МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

А.А. Мелещенко

Основы программирования на языке С

Учебное пособие

по курсу «Конструирование программ и языки программирования»
для студентов специальности 31 03 04 “Информатика”
дневной формы обучения

Минск 2004

УДК 004.434 (075.8)
ББК 32.973-018.1я73
М 47

Рецензент:

зав. кафедрой математики и информатики факультета информационных технологий Европейского гуманитирного университета
кандидат технических наук В.И. Романов

Мелещенко А.А.

М 47 Основы программирования на языке С: Учеб. пособие по курсу «Конструирование программ и языки программирования» для студ. спец. 31 03 04 «Информатика» дневной формы обуч. / А.А. Мелещенко. – Мн.: БГУИР, 2004. – 232 с.: ил.

ISBN 985-444-696-4

Пособие является учебным материалом по курсу КПиЯП (1 семестр обучения, специальность «Информатика»). Будет полезно студентам, изучающим язык программирования С. Материал пособия может использоваться в качестве теоретической части лабораторных работ.

УДК 004.434 (075.8)
ББК 32.973-018.1я73

ISBN 985-444-696-4

Содержание

Предисловие 8

1. Данные программы 9

Анатомия С-программы 9

Функция main() 10

Сообщения об ошибках 11

Заголовочные файлы 13

Комментарии 13

Отладка с комментариями 14

Переменные и типы переменных 15

Ключевые слова 17

Идентификаторы 17

Целые типы 18

Знаковые и беззнаковые значения 19

Множественные объявления переменных 20

Литеральные целые значения 21

Ключевое слово *const* 22

Типы данных с плавающей запятой 23

Точность 25

Символьные типы 26

Использование отдельных символов 26

Работа со строками 27

Размеры переменных 28

Символические константы 29

Определение собственных символических констант 30

Предопределенные символические константы 31

Перечисления 31

Преобразование типов 34

Использование операции приведения типа 35

Резюме 35

Обзор функций 37

Функция *printf()* (stdio.h) 37

Прототип и краткое описание 37

Спецификаторы преобразования 37

Задание ширины поля и точности представления 38

Флаги 38

Функция *scanf()* (stdio.h) 39

Прототип и краткое описание 39
2. Действия программы 40
Выражения 40
Операторы 40
Арифметические операторы 40
Операторы отношений 42
Логические операторы 42
Оператор отрицания 43
Операторы инкремента и декремента 43
Оператор присваивания 44
Множественное присваивание 45
Сокращенный оператор присваивания 45
Приоритеты операторов 46
Оператор *if* 47
Оператор *else* 49
Условные выражения 52
Оператор *switch* 52
Оператор *while* 55
Оператор *do-while* 57
Оператор *for* 58
Бесконечный цикл 60
Оператор *break* 60
Оператор *continue* 62
Оператор *goto* 62
Резюме 64
Обзор функций 65
3. Функции 68
Нисходящее программирование 68
Функции, которые возвращают пустоту 68
Локальные и глобальные переменные 71
Область видимости переменных 72
Область видимости локальных переменных 72
Область видимости глобальных переменных 74
Функции, которые возвращают значение 75
Целые функции 75
Функции с плавающей запятой 77
Другие типы функций 81
Распространенные ошибки в функциях 81
Параметры и аргументы функций 82

Безымянные параметры ... 87
Рекурсия: то что сворачивается, должно разворачиваться ... 88
Резюме ... 90
Обзор функций ... 91
4. Массивы ... 92
Введение в массивы ... 92
Инициализация массивов ... 93
Использование *sizeof* с массивами ... 99
Использование массивов констант ... 99
Символьные массивы ... 100
Многомерные массивы ... 100
Двумерные массивы ... 100
Трехмерные массивы ... 101
Инициализация многомерных массивов ... 102
Передача массивов функциям ... 105
Передача многомерных массивов функциям ... 106
Указатели ... 108
Введение в указатели ... 108
Объявление и разыменование указателей ... 109
Указатели в качестве псевдонимов ... 109
Нулевые указатели ... 110
Указатели типа void ... 111
Указатели и функции ... 112
Указатели и динамические переменные ... 113
Резервирование памяти в куче ... 114
Удаление памяти в куче ... 115
Указатели и массивы ... 115
Динамические массивы ... 117
Резюме ... 118
Обзор функций ... 120
5. Строки ... 121
Что такое строка ... 121
Строковые литералы ... 123
Строковые переменные ... 124
Строковые указатели ... 124
Нулевые строки и нулевые символы ... 125
Строковые функции ... 125
Отображение строк ... 126
Чтение строк ... 126

Преобразование строк в значения ... 127
Определение длины строк ... 128
Копирование строк ... 129
Дублирование строк ... 129
Сравнение строк ... 132
Конкатенация строк ... 134
Поиск элементов строк ... 136
Поиск символов ... 136
Поиск подстрок ... 138
Разложение строк на подстроки ... 139
Резюме ... 142
Обзор функций ... 143
6. Структуры ... 148
Сравнивание и присваивание структур ... 149
Инициализация структур ... 149
Использование вложенных структур ... 151
Структуры и функции ... 152
Структуры и массивы ... 154
Массивы структур ... 154
Структуры с членами, являющимися массивами ... 155
Динамические структуры данных ... 155
Самоссылочные структуры ... 156
Стеки ... 156
Очередь ... 161
Списки ... 164
Двунаправленные списки ... 168
Резюме ... 172
7.Сортировка и поиск данных ... 174
Методы и алгоритмы сортировки ... 174
Прямые сортировки ... 175
1. Сортировка методом обмена ... 175
2. Сортировка методом выбора ... 176
3. Сортировка методом вставки ... 177
Усовершенствованные методы сортировки ... 178
4. Сортировка методом Шелла ... 178
5. Быстрая сортировка ... 179
Выбор метода сортировки ... 185
Поиск данных ... 186
Линейный поиск ... 186

Бинарный поиск ... 187
Поиск строки в тексте ... 190
Прямой поиск строки ... 190
Алгоритм Боуера–Мура ... 191
Какой метод выбрать? ... 194
Резюме ... 194
Обзор функций ... 195
8. Файлы ... 197
Что такое файл? ... 197
Текстовые файлы ... 198
Чтение в посимвольном режиме ... 198
Чтение в построчном режиме ... 200
Посимвольная запись ... 201
Построчная запись ... 202
Функция *printf()* и родственные ей функции ... 202
Функция *scanf()* и родственные ей функции ... 204
Двоичные файлы ... 206
Обработка двоичных файлов ... 206
Файлы с последовательным доступом ... 207
Файлы с произвольным доступом ... 208
Программирование баз данных ... 211
Проектирование баз данных ... 211
Создание файла базы данных ... 216
Добавление записей в базы данных ... 217
Редактирование записей базы данных ... 218
Создание отчетов о содержимом базы данных ... 219
Резюме ... 220
Обзор функций ... 221
Приложение 1: Таблицы кодов ASCII ... 225
Приложение 2: Хороший стиль программирования ... 230
Приложение 3: Рекомендуемая литература ... 232

Предисловие

Научиться программировать – непростое дело; в какой-то момент программирование может показаться вам утомительным и «неподъемным». Но его освоение принесет вам ни с чем не сравнимое удовлетворение, а кроме того, возможность стабильного заработка во взрослой части вашей жизни.

Это пособие – для тех, кто *хочет* научиться программировать на С. Последовательно и подробно рассматриваются основные конструкции языка, функции, указатели и массивы, строки, структуры и файлы. Большое внимание уделяется алгоритмам сортировки и поиска данных, разработке баз данных средствами языка С.

Основная цель пособия – научить *основам* программирования на С, поэтому некоторые элементы языка, например, бинарные операции, объединения, классы переменных, макросы – не рассматриваются. При необходимости обратитесь к приложению 3 «Рекомендуемая литература» для их самостоятельного изучения.

Вряд ли вы всю жизнь будете программировать в одиночку – вы будете работать в коллективе программистов. Хорошо, если с самого начала вы привыкните писать программы в общепринятом стиле программирования, что облегчит их понимание другими людьми. Этому посвящено приложение 2 «Хороший стиль программирования».

Хотелось бы выразить благодарность студентам, внесшим существенный вклад в эту работу:

Болбас Татьяне и Мазохе Ольге, которые помогли перевести материалы пособия в электронную форму;

Прудникову Михаилу, который подготовил материал, вошедший в главы 6 и 7;

Михасеву Артему, который помог протестировать примеры программ, входящие в пособие.

Сотрудники кафедры информатики В.Л. Балащенко и С.И. Сиротко консультировали меня во время работы над рукописью. Нашли время прочесть рукопись перед ее сдачей в печать и внесли много ценных замечаний Карасик Е.А. и Зиссер Ю.А.

Самая большая благодарность – Тому Свану, американскому программисту, известному лектору, на работах которого основана большая часть материала данного пособия.

А. Мелещенко

1. Данные программы

Данные – это факты, которые известны компьютерной программе. Запомнив данные в программе, вы передаете эти знания, например, номер телефона или адрес вашего друга, компьютеру.

Как вы узнаете из этой главы, данные С-программы могут быть либо целыми числами, либо значениями с плавающей запятой, либо же символьными строками. Однако перед тем как начать исследование этих и других типов данных, давайте поближе познакомимся с составными частями типичной С-программы.

Анатомия С-программы

Лучший способ изучить язык программирования – это ввод, компилирование и запуск на выполнение листингов небольшого размера. Введите листинг 1.1 (исключая номера строк с двоеточиями слева), сохраните его под именем welcome.c, затем скомпилируйте и, наконец, выполните программу.

Листинг 1.1. welcome.c (пример простой программы на С)

```
1: #include <stdio.h>
2:
3: main()
4: {
5:   printf("Welcome to C Programming!\n");
6:   return 0;
7: }
```

Рассмотрим, из чего состоит программа welcome.c.

Строка 1 – это директива включения. Имя директивы (**#include** – «включить») указывает компилятору на чтение текста из заданного файла. Используемый в листинге 1.1 включаемый файл stdio.h называется заголовочным и обычно имеет расширение “.h”. Файл stdio.h содержит различные, необходимые для вашей программы стандартные объявления, связанные с вводом и выводом.

Строка 2 – пустая. Пустые строки также выполняют свою роль в программировании, – например, помогают выделить ключевые разделы программ. Компилятор «не обращает внимания» на пустые строки.

Открывающая и закрывающая фигурные скобки в строках 4 и 7 образуют блок – группу элементов, с которой компилятор будет обращаться как с единым целым. Блок также называется *составным оператором*. Блок

из листинга 1.1 содержит два оператора (строки 5 и 6). Операторы представляют собой действия, которые программа должна выполнить.

Первый оператор (строка 5) вызывает стандартную функцию форматированного вывода **printf()** (произносится *принт-эф*; функция объявлена в файле stdio.h), чтобы отобразить на экране строку заключенного в кавычки текста. Этот текст заканчивается управляющим символом '**\n**', который обозначает переход на новую строку.

Строка 6 содержит оператор возврата **return**. Он завершает выполнение программы и возвращает некоторое значение операционной системе. Обычно ненулевые значения свидетельствуют об ошибках, нулевые – об успешном завершении программы.

В этой короткой программе лишь одна строка 3 осталась без объяснения. В ней расположен один из самых важных элементов каждой С-программы – функция **main()**. Она заслуживает более пристального внимания.

Функция main()

С-программы выполняются последовательно. Они начинаются с одного оператора, выполняют ряд других и, если не ударит молния и не отключится питание, благополучно завершаются в запланированное время.

Все С-программы начинаются одинаково: первый оператор содержит функцию, называемую **main()**. (Вы узнаете больше о функциях и о том, как их писать, в главе 3). Именно с этой функции начинает выполняться любая скомпилированная С-программа, независимо от ее размера.

Листинг 1.2 представляет собой минимально полную С-программу. Несмотря на то что эта программа – «лилипут», она все-таки имеет функцию **main()**.

Листинг 1.2. smallest.c (минимально возможная С-программа)

```
1: main()
2: {
3:   return 0;
4: }
```

Функции начинаются с имени, за которым следует пара круглых скобок. За именем функции следует блок операторов (строки 2-4). Все, что находится внутри блока, принадлежит функции. В данном случае, у функции есть только один оператор **return** (строка 3), хотя их может быть и больше. Каждый оператор должен заканчиваться точкой с запятой (;), так что компилятор легко может найти их концы.

Выполнение этой суперпростой программы под неусыпным оком отладчика показывает, как программа начинается, выполняется и заканчивается (рис. 1.1).

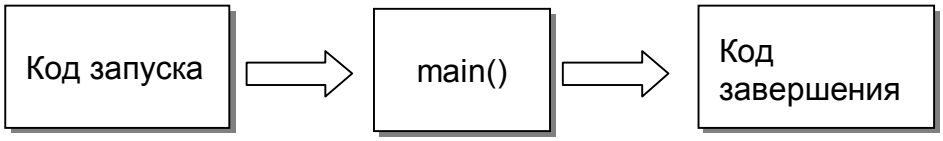

Рис. 1.1. Выполнение типичной С-программы

Введите листинг 1.2 и вместо обычной компиляции выполните следующие шаги.

1. Нажмите <F8> (если вы работаете в среде Borland C++) или <F10> (Visual C++) для компиляции и пошагового выполнения программы. По первому нажатию выделится оператор **main()**. Программа замирает на старте, подобно гонщику, напряженно ожидающему зеленого света.
2. Нажмите <F8> снова. Выделится оператор **return**. Вы только что выполнили невидимую секцию, называемую кодом запуска. Код запуска программы выполняет различные инициализации, которые обычно нас мало интересуют.
3. Нажмите <F8> еще раз, чтобы выполнить оператор **return**. Сейчас выделена последняя скобка и программа готова выполнить другую невидимую секцию, называемую кодом завершения. Подобно коду запуска, код завершения выполняет разнообразную вспомогательную работу, которую вы можете пока игнорировать.
4. Нажмите <F8> в последний раз, чтобы завершить программу.

Сообщения об ошибках

Не удивляйтесь, если при написании и вводе текста программы вы делаете много ошибок. Программирование требует совершенства. Пропустите хотя бы один, казалось, незначительный символ – и компилятор запищит от недовольства.

Существует две категории «жалоб» компилятора.

- *Errors (Ошибки)* означают серьезные ошибки, которые мешают компилятору закончить его работу.
- *Warnings (Предупреждения)* обращают ваше внимание на потенциальные проблемы, которые могут вызвать сбой в работе программы. Вы, конечно, можете запустить программу, для которой компилятор выдал предупреждения, но в этом случае результаты будут непредсказуемы. Если вы получили от компилятора предупреждение, отнеситесь к нему со вниманием и ликвидируйте его причину как можно скорее.

Поскольку в ходе деятельности на ниве программирования вы, несомненно, будете допускать бесчисленное количество ошибок, сейчас будет полезно специально сделать несколько ошибок, чтобы знать, как

исправлять их в дальнейшем. Скомпилируйте листинг 1.3, который содержит несколько грубых ошибок.

Листинг 1.3. bad.c (хорошая программа с большим количеством ошибок)

```
1: include <stdio.h)
2:
3: main{}
4: (
5:   printf("Problems...\n");
6:   printf("Problems, problems\n);
7:   printf("Problems all day long!\n")
8: )
```

Во время компиляции листинга компилятор сообщит вам, что программа bad.c имеет несколько ошибок:

```
Error bad.c 1: Declaration syntax error
Error bad.c 6: ) expected
Error bad.c 6: Unterminated string or character constant
```

Каждое сообщение об ошибке содержит имя файла (что важно для многофайловых программ), номер строки и короткое объяснение. Как видите, проблемы гнездятся в строках 1 и 6. Теперь исправьте ошибки в указанных строках. Замените строку 1 на

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

Строка 6 упоминается сразу в двух сообщениях. Первое предполагает, что пропущена круглая скобка. Это понятно. Следующее сообщение «Unterminated string or character constant» («Незаконченная строковая или символьная константа») говорит о том, что строка символов в операторе **printf()** осталась без заключающих кавычек. Замените строку 6 на

```
printf("Problems, problems\n");
```

и скомпилируйте измененную программу. И снова компилятор открывает свою «жалобную книгу»:

```
Error bad.c 3: Declaration syntax error
Error bad.c 6: ) expected
Error bad.c 7: ) expected
```

Теперь уже строка 3 попала в «черный список». Это обычное дело: вы исправляете одни ошибки и возникают другие. Постарайтесь исправить остальные ошибки самостоятельно (используйте листинг 1.1 в качестве образца). Листинг 1.4 демонстрирует правильную программу.

Листинг 1.4. good.c (исправленная версия bad.c)

```
1: #include <stdio.h>
2:
3: main()
4: {
```

```
5:    printf("Problems...\n");
6:    printf("Problems, problems\n");
7:    printf("Problems all day long!\n");
8:    return 0;
9: }
```

Заголовочные файлы

Такие директивы, как **#include <stdio.h>**, дают указание компилятору включить текст заданного файла в вашу программу. Просмотрите файл stdio.h (его легко найти с помощью средств поиска на вашем компьютере).

При обработке директивы **#include** компилятор ищет заданные файлы в соответствии со следующими правилами:

- если имя файла заключено в угловые скобки (**#include <stdio.h>**), компилятор осуществляет поиск в стандартных каталогах, заданных в настройках среды разработки программ;
- если имя файла заключено в кавычки (**#include "anyfile.h"**), компилятор ищет файл в текущем каталоге;
- если компилятор не может найти файл, имя которого заключено в кавычки, то он обращается с ним так, как будто его имя окружено угловыми скобками (другими словами, если файла "anyfile.h" нет в текущем каталоге, компилятор ищет его в стандартных каталогах);
- при указании имен включаемых файлов регистр букв не имеет никакого значения. Все имена MyFile.H, MYFILE.H и myfile.h относятся к одному и тому же файлу. Сама же директива **#include** вводится на нижнем регистре (строчными буквами).

Имена файлов могут включать и маршруты, как, например, в директиве `#include "c:\mydir\myfile.h"`. Имена каталогов отделяются обратной косой чертой.

Комментарии

Комментарии – это данные, предназначенные только для вас. Компилятор их полностью игнорирует. Цель комментариев – пояснить содержание программы.

Заключайте комментарии в двухсимвольные ограничители /* и */. Вот пример:

```
/* Моя первая программа */
```

Компилятор полностью игнорирует все символы в комментариях, включая ограничители. Комментарии могут стоять в конце оператора:

```
printf("Print this text"); /* отображение текста на экране */
```

Однако их нельзя ставить внутри текста, заключенного в кавычки, или внутри идентификаторов:

```
printf("Not /* комментарий */");     /* ??? */
int bad/* комментарий */Identifier; / *??? */
```

Мы будем использовать комментарии /* ??? */ для обозначения неправильных конструкций или сомнительных действий. Приведенные выше две строки никуда не годятся.

