы уже ввели
                         80 названий.");
                        break;
                        }
                }
        fclose(fp);
        }
```

3.Напишите программу, в которой функция fprintf() используется для создания файла инвентарной описи, содержащей сведения о наименовании товара, его цене и количестве единиц, имеющихся в наличии.

```
#include "stdio.h"
main()
        {
        FILE *fp;
struct product
        {
        char name[20];
        float cost;
        int quant;
        } item;
        if((fp = fopen("MYFILE","w"))==NULL)
                {
                puts("Cannot open the file");
                exit();
                }
        puts("\nВведите сведения о товаре\n\n");
        printf("Введите наименование: ");
        gets(item.name);
        while(strlen(item.name)>0)
                {
                printf("Введите цену: ");
        canf("%f", &item.cost);
                printf("Введите количество единиц: ");
                scanf("%d", &item.quant);
```

```
                fprintf(fp, "%s %f %d\n", item.name,
                        item.cost, item.quant);
                printf("Введите наименование: ");
                gets(item.name);
                }
        fclose(fp);
        }
```

4.Напишите программу, которая читает файл инвентарной описи, созданный в упражнении 3.

```
#include "stdio.h"
main()
        {
        FILE *fp;
        struct product
                {
                char name[20];
                float cost;
                int quant;
                } item;
        if((fp = fopen("MYFILE","r"))==NULL)
                {
                puts("Невозможно открыть файл");
                exit();
                }
        while(fscanf(fp,"%s %f %d", item.name,
                        &item.cost, &item.quant) != EOF)
                {
                printf("Наименование: %s\n", item.name);
                printf("Цена %f\n", item.cost);
                printf("Количество: %d\n", item.quant);
                }
        fclose(fp);
        }
```

5.Отредактируйте программы из упражнений 3 и 4 так, чтобы они читали данные как структуры.

```
#include "stdio.h"
main()
        {
        FILE *fp;
        struct product
                {
                char name[20];
        loat cost;
                int quant;
                } item;
```

```
        if((fp = fopen("MYFILE","w"))==NULL)
                {
                puts("Невозможно открыть файл");
                exit();
                }
        puts("\nВведите сведения о товаре\n\n");
        printf("Введите наименование: ");
        gets(item.name);
        while(strlen(item.name)>0)
                {
                printf("Введите цену: ");
                scanf("%f", &item.cost);
                printf("Введите количество: ");
                scanf("%d", &item.quant);
                fwrite(&item, sizeof(item), 1, fp);
                printf("Введите наименование: ");
                gets(item.name);
                }
        fclose(fp);
        }

#include "stdio.h"
main()
        {
        FILE *fp;
        struct product
                {
                char name[20];
                float cost;
                int quant;
                } item;
        if((fp = fopen("MYFILE","r"))==NULL)
                {
                puts("Невозможно открыть файл");
                exit();
                }
        while(fread(&item, sizeof(item), 1, fp)==1)
                {
                printf("Наименование: %s\n", item.name);
                printf("Цена: %f\n", item.cost);
                printf("Количество: %d\n", item.quant);
                }
        fclose(fp);
```

```
}
```

6.Объясните, почему следующая программа написана неверно:

```
#include "stdio.c"
main()
{
FILE fp;
char letter;
if ((fp = fopen("MYFILE", "w")) == NULL)
{
puts("Невозможно открыть файл");
exit();
}
do
{
letter = getchar();
fputc(letter, fp);
}
while(letter != '\n');
fclose(fp);
}
```

*Файл заголовков называется STDIO.H, а не STDIO.C. Указатель на файл должен определяться со звездочкой перед именем: *fp. В цикле while должен быть указан код \r, а не \n.*

ПРИЛОЖЕНИЕ II

Прикладная программа

Листинг к главе 13. Программа составления каталога компакт-дисков.