Вы можете писать комментарии, которые занимают одну или несколько строк, используя одну пару ограничителей. Следующие три строки текста представляют собой один комментарий:

```
/* Моя вторая С-программа
   Написана 10.09.2004
   Автор: имя */
```

Как и в случае однострочных комментариев, компилятор игнорирует весь текст, включая ограничители. Пользуясь этим, программисты часто пишут причудливые комментарии, аналогичные следующему:

```
/*
** Программа: Моя третья С-программа
** Дата: 12.09.2004
** Автор: имя
*/
```

Дополнительные звездочки здесь только для красоты оформления.

Отладка с комментариями

Комментарии часто бывают полезны для отладки отдельных фрагментов программы. Иногда, удалив один или два оператора и выполнив программу без них, можно выявить причину возникшей проблемы. Этот метод можно сравнить с ситуацией, когда воображаемый врач удаляет у пациента сердце и, убедившись, что тот остался больным, делает вывод, что с сердцем как раз было все в порядке (не следуйте этому примеру в жизни).

Предположим, вы хотите узнать, что случится, если в листинге 1.1 пренебречь оператором возврата. Вместо удаления строки, запретите ее выполнение с помощью комментария.

```
main()
{
  printf("Welcome to C Programming!\n");
/*
  return 0;
*/
}
```

Компилятор теперь проигнорирует этот оператор, который можно легко восстановить, убрав ограничители.

Замечание. *Пользуйтесь комментариями умеренно. Хорошие программы удобочитаемы сами по себе, и вам следует привыкать к чтению программного кода, а не только комментариев. Используйте комментарии, чтобы прояснить значение ключевых мест в ваших программах или для*

сообщения информации (как правило, в начале файла) о назначении программы, параметрах, авторе и т.п.

Переменные и типы переменных

С помощью переменных происходит запоминание информации в программе. Переменная представляет собой имя, связанное с областью памяти, отведенной для хранения значения переменной.

Перед тем как использовать переменные, программа должна объявить их. Это правило помогает компилятору быстро справляться со своими обязанностями и заставляет программистов тщательнее планировать свою работу. Листинг 1.5 демонстрирует, как нужно объявлять, инициализировать и использовать простую переменную, позволяющую хранить целые числа. В языке С целые переменные обозначаются сокращением **int**.

Листинг 1.5. integer.c (пример использования целой переменной)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   int value;
 6:
 7:   value = 1234;
 8:   printf("Value = %d\n", value);
 9:   return 0;
10: }
```

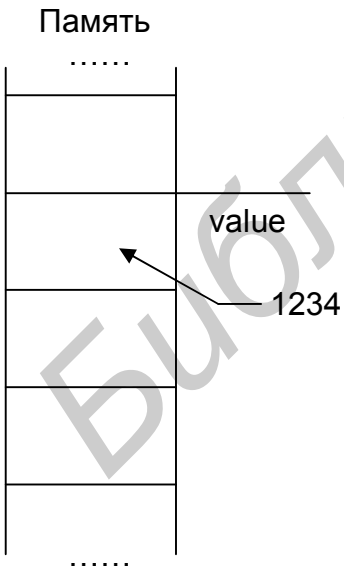

Рис. 1.2. Значение 1234 сохраняется в памяти

Строка 5 объявляет переменную **int** с именем **value**. Объявление переменной заканчивается точкой с запятой. Слово **int** является именем типа данных, оно встроено в язык С. Слово **value** является идентификатором, т.е. именем, которое вы придумали сами. По договоренности, пустая строка (строка 6) отделяет объявления переменных программы от ее операторов.

Строка 7 присваивает переменной **value** значение **1234**. При выполнении оператора присваивания программа сохраняет двоичное представление числа **1234** в области памяти, принадлежащей **value** (рис. 1.2).

Строка 8 отображает значение переменной **value** с помощью оператора

```
printf("Value = %d\n", value);
```

Рассмотрим оператор **printf()** подробнее. Внутри строки находится спецификатор **%d** –

специальная команда, которая «объясняет» функции **printf()**, где и в каком виде должна появиться переменная **value**. Функция **printf()** заменяет спецификатор значением переменной, записанной после строки, заключенной в кавычки. Символ, стоящий после знака процента, указывает, какого типа должна быть выведена переменная. В нашем случае – это **d**, значит, вывести нужно целую переменную.

Листинг 1.6. отображает значения переменных различных типов данных.

Листинг 1.6. variable.c (переменные различных типов)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   char slash = '/';
 6:   short month = 1;
 7:   int year = 2010;
 8:   long population = 10000000L;
 9:   float pi = 3.14159;
10:   double velocity = 299791336.2;
11:   long double lightYear = 9.5e15;
12:
13:   printf("Date = %02d%c%d\n", month, slash, year);
14:   printf("Population of the Belarus = %ld\n", population);
15:   printf("Pi = %f\n", pi);
16:   printf("Velocity of light = %13.2lf meter/sec\n",velocity);
17:   printf("One light year = %.0Lf meters\n", lightYear);
18:   return 0;
19: }
```

Далее в этой главе мы рассмотрим все типы данных, представленные в листинге 1.6 (строки 5-11).

Чтобы отобразить значения переменных, функция **printf()** использует множество спецификаторов (**%d**, **%c**, **%f** и т.д.), каждый из которых вставляет в отображаемые строки значение соответствующего типа. Раздел «Обзор функций» в конце этой главы содержит подробное описание этой могущественной функции.

Строка 7 демонстрирует удобный способ объявления и инициализации переменной за один прием. Вы также могли бы записать эту строку в виде двух строк:

```
int year;
...
year = 2010;
```

Конечный результат тот же, но перед тем, как число **2010** будет присвоено переменной **year**, значение переменной имеет непредсказуемое значение, равное комбинации нулей и единиц, оставленной в ее области

памяти от предыдущих операций. Во избежание проблем всегда инициализируйте свои переменные перед их использованием (компилятор предупредит вас, если вы забудете об этом).

Упражнение. *Сократите листинг 1.5, объявив и инициализировав переменную* ***value*** *за один прием.*

Ключевые слова

Ключевые слова, например **int** в листинге 1.5, являются встроенными символами языка. Вы не должны их использовать для своих собственных идентификаторов. Ключевые слова нужно вводить строчными буквами. В табл. 1.1 перечислены ключевые слова языка С.

Таблица 1.1

Ключевые слова ANSI C

`asm`	`default`	`for`	`short`	`Union`
`auto`	`do`	`goto`	`signed`	`Unsigned`
`break`	`double`	`if`	`sizeof`	`void`
`case`	`else`	`int`	`static`	`volatile`
`char`	`enum`	`long`	`struct`	`while`
`const`	`extern`	`register`	`switch`	
`continue`	`float`	`return`	`typedef`	

Идентификаторы

Слово **value**, встретившееся в листинге 1.5, называется идентификатором. Идентификаторы – это слова, которые вы придумываете для обозначения своих собственных переменных и функций.

Идентификаторы должны начинаться с буквы английского алфавита и могут содержать только английские буквы, цифры и символы подчеркивания. Они могут быть практически любой длины, но должны различаться в первых 32 символах, чтобы компилятор воспринимал их как разные идентификаторы (количество различаемых символов может быть задано с помощью настроек вашей среды разработки).

Замечание. *Идентификаторы также могут начинаться с одного или нескольких знаков подчеркивания, например,* `_myIdentifier` *или* `__systemVar__`*. Обычно идентификаторы такого вида зарезервированы для системного уровня. Если вы не будете начинать свои идентификаторы со знаков подчеркивания, у вас никогда не будет конфликтов с предопределенными системными идентификаторами.*

Язык С чувствителен к регистру букв, то есть имеет значение, в каком регистре – верхнем или нижнем – написаны буквы. **MyVar**, **myvar** и **MYVAR** – это совершенно разные идентификаторы. Обычно программисты, пишущие на С, начинают идентификаторы переменных и стандартных

функций со строчных букв. Свои собственные функции они пишут с прописной буквы, поэтому их легко отличить от библиотечных. Например, вы можете объявить несколько переменных с именами

```
value          balance        result         name
```

и вызывать функции с именами

```
printf()       MyFunc()       getchar()      GetUp()
```

Помните, что все это соглашения, а не правила. В своих идентификаторах вы можете использовать буквы любого регистра и в любом порядке. Но стандартные функции (такие, как **printf()** или **getchar()**) необходимо писать строчными буквами – точно так, как они объявлены. Данные соглашения помогут вам разрабатывать свои и понимать чужие программы.

Неправильно записанные идентификаторы являются причиной «жалоб» компилятора типа «Declaration terminated incorrectly». Подобное сообщение об ошибке могут вызвать следующие примеры «горе-идентификаторов»:

```
3rdValue    $balance    my-Var    Mistake!    /* ??? */
```

Идентификаторы никогда не должны начинаться с цифры и не должны содержать никаких знаков препинания, кроме символа подчеркивания.

Иногда программисты используют символы подчеркивания в идентификаторах, состоящих из нескольких слов. В этом случае значение идентификаторов становится очевидным, и они легко читаются:

```
Balance_of_power                speed_of_light
```

Существует другое популярное соглашение, называемое «*верблюжий горб*». Идентификатор, начинающийся со строчной буквы и имеющий прописную где-то в середине, действительно напоминает горб верблюда. С учетом этого соглашения два предыдущих идентификатора примут следующий вид:

```
balanceOfPower                  speedOfLight
```

В своих программах вы можете использовать любой из этих стилей.

Целые типы

Целые типы данных (такие как **int**) занимают фиксированное число ячеек памяти, что ограничивает диапазон значений, которые они могут представлять. Переменные типа **int** обычно занимают два байта и могут хранить значения в диапазоне от –32 768 до +32 767 (включая 0).

Если вам нужен больший диапазон целых чисел, можно объявить переменные типа **long int**, например:

```
long int bigValue;
```

Объявленная таким образом переменная обычно занимает четыре байта и может хранить значения от –2 147 483 648 до +2 147 483 647, включая 0. Вы

можете сократить запись **long int** до **long**:

```
long bigValue;
```

Также вы можете объявить значения типа **short int** как

```
short int smallValue;
```

и сократить их до

```
short smallValue;
```

Как правило, тип **short int** аналогичен типу **int** и представляет тот же диапазон значений, поэтому используется редко.

Замечание. *Точные диапазоны и размеры памяти для разных типов данных изменяются от компилятора к компилятору, особенно если компиляторы работают в различных операционных системах. ANSI C гарантирует, что значение* ***int*** *занимает не меньше памяти, чем* ***short int****, и что* ***long int*** *занимает не меньше памяти, чем* ***int****.*

Для очень маленьких значений используйте тип **char**. Несмотря на то что переменные **char** предназначены для представления символов, они могут также хранить значения, которые помещаются в одном байте, т.е. в диапазоне от –128 до +127 (включая 0), например:

```
char oneByte;
```

Знаковые и беззнаковые значения

Все целые типы – **char**, **short**, **int** и **long** – являются знаковыми по умолчанию. Другими словами, переменные этих типов могут иметь положительные и отрицательные значения. Вы можете использовать ключевое слово **signed,** чтобы явно указать на это:

```
signed int plusOrMinus;
```

Простой тип **int** означает то же самое, слово **signed** обычно не пишут.

Если вам не нужно хранить отрицательные значения, вы можете перед любым целым типом использовать ключевое слово **unsigned**. Вот несколько примеров беззнаковых переменных:

```
unsigned char oneChar;
unsigned short oneShort;
unsigned int oneInt;
unsigned long oneLong;
```

Поскольку в беззнаковых переменных отрицательные значения не запоминаются, они могут представлять вдвое больше положительных значений, чем их знаковые эквиваленты. В табл. 1.2 сведены знаковые и беззнаковые целые типы данных с указанием размера занимаемой памяти и диапазонов значений.

Загляните в стандартный библиотечный заголовочный файл limits.h, чтобы узнать точные диапазоны значений целых типов данных для вашего типа компилятора.

Таблица 1.2

Целые типы данных

Тип данных	Размер в байтах	Размер в битах	Минимальное значение	Максимальное значение
signed char	1	8	-128	127
unsigned char	1	8	0	255
signed short	2	16	-32768	32767
unsigned short	2	16	0	65535
signed int	2	16	-32768	32767
unsigned int	2	16	0	65535
signed long	4	32	-2147483647	2147483647
unsigned long	4	32	0	4294967295

Поскольку переменные целых типов занимают фиксированный объем памяти, при использовании в выражениях они, подобно автомобильному одометру, могут переполняться, т.е. доходить до максимума и затем сбрасываться в нуль. Например, если беззнаковая целая переменная (**unsigned int**) **i** равна 15000, то выражение

```
j = i + 60000;
```

присвоит беззнаковой целой переменной **j** значение 9464, а не 75000, которое лежит за границами представления значений типа **unsigned int**, т.е. результат «сворачивается», чтобы поместиться в меньшее пространство. Это не ошибка, а специальный эффект, позволяющий избежать затрат времени на проверку возможного переполнения. Однако рассчитывать на эффект «сворачивания» значения – это плохая практика. Ведь ваши программы могут работать и в других системах, где используется другой объем памяти для представления значений типа **int**.

Замечание. *При переполнении беззнаковые целые переменные обнуляются. Знаковые переменные обычно приводятся к максимальному в их диапазоне отрицательному числу.*

Упражнение. *Если **k** является переменной типа **int**, то чему будет равна **k** после выполнения выражения **k = 1000 * 2000**? А если **k** объявить как переменную типа **long**, то чему будет равно значение переменной в этом случае? (Постарайтесь угадать ответ, а затем напишите программу, чтобы проверить свои предположения.)*

Множественные объявления переменных

При объявлении нескольких переменных одного и того же типа вы можете записать их либо одну за другой:

```
int v1; int v2; int v3; int v4;
```

либо отдельными строками:

```
int v1;
```

```
int v2;
int v3;
int v4;
```

Гораздо удобнее вместо бесконечных повторений **int** ограничиться короткой записью:

```
int v1, v2, v3, v4;
```

Во всех трех случаях переменные **v1**, **v2**, **v3** и **v4** объявлены переменными типа **int**.

Литеральные целые значения

Литеральными являются значения, которые вы вводите непосредственно в текст программы. Поскольку после компиляции программы вы не можете изменить их значения, литералы также называются *константами.* Чтобы обработать такие константы, как **1234** и **–96**, компилятор использует типы данных с наименьшим возможным диапазоном значений.

Иногда необходимо заставить компилятор хранить константы как данные определенного типа. Например, чтобы хранить константу **1234** не как **int**, а как **long**, добавьте букву **L** после последней цифры:

```
long bigValue = 1324L;
```

Чтобы задать беззнаковое значение, добавьте **U**. Константа **1234U** запоминается как значение типа **unsigned int**. Константа **1234UL** будет храниться как значение типа **unsigned long int**.

Вы можете также задавать литеральные значения в шестнадцатеричном и восьмеричном форматах. Константы, которые начинаются с цифры 0, интерпретируются как восьмеричные числа (десятичные константы не должны начинаться с нуля). Константы, начинающиеся с символов 0x, являются шестнадцатеричными. Листинг 1.7 отображает шестнадцатеричные, восьмеричные и десятичные числа в различных форматах.

Листинг 1.7. hexoct.c (шестнадцатеричные и восьмеричные константы)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   int hexValue = 0xf9a;
 6:   int octalValue = 0724;
 7:   int decimalValue = 255;
 8:
 9:   printf("As decimal integers: \n");
10:   printf(" hexValue = %d\n", hexValue);
11:   printf(" octalValue = %d\n", octalValue);
12:   printf(" decimalValue = %d\n", decimalValue);
13:
14:   printf("\nAs formatted integers:\n");
15:   printf(" hexValue = %x\n", hexValue);
16:   printf(" octalValue = %o\n", octalValue);
```

```
17:   printf(" decimalValue = %#x\n", decimalValue);
18:   return 0;
19: }
```

Выполнив программу, приведенную в листинге 1.7, вы увидите на экране следующие строки:

```
As decimal integers:
 hexValue = 3994
 octalValue = 468
 decimalValue = 255

As formatted integers:
 hexValue = f9a
 octalValue = 724
 decimalValue = 0xff
```

Форматирование шестнадцатеричных и восьмеричных значений с помощью команд **%d** в операторах **printf()** отображает их эквивалентные десятичные значения (строки 10-12). Задавайте **%o** для получения восьмеричного формата (строка 16). Чтобы отобразить шестнадцатеричные значения, используйте формат **%x** (для вывода строчных букв от a до f, используемых в шестнадцатеричных числах) или **%X** (для вывода прописных букв от A до F). Значок # перед символами **x** или **X** позволяет выводить шестнадцатеричные числа в альтернативном формате. Например, строка 17 использует **%#x** для отображения десятичного значения 255 в виде шестнадцатеричного 0xff.

Ключевое слово *const*

Как вам уже известно, вы можете запоминать значения в целых переменных с помощью операторов типа

```
intValue = 1234;
```

Любой другой оператор присваивания может изменить значение **intValue**. Например, оператор

```
intValue = 4321;
```

присваивает **4321** переменной **intValue**, заменяя ее текущее значение. Если существует другая целая переменная **newValue**, то оператор

```
intValue = newValue;
```

делает значение переменной **intValue** равным значению переменной **newValue**.

Чтобы запретить операторам изменять значения переменных, предварите их объявления ключевым словом **const**. Например, попробуйте заменить строку 5 листинга 1.7 на

```
const int hexValue = 0xf9a;
```

и затем добавить оператор

```
hexValue = 0xa9b;
```

Компилятор выдаст сообщение об ошибке «Cannot modify a const

object» («Нельзя изменять константу»). Переменным, объявленным как константы, можно присваивать начальные значения, но другие операторы не могут изменять их – мера безопасности, гарантирующая, что эти значения останутся постоянными на протяжении всего времени работы программы.

Типы данных с плавающей запятой

Существуют три типа данных с плавающей запятой: **float**, **double** и **long double**. (Типа **long float** не существует). В табл. 1.3 представлены размеры памяти и диапазоны значений для типов данных с плавающей запятой.

Таблица 1.3

Типы данных с плавающей запятой

Тип данных	Размер в байтах	Размер в битах	Минимальное значение	Максимальное значение
float	4	32	-3.4E+38	3.4E+38
double	8	64	-1.7E+308	1.7E+308
long double	10	80	-3.4E+4932	1.1E+4932

Замечание. *Обратитесь к стандартному библиотечному заголовочному файлу float.h для уточнения диапазонов представления и других деталей, касающихся чисел с плавающей запятой.*

Инициализируйте значения с плавающей запятой, присваивая им литеральные константы в объявлении

```
double balance = 525.49;
```

либо используя отдельный оператор присваивания

```
balance = 99.99;
```

Константы с плавающей запятой, такие как **525.49** или **99.99**, имеют тип **double**, если за ними не следуют буквы **f** или **F**. Значение **3.14159F** имеет тип **float**. Используйте букву **l** или **L** для получения значения типа **long double**, например **3.14159L**.

Листинг 1.8 демонстрирует типичное применение переменных с плавающей запятой. Эта программа решает простую математическую задачу: вычисляет площадь и длину окружности по заданному радиусу.

Листинг 1.8. circle.c (вычисление площади и длины окружности)

```
1: #include <stdio.h>
2: const double PI = 3.14159;
3:
4: void main()
5: {
6:   double r, s, l;
7:   printf("\nEnter radius: ");
8:   scanf("%lf", &r);
```

```
 9:    s = PI * r * r;
10:    l = 2 * PI * r;
11:    printf("Square of circle = %lf, circuit = %lf", s, l);
12: }
```

В строке 2 объявляется глобальная константа **PI**, которая будет использоваться в вычислениях. Хороший прием – записывать константы прописными буквами, что явно указывает на то, что их значение не может быть изменено.

Строка 6 объявляет 3 переменных с плавающей запятой типа **double**. Оператор **printf()** на следующей строке предлагает ввести радиус. За ввод данных в программе отвечает функция **scanf()**, объявленная в файле stdio.h и по форме аналогичная функции **printf()**. Оператор

```
scanf("%lf", &r);
```

ожидает, пока вы не введете значение радиуса, после чего сохраняет введенное значение в переменной **r**. Оператор содержит один или несколько спецификаторов, заключенных в кавычки. В нашем случае это "**%lf**", т.к. мы хотим ввести одно значение типа **double**.

За строкой формата следует переменная, предназначенная для приема данных. Выражение **&r** передает адрес переменной **r** функции **scanf()**. Подробнее тема адресов и указателей будет раскрыта в главе 4, а пока просто запомните, что перед именами всех переменных (за исключением строк) в функции **scanf()** необходимо поставить знак взятия адреса (**&**).

Листинг 1.8 содержит еще один новый элемент – выражение. В строках 9 и 10 вычисляется площадь и длина окружности по формулам: $\mathbf{S = \pi r^2}$, $\mathbf{L = 2\pi r}$. Обратите внимание, что все переменные в программе circle имеют тип **double**. Это не случайно. Целые переменные плохо подходят для использования в арифметических выражениях. Особенно «опасны» для них выражения, в которых используются операции умножения и деления.

Так как целые переменные имеют относительно небольшой диапазон значений, умножение даже небольших чисел друг на друга может привести к переполнению и эффекту «сворачивания» целой переменной. Например, следующий «безобидный» оператор

```
int i = 170 * 200;
```

на самом деле содержит потенциальную ошибку, т.к. переменная типа **int** не может вместить значение 34 000.

При делении целых чисел дробная часть результата отбрасывается и, если вы не будете внимательны, то можете допустить ошибку, подобную следующей:

```
float f = 170 / 200;  /* f равно 0, а не 0.85 */
```

Чтобы не потерять значащую информацию, используйте в арифметических выражениях переменные и константы с плавающей запятой.

Точность

Использование значений с плавающей запятой можно сравнить с измерениями, выполняемыми с помощью линейки. В отличие от целочисленных вычислений, представляющих точные расчеты, точность вычислений значений с плавающей запятой зависит от размера переменной этого типа данных в памяти. Чем больший размер имеет переменная в памяти, тем ближе друг к другу расположены деления воображаемой линейки и тем с большей точностью происходят вычисления.

Значения типа **float** имеют около шести или семи значащих цифр (т.е. различаемых цифр после десятичной точки); значение типа **double** – 15 или 16, а **long double** – 19. Ошибка округления, вышедшая за пределы установленных значащих цифр, может привести к потере информации.

Концептуально существует бесконечное число значений между двумя вещественными числами. Между 0.0 и 1.0 находится значение 1.5; между 1.0 и 1.5 есть 1.25 и т.д. Однако, поскольку для представления значения с плавающей запятой используется ограниченное количество битов, некоторые значения не могут быть точно представлены в памяти. Значения с плавающей запятой в компьютере лишь *аппроксимируют* истинные вещественные числа в математике. Вы можете представить любое значение в пределах диапазона выбранного типа данных с плавающей запятой, но не можете представить каждое значение с абсолютной точностью.

Листинг 1.9 демонстрирует особенности выражений с плавающей запятой.

Листинг 1.9. precise.c (точность вычислений значений с плавающей запятой)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   double d1 = 4.0 / 2.0;
 6:   double d2 = 1.0 / 3.0;
 7:   double d3 = 2.0E40 * 2.0E30 + 1;
 8:
 9:   printf("d1 = %lf\n", d1);
10:   printf("d2 = %lf\n", d2);
11:   printf("d3 = %le\n", d3);
12:   return 0;
13: }
```

Строки 5-7 присваивают значения результатов трех выражений переменным типа **double**. Деление 4.0 на 2.0 дает результат 2.0, который может точно представлен. Однако при делении 1.0 на 3.0 получается нечто вроде 0.33333 – близко, но не точно. Десятичная цифра повторяется до бесконечности, и любое отсечение уменьшает общую точность. Третье выражение прибавляет 1 к произведению двух очень больших чисел.

Поскольку результат требует более 64 битов для его представления, полученное значение **d3** (**4E+70**) будет на 1 меньше, чем истинный ответ.

Пусть это не удивляет вас. Несколько простых советов помогут вам правильно использовать значения с плавающей запятой.

- Всюду, где только можно, объявляйте вместо 32-битовых переменных типа **float** 64-битовые переменные типа **double**. Тип данных **double** может безопасно представлять больше значащих цифр, чем **float**, и в большинстве случаев дает более точные результаты.
- Не используйте **long double**, если вы в самом деле не нуждаетесь в дополнительной точности, обеспечиваемой этим 80-битовым значением с плавающей запятой, которое требует больше памяти и времени на обработку. Для большинства приложений простые значения типа **double** позволяют достичь идеального компромисса между размером и точностью.
- Не используйте значения с плавающей запятой для хранения важных финансовых данных. Ошибки округления здесь неприемлемы. Для работы с финансовой и другой важной информацией существуют специальные классы чисел.

Символьные типы

Для работы с символами и строками предназначен символьный тип **char**. Переменные этого типа занимают в памяти компьютера один байт и этого как раз достаточно, чтобы хранить *номер* символа в стандартной для всех моделей PC таблице символов ASCII (см. прил. 1). Таким образом, в памяти компьютера все символы хранятся как числа, что облегчает работу с ними.

Использование отдельных символов

Итак, переменные типа **char** могут запоминать небольшие значения в диапазоне от –128 до +127. Например, объявление

```
char c = 'S';
```

инициализирует переменную **c** типа **char** ASCII-значением символа **S** (равным 83). Односимвольные константы заключаются в апострофы (одинарные кавычки).

Листинг 1.10 отображает символы в различных форматах.

Листинг 1.10. ascii.c (символы в десятичном и шестнадцатеричном форматах)

```
1: #include <stdio.h>
2:
3: main()
4: {
5:   char c;
6:
7:   printf("Enter character: ");
```

```
 8:    c = getchar();
 9:    printf("Character = %c\n", c);
10:    printf("ASCII (dec) = %d\n", c);
11:    printf("ASCII (hex) = %#x\n", c);
12:    return 0;
13: }
```

В строке 5 объявляется переменная типа **char**. В строке 8 вызывается описанная в файле stdio.h библиотечная функция **getchar()** для чтения символа с клавиатуры. Функция ожидает, пока вы не введете символ и не нажмете <Enter>, затем возвращает значение символа, присвоенное здесь переменной **c**. После этого три оператора **printf()** отображают значение переменной в символьном (**%c**), десятичном (**%d**) и шестнадцатеричном (**%#x**) форматах.

Поскольку литеральные символы хранятся в памяти как значения **int**, а не как данные типа **char**, что можно было бы предположить, вы можете заменить строку 5 на

```
int c;
```

после чего программа продолжает прекрасно работать. Действительно, многие библиотечные функции (и многие программы) разработаны в расчете на сохранение одиночных символов как значений **int** – это дань ранним версиям языка С.

Работа со строками

Последовательность символов образует строку. В этой главе вы уже видели много примеров строк. Строка в С хранится как последовательность значений типа **char**, которые заканчиваются нулевым символом ‘**\0**’. Нулевой символ позволяет функциям, работающим со строками, обнаруживать концы строк.

Самое простое, что можно сделать, чтобы создать строковую переменную, это добавить к объявлению **char** заключенную в квадратные скобки максимальную длину строки. Например, запись

```
char string[80];
```

объявляет переменную с именем **string**, которая может запоминать от 0 до 80 символов, включая завершающий нулевой символ.

Чтобы прочитать строку символов с клавиатуры, выполните оператор типа

```
scanf("%s", string);
```

Спецификатор **%s** указывает, что будет читаться строка. Операция взятия адреса перед именем строки в данном случае не нужна. Затем вы можете отобразить результат с помощью функции **printf()**:

```
printf("Вы ввели %s", string);
```

Мы остановимся подробнее на строках и строковых функциях в главе 5.

Упражнение. *Напишите программу, которая предлагала бы ввести имя, затем выводила бы сообщение «Привет, X», где X должен быть заменен на введенный текст.*

Размеры переменных

Компиляторы ANSI C могут отличаться друг от друга способом хранения данных различных типов. Вместо того чтобы строить предположения о количестве байтов, занимаемых конкретной переменной, воспользуйтесь услугами оператора **sizeof()**.

Листинг 1.11 является удобной тестовой программой, которая с помощью оператора **sizeof()** отображает размер всех основных типов данных компилятора ANSI C.

Листинг 1.11. sizeof.c (применение оператора **sizeof()**)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   char c;
 6:   short s;
 7:   int i;
 8:   long l;
 9:   float f;
10:   double d;
11:   long double ld;
12:
13:   printf("Size of char ..... = %2d byte(s)\n", sizeof(c));
14:   printf("Size of short ..... = %2d byte(s)\n", sizeof(s));
15:   printf("Size of int ..... = %2d byte(s)\n", sizeof(i));
16:   printf("Size of long ..... = %2d byte(s)\n", sizeof(l));
17:   printf("Size of float ..... = %2d byte(s)\n", sizeof(f));
18:   printf("Size of double .... = %2d byte(s)\n", sizeof(d));
19:   printf("Size of long double = %2d byte(s)\n",sizeof(ld));
20:   return 0;
21: }
```

Замечание. *Компиляция программы sizeof.c может вызвать многочисленные предупреждения о переменных, которые были объявлены, но не использовались («declared but never used»). Это предупреждение поможет вам избавиться от переменных, которые стали ненужными, и вы про них забыли. В данном случае можете не обращать на него внимания.*

Программа объявляет переменные семи типов (строки 5-11), а затем отображает размер элемента каждого типа в байтах (строки 13-19). Например, оператор

```
printf(..., sizeof(c));
```

отображает размер переменной **c** типа **char**. Компилятор преобразует выражение **sizeof(элемент)** в целое значение, равное размеру элемента в байтах. Этот элемент может быть именем типа данных вроде **int** или **float** либо именем любой вашей переменной.

Символические константы

Некоторые С-программы можно сравнить с криптограммами. Другие кажутся легкими для анализа и понимания. А разница между ними состоит не столько в сложности программы, сколько в том, какие идентификаторы, комментарии и символические константы использует программист.

Символическая константа представляет собой идентификатор, который что-нибудь означает. Например, в автоматах (большинство из которых представляет собой хитро замаскированные компьютеры) идентификатор **JACKPOT** может означать максимальную сумму выигрыша. Такой идентификатор обычно объявляется в начале программы с помощью директивы **#define** по следующему образцу:

```
#define JACKPOT 45000
```

Для удобства символические константы вводятся прописными буквами, чтобы их легко было различить в сложной программе. Вы должны понимать, что **JACKPOT** – это *не* переменная. Директива **#define** связывает некоторый символ (в данном случае **JACKPOT**) с заданным текстом (например, **45000**). Другими словами, везде, где в тексте программы встречается слово **JACKPOT**, при запуске программы препроцессор заменит его на текст **45000**.

Вы можете использовать символические константы в любом месте, где связанные с ними литералы имеют смысл. Например, если в программе объявлена символическая константа

```
#define NAME "Jack Pot"
```

то компилятор будет интерпретировать оператор

```
printf(NAME);
```

так, как если бы вы написали

```
printf("Jack Pot");
```

Очень часто совершают ошибку, заканчивая определение символической константы точкой с запятой. Следующая запись неверна:

```
#define JACKPOT 45000;   /* ??? */
```

Когда вы попытаетесь использовать этот неудачный символ, например в операторе

```
printf("JackPot = $%d\n", JACKPOT);   /* ??? */
```

компилятор попытается скомпилировать эту строку как

```
printf("Jack Pot = $%d\n", 45000;);   /* ??? */
```

что вызовет ошибку неверного синтаксиса.

Определение собственных символических констант

Определение символических констант в начале программы или в заголовочном файле создает набор легко модифицируемых символов. Листинг 1.12 демонстрирует типичные способы использования директивы **#define**.

Листинг 1.12. define.c (использование символических констант)

```
 1: #include <stdio.h>
 2:
 3: #define CHARACTER '@'
 4: #define STRING "Mastering ANSI C\n"
 5: #define OCTAL 0233
 6: #define HEXADECIMAL 0x9b
 7: #define DECIMAL 155
 8: #define FLOATING_POINT 3.14159
 9:
10: main()
11: {
12:   printf("CHARACTER = %c\n", CHARACTER);
13:   printf("STRING = " STRING);
14:   printf("OCTAL = %o\n", OCTAL);
15:   printf("HEXADECIMAL = %#x\n", HEXADECIMAL);
16:   printf("DECIMAL = %d\n", DECIMAL);
17:   printf("FLOATING_POINT = %f\n", FLOATING_POINT);
18:   return 0;
19: }
```

Если у вас большая программа, то сохранение многочисленных символических констант в одном месте облегчит изменение параметров программы и вам не придется делать изменения во всех операторах, которые их используют.

Рассмотрите внимательно операторы **printf()** в строках 12-17. Строка 13 не является ошибочной. При обработке строки

```
printf("STRING = ", STRING);
```

константа **STRING** заменяется на соответствующий текст:

```
printf("STRING = " "Mastering ANSI C\n");
```

При компиляции программы эти смежные строки соединяются в одну длинную строку. Таким образом, этот оператор отобразит одну строку:

```
STRING = Mastering ANSI C
```

В качестве полезного упражнения перепишите каждый из операторов **printf()** (строки 12-17), заменив вручную символические константы соответствующими им значениями. Это поможет вам разобраться, как обрабатываются операторы подобного рода.

Предопределенные символические константы

Кроме ваших собственных, вы можете использовать предопределенные символические константы из различных заголовочных файлов. Например, в заголовочном файле math.h есть несколько очень полезных констант:

```
#define M_E             2.71828182845904523536
#define M_LOG2E         1.44269504088896340736
#define M_PI            3.14159265358979323846
#define M_PI_2          1.57079632679489661923
#define M_PI_4          0.78539816339744830961 6
#define M_SQRT2         1.41421356237309504880
```

Если вы включите в программу директиву **#include <math.h>**, то вам станут доступны эти и другие математические константы. Например, если вам потребуется использовать значение π в выражении, вы можете записать:

```
result = M_PI * value;
```

Это поможет вам избежать возможных опечаток при самостоятельном вводе числа π.

Упражнение. *Модифицируйте листинг 1.8, используя в выражениях более точное значение числа π, представленное константой из файла math.h.*

Замечание. *Директива* ***#define*** *позволяет создавать текстовые макроопределения (или макросы), представляющие собой вид мини-языка, встроенного во все компиляторы ANSI C. Некоторые библиотечные функции низкого уровня определяются как макросы, чтобы избежать многочисленных вызовов подпрограмм для таких простых операций, как, например, чтение одного символа с клавиатуры. Может быть, в свое время и вы захотите узнать, как строить свои собственные макросы, но сейчас благоразумнее отложить эту задачу до тех пор, пока вы не почувствуете себя увереннее. Многие опытные программисты за свою жизнь не написали ни одного макроса, и вы ничего не потеряете, если присоединитесь к их числу.*

Перечисления

Предположим, вашей программе необходимо работать с цветами радуги. Вы могли бы представить их в виде символических констант следующим образом:

```
#define RED    0
#define ORANGE 1
#define YELLOW 2
#define GREEN  3
#define BLUE   4
#define INDIGO 5
#define VIOLET 6
```

Это удобный способ представления названия цветов в виде уникальных целых значений. После определения этих символических констант программа может объявить, например, целую переменную

```
int color;
```
и затем присвоить ей любой из «цветов» вашей «радуги»:
```
color = GREEN;
```

Компилятор заменит символическую константу **GREEN** связанным с ней текстом, в данном случае цифрой **3**. В результате будет скомпилирован оператор
```
color = 3;
```

Но, как правило, символ **GREEN** воспринимается лучше литерального значения **3**.

Ввод длинной последовательности названий цветов и других относительно устойчивых имен (например названий месяцев) – это довольно нудная работа. Кроме того, подобное определение символических констант никак не отражает их родства. Чтобы облегчить процесс создания подобных списков, ANSI C имеет полезное средство, называемое перечислимым типом. Используя ключевое слово **enum**, вы можете создать семь упомянутых выше констант цвета следующим образом:
```
enum {RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET};
```

За ключевым словом **enum** следует заключенный в фигурные скобки список идентификаторов (обычно прописными буквами), разделенных запятыми. Список завершает обязательная точка с запятой. Каждому символу последовательно присваивается целое значение, начиная с нуля. **RED** получает значение 0, **ORANGE** – 1, **YELLOW** – 2 и т.д.

После сделанного с помощью **enum** объявления можно объявить переменную с именем **color** и присвоить ей значение **BLUE**:
```
int color = BLUE;
```

Перечислимые типы данных имеют внутреннее представление с минимально возможным объемом памяти – в знаковых либо беззнаковых типах **char** или **int** – и, следовательно, всегда могут быть присвоены целым переменным.

С помощью другого ключевого слова – **typedef** (сокращение от «Type definition» – «Определение типа») – можно определить новое, более информативное имя типа данных. Используя **typedef**, более формализованное перечисление цветов радуги можно записать следующим образом:
```
typedef enum
{
  RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
} Colors;
```

Вопреки распространенному мифу (и звучанию ключевого слова – оно как раз и вводит в заблуждение), **typedef** не определяет новый тип данных, а лишь создает для него *псевдоним* (**Colors**). Псевдоним обычно начинается с большой буквы, хотя это не является жестким правилом.

После определения псевдонима вы можете объявить переменную как

```
Colors oneColor;
```

Теперь **oneColor** является переменной типа **Colors**, и ей можно присваивать символические константы цветов с помощью оператора, аналогичного следующему:

```
oneColor = INDIGO;
```

Но так как **Colors** является перечислимым типом, вы также можете присваивать переменным этого типа целые значения:

```
oneColor = 5;
```

Как вы уже знаете, компилятор автоматически нумерует элементы в объявлении **enum**. Но иногда нужно изменить эту нумерацию и присвоить заданные значения одному или нескольким символам. Например, рассмотрим проблему создания перечислимого типа для названий месяцев. Вы могли бы сделать следующее объявление:

```
enum {JAN, FEB, MAR, APR, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

В этом объявлении нет ошибки. Но так как первому символу перечислимого списка присваивается значение 0, элемент **JAN** получает значение 0, а не 1, которое принято связывать с первым месяцем года.