```
/*CDAPP.C*/
#include "stdio.h"
#define FILENAME "CDfile"
#define TEMPFILE "Temp"
#define MAX 20
FILE *fp, *tp, *printer;
struct CD
        {
        char  name[20];
        char  description[40];
        char  category[12];
        float cost;
        int  number;
        } disc;
int slots[MAX];
int count;
main()
        {
        char select;
        getslots();
        do
                {
                puts("Моя коллекция компакт-дисков\n");
                puts("1 Добавить карточку\n");
                puts("2 Удалить карточку\n");
                puts("3 Редактировать
 содержимое карточки\n");
                puts("4 Изменить номер ячейки\n");
                puts("5 Сортировка карточек\n");
                puts("6 Найти карточку\n");
                puts("7 Вывести на печать\n");
                puts("8 Выйти из программы\n");
                printf("Пожалйста, введите Ваш выбор: ");
                select = getchar();
                putchar('\n');
                switch(select)
                        {
                        case '1' :
                                addcd();
```

```
                        break;
                case '2':
                        delcd();
                        break;
                case '3':
                        chcd();
                        break;
                case '4':
                        chloc();
                        break;
                case '5':
                        sort();
                        break;
                case '6':
                        locate();
                        break;
                case '7':
                        plist();
                        break;
                case '8':
                        break;
                default:
                        puts("Ошибка,
         повторите ввод\n\n");
                }
        }
        while(select!='8');
        return(0);
}
addcd()
{
int pause;
if(count>>=MAX)
        {
        puts("К сожалению
         свободных ячеек нет\n");
        pause=getchar();
        return;
        }
if((fp = fopen(FILENAME,"a")) == NULL)
        {
            printf("Невозможно открыть
         файл %s\n",FILENAME);
```

```
                exit();
            }
        puts("\n");
        printf("Введите название диска: ");
        gets(disc.name);
        printf("Введите описание:");
        gets(disc.description);
        printf("введите категорию:");
        gets(disc.category);
        printf("Введите цену:");
        scanf("%f", &disc.cost);
        getslot();
        fwrite(&disc, sizeof(disc), 1, fp);
        fclose(fp);
        getslots();
        return;
        }
delcd()
        {
        char  delname[20];
        char  fflag;
        fflag='n';
        openrw();
        puts("Удаление информации о диске\n");
        printf("Введите название диска: ");
        gets(delname);
        while(fread(&disc, sizeof(disc), 1, fp)==1)
            {
            if(strcmp(disc.name, delname) != 0)
                fwrite(&disc, sizeof(disc), 1, tp);
            else
                fflag='y';
            }
        fclose(fp);
        fclose(tp);
        if(fflag=='n')
            nofind();
        else
            {
            openwr();
            while(fread(&disc, sizeof(disc), 1, tp)==1)
                fwrite(&disc, sizeof(disc), 1, fp);
            fclose(fp);
```

```
                fclose(tp);
                }
        getslots();
        return;
        }
chcd()
        {
        char  chname[20];
        char  fflag;
        int pause;
        fflag='n';
        openrw();
        puts("Редактирование сведений о диске\n");
        printf("Введите название диска: ");
        gets(chname);
        while(fread(&disc, sizeof(disc), 1, fp)==1)
                {
                if(strcmp(disc.name,chname)!=0)
                        fwrite(&disc, sizeof(disc), 1, tp);
                else
                        {
                        fflag='y';
                        puts("Текущая информация\n");
                        showdisc();
                        puts("Новая информация\n");
                        printf("Введите название диска: ");
                        gets(disc.name);
                        printf("Введите описание:");
                        gets(disc.description);
                        printf("Введите категорию:");
                        gets(disc.category);
                        printf("Введите цену:");
                        scanf("%f", &disc.cost);
                        if(count>>=MAX)
                                {
                                puts("К сожалению
                 свободных ячеек нет\n");
                                pause=getchar();
                                }
                        else
                                {
                                getslot();
                                }
```