Эту проблему можно решить, явно присвоив значение любому элементу перечислимого списка. Например, чтобы установить **JAN** равным 1, вы можете заменить предыдущее объявление на

```
enum {JAN = 1, FEB, MAR, APR, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

Выражение **JAN = 1** присваивает единицу символу **JAN**. Следующие символы продолжают эту последовательность с этого места, присваивая **FEB** значение 2, **MAR** – 3, **APR** – 4 и т.д. Вы можете поступить аналогичным образом с любым элементом перечислимого списка. Можно даже пронумеровать месяцы десятками:

```
enum {JAN = 10, FEB = 20, ..., DEC = 120};
```

но такое насилие над элементами списка вряд ли когда-либо потребуется. Лучше всего предоставить работу по присваиванию значений компилятору, а самим пользоваться преимуществом легко узнаваемых имен символических констант.

Замечание. *Использование перечислимых типов делает более наглядным текст вашей программы, но не текст, который программа отображает на экране. Вы не сможете отобразить на экране символы **JAN**, **FEB** или **INDIGO** только благодаря использованию перечислимых типов. Хранясь в памяти как целые типы данных, на экране перечисления будут отображаться так же, как значения **int** или **char**.*

Упражнение. *Создайте перечислимый тип данных для дней недели.*

Преобразование типов

Хотя значения **int** и **long** являются целыми, их не назовешь лепестками одного и того же цветка. Нам осталось разобраться в том, что случится, если в выражении смешаются значения различных типов данных.

Не забывайте, что все переменные занимают в памяти фиксированное количество байтов. Если проигнорировать этот факт, то программа может преподнести неприятный сюрприз. Рассмотрим следующие объявления:

```
long aLong = 12345678;
int anInt;
```

Что же произойдет, если использовать следующий оператор присваивания:

```
anInt = aLong; /* ??? */
```

Поскольку переменная **anInt** может запоминать значения только в диапазоне от –32 768 до +32 767 (табл. 1.2), она не сможет вместить значение переменной **aLong**. Несмотря на очевидную ошибку, компилятор позволит свершиться этой «несправедливости», присвоив **anInt** значение 24 910 – совсем не то, что вы ожидали.

Оказывается, число 24 910 – это остаток от деления числа 12 345 678 на число 65 536, последнее является наибольшим беззнаковым значением, которое может содержаться в двух байтах. Это отнюдь не случайно. Значение **long**, сохраняемое в переменной **aLong**, усекается, чтобы поместиться в меньшей области памяти, отведенной для переменной **anInt**.

Присваивание значений больших переменных меньшим может вызвать аналогичную потерю информации, и компилятор может отреагировать предупреждающим сообщением, а может и «закрыть на это глаза». Присваивание меньших значений большим обычно не вызывает никаких неприятностей. Например, если значение переменной **anInt** равно 4321, то оператор

```
aLong = anInt;
```

присвоит переменной **aLong** то же самое значение 4321, которое великолепно вписывается в диапазон типа **long int**.

В приведенном примере компилятор выполнил расширение двухбайтового значения **int** до четырехбайтового значения **long**. Расширения также встречаются в выражениях, включающих значения с плавающей запятой. Если **fp** является переменной типа **float**, а **db** – типа **double**, то оператор

```
db = fp;
```

расширит **fp** до **db** без потери информации. Однако обратное присваивание

```
fp = db;
```

может вызвать проблемы, так как значение типа **float** не в состоянии охватить тот же диапазон, что и **double**.

Встречается и другой вид расширения, когда целым присваиваются значения с плавающей запятой. Такое присваивание разрешается, но, конечно, теряется дробная часть. Обратная операция – присваивание целых переменным с плавающей запятой – также разрешена.

Смешение целых значений и значений с плавающей запятой вызывает преобразование типов, результаты которого не всегда легко предсказать. Рассмотрим следующие объявления:

```
int i = 4;
double d = 2.8;
```

Затем выполним оператор

```
i = d * i;
```

Чему будет равно **i**? В смешанных выражениях значения с меньшим диапазоном расширяются до значений с большим диапазоном, и, чтобы обработать выражение **d * i**, компилятор сначала преобразует целую переменную **i** (равную 4) в значение типа **double**, умножит это значение на **d** (**4.0 * 2.8 = 11.2**), а затем присвоит результат переменной **i**. На последней стадии значение с плавающей запятой 11.2 сужается до типа **int**, присваивая в итоге переменной **i** значение 11.

Использование операции приведения типа

В наших силах отменять или, по крайней мере, более гибко управлять действующими по умолчанию правилами компилятора, касающимися расширения типов данных. Возьмем объявления из предыдущего примера:

```
int i = 4;
double d = 2.8;
```

Если вы хотите умножить неокругленный целый эквивалент значения переменной **d** на **i** (т.е. в итоге получить 8 вместо 11), используйте операцию приведения типов:

```
i = (int)d * i;
```

Выражение **(int)d** заставляет компилятор преобразовать переменную **d** в значение типа **int** (при этом дробная часть числа 2.8 отбрасывается и получается целое значение 2). После этого 2 умножается на значение переменной **i** (равное 4), в результате чего получается 8.

Совет. *Проявляйте осторожность при использовании смешанных выражений в ваших программах. Чтобы избежать потери информации, не присваивайте значения больших переменных меньшим. Также не рекомендуется смешивать в выражениях знаковые и беззнаковые переменные.*

Резюме

- Все С-программы должны иметь функцию **main()**. С этой функции начинается выполнение программы.

- Сообщения об ошибках (Errors) указывают на серьезные промахи, которые не позволяют компилятору завершить работу. Предупреждения (Warnings) свидетельствуют о менее серьезных ошибках и разрешают программам работать, но их следует исправить как можно скорее.
- Включение в программу заголовочных файлов с помощью директивы **#include** позволяет вам получить доступ к объявлениям стандартных (или написанных вами) функций, константам и другим элементам.
- Комментарии заключаются в ограничительные символы /* и */. Компилятор игнорирует эти ограничители и весь текст между ними.
- Ключевые слова являются зарезервированными символами языка. Идентификаторы – это слова, которые вы придумываете сами. Идентификаторы должны начинаться с буквы или символа подчеркивания и могут включать только буквы английского алфавита, цифры и символы подчеркивания.
- Язык С чувствителен к регистру букв. Идентификаторы **myVar** и **MYVAR** рассматриваются как разные слова.
- Целыми типами данных являются **char**, **short**, **int** и **long**, которые считаются знаковыми по умолчанию. Этим типам могут предшествовать ключевые слова **signed** или **unsigned**. Само по себе слово **unsigned** является синонимом типу **unsigned int**.
- Типами данных с плавающей запятой являются **float**, **double** и **long double**. Используя их, вы не можете представить вещественные числа с абсолютной точностью. Для большинства приложений тип **double** является удачным компромиссом между размером и точностью.
- Литеральные символы записываются подобно ‘**a**’. Литеральные строки записываются аналогично “**abcdef**”.
- Используйте оператор **sizeof** для определения объема памяти, занимаемого типами данных и переменными.
- Используйте директивы **#define** для определения символических констант.
- Используйте ключевое слово **enum** для определения перечисляемых символических констант.
- Будьте осторожны в использовании выражений, в которых смешиваются типы данных.

Обзор функций

Функция *printf()* (stdio.h)

Прототип и краткое описание

```
int printf(const char *format, ...);
```

Функция записывает форматированную строку в стандартный поток вывода и возвращает количество выводимых на печать байтов информации. Если произошла ошибка вывода, **printf()** возвращает значение **EOF**.

Спецификаторы преобразования

С помощью *спецификаторов преобразования* функция **printf()** может выводить на печать значения различных типов данных. В табл. 1.4 представлены наиболее употребительные спецификаторы.

Таблица 1.4

Спецификаторы преобразования функции printf()

Спецификатор	Тип данных	Описание
%d	int	Десятичное целое число со знаком
%u	unsigned int	Десятичное целое число без знака
%ld	long int	Длинное десятичное целое число со знаком
%lu	unsigned long int	Длинное десятичное целое число без знака
%o	целые типы	Восьмеричное целое число без знака
%x или %X	целые типы	Шестнадцатеричное целое число без знака. Спецификатор **X** используется для вывода числа цифрами 0-9 и буквами A-F, а **x** – для вывода числа цифрами 0-9 и буквами a-f
%f или %e	float	Число с плавающей точкой, десятичное или экспоненциальное представление
%lf или %le	double	Число с плавающей точкой двойной точности, десятичное или экспоненциальное представление
%Lf или %Le	long double	Число **long double**, десятичное или экспоненциальное представление
%c	char	Одиночный символ
%s	char *	Строка символов
%%		Печать знака процента

Задание ширины поля и точности представления

Точный размер поля, в котором печатаются данные, задается *шириной поля*. Если ширина поля больше, чем необходимо для печати данных, то данные обычно выравниваются внутри поля по его правому краю. Целое число, задающее ширину поля, может быть вставлено между знаком процента (**%**) и спецификатором преобразования.

Пример: `"%4d"` – печать целого значения в поле шириной 4.

Функция **printf()** дает также возможность задать *точность представления*, с которой будут напечатаны данные. Для этого поместите между знаком процента и спецификатором преобразования десятичную точку (.) с последующим числом.

Пример: `"%.4f"`

Точность имеет различный смысл для различных типов данных. Если она используется при выводе чисел с плавающей запятой, то точность – это количество цифр, которые будут напечатаны после десятичной точки.

Пример: `"%5.2Lf"` – вывод значения типа **long double** в поле шириной 5 символов с двумя цифрами после десятичной точки.

Если точность используется со спецификаторами преобразования целых чисел, то она обозначает минимальное количество цифр, которое должно быть выведено. Если выводимое значение содержит меньше цифр, чем задано точностью, то перед ним будут напечатаны префиксные нули, так, чтобы общее количество цифр стало равно заданной точности. Для спецификатора **%s** точность – это максимальное число символов строки, которое будет напечатано.

Флаги

Функция **printf()** предусматривает использование *флагов*, дополняющих возможности форматирования. Чтобы использовать в спецификации флаг, поместите его непосредственно справа от знака процента. В одной спецификации могут быть объединены несколько флагов.

Таблица 1.5

Флаги функции printf()

Флаг	**Описание**
1	2
–	Выравнивание выводимой строки по левому краю в пределах заданной ширины поля. *Пример*: `"%-20s"`
+	Значения со знаком печатаются со знаком «плюс», если они положительны, и со знаком «минус», если они отрицательны. *Пример*: `"%+6.2f"`

Окончание табл. 1.5

1	2
Пробел	Значения со знаком «плюс» печатаются с пробелом (но без знака), если они положительны, и со знаком «минус», если они отрицательны. *Пример*: `"% 6.2f"`
#	Альтернативный формат. Выводит в начале 0 для спецификатора **%o** и 0x или 0X для спецификаторов **%x** и **%X**. *Пример:* `"%#o"`
0	Дополняет поле до заданной ширины нулями слева. *Пример*: `"%04d"`

Функция *scanf()* (stdio.h)

Прототип и краткое описание

```
int scanf(const char *format, ...);
```

Считывает форматированный ввод со стандартного потока ввода (как правило, с клавиатуры); ввод завершается на первом встретившемся символе пробела, табуляции или при нажатии <Enter>. Для ввода значений с помощью **scanf()** вы можете применять те же спецификаторы преобразования, что и в **printf()** (табл. 1.4).

2. Действия программы

Компьютерные программы, написанные на С, могут сортировать базы данных, делать вычисления по заданным формулам, распечатывать таблицы и многое другое. Разве можно представить себе кино с неподвижными и немыми актерами? Точно так же программа без действий – это уже не программа, а нечто застывшее и скучное. Как действия в кино оживляют неподвижные картинки, так и программы обрабатывают данные, вычисляя выражения и выполняя операторы.

Выражения

Выражения – это операции, которые выполняют программы. Выражения бывают разные – от очень простых до невероятно сложных, но все они имеют свои значения. Например, переменная **A** типа **int** является простейшим выражением, которое представляет значение переменной **A**. Выражение **(A + B)** равно значению суммы слагаемых **A** и **B**. Обычно результат выражения присваивается какой-нибудь переменной с помощью оператора присваивания:

```
C = A + B;
```

Значение **C** теперь равно сумме **A** и **B**.

Операторы

Язык С включает несколько операторов, без которых не обходится практически ни одна программа. В следующих разделах вы познакомитесь с арифметическими и логическими операторами, а также операторами отношений, отрицания, инкремента и декремента.

Арифметические операторы

Еще сидя за школьной партой, вы использовали арифметические операторы +, –, * и / (табл. 2.1). В языке С эти операторы работают аналогичным образом. Оператор **%** используется для вычисления остатка от деления целых чисел. Например, значение выражения **24 % 11** (читается «24 по модулю 11») равно 2, т.е. остатку от целочисленного деления **24 / 11**. Значение выражения **8 % 2** равно 0, так как 8 делится на 2 без остатка.

Таблица 2.1

Арифметические операторы

Оператор	Описание	Пример
*	Умножение	a * b
/	Деление	a / b
+	Сложение	a + b
-	Вычитание	a – b
%	Деление по модулю	a % b

Листинг 2.1 содержит пример выражений, использующих арифметические операторы. В примере выполняется преобразование значений градусов по Фаренгейту в градусы по Цельсию.

Листинг 2.1. celsius.c (преобразование градусов)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3:
 4: main()
 5: {
 6:   double fdegrees, cdegrees;
 7:
 8:   printf("Fahrenheit to Celsius conversion\n\n");
 9:   printf("Degrees Fahrenheit? ");
10:   scanf("%lf", &fdegrees);
11:   cdegrees = ((fdegrees - 32.0) * 5.0) / 9.0;
12:   printf("Degrees Celsius = %lf\n", cdegrees);
13:   return 0;
14: }
```

Программа использует две переменные с плавающей запятой типа **double** – **fdegrees** и **cdegrees**. После ввода количества градусов по Фаренгейту с помощью функции **scanf()** выражение

```
((fdegrees - 32.0) * 5.0) / 9.0
```

вычисляет эквивалентную температуру в градусах по Цельсию. При этом наличие круглых скобок в выражении влияет на порядок выполнения операций: прежде всех выполняются операции, заключенные во «внутренние» круглые скобки. В этом примере выражение вычисляется в такой последовательности:

1) **32.0** вычитается из **fdegrees**;
2) результат первого шага умножается на **5.0**;
3) результат второго шага делится на **9.0**.

Совет. *Всегда используйте столько круглых скобок, сколько нужно для того, чтобы сделать ясной структуру выражения. Круглые скобки совершенно не*

влияют на быстродействие, с их помощью вы просто задаете желаемый порядок вычисления.

Операторы отношений

Операторы отношений обрабатывают свои операнды таким образом, чтобы получить либо истинный (ненулевой), либо ложный (нулевой) результат. В языке С ложный результат соответствует нулю, а истинный – ненулевому значению. Выражение

```
(A < B)
```

истинно только в том случае, если **A** меньше **B**. Переменные **A** и **B** должны принадлежать к таким типам данных, которые можно сравнивать. Обычно это целые или значения с плавающей запятой. (Для сравнения строк операторы отношений использовать нельзя; в главе 5 объясняется, как сравнивать строки с учетом порядка букв.)

В табл. 2.2 перечислены все операторы отношений языка С.

Таблица 2.2

Операторы отношений

Оператор	Описание	Пример
<	Меньше	a < b
<=	Меньше или равно	a <= b
>	Больше	a > b
>=	Больше или равно	a >= b
==	Равно	a == b
!=	Не равно	a != b

Замечание. *Оператор равенства в С представлен двойным знаком равенства, таким образом, выражение* ***(A == B)*** *будет истинно, только если* ***A*** *равно* ***B****. Не путайте такие выражения с операторами типа* ***A = B****, которые присваивают значение* ***B*** *переменной* ***A****. Для операторов присваивания всегда используйте одиночный знак равенства (=), а для сравнения – двойной (==).*

Логические операторы

Логические операторы **&&** и || объединяют выражения отношений в соответствии с правилами логического И и ИЛИ. Используйте логический оператор И (**&&**) в сложных выражениях отношений, аналогичных следующему:

```
(A < B) && (B < C)
```

которое будет истинным, только если **A** меньше **B** и **B** меньше **C**.

Таким же образом используйте оператор ИЛИ (||). Выражение

```
(A < B) || (B < C)
```

будет истинным, только если **A** меньше **B** или **B** меньше **C**.

Сложные логические выражения вычисляются рациональным способом. Например, если при вычислении выражения

```
(A <= B) && (B == C)
```

оказалось, что **A** больше **B**, то все выражение примет значение «ложь», и вторая часть **(B == C)** вычисляться не будет. Такая сокращенная обработка сложных логических выражений способствует быстрому выполнению программы.

Оператор отрицания

Вы можете инвертировать результат любого логического выражения с помощью унарного (т.е. имеющего только один операнд) оператора отрицания '**!**'. Выражение

```
!(A < B)
```

(«неверно, что **A** меньше **B**») эквивалентно выражению

```
(A >= B).
```

Оператор неравенства **!=** связан с оператором отрицания. Выражение

```
(A != B)
```

(«**A** не равно **B**») эквивалентно выражению

```
!(A == B)
```

(«не верно, что **A** равно **B**»).

Операторы инкремента и декремента

Двумя самыми интригующими операторами языка C являются ++ (инкремент) и **--** (декремент). Оператор ++ («плюс плюс») прибавляет к операнду единицу. Оператор **--** («минус минус») вычитает единицу.

Несколько примеров проясняют, как работают эти операторы. Следующих две строки дают идентичные результаты:

```
i = i + 1;  /* прибавление единицы */
i++;        /* то же самое */
```

Следующие два выражения также дают одинаковые результаты:

```
i = i - 1; /* вычитание единицы из i */
i--;       /* то же самое */
```

В работе с операторами инкремента и декремента есть одна существенная деталь. Значения выражений, подобных **i++** или **i--**, зависят от *расположения* операторов ++ и --. Если оператор следует за своим операндом, то значение выражения **i++** или **i--** равно значению **i** до модификации. Другими словами, оператор

```
j = i++;
```

присваивает переменной **j** первоначальное значение переменной **i**. Например, если значение **i** равно 7, то после выполнения оператора инкремента значение **i** будет равно 8, а **j** получит значение 7.

Если же операторы инкремента и декремента предшествуют своим операндам, значение выражения будет равно значению операнда после модификации, т.е. оператор

```
j = ++i;
```

присваивает переменной **j** инкрементированное значение **i**. Если значение **i** равно 7, то после выполнения этого оператора обе переменные **j** и **i** станут равными 8.

Чтобы лучше понять эти различия, можно обратиться к длинным формам записи выражений инкремента и декремента. Действие оператора

```
j = i++;
```

аналогично действию двух следующих операторов:

```
j = i;
i++;
```

А оператор

```
j = ++i;
```

действует подобно такой паре операторов:

```
++i;
j = i;
```

Если вы просто хотите прибавить или отнять от переменной единицу:

```
i++;
```

расположение операторов инкремента и декремента не имеет значения.

Оператор присваивания

Вы уже знакомы с оператором присваивания, а теперь, пора узнать поближе этот важный символ. Рассмотрим следующий простой оператор, который присваивает значение **B** переменной **A**:

```
A = B;  /* присвоить значение B переменной A */
```

В результате выполнения этого оператора значение **B** не изменяется, а прежнее значение **A** исчезает, как утренний туман в лучах солнца.

Выражение

```
A = 1234;
```

вполне законно при условии, что переменная **A** может хранить числовые значения. Выражение

```
1234 = A;  /* ??? */
```

не имеет смысла и не будет компилироваться, потому что литеральная константа (**1234**) не является переменной и не может принимать значения.

Множественное присваивание

Как вы уже знаете, каждое выражение имеет свое значение. Выражение **(A = B)** имеет значение, равное значению присваиваемого операнда. Следовательно, выражение

```
C = (A = B);
```

полностью допустимо. Сначала заключенное в круглые скобки подвыражение **(A = B)** присваивает значение **B** переменной **A**. Затем значение этого подвыражения (равное **B**) присваивается переменной **C**. Таким образом, как переменная **C**, так и **A,** сейчас равна **B**. Можно даже обойтись без круглых скобок. Выражение

```
C = A = B;
```

присваивает значение **B** переменным **A** и **C**.

Принимая это во внимание, возьмите на вооружение один из способов инициализации многих переменных. Выражение

```
C = A = B = 451;
```

присваивает число **451** переменным **A**, **B** и **C**.

Сокращенный оператор присваивания

Операторы присваивания иногда выполняют больше работы чем нужно. В операторе

```
A = A + 45;
```

компилятор сначала генерирует код для вычисления выражения **A + 45**, результат которого затем присваивается опять переменной **A**. То же самое выражение можно записать проще:

```
A += 45;
```

Эта запись помогает компилятору сгенерировать программный код более рационально. Двойной символ += называется сокращенным оператором присваивания. Вот полный набор сокращенных операторов:

```
*=, /=, +=, -=, %=, <<=, >>=, &=, ^=, |=.
```

Следующие несколько примеров демонстрируют, как работает сокращенная форма операторов (переменная **count** имеет тип **int**).

```
count += 10; /* count = count + 10; */
count *= 2;  /* count = count * 2;  */
count /= 3;  /* count = count / 3;  */
count %= 16; /* count = count % 16; */
```

Большинство современных компиляторов способны генерировать эффективный программный код и для полных форм операторов присваивания, но в сложных выражениях, где участвуют не такие простые переменные, как **count**, множественные ссылки на переменные могут снизить быстродействие программы. Используйте сокращенные операторы присваивания, где только это возможно. Они ускорят выполнение программы.

Приоритеты операторов

В сложных выражениях операторы с более высоким приоритетом выполняются в первую очередь. Причем одни операции выполняются слева направо, а другие – справа налево. Это свойство операторов называется *ассоциативностью.*

Для получения точных результатов, т.е. для указания нужного порядка выполнения операций, используйте круглые скобки, например:

```
y = ((a * x) + b) / (x + c);
```

Так как операторы ***** и / имеют более высокий приоритет, чем + и –, то представленное выше выражение можно записать проще и получить при этом правильный результат, а именно:

```
y = (a * x + b) / (x + c);
```

Изучите приоритеты операторов и правила ассоциативности, приведенные в табл. 2.3. Но чем полагаться исключительно на приоритет операторов, добавьте лучше лишнюю пару круглых скобок – они не повлияют на быстродействие, а программа станет более понятной.

Таблица 2.3

Приоритет операторов и порядок вычисления (ассоциативность)

Уровень приоритета	Операторы	Порядок вычислений
1	`() . [] ->`	Слева направо
2	`* & ! ~ ++ -- + - (тип) sizeof`	Справа налево
3	`* / %`	Слева направо
4	`+ -`	Слева направо
5	`<< >>`	Слева направо
6	`< <= > >=`	Слева направо
7	`== !=`	Слева направо
8	`&`	Слева направо
9	`^`	Слева направо
10	`|`	Слева направо
11	`&&`	Слева направо
12	`||`	Слева направо
13	`? :`	Справа налево
14	`= *= /= += -= %= <<= >>= &= ^= |=`	Справа налево
15	`,`	Слева направо

Комментарий. *Унарный плюс (+) и унарный минус (–) находятся на уровне 2, и их приоритет выше, чем у арифметических плюса и минуса, находящихся на уровне 4. Символ* ***&*** *на уровне 2 является оператором взятия адреса, а*

символ ***&*** *на уровне 8 представляет поразрядный оператор И. Символ * на уровне 2 – это оператор разыменования, а символ * на уровне 3 означает оператор умножения. При отсутствии круглых скобок операторы одного уровня вычисляются в соответствии с порядком вычислений.*

Назначение некоторых операторов сейчас может показаться вам непонятным. Мы подробно рассмотрим их в следующих главах.

Оператор *if*

Оператор **if** («если») действует точно так, как ожидается. Если выражение истинно, оператор выполнит действие; в противном случае оператор выполняться не будет.

В общем виде оператор **if** можно записать так:

```
if (выражение)
  оператор;
```

Выражением может служить любое выражение, которое в результате дает значение «истина» или «ложь». Если *выражение* истинно (ненулевое), то *оператор* выполняется. Если *выражение* ложно (нуль), то *оператор* пропускается.

Оператор **if** может быть составным:

```
if (выражение) {
  оператор1;
  оператор2;
}
```

Расположение фигурных скобок в составном операторе **if** строго не регламентируется, однако правила хорошего стиля написания программ (см. прил. 2) рекомендуют выравнивать скобки, как в предыдущем фрагменте, либо следующим образом:

```
if (выражение)
{
  оператор1;
  оператор2;
}
```

Для записи сравниваемых выражений вы можете использовать оператор отношения:

```
if (выражение == значение)
  оператор;
```

Оператор будет выполнен только в том случае, если *выражение* равно *значению*. Не следует записывать оператор **if** в таком виде:

```
if (выражение = значение) /* ??? */
  оператор;
```

в противном случае вы получите предупреждение компилятора о неверном, возможно, присваивании: вы, вероятно, не хотели присваивать *значение*

выражению, а скорее всего случайно ввели один знак равенства вместо двух – это одна из самых распространенных ошибок.

Листинг 2.2 показывает, как использовать оператор **if**. Скомпилируйте и запустите программу, затем введите число от 1 до 10. Если вы введете число, выходящее за пределы этого диапазона, программа выдаст сообщение об ошибке.

Листинг 2.2. choice.c (использование оператора **if**)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   int number;
 6:   int okay;
 7:
 8:   printf("Enter a number from 1 to 10: ");
 9:   scanf("%d", &number);
10:   okay = (number >= 1) && (number <= 10);
11:   if (!okay)
12:     printf("Incorrect answer!\n");
13:   return okay;
14: }
```

Строка 10 присваивает переменной **okay** типа **int** истинное или ложное значение результата выражения отношения

```
(number >= 1) && (number <= 10);
```

Результат выражения представляет собой целое значение – ноль, если вычисление даст «ложь», или ненулевое значение в случае «истины». Это значение присваивается переменной **okay**, после чего строка 11 проверяет выражение **!okay** («не о'кей»). Если **okay** ложно (в случае ошибки), то **!okay** даст значение «истина», и тогда строка 12 выведет сообщение об ошибке.

Строка 13 возвращает значение переменной **okay** («истину» или «ложь») обратно в операционную систему, хотя в данном случае в этом нет необходимости – это просто демонстрация такой возможности.

Вы можете также записывать сложные логические выражения непосредственно в операторах **if**. Например**,** вы могли бы записать**:**

```
if ((number < 1) || (number > 10))
  printf("Incorrect answer!\n");
```

Выражения **(number < 1)** и **(number > 10)** заключены в скобки лишь для удобства. Следующая запись вполне корректна**.**

```
if (number < 1 || number > 10)
  printf("Incorrect answer!\n");
```

Оператор *else*

Оператор **else** («иначе») является расширением оператора **if**. После любого оператора **if** вы можете вставить оператор **else** для выполнения альтернативного действия. Используйте **else** следующим образом:

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

Если *выражение* истинно (ненулевое), выполняется *оператор1*; в противном случае – *оператор2*. Операторы могут быть простыми или составными. Вы можете записать:

```
if (выражение) {
  оператор1;
  оператор2;
} else {
  оператор3;
  оператор4;
}
```

Составные операторы могут иметь любую глубину вложенности, поэтому *оператору1* разрешено иметь еще один оператор **if**, который, в свою очередь, может выполнять другие операторы **if**. Логика глубоко вложенных операторов **if** вряд ли будет прозрачной, но компилятор не налагает никаких ограничений на их количество. Лучше всего ограничиться двумя уровнями вложения.

Как и в простых операторах **if**, расположение фигурных скобок в сложных конструкциях **if-else** жестко не регламентируется. Некоторые программисты предпочитают выравнивать фигурные скобки так:

```
if (выражение)
{
  оператор1;
  оператор2;
}
else
{
  оператор3;
  оператор4;
}
```

С помощью конструкций **if-else** вы можете создать многовариантный выбор:

```
if (выражение1)
  оператор1;
else if (выражение2)
  оператор2;
else if (выражениеN)
  операторN;
else                        /* необязательно */
  оператор_умолчания;       /* необязательно */
```

Давайте разберемся, как работает такая конструкция. Если *выражение1* истинно, то выполняется *оператор1;* если истинно *выражение2*, то выполняется *оператор2* и т.д. Во всех остальных случаях (т.е. если ни одно из выражений не имеет значения «истина») выполняется *оператор_умолчания.*

Листинг 2.3, используя вложенные операторы if-else, определяет високосные годы. Скомпилируйте и запустите программу, затем введите год, например 2004. Год, открывающий новое столетие, является високосным, если без остатка делится на 400. Промежуточные годы будут високосными, если они без остатка делятся на 4.

Листинг 2.3. leap.c (определение високосного года)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   int leapYear;
 6:   int year;
 7:
 8:   printf("Leap Year Calculator\n");
 9:   printf("Year? ");
10:   scanf("%d", &year);
11:   if (year > 0) {
12:     if ((year % 100) == 0)
13:       leapYear = ((year % 400) == 0);
14:     else
15:       leapYear = ((year % 4) == 0);
16:     if (leapYear)
17:       printf("%d is a leap year\n", year);
18:     else
19:       printf("%d is not a leap year\n", year);
20:   }
21:   return 0;
22: }
```

Оператор **if** в строке 11 помогает избежать ошибки. Если введенный год оказался меньше нуля, вычисления производиться не будут. Оператор **if** в следующей строке определяет, является ли введенный год первым годом столетия. Если да, то строка 13 определяет, делится ли год на 400 без остатка. Если нет, то нужно определить, делится ли год без остатка год на 4 (строка 15).

Оператор **if-else**, содержащий ошибку (чаще всего это неправильное расположение фигурных скобок или их отсутствие), может привести к неправильной работе всей программы. Листинг 2.4 демонстрирует испорченную программу.

Листинг 2.4. badif.c (неправильная вложенность операторов **if-else**)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   int value;
 6:
 7:   printf("Value (1..10)? ");
 8:   scanf("%d", &value);
 9:   if (value >= 1)
10:     if (value > 10)
11:       printf("Error: value > 10 \n");
12:   else
13:     printf("Error: value < 1 \n");
14:   return 0;
15: }
```

Запустите программу badif и введите число от 1 до 10. Предполагалось, что числа, не входящие в этот диапазон, будут отброшены, но эта программа бракует числа именно из заданного диапазона и некоторые, лежащие за его пределами. Оператор **if** в строке 9 сравнивает введенное число с единицей, и если оно больше или равно 1, то внутренний оператор **if** сравнивает его с 10. Если ваше число больше 10, будет выдано сообщение об ошибке. Проблема в том, что код не работает так, как ожидается.

Во всем виноват оператор **else** на строке 12. Несмотря на то что он выравнивается с оператором **if** в строке 9, логически он связан с ближайшим оператором **if**, находящимся на строке 10. Здесь и заключается ошибка. Теперь ваша задача – исправить программу так, чтобы оператор **else** работал в одной связке с нужным оператором **if**.

Совет. *Чтобы проверить логику программы, никогда не полагайтесь на сделанные вами отступы. Они только для вас, компилятор не обращает на них никакого внимания. А вот фигурные скобки заставят его прислушаться к вашим пожеланиям относительно вложенных операторов* ***if-else***.

Листинг 2.5 демонстрирует исправленный вариант программы.

Листинг 2.5. goodif.c (правильная вложенность операторов **if-else**)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   int value;
 6:
 7:   printf("Value (1..10)? ");
 8:   scanf("%d", &value);
 9:   if (value >= 1) {
10:     if (value > 10)
11:       printf("Error: value > 10 \n");
12:   } else
```

```
13:     printf("Error: value < 1 \n");
14:   return 0;
15: }
```

Условные выражения

Сокращенный оператор **if-else**, называемый *условным выражением*, иногда очень полезен. В общем виде его можно записать так:

```
выражение1 ? выражение2 : выражение 3;
```

Программа вычисляет *выражение1*. Если оно истинно, то результат всего выражения равен *выражению2*. Если же *выражение1* ложно, результат равен *выражению3*. Условное выражение в точности эквивалентно оператору **if-else**:

```
if (выражение1)
  выражение2;
else
  выражение3;
```

Обычно вы присваиваете результат условного выражения какой-нибудь переменной. Предположим, вы пишете программу, управляющую меню. Если переменная типа **int menuChoice** будет равна '**Y**', вы хотите установить другую переменную **testValue** равной 100, в противном случае – равной нулю. Одно из решений использует оператор **if-else**:

```
if (menuChoice == 'Y')
  testValue = 100;
else
  testValue = 0;
```

Эта конструкция достаточно ясна, но данный оператор требует двух ссылок на переменную **testValue**. Чтобы обратиться к **testValue** только один раз и выполнить ту же самую работу, вы можете записать:

```
testValue = (menuChoice == 'Y') ? 100 : 0;
```

Если выражение (**menuChoice** == '**Y**') будет истинным, то оператор присвоит переменной **testValue** значение 100, если ложным – значение 0.

Не следует использовать условные выражения только потому, что они занимают одну строку вместо четырех. В большинстве случаев компилятор генерирует аналогичный код как для оператора **if-else**, так и для эквивалентного условного выражения. С другой стороны, чтобы избежать двух ссылок на одно и то же сложное выражение (особенно, если оно имеет сложную и громоздкую форму), предпочтительнее использовать условное выражение.

Оператор *switch*

Вложенный набор операторов **if-else** может выглядеть как запутанный водопровод старого дома. Система работает, но трудно понять, какая труба

куда ведет. Рассмотрим следующую серию операторов **if-else**, каждый из которых сравнивает выражение с определенным значением:

```
if (выражение == значение1)
  оператор1;
else if (выражение == значение2)
  оператор2;
else if (выражение == значениеN)
  операторN;
else                    /* необязательно */
  оператор_умолчания;   /* необязательно */
```

Эту конструкцию можно сократить с помощью более простого оператора **switch**:

```
switch (выражение) {
  case значение1:
    оператор1;     /* выполняется, если выражение == значение1 */
    break;         /* выход из оператора switch */
  case значение2:
    оператор2;     /* выполняется, если выражение == значение2 */
    break;         /* выход из оператора switch */
  case значениеN:
    операторN;     /* выполняется, если выражение == значениеN */
    break;         /* выход из оператора switch */
  default:
    оператор_умолчания; /*выполняется, если не было совпадений*/
}
```

На первый взгляд эта запись может показаться более длинной, но на практике оператор **switch** использовать проще, чем эквивалентную конструкцию **if-else**.

За ключевым словом **switch** следует выражение, которое будет сравниваться с заданными значениями внутри оператора. Строка

```
case значение1:
```

сравнивает *значение1* с результатом вычисления выражения оператора **switch**. Если сравнение даст значение «истина», то будет выполнен оператор (или блок операторов), следующий за строкой **case**. Если же результатом сравнения будет «ложь», аналогичным образом будет обрабатываться следующая строка **case**. Последняя строка оператора **switch** (**default**) задает действия, выполняемые в случае, если ни одно из *значений* **case** не совпало со значением *выражения*.

Оператор **break** в каждом блоке выбора осуществляет немедленный выход из оператора **switch**. Операторы **switch** разработаны таким образом, что если вы напишите их без оператора **break**

```
switch (выражение) {
  case значение1:
    оператор1;      /*  ??? */
  case значение2:
    оператор2;      /*  ??? */
  case значение3:
```

```
    оператор3;
}
```

и *выражение* будет равно *значению1*, то после выполнения *оператора1* последует выполнение *оператора2* и *оператора3*, хотя вряд ли именно это входило в ваши планы. Первый же случай сравнения, давший в результате «истину», приведет к выполнению всех последующих операторов, вплоть до конца оператора **switch**.

Замечание. *Некоторые программисты иногда намеренно опускают операторы **break** из операторов **switch**, чтобы позволить одному блоку **case** выполнять действия, записанные и в других блоках. Технически это разрешено, но затрудняет процесс отслеживания результатов и создает трудности при внесении изменений. Лучше всего завершать каждый блок **case** своим оператором **break**.*

Одним из самых распространенных программных интерфейсов является меню команд. Листинг 2.6 демонстрирует, как писать программу, обслуживающую меню, используя оператор **switch**. Скомпилируйте и запустите программу, затем введите букву **A**, **D**, **S** или **Q**, чтобы выбрать команду. (Команды не делают ничего полезного, и вы можете развить эту программу по своему вкусу.) Введите букву, которая не входит в этот маленький список, и вы увидите, как оператор **switch** обнаружит ошибку ввода.