```
                        fwrite(&disc, sizeof(disc), 1, tp);
                        }
                }
        fclose(fp);
        fclose(tp);
        if(fflag=='n')
                nofind();
        else
                {
                openwr();
                while(fread(&disc, sizeof(disc), 1, tp)==1)
                fwrite(&disc, sizeof(disc), 1, fp);
                fclose(fp);
                fclose(tp);
                }
        getslots();
        return;
        }
chloc()
        {
        char  chname[20];
        char  fflag;
        int pause;
        fflag='n';
        if(count>>=MAX)
                {
                puts("К сожалению свободных ячеек нет\n");
                pause=getchar();
                return;
                }
        openrw();
        puts("Изменение номера ячейки\n");
        printf("Введите название диска: ");
        gets(chname);
        while(fread(&disc, sizeof(disc), 1, fp)==1)
                {
                if(strcmp(disc.name,chname)!=0)
                        fwrite(&disc, sizeof(disc), 1, tp);
                else
                        {
                        fflag='y';
                        puts("Текущая информация\n");
                        showdisc();
```

```
                        puts("\nНовый номер ячейки\n");
                        getslot();
                        fwrite(&disc, sizeof(disc), 1, tp);
                        }
                }
        fclose(fp);
        fclose(tp);
        if(fflag=='n')
                nofind();
        else
                {
                openwr();
                 while(fread(&disc, sizeof(disc), 1, tp)==1)
                  fwrite(&disc, sizeof(disc), 1, fp);
                fclose(fp);
                fclose(tp);
                }
        getslots();
        return;
        }
locate()
        {
        char name[20];
        char  fflag, pause;
        fflag='n';
        if((fp = fopen(FILENAME,"r")) == NULL)
                {
                printf("Невозможно
                 открыть файл %s\n",FILENAME);
                exit();
                }
        puts("Поиск диска\n");
        printf("Введите название диска: ");
        gets(name);
        while(fread(&disc, sizeof(disc), 1, fp)==1)
                {
                if(strcmp(disc.name,name)==0)
                        {
                        fflag='y';
                        showdisc();
                        printf("Для продолжения
                 нажмите Enter");
                        pause=getchar();
```

```
                        putchar('\n');
                        }
                }
        fclose(fp);
        fclose(tp);
        if(fflag=='n')
                nofind();
        return;
        }
plist()
        {
        if((fp = fopen(FILENAME,"r")) == NULL)
                {
                printf("Невозможно открыть
                 файл %s\n",FILENAME);
                exit();
                }
        if((printer = fopen("prn","w")) == NULL)
                {
                printf("Печатающее устройство
                 не готово к работе\n");
                fclose(fp);
                exit();
                }
        while(fread(&disc, sizeof(disc), 1, fp)==1)
                {
                fprintf(printer,"Название:
                  %s\n",disc.name);
                fprintf(printer,"Описание:
                  %s\n",disc.description);
                fprintf(printer,"Категория:
                 %s\n",disc.category);
                fprintf(printer,"Цена:
                     %6.2f\n",disc.cost);
                fprintf(printer,"Номер п/п:
                 %d\n",disc.number);
                fprintf(printer,"\n\n");
                                }
        fclose(printer);
        fclose(fp);
        return;
        }
sort()
```

```
        {
        struct CD temp[MAX];
        int index, loop1, loop2, endloop;
        loop1=0;
        loop2=0;
        endloop=0;
        index = 0;
        if((fp = fopen(FILENAME,"r")) == NULL)
                {
                printf("Невозможно открыть
                 файл %s\n",FILENAME);
                exit();
                }
        while(fread(&disc, sizeof(disc), 1, fp)==1)
                {
                temp[index]=disc;
                index++;
                }
        fclose(fp);
        if((fp = fopen(FILENAME,"w")) == NULL)
                {
                printf("Невозможно открыть
                 файл %s\n",FILENAME);
                exit();
                }
        for(loop1=1;loop1<<< || disc.number>>MAX || flag==1);
                count++;
        slots[count]=disc.number;
        return;
        }
getslots()
        {
        int index;
        index=0;
        count=0;
        if((fp = fopen(FILENAME,"r")) != NULL)
                {
                while(fread(&disc, sizeof(disc), 1, fp)==1)
                        {
                        slots[index]=disc.number;
                        index++;
                        count++;
                        }
```

```
        fclose(fp);
        }
}
```