Листинг 2.6. menu.c (демонстрация оператора **switch**)

```
 1: #include <stdio.h>
 2: #include <ctype.h>
 3:
 4: main()
 5: {
 6:   int choice;
 7:
 8:   printf("Menu: A)dd, D)elete, S)ort, Q)uit: ");
 9:   choice = toupper(getchar());
10:   switch (choice) {
11:     case 'A':
12:       printf("You selected Add\n");
13:       break;
14:     case 'D':
15:       printf("You selected Delete\n");
16:       break;
17:     case 'S':
18:       printf("You selected Sort\n");
19:       break;
20:     case 'Q':
21:       printf("You selected Quit\n");
22:       break;
23:     default:
24:       printf("\nIllegal choice!\n");
```

```
 25:    }
 26:    return 0;
 27: }
```

Чтобы прочитать символ, введенный с клавиатуры, строка 9 вызывает стандартную библиотечную функцию **getchar()**. Функция **toupper()** переводит этот символ в верхний регистр (делает прописным), упрощая таким образом определение нажатой клавиши.

Оператор **switch** начинается на строке 10. Выбранный символ рассматривается в качестве анализируемого выражения. Последующие блоки **case** сравнивают переменную **choice** с символами '**A**', '**D**', '**S**' и '**Q**', выдавая подтверждающее сообщение в случае, если сравнение было удачным. Если совпадения с заданными буквами не произошло, блок **default** (строки 23-24) выведет сообщение об ошибке.

Оператор *while*

Оператор **while** – один из трех операторов цикла языка С. Программы используют его для повторного выполнения операторов в течение всего времени, пока заданное условие истинно.

```
while (выражение)
  оператор;
```

В переводе с английского эту запись можно прочитать так: «Пока *выражение* истинно, выполнять *оператор*». Как обычно, оператор может быть простым или составным:

```
while (выражение) {
  оператор1;
  оператор2;
  .........
}
```

Листинг 2.7 использует цикл **while** для счета от 1 до 10.

Листинг 2.7. wcount.c (счет от 1 до 10 с помощью цикла **while**)

```
  1: #include <stdio.h>
  2:
  3: main()
  4: {
  5:    int counter;
  6:
  7:    printf("while count\n");
  8:    counter = 1;
  9:    while (counter <= 10) {
 10:      printf("%d\n", counter);
 11:      counter++;
 12:    }
 13:    return 0;
 14: }
```

Строка 8 инициализирует целую переменную **counter**, называемую управляющей переменной цикла. Оператор **while** в строках 9-12 сравнивает значение переменной **counter** с числом 10. Пока **counter** меньше или равно 10, будут выполняться операторы на строках 10-11. Строка 11 увеличивает **counter** на единицу на каждой итерации цикла, гарантируя таким образом, что цикл в конце концов закончится. (Цикл обычно должен выполнять хотя бы один оператор, который влияет на его условие, в противном случае он превратится бесконечный цикл.)

Чему будет равно значение переменной **counter** после окончания работы оператора **while**? Проверьте свою догадку, вставив следующий оператор между строками 12 и 13:

```
printf("counter = %d\n", counter);
```

Почему переменная **counter** имеет конечное значение 11? Чему оно будет равно, если заменить выражение в строке 9 на **(counter < 10)**? Что произойдет, если в строке 8 присвоить переменной **counter** значение, равное 11? Будет ли цикл выполняться?

Ответив на эти вопросы, вы определите важное свойство цикла **while**: если анализируемое выражение с самого начала ложно, ни один из его операторов не будет выполнен.

Упражнение. *Модифицируйте листинг 2.7 в направлении обратного счета от 10 до 1.*

В условии цикла **while** можно использовать и другие виды переменных. Листинг 2.8 выводит английский алфавит с помощью символьной переменной.

Листинг 2.8. walpha.c (вывод алфавита с помощью цикла **while**)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   char c;
 6:
 7:   printf("while alphabet\n");
 8:   c = 'A';
 9:   while (c <= 'Я') {
10:     printf("%c", c);
11:     c++;
12:   }
13:   return 0;
14: }
```

Выражение в строке 9 **(c <= ‘Z’)** истинно, если ASCII-значение символа (т.е. его номер в таблице ASCII – см. приложение 1) меньше или равно ASCII-значению буквы ‘**Z**’.

Упражнение. *Измените листинг 2.8 таким образом, чтобы программа выводила на экран русский алфавит.*

Оператор *do-while*

Оператор **do-while** («делай – пока») является в некотором роде перевернутым циклом **while**.

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

Выполняется *оператор*, затем проверяется *выражение*. До тех пор пока значение *выражения* будет истинным, *оператор* будет выполняться.

Сравните эту схему работы с рассмотренным выше оператором **while**, который вычисляет свое выражение *прежде,* чем выполнить какой-нибудь оператор. Цикл **while** вообще может не выполнить ни одного оператора, если анализируемое выражение с самого начала было ложным. Однако **do-while** всегда выполняет свои операторы, по крайней мере, один раз, т.к. проверка *выражения* осуществляется в *конце* каждой итерации цикла. Из этого следует основное правило, необходимое при выборе между **while** и **do-while**.

- Спросите себя: «Существуют ли такие условия, при которых операторы цикла не должны выполняться ни разу?» Если ответом будет «да», то вам, скорее всего, следует выбрать цикл **while**.
- Если ответ на предыдущий вопрос будет «нет», то вам, вероятно, подойдет цикл **do-while**.

Листинг 2.9 аналогичен приведенной выше программе wcount, однако использует для счета от 1 до 10 цикл **do-while**.

Листинг 2.9. dwcount.c (счет от 1 до 10 с помощью цикла **do-while**)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   int counter;
 6:
 7:   printf("do-while count\n");
 8:   counter = 0;
 9:   do {
10:     counter++;
11:     printf("%d\n", counter);
12:   } while (counter < 10);
13:   return 0;
14: }
```

Задание. *Кроме выполнения программы dwcount на компьютере, попытайтесь выполнить ее вручную, записывая значения переменной* ***counter*** *на бумаге. Выясните, почему в строке 12 используется выражение* ***(counter***

***< 10)**, а не **(counter <= 10)**, как это было в программе wcount. Проверьте свои предположения, вставив операторы **printf()** для отображения значения переменной **counter** в интересующие вас места программы (это неплохое средство отладки).*

Условием цикла **do-while** может быть любое логическое выражение. Чтобы продемонстрировать такую многогранность, листинг 2.10 использует цикл **do-while** для вывода на экран алфавита.

Листинг 2.10. dwalpha.c (вывод алфавита с помощью цикла **do-while**)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   char c;
 6:
 7:   printf("do-while alphabet\n");
 8:   c = 'A' - 1;
 9:   do {
10:     c++;
11:     printf("%c", c);
12:   } while (c < 'Z');
13:   return 0;
14: }
```

Оператор *for*

Когда вам точно известно, сколько раз должны выполняться операторы цикла, лучше всего использовать третий оператор цикла языка С – **for**.

```
for (выражение1; выражение2; выражение3;) {
  оператор;
}
```

На первый взгляд, оператор **for** может показаться сложным, но он станет понятнее, если его представить в виде эквивалентного ему оператора **while**:

```
выражение1;
while (выражение2;) {
  оператор;
  выражение3;
}
```

Например, для того чтобы с помощью цикла **for** посчитать от 1 до 10, можно взять переменную **i** типа **int** и записать:

```
for (i = 1; i <= 10; i++)   /* выражения 1, 2, и 3 */
  printf("i = %d\n", i);    /* оператор */
```

Первое выражение цикла **for** в этом примере присваивает начальное значение переменной **i**, равное 1. Это действие выполняется только один раз, перед самым началом цикла. Второе выражение обычно является выражением отношения. В нашем случае оно будет равно «истине», если

значение **i** меньше или равно 10. Третье, и последнее, выражение увеличивает на единицу значение переменной **i**, приближая, таким образом, цикл к концу.

Предыдущий оператор **for** можно изобразить в виде эквивалентного цикла **while**:

```
i = 1;                          /* выражение 1 */
while (i <= 10) {               /* выражение 2 */
  printf("i = %d\n", i);        /* оператор    */
  i++;                          /* выражение 3 */
}
```

Листинг 2.11 использует цикл **for** для отображения набора видимых символов из основной части таблицы ASCII (это символы со значениями от 32 до 127).

Листинг 2.11. fascii.c (отображение ASCII-символов с помощью цикла **for**)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   char c;
 6:
 7:   for (c = 32; c < 128; c++) {
 8:     if ((c % 32) == 0) printf("\n");
 9:     printf("%c", c);
10:   }
11:   printf("\n");
12:   return 0;
13: }
```

Когда вы запустите программу fascii, она отобразит на экране следующую 96-символьную таблицу:

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

Цикл **for** (строки 7-10) первоначально устанавливает управляющую переменную **c** типа **char** равной ASCII-значению символа пробела (32). Управляющее выражение **(c < 128)** завершает цикл после отображения последнего символа, т.е. когда переменная **c** станет равной 128. Выражение **c++** увеличивает управляющую переменную на каждом проходе цикла на единицу.

Оператор **if**, находящийся внутри оператора **for**, сравнивает выражение (**c % 32**), равное остатку от деления **c** на 32, с нулем. Если результат сравнения – значение «истина», то оператор **printf()** начинает новую строку. Замените 32 на 16, и вы получите вывод по 16 символов в строке.

Упражнение. *Замените цикл **for** в листинге 2.11 эквивалентным циклом **while**.*

Бесконечный цикл

Выражения оператора **for** в предыдущем примере можно было бы опустить, но в этом случае мы получаем бесконечный цикл:

```
for ( ; ; ) ; /* бесконечный цикл */
```

Этот цикл не инициализирует управляющую переменную, не содержит управляющего выражения и не выполняет никаких действий, чтобы завершиться. Он также не имеет никаких операторов. Во время выполнения программа «зависает», пока не произойдет какое-нибудь внешнее событие: например, пропадет электропитание или сигнал прерывания заставит процессор начать выполнение другой программы.

Пробел между закрывающей круглой скобкой и завершающей точкой с запятой поставлен не случайно. Без этого пробела некоторые компиляторы С выдают предупреждение, которое помогает избежать следующей ошибки:

```
for (i = 0; i < 100; i++); /* ??? */
  оператор;
```

Внимательно рассмотрите этот фрагмент. Цикл не выполняет никаких полезных действий – сто раз выполняется «пустой» оператор (;), следующий после закрывающей круглой скобки **for**; *оператор* выполнится только один раз.

Оператор *break*

Иногда бывает нужно прервать выполнение цикла **while**, **do-while** или **for**. Листинг 2.12 демонстрирует, как оператор **break** может прервать программу подобно тому, как срочный выпуск новостей прерывает телевизионную передачу.

Листинг 2.12. breaker.c (действие оператора **break**)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   int count;
 6:
 7:   printf("\n\nfor loop:\n");
 8:   for (count = 1; count <= 10; count++) {
 9:     if (count > 5) break;
10:     printf("%d\n", count);
11:   }
12:
13:   printf("\n\nwhile loop:\n");
14:   count = 1;
15:   while (count <= 10) {
16:     if (count > 5) break;
17:     printf("%d\n", count);
18:     count++;
19:   }
```

```
20:
21:   printf("\n\ndo/while loop:\n");
22:   count = 1;
23:   do {
24:     if (count > 5) break;
25:     printf("%d\n", count);
26:     count++;
27:   } while(count <= 10);
28:
29:   return 0;
30: }
```

Программа breaker выполняет операторы **for** (строки 8-11), **while** (строки 15-19) и **do-while** (строки 23-27). Каждый оператор считает от 1 до 10, используя переменную **count**. Каждый цикл также выполняет оператор **break** *до* того, как переменная **count** достигнет своего конечного значения (строки 9, 16, 24), поэтому эти циклы заканчиваются преждевременно. Данный метод очень полезен, особенно, когда нужно в спешке «спасаться бегством» из цикла: например, для того, чтобы отреагировать на ошибочную информацию.

Операторы **break** можно использовать и в других случаях (вы уже встречали их раньше внутри оператора **switch**). Например, широко распространен метод программирования, при котором бесконечный цикл использует оператор **break**, чтобы выйти из «вечности»:

```
for ( ; ; ) {          /* выполнять цикл "вечно" */
  .............        /* различные операторы */
  if (выражение)       /* если выражение окажется истинным */
    break;             /* прервать цикл */
}
```

Так обычно пишется главный цикл приложений для Windows, программируются контроллеры, программы с пользовательским интерфейсом. Многие программисты предпочитают организовывать бесконечный цикл с помощью ы **while**:

```
while (1) {          /* бесконечный цикл while */
  .............
  if (выражение)
    break;
}
```

Упражнение. *В листинге 2.1, чтобы преобразовать не одно значение температуры, а несколько, вам приходится каждый раз заново запускать программу. Добавьте в программу цикл для повторения преобразований до тех пор, пока вы не захотите выйти из программы.*

Оператор *continue*

Оператор **continue** похож на **break**, но он заставляет цикл прервать текущую итерацию и начать следующую. Листинг 2.13 демонстрирует различие между операторами **break** и **continue**.

Листинг 2.13. continue.c (действие оператора **continue**)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   int count;
 6:
 7:   printf("\nStarting for loop with continue...\n");
 8:   for (count = 1; count <=10; count++) {
 9:     if (count > 5) continue;
10:     printf("%d\n", count);
11:   }
12:   printf("After for loop, count = %d\n", count);
13:
14:   printf("\n\nStarting for loop with break...\n");
15:   for (count = 1; count <= 10; count++) {
16:     if (count > 5) break;
17:     printf("%d\n", count);
18:   }
19:   printf("After for loop, count = %d\n", count);
20:   return 0;
21: }
```

Чтобы посчитать от 1 до 10, программа continue использует два цикла **for** (строки 8-11 и 15-18). Строка 9 первого цикла **for** выполнит оператор **continue**, если переменная **count** будет больше 5. Строка 16 выполнит оператор **break** при том же условии. За исключением этого различия приведенные циклы идентичны.

Оператор **continue** прерывает последовательное выполнение программы и передает управление в начало цикла. Оператор **printf()** в строке 10, таким образом, пропускается. Оператор **break** немедленно осуществляет выход из цикла, передавая управление следующему после цикла оператору (строка 19).

Программа отображает значение переменной **count** внутри и снаружи циклов. После выхода из первого цикла оно равно 11, после выхода из второго – 6.

Оператор *goto*

Оператор **goto** позволяет перейти в указанное место программы, и начиная с этой позиции, продолжить выполнение операторов программы. Поскольку **goto** позволяет «прыгать» в любое место программы, его

использование напоминает желание оставить проторенную дорогу ради замеченной вами извилистой тропинки в поле.

На первый взгляд, **goto** кажется очень полезным оператором. Однако на практике он дает программистам слишком большую свободу «скакать» по программе куда угодно. В лучшем случае, результаты программы, которая имеет несколько **goto** операторов, просто трудно понять, в худшем – такая программа вообще работает плохо.

Если вы не можете обойтись без **goto**, поставьте метку (неиспользованный ранее идентификатор и двоеточие) в нужном месте программы (перед оператором, на который вы хотите «прыгнуть»). Выполните **goto** МЕТКА, и «течение» вашей программы будет направлено «по новому руслу». Листинг 2.14 продемонстрирует, как использовать **goto** для счета от 1 до 10.

Листинг 2.14. gcount.c (счет от 1 до 10 с помощью оператора **goto**)

```
 1: #include <stdio.h>
 2:
 3: main()
 4: {
 5:   int count = 1;
 6:
 7:   printf("\ngoto count\n");
 8:   TOP:
 9:   printf("%d\n", count);
10:   count++;
11:   if (count <= 10) goto TOP;
12:   return 0;
13: }
```

Метка **TOP** в строке 8 помечает «пункт назначения» для оператора **goto**. Оператор **if** на строке 11 проверяет значение целой переменной **count**. Если оно меньше или равно 10, **goto** передает управление на метку **TOP**, чтобы снова выполнить операторы **printf()** и **count++**, пока переменная **count** не станет больше 10.

Программа, конечно, работает, но ей не хватает наглядности циклов **while**, **do-while** и **for**, описанных в этой главе. Понимание **goto**-варианта требует отслеживания вручную работы каждого оператора. Отладка сложных программ с оператором **goto** – это кропотливый и неблагодарный труд. Тем не менее вы должны знать, как использовать операторы **goto**, т.к. можете встретиться с ними в чужой программе. Но не используйте их в своих собственных – это тот оператор, без которого можно обойтись.

Резюме

- Программы выполняют действия, вычисляя выражения и выполняя операторы.
- Все выражения имеют свои значения. Например, значение выражения **(A = B)** равно значению, присвоенному переменной **A**.
- Операторы языка C *****, /, +, – и **%** выполняют арифметические операции над операндами. Операторы отношений <, <=, >, >=, == и **!=** сравнивают два операнда между собой. Логические операторы **&&** и || объединяют выражения в соответствии с правилами логического И и ИЛИ. Оператор отрицания **!** инвертирует значение логического выражения.
- Оператор инкремента ++ прибавляет единицу к своему операнду. Оператор декремента **––** вычитает единицу из своего операнда. Эти операторы могут стоять как перед, так и после своих операндов.
- Сокращенные операторы присваивания позволяют сократить операторы типа **value** = **value** + **10** до **value** += **10**. Наиболее часто в языке C используются сокращенные операторы: ***=**, **/=**, **+=**, **–=** и **%=**.
- Все операторы имеют приоритеты. Выражения, использующие операторы с более высоким приоритетом, вычисляются раньше выражений, которые используют операторы с низким приоритетом.
- Для управления порядком вычисления в выражениях всегда можно использовать круглые скобки, изменяя, таким образом, действующие по умолчанию приоритеты операторов и их ассоциативность.
- Операторы **if** и **if-else** позволяют принимать решения в зависимости от того, является ли анализируемое выражение истинным или ложным.
- Используйте операторы **switch** для упрощения длинных конструкций **if-else**. Убедитесь, что каждый блок **case** заканчивается оператором **break**.
- Используйте операторы **while** для создания циклов, которые выполняются, пока проверяемое условие остается истинным. В цикле **while** условное выражение проверяется в самом начале и, следовательно, если оно изначально ложно, то операторы цикла выполняться не будут.
- Используйте операторы **do-while** для создания циклов, выполняющихся до тех пор, пока проверяемое условие не станет ложным. В циклах **do-while** условное выражение вычисляется в конце, и, следовательно, их операторы выполняются, по крайней мере, один раз.
- Оператор **for**, возможно, самый популярный оператор в языке C. Используйте его, когда вы знаете или можете вычислить заранее, сколько итераций должен иметь цикл.
- Оператор **break** немедленно завершает цикл.
- Оператор **continue** заставляет цикл начать следующую итерацию.

- Оператор **goto** передает управление на любой помеченный оператор. Поскольку программы, использующие много операторов **goto**, как правило, трудны для понимания и отладки, использовать **goto** не рекомендуется.

Обзор функций

Таблица 2.4

Математические функции (math.h)

Функция	**Прототип и краткое описание**
1	2
abs()	`int abs(int i);` Возвращает абсолютное значение целого аргумента **i**.
acos()	`double acos(double x);` Функция арккосинуса**.** Значение аргумента должно находиться в диапазоне от –1 до +1.
asin()	`double asin(double x);` Функция арксинуса**.** Значение аргумента должно находиться в диапазоне от –1 до +1.
atan()	`double atan(double x);` Функция арктангенса**.**
atan2()	`double atan2(double y, double x);` Функция арктангенса от значения **y / x**.
ceil()	`double ceil(double x);` Вычисляет ближайшее целое, не меньшее, чем аргумент **x**.
cos()	`double cos(double x);` Функция косинуса**.** Угол (аргумент) задается в радианах.
exp()	`double exp(double x);` Экспоненциальная функция $\mathbf{e^x}$.

Окончание табл. 2.4

1	2
fabs()	`double fabs(double x);` Возвращает абсолютное значение вещественного аргумента **x**.
floor()	`double floor(double x);` Находит наибольшее целое, не превышающее значение **x**. Возвращает его в форме **double**.
fmod()	`double fmod(double x, double y);` Возвращает остаток от деления нацело **x** на **y**.
labs()	`long labs(long x);` Возвращает абсолютное значение аргумента типа **long**.
log()	`double log(double x);` Возвращает значение натурального логарифма (**ln x**).
log10()	`double log10(double x);` Возвращает значение десятичного логарифма (**$\log_{10}$x**).
pow()	`double pow(double x, double y);` Возвращает значение **x** в степени **y**.
sin()	`double sin(double x);` Функция синуса**.** Угол (аргумент) задается в радианах.
sqrt()	`double sqrt(double x);` Возвращает значение квадратного корня из **x**.
tan()	`double tan(double x);` Функция тангенса**.** Угол (аргумент) задается в радианах.

Таблица 2.5

Ввод и вывод символов (stdio.h)

Функция	**Прототип и краткое описание**
getchar()	`int getchar(void);` Считывает следующий символ со стандартного потока ввода (обычно с клавиатуры) и возвращает его.
putchar()	`int putchar(int c);` Записывает символ **c** в стандартный поток вывода (как правило, на экран) и возвращает его.

Таблица 2.6

Функции проверки и преобразования символов (ctype.h)

Функция	Прототип и краткое описание
isalnum()	`int isalnum(int c);` Возвращает ненулевое значение, если **c** – код буквы или цифры (A-Z, a-z, 0-9), и нуль – в противном случае.
isalpha()	`int isalpha(int c);` Возвращает ненулевое значение, если **c** – код буквы (A-Z, a-z), и нуль – в противном случае.
isdigit()	`int isdigit(int c);` Возвращает ненулевое значение, если **c** – код цифры (0-9), и нуль – в противном случае.
islower()	`int islower(int c);` Возвращает ненулевое значение, если **c** – код буквы в нижнем регистре (a-z), и нуль – в противном случае.
isspace()	`int isspace(int c);` Возвращает ненулевое значение, если **c** – обобщенный пробел: пробел, символ табуляции, символ новой строки или новой страницы, символ возврата каретки (коды 0x09-0x0D, 0x20), и нуль – в противном случае.
isupper()	`int isupper(int c);` Возвращает ненулевое значение, если **c** – код буквы в верхнем регистре (A-Z), и нуль – в противном случае.
tolower()	`int tolower(int c);` Если аргумент является символом верхнего регистра, возвращает соответствующий символ нижнего регистра, иначе возвращает исходный аргумент.
toupper()	`int toupper(int c);` Если аргумент является символом нижнего регистра, возвращает соответствующий символ верхнего регистра, иначе возвращает исходный аргумент.

Примечание. *В разделе «Обзор функций» в этой и следующих главах приводятся только наиболее употребительные функции ANSI C. Для получения информации обо всех доступных функциях и примерах их использования обратитесь к справочной системе (Help).*

3. Функции

Создание компьютерных программ подобно строительству мостов. Нельзя сразу заливать бетон в пустоту; прежде чем перекрыть воду, нужно поставить опоры на земле.

Функции для программистов – то же, что балки, канаты и камни для строителей мостов. Функции делят большую программу на поддающиеся более простому решению составляющие. Кроме того, они экономят память, аккумулируя в себе повторяющиеся операции.

Нисходящее программирование

Чтобы написать сложную программу на С, немногие специалисты сядут за компьютер и просто начнут вводить программный текст. Опытные программисты делят большие проблемы на маленькие части и справляются с каждой из них по очереди, пока не будет выполнена вся работа.

В основу нисходящего программирования положен метод «разделяй и властвуй»: начните с главной задачи, разделите ее на более мелкие, те, в свою очередь, – на подзадачи, пока не получите относительно простые для решения проблемы. Затем для каждой из них напишите свою функцию, и дело сделано.

В следующих разделах вы узнаете, как писать функции, а затем научитесь применять их во время создания программ методом нисходящего программирования.

Функции, которые возвращают пустоту

У каждой С-программы есть, по крайней мере, одна функция, а именно **main()**. Большинство же программ имеют много функций, каждая из которых выполняет свою задачу. И чем *уже* будет эта задача, тем лучше. Запомните, в данном случае золотым правилом является простота.

Листинг 3.1 показывает корректный способ написания и использования функций. Программа считает от 1 до 10, а затем вниз – от 10 до 1.

Совет. *Несмотря на простоту, программы, подобные fncount, помогают демонстрировать новые понятия. Если вы сталкиваетесь с каким-то термином программирования, который не понимаете, напишите аналогичную тестовую программу. Учитесь быть своим собственным учителем, а компилятор станет вашим гидом.*

Листинг 3.1. fncount.c (демонстрация функций)

```
1: #include <stdio.h>
2:
```

```
 3: void CountUp(void);
 4: void CountDown(void);
 5:
 6: main()
 7: {
 8:   CountUp();
 9:   CountDown();
10:   return 0;
11: }
12:
13: void CountUp(void)
14: {
15:   int i;
16:
17:   printf("\n\nCounting up to 10\n");
18:   for (i = 1; i <= 10; i++)
19:     printf("%4d", i);
20: }
21:
22: void CountDown(void)
23: {
24:   int i;
25:
26:   printf("\n\nCounting down from 10\n");
27:   for (i = 10; i >= 1; i--)
28:     printf("%4d", i);
29: }
```

Строки 3-4 объявляют прототипы функций, которые сообщают компилятору имя и форму каждой функции в программе. Внимательнее рассмотрите строку 3:

```
void CountUp(void);
```

CountUp представляет собой имя функции; постарайтесь выбирать для функций такие имена, чтобы с первого взгляда было понятно ее назначение. Поскольку функция **CountUp()** выполняет действие, но не возвращает значение, ее имени предшествует ключевое слово **void** (слово «**void**» означает «пустой»). Второе слово **void** внутри круглых скобок, где обычно находится список параметров, сообщает компилятору, что функции **CountUp()** не требуется передача аргументов. **CountUp()** – функция простейшего вида: она ничего не принимает и ничего не возвращает.

Чтобы использовать (вызвать) функцию типа **void**, просто напишите ее имя, как показано в строках 8-9. Оператор

```
CountUp();
```

выполняет функцию **CountUp()**, временно останавливаясь в этом месте программы, пока не выполнятся операторы функции. Пустые круглые скобки в данном случае нужны обязательно, т.к. в отличие от языка Pascal С-функции всегда записываются с круглыми скобками. Когда функция будет

выполнена (программисты говорят: «Когда функция вернет управление»), программа продолжит свою работу с того места, на котором она остановилась. В данном случае программа выполнит оператор в строке 9, который вызовет другую функцию, а именно **CountDown()**. Когда и эта функция вернет управление, будет выполнен оператор в строке 10 и программа завершится (рис. 3.1).

Каждая объявленная функция (строки 3-4) должна быть определена

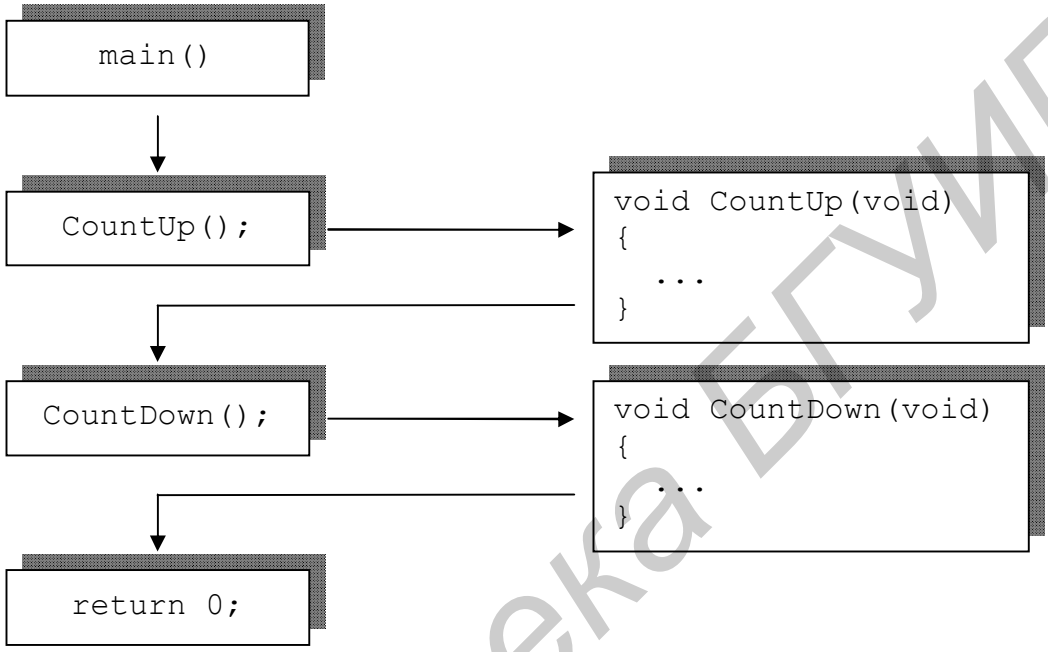

Рис. 3.1. При вызове функций **CountUp()** и **CountDown()** управление временно передается операторам этих функций

где-нибудь в программе (строки 13-29). Функция может быть определена в любом месте программы *после* объявления прототипа, но не внутри другой функции.

Определение функции **CountUp()**:

```
void CountUp(void)
{
  int i;

  printf("\n\nCounting up to 10\n");
  for (i = 1; i <= 10; i++)
    printf("%4d", i);
}
```

напоминает функцию **main()**. Сначала идет заголовок функции, который должен полностью совпадать с объявленным ранее прототипом, но без завершающей точки с запятой (строки 3 и 13). В больших программах это правило заставляет вас тщательно проектировать функции и реализовать их так, как они были спланированы. Подобно обложке книги, фигурные скобки

ограничивают тело функции, в котором могут объявляться переменные и выполняться операторы.

Функцию могут вызывать любые операторы, включая расположенные внутри других функций. Например, вставьте оператор

```
CountDown();
```

между строками 19 и 20 в листинге 3.1. Когда вы запустите программу, в строке 8 произойдет вызов функции **CountUp()**, которая перед тем, как вернуть управление, вызовет функцию **CountDown()**.

Чтобы выйти из функции в нужный момент, выполните оператор **return**.

```
void AnyFn(void)
{
  if (условие)
    return;
  оператор;
}
```

Функция **AnyFn()** немедленно завершается, если *условие* будет истинным. *Оператор* же выполнится только в случае, если *условие* окажется ложным.

Локальные и глобальные переменные

Любые переменные, объявленные внутри функции, имеют сравнительно короткое время жизни, существуя в памяти компьютера только тогда, когда функция работает. Такие переменные называются *локальными*. Например, целая переменная **i** локальна для функции **CountUp()** в примере, рассмотренном выше. Переменная **i** в строке 15 и переменная **i** в строке 24 листинга 3.1 являются двумя разными переменными.

Переменные, объявленные вне функций, – это глобальные переменные. Из записи программного фрагмента:

```
int global;    /* global – вне функции main() */

main()
{
  int local;  /* local – внутри main() или другой функции */
  ...
}
```

следует, что любые программные операторы могут пользоваться переменной **global**, а к переменной **local** имеют доступ только операторы внутри функции **main()**. Важная особенность: в отличие от локальных, глобальные переменные всегда инициализируются нулевыми значениями. Таким образом, значение переменной **global** в приведенном фрагменте равно нулю, а значение **local** – не определено.

Допустим, та же самая программа определяет функцию вида

```
void AnyFn(void)
```

```
{
  global = 5;  /* все нормально */
  local = 6;   /* ??? */
   ...
}
```

При этом присваивание переменной **local** не будет скомпилировано. Эта локальная переменная определена в функции **main()** и нигде больше не существует.

Умение правильно пользоваться локальными переменными определит ваш успех в программировании на С. Рассмотрим подробнее их некоторые важные характеристики.

Область видимости переменных

Переменные обладают свойством, называемым *областью видимости*. Операторы внутри этой области могут «видеть» значение переменной и работать с ней. Но операторы вне этой области не имеют доступа к переменной и, таким образом, не могут ее использовать.

Область видимости локальных переменных

В начале работы функция выделяет память в стеке для запоминания своих локальных переменных. Эта память существует, только пока функция активна. После своего «возврата» функция удаляет выделенную стековую память, отбрасывая за ненадобностью все запомненные там переменные. Таким образом, стек динамически то растет, то сокращается по мере того, как операторы вызывают функции и происходит возврат из них.

Сохраняясь в стеке, локальные переменные «помогают» функциям эффективно использовать память. В некоторых программах, использующих множество функций, суммарное пространство, занимаемое всеми локальными переменными, может быть очень велико (иногда даже больше доступной памяти). Однако благодаря тому, что локальные переменные заносятся и удаляются из стека по мере выполнения функций, переполнения памяти не происходит. Глобальные переменные, которые нужно хранить в памяти все время, пока выполняется программа, используют память компьютера с гораздо меньшей эффективностью.

Поскольку область видимости локальной переменной ограничена функцией, в которой она объявлена, две функции могут бесконфликтно объявлять локальные переменные с одинаковыми именами. Таким образом, нет необходимости подыскивать различные идентификаторы для ваших локальных переменных, используемых в разных функциях. Сорок функций могут объявить локальную переменную **i**, использующуюся в цикле **for**, без каких-либо проблем. Листинг 3.2 демонстрирует эту идею.

Листинг 3.2. local.c (неконфликтующие локальные переменные)

```
 1: #include <stdio.h>
 2: #include <conio.h>
 3:
 4: void Pause(void);
 5: void Function1(void);
 6: void Function2(void);
 7:
 8: main()
 9: {
10:   Function1();
11:   return 0;
12: }
13:
14: void Pause(void)
15: {
16:   printf("Press <Spacebar> to continue...");
17:   while (getch() != ' ') ;
18: }
19:
20: void Function1(void)
21: {
22:   char s[15] = "Grodno\n";
23:
24:   printf("\nBegin function #1. s = %s", s);
25:   Pause();
26:   Function2();     /* вызов Function2 из Function1 */
27:   printf("\nBack in function #1. s = %s", s);
28: }
29:
30: void Function2(void)
31: {
32:   char s[15] = "Brest\n";
33:
34:   printf("\nBegin function #2. s = %s", s);
35:   Pause();
36: }
```

Скомпилируйте и запустите программу local. Программа отобразит следующее:

```
Begin function #1, s = Grodno
Press <Spacebar> to continue...
Begin function #2, s = Brest
Press <Spacebar> to continue...
Back in function #1, s = Grodno
```

Как функция **Function1()**, так и функция **Function2()** объявляют и отображают свою строковую переменную **s** (строки 22 и 32). **Function1()** вызывает **Function2()** в строке 26. Если бы переменные конфликтовали между собой, **Function2()** изменила бы строковое значение другой функции и, таким образом, при выполнении строки 27 вы бы увидели “**Brest”** вместо “**Grodno**”. Но этого не происходит, так как, несмотря на то что эти две

переменные имеют одинаковые имена, их область видимости охватывает только функции, в которых они объявлены. Одна переменная не зависит от другой и имеет свое собственное значение.

Кроме того, в программе есть еще одна функция – **Pause()** (строки 14-18), которую вы можете использовать для своих собственных разработок. **Pause()** выводит сообщение «Press <Spacebar> to continue…», которое приглашает вас нажать клавишу <Пробел>, и выполняет цикл **while**

```
while (getch() != ' ') ;
```

Этот оператор циклически выполняется до тех пор, пока функция **getch()** не вернет символ пробела. Цикл не выполняет никаких других действий, поэтому перед точкой с запятой стоит пробел. Функция **getch()** не отображает введенный символ, для этого есть аналогичная функция **getche()**.

Локальные переменные не сохраняют свои значения между вызовами функций, в которых они объявлены. В следующем фрагменте

```
void AnyFn(void)
{
  int anyInt;
  ...
}
```

переменная **anyInt** создается каждый раз заново при вызове функции **AnyFn()**. При завершении работы функции значения переменной **anyInt** и других локальных переменных теряются.

Менее распространенный способ объявления локальных переменных – вставка объявления прямо в операторный блок. Например, вы можете объявить локальную переменную в операторе **if** следующим образом:

```
if (условие) {
  int i = 1;
  ...
}
```

Целая переменная **i** разместится в памяти, только если *условие* окажется истинным. Если переменная **i** будет создана, ее область видимости и время жизни будут ограничены закрывающей фигурной скобкой оператора **if**.

Область видимости глобальных переменных

Глобальные переменные запоминаются в сегменте данных программы и существуют в течение всего жизненного цикла программы. Вы можете объявлять глобальные переменные в любом месте программы, но за пределами функции **main()** и других функций. Обычно их объявления располагаются непосредственно перед **main()**. Следующий пример

```
#include <stdio.h>

int globalInt;
double globalDouble;
```

```
main()
{
  оператор;
}
```

объявляет две глобальных переменных **globalInt** и **globalDouble**. Любой оператор в любой функции может работать со значениями этих переменных, другими словами, эти переменные имеют глобальную область видимости.

Совет. *Используйте локальные переменные для действий, присущих только данной функции. Для действий, характерных для всей программы, используйте глобальные переменные.*

Функции, которые возвращают значение

Функции могут возвращать некоторое значение операторам, которые их вызвали. Функции такого вида можно сравнить со специализированными калькуляторами, которые решают уравнения, выполняют математические операции и возвращают результаты вычислений в точку вызова.

Целые функции

Листинг 3.3 демонстрирует использование функции, которая возвращает целое значение, представляющее «истину» или «ложь».

Листинг 3.3. quitex.c (функция, возвращающая «истину» или «ложь»)

```
 1: #include <stdio.h>
 2: #include <ctype.h>
 3:
 4: #define FALSE 0
 5: #define TRUE 1
 6: int UserQuits(void);
 7:
 8: main()
 9: {
10:   int i = 0;
11:   int quitting = FALSE;
12:
13:   printf("\nQuit example\n");
14:   while (!quitting) {
15:     i++;
16:     printf("i = %d\n", i);
17:     quitting = UserQuits();
18:   }
19:   return 0;
20: }
21:
22: int UserQuits()
23: {
24:   char c;
```

```
25:
26:    printf("Another value? (y/n) ");
27:    do {
28:      c = toupper(getchar());
29:    } while ((c != 'Y') && (c != 'N'));
30:    return (c == 'N');
31: }
```

Если вам трудно запомнить, что нуль означает «ложь», а любое ненулевое число – «истину», определите символы **TRUE** и **FALSE**, как это делает программа quitex в строках 4-5. Кроме того, эти символы делают текст программы более читабельным.

Например, обратите внимание на строку 11 в функции **main()**. Здесь объявляется целая переменная **quitting**, и одновременно ей присваивается значение **FALSE**. Эта символическая константа проясняет смысл строки и напоминает вам, что переменная **quitting** используется в качестве булевой переменной, принимающей значения «истина» или «ложь».

Совет. *При разборе любых программ с функциями, вместо того чтобы читать строка за строкой, лучше всего начать с функции* ***main()****, а затем, следуя программной логике, перейти на другие ветви. Хорошая идея – отследить работу сложной программы в пошаговом режиме отладчика.*

Когда вы запустите программу quitex, она задаст вопрос, не желаете ли вы увидеть еще одно значение. Если вы введете **Y**, означающее «да», программа отобразит предыдущее значение, увеличенное на единицу. Если же вашим ответом будет **N**, программа завершится. Quitex демонстрирует обычную задачу: прием ответов типа «да» или «нет» на предложенный вопрос, и вы можете использовать этот алгоритм в своих собственных разработках.

Рассмотрите программный цикл **while** в строках 14-18. Работа цикла продолжается, пока значение переменной **quitting** не станет истинным. Внутри цикла, после инкрементирования переменной и отображения ее значения строка 17 присваивает переменной **quitting** результат работы функции **UserQuits()**.

Прототип функции находится в строке 6. Объявление

```
int UserQuits(void);
```

определяет **UserQuits()** как функцию, которая возвращает целое значение и не требует никаких аргументов. Переменная **quitting,** так же как и функция **UserQuits()**, имеет тип **int** и поэтому оператор

```
quitting = UserQuits();
```

непосредственно присваивает переменной **quitting** значение, возвращенное функцией.

Функция **UserQuits()** определена в строках 22-31. Как обычно, заголовок функции (строка 22) дублирует ее прототип, за исключением

завершающей точки с запятой. Тело функции (строки 23-31), заключенное в фигурные скобки, содержит объявление локальной переменной (строка 24) и несколько операторов (строки 26-30), которые отображают подсказку и ожидают вашего ответа. Результат анализируется в строке 29. Только в случае, когда переменная **c** равна '**Y**' или '**N**', цикл **do-while** прекращает ввод и проверку символов, что обеспечивает, таким образом, прием только разрешенных ответов.

В строке 30, относящейся к функции **UserQuits()**, выполняется возврат результата функции. Оператор

```
return (c == 'N');
```

вычисляет выражение в круглых скобках, принимая значение «истина» или «ложь». Если значение не равно '**N**', возвращается нуль («ложь»), в противном случае возвращается ненулевое значение («истина»).

Замечание. *Все функции, которые возвращают значения, должны выполнять, по крайней мере, один оператор **return**, иначе компилятор выдаст предупреждение: «Functions should return a value…» («Функции должны возвращать значение…»). Не игнорируйте это предупреждение! Функции, которые завершаются без явного возврата значения, на самом деле возвращают значение, выбранное случайным образом, и поэтому такая функция может нанести вред.*

Функции с плавающей запятой

Листинги 3.4 и 3.5 демонстрируют нисходящий метод разработки на примере программы обслуживания меню. Кроме того, программа показывает, как писать и использовать функции с плавающей запятой. Metrics является незаконченной программой – хороший пример нисходящего программирования в стадии разработки. Программа синтаксически полна, и вы можете ее скомпилировать и запускать, но работает только ее первая команда. Рассмотрим сначала заголовочный файл metrics.h.

Замечание. *Файл metrics.h не является полной программой – вы не сможете скомпилировать и запустить ее. Скомпилируйте листинг metrics.c, который использует заголовочный файл metrics.h.*

Листинг 3.4. metrics.h (заголовочный файл для metrics.c)

```
 1: #include <conio.h>
 2:
 3: #define FALSE 0
 4: #define TRUE 1
 5: #define CENT_PER_INCH 2.54;
 6:
 7: /* Прототипы функций */
 8:
 9: void DisplayMenu(void);
10: int MenuSelection(void);
```

```
11: double GetValue(void);
12: void InchesToCentimeters(void);
```

Структурируйте ваши программы, запоминая директивы типа **#include** и **#define** в заголовочном файле, подобном metrics.h. Это также хорошее место для размещения прототипов функций, как это сделано в строках 9-12. Все эти строки можно размещать и в программном модуле (например metrics.c), но тогда они будут доступны только в этом файле. Имея отдельный заголовочный файл, несколько модулей могут получить доступ к объявлениям и прототипам с помощью директивы

```
#include "metrics.h"
```

Заголовочный файл metrics.h объявляет четыре прототипа функции. Первая и последняя – **DisplayMenu()** и **InchesToCentimmeters()** – являются функциями типа **void**. Вторая – **MenuSelection()** – возвращает значение типа **int**, представляющее выбранную команду. Третья – **GetValue()** – возвращает значение с плавающей запятой типа **double**. Программа metrics вызывает функцию **GetValue()**, чтобы предложить вам ввести значения для преобразования из одной системы в другую. Ниже представлен основной листинг.

Листинг 3.5. metrics.c (программа обслуживания меню)

```
 1: #include <stdio.h>
 2: #include "metrics.h"
 3:
 4: main()
 5: {
 6:   int quitting = FALSE;
 7:
 8:   printf("Welcome to Metrics\n");
 9:   while (!quitting) {
10:     DisplayMenu();
11:     switch(MenuSelection()) {
12:       case 1:
13:         InchesToCentimeters();
14:         break;
15:       case 9:
16:         quitting = TRUE;
17:         break;
18:       default:
19:         printf("\nSelection error!\n");
20:     }
21:   }
22:   return 0;
23: }
24:
25: /* Описание функций */
26:
27: void DisplayMenu(void)
```

```
28: {
29:   printf("\nMenu\n");
30:   printf("----\n");
31:   printf("1 - Inches to centimeters\n");
32:   printf("2 - Centimeters to inches\n");
33:   printf("3 - Feet to meters\n");
34:   printf("4 - Meters to feet\n");
35:   printf("5 - Miles to kilometers\n");
36:   printf("6 - Kilometers to miles\n");
37:   printf("9 - Quit\n");
38: }
39:
40: int MenuSelection(void)
41: {
42:   printf("\nSelection? (Don't press ENTER!): ");
43:   return (getche() - '0');
44: }
45:
46: double GetValue(void)
47: {
48:   double value;
49:
50:   printf("\nValue to convert? ");
51:   scanf("%lf", &value);
52:   return value;
53: }
54:
55: void InchesToCentimeters(void)
56: {
57:   double value;
58:   double result;
59:
60:   printf("\nInches to Centimeters\n");
61:   value = GetValue();
62:   result = value * CENT_PER_INCH;
63:   printf("%.3lf inches = %.3lf cents\n", value, result);
64: }
```

Строка 1 содержит уже привычный заголовочный файл stdio.h, а строка 2 – metrics.h – заголовочный файл из листинга 3.4.

Чтобы понять, как работает metrics, рассмотрим операторы функции **main()**. Оператор **while** в строке 9 проверяет флажок **quitting**, повторяя цикл до тех пор, пока его значение не станет истинным. Строка 10 вызывает функцию **DisplayMenu()**, которая (как нетрудно догадаться) отображает меню команд. Хорошо подобранные имена функций могут немало сообщить о том, что скрывается за их «вывеской».

В строках 11-20 разместился оператор **switch**. Выражение в заголовке этого оператора вызывает функцию **MenuSelection()**, чтобы получить информацию о выборе пользователя. Затем полученное значение

сравнивается с заданными значениями селекторов **case**. При выборе значения 1 вызывается функция **InchesToCentimeters()** (строка 13); при выборе 9 – значение флажка **quitting** устанавливается равным «истине» в строке 16; наконец, для всех остальных значений отображается сообщение об ошибочном вводе (строка 19). Пропущенные значения выбора 2-8 в операторе **switch** иллюстрируют метод нисходящего программирования. Несмотря на то что программа metrics не завершена, отдельные ее части уже можно выполнять и тестировать.

Функции программы описаны в строках 27-64. Следующие замечания поясняют некоторые не вполне ясные моменты листинга.

- Функция **DisplayMenu()** (строки 27-38) выполняет ряд операторов **printf()** для отображения меню программы. Управляющие символы '**\n**' в строке 29 гарантируют, что меню начнется с новой строки.
- Функция **MenuSelection()** (строки 40-44) предлагает пользователю выбрать команду меню. Для получения кода нажатой клавиши вызывается объявленная в файле conio.h функция **getche()**, которая не требует от пользователя нажатия <Enter>. Вычитание ASCII-значения символа цифры '**0**' в строке 43 (**getche() – '0'**) преобразует введенный символ в десятичное значение в диапазоне от 0 до 9. Рассмотрим, как это происходит. ASCII-значение нуля равно 48. Таким образом, если пользователь введет символ '**0**', то выражение '**0**' – '**0**' (48 – 48) дает 0. Поскольку значения ASCII-цифр возрастают последовательно, '**1**' – '**0**' (49 – 48) дает 1, '**2**' – '**0**' (50 – 48) равно 2 и т.д.
- Функция **GetValue()** (строки 46-53) предлагает пользователю ввести значение для последующего преобразования, вызывая функцию **scanf()**, чтобы прочесть число с плавающей точкой в переменную **value**. Строка 52 возвращает это значение в качестве результата работы функции.
- Функция **InchesToCentimeters()** (строки 55-64) в строке 61 вызывает функцию **GetValue()**, присваивая возвращаемое ею значение переменной **value**. Другой переменной **result** присваивается результат преобразования исходного значения из дюймов в сантиметры. Оба значения отображаются на экране с помощью функции **printf()** в строке 63.

Упражнение. *Завершите листинг 3.5. Используйте нисходящий метод программирования, т.е. добавляйте команды постепенно, сопровождая этот процесс проверкой их работы, пока полностью не завершите программу. (Подсказка: в одном сантиметре 0,3937 дюйма, в одном фунте 0,3048 метра, в одном метре 3,28084 фута, в одной миле 1,609 километра, в километре 0,621 мили.)*

Замечание. *Нисходящее программирование позволяет локализовать ошибки в программе на начальном этапе ее создания. Не откладывайте отладку до тех пор, когда размеры создаваемой программы станут угрожающе большими. Выполняйте тестирование по мере продвижения вперед,*

убеждаясь в том, что каждая законченная часть работает, как задумано, и только потом переходите к следующей.

Другие типы функций

Функции могут возвращать значения любого типа из описанных в главе 1. Например, вы можете объявить функцию типа **long int** следующим образом:

```
long AnyLongFn(void);  /* long и long int – синонимы */
```

Или у вас может быть функция, возвращающая значение типа **unsigned long** и объявленная как

```
unsigned long AnyUnsignedLongFn(void);
```

Большинству функций с плавающей запятой следовало бы возвращать значение типа **double**, так как оно точнее **float** и занимает меньше места, чем **long double**. Хотя вы, конечно, можете объявлять функции следующих типов:

```
float AnyFloatFn(void);
double AnyDoubleFn(void);
long double AnyLongDoubleFn(void);
```

Функции также могут возвращать строки и указатели, о которых речь пойдет в следующих главах.

Распространенные ошибки в функциях

Причиной некорректной работы вашей функции может служить одна из следующих распространенных ошибок.

- *No return* (нет оператора **return**). Компилятор делает подобное предупреждение относительно любой возвращающей значение функции (т.е. отличной от типа **void**), не содержащей оператора **return**. Если подобная функция завершается без выполнения **return**, она возвращает непредсказуемое значение, которое может вызвать серьезные неприятности.
- *Skipped return* (невыполнимый оператор **return**). Компилятор предупреждает о функциях, которые не смогут выполнить оператор **return** ни при каких условиях. Обратите особое внимание на функции, имеющие операторы **if**, и убедитесь, что оператор **return** выполняется для каждого возможного случая выхода.
- *No prototype* (нет прототипа). Считается, что функции, у которых нет прототипа, имеют тип **int**, даже если они определены как возвращающие значения другого типа. Не следует полагаться на это правило, лучше явно объявлять прототипы для всех функций.
- *Side effect* (побочный эффект). Эта проблема обычно вызвана функциями, которые изменяют значения глобальных переменных. Поскольку функции можно вызывать из выражений и другие операторы могут использовать те

же самые переменные, то изменение значений глобальных переменных может вызвать трудно обнаруживаемые ошибки.

Первые два типа перечисленных выше ошибок, отсутствие и обход оператора **return**, легко устраняются при внимательном отношении к предупреждениям компилятора. Следующая запись иллюстрирует самую распространенную ошибку:

```
int AnyIntFn(void)
{
  if (условие) {
    оператор1;
    return 0;
  } else
    оператор2;    /* ??? */
}
```

Эта функция выполняет оператор **return**, только если *условие* истинно. Если же *условие* ложно, *оператор2* выполнится нормально, но функция завершится без выполнения оператора **return**.

Третья из представленного выше списка распространенных ошибок (отсутствие прототипа) восходит к старым С-программам. Функции без прототипов рассматриваются как имеющие тип **int**. Таким образом, вы могли бы без отрицательных последствий убрать прототипы, подобные представленным в строке 6 листинга 3.3, но делать это не рекомендуется.

Замечание. *Функция **main()**, обычно записываемая без явного задания возвращаемого типа, на самом деле возвращает значение **int**, и, следовательно, ее можно объявить как **int main()**. Этим объясняется, почему функция **main()** должна выполнять оператор **return**, чтобы избежать предупреждения компилятора.*

Из всех ошибок, допускаемых в функциях, наиболее коварны побочные эффекты. Изменяйте значения глобальных переменных внутри функций с большой осторожностью.

Параметры и аргументы функций

Функции могут принимать входные значения при их вызове. Предположим, вам надо вычислить куб от значения переменной **r**, объявленной как **double**. Чтобы получить результат, вы можете использовать выражение вида

```
r = r * r * r;
```

Предположим, что вам нужно вычислять третью степень для многих переменных. Не стоит усложнять и загромождать программу, используя это выражение в разных местах, где требуется выполнить вычисление. Функции как раз и являются тем идеальным средством, которое поможет вам при выполнении повторяющихся операций. Вы можете объявить функцию, вычисляющую третью степень аргумента, следующим образом:

```
double Cube(double r);
```

Данная функция возвращает значение типа **double**. Определение в круглых скобках задает значение типа **double** с именем **r**, которое называется параметром. Реализация функции может иметь следующий вид:

```
double Cube(double r)
{
  return (r * r * r);
}
```

Если **x** и **y** – переменные типа **double**, то с помощью следующего оператора переменной **y** можно присвоить третью степень **x**:

```
y = Cube(x); /* y = x * x * x */
```

Переменная **x** называется аргументом. Ее значение передается параметру **r** в момент вызова функции **Cube()**.

Замечание. *Некоторые авторы называют параметры функции «формальными параметрами», а аргументы, передаваемые в функцию, – «фактическими параметрами». Вы можете встретить эти устаревшие термины в различных изданиях.*

Функции могут объявлять несколько параметров, в этом случае операторы могут передавать им несколько аргументов. Рассмотрим проблему вычисления стоимости электроэнергии на производстве. При известном тарифе (стоимости за киловатт в час), времени и мощности потребления стоимость электроэнергии в тысячах рублей равна:

```
cost = (rate * power * time) * 0.001;
```

Представить эту формулу в виде функции можно так:

```
double Cost(double time, double power, double rate);
```

Функция **Cost()** объявляет три параметра – **time**, **power** и **rate** – типа **double**. Параметры разделяются запятыми. Возвращает функция тоже значение типа **double**:

```
double Cost(double time, double power, double rate);
{
  return (rate * power * time) * 0.001;
}
```

Если переменная **result** имеет тип **double**, то простой вызов функции вычислит стоимость электроэнергии в тысячах рублей при мощности, равной 100 кВт, времени потребления, равном 10 часам, и тарифе 47.5:

```
result = Cost(10.0, 100.0, 47.5);
```

При вызове функции параметры получают копии значений переданных аргументов, используя механизм, подобный операторам присваивания. Другими словами, функция начинается так, как будто сначала выполняются следующие операторы:

```
time = 10.0;
power = 100.0;
rate = 47.5;
```

Подобно локальным переменным, параметры функции **Cost()** (**time**, **power**, **rate**) запоминаются в стеке и обрабатываются аналогичным образом с единственным различием – они инициализируются значениями передаваемых функции аргументов.

Это очень важная концепция. Запишите функцию **Cost()** следующим образом**:**

```
double Cost(double time, double power, double rate)
{
  power = 150.51;  /* ??? */
  return (rate * power * time) * 0.001;
}
```

При этом значение **150.51** не будет передано обратно аргументу, с которым была вызвана функция, т.к. присваивание выполнится только для локальной переменной **power**. Другими словами, если **q**, **x**, **y**, **z** – переменные типа **double**, то оператор

```
q = Cost(x, y, z);
```

гарантирует неприкосновенность значений переменных **x**, **y** и **z**. Значения этих аргументов копируются в параметры функции, а сами аргументы остаются неизменными.

Листинг 3.6 рассчитывает стоимость электроэнергии с помощью функции **Cost()**.

Листинг 3.6. electric.c (отображение таблицы стоимости электроэнергии)

```
 1: #include <stdio.h>
 2: #include <ctype.h>
 3: #include <conio.h>
 4:
 5: #define MAXROW 8    /* число строк в таблице */
 6: #define MAXCOL 6    /* число столбцов в таблице */
 7:
 8: /* Прототипы функций */
 9:
10: void Initialize(void);
11: double Cost(double time, double power, double rate);
12: void PrintTable(void);
13: int Finished(void);
14:
15: /* Глобальные переменные */
16:
17: double startHours;
18: double hourlyIncrement;
19: double startWatts;
20: double wattsIncrement;
21: double costPerKwh;
22:
23: main()
24: {
```

```
25:   do {
26:     Initialize();
27:     PrintTable();
28:   } while (!Finished());
29:   return 0;
30: }
31:
32: void Initialize(void)
33: {
34:   printf("Cost of electricity\n\n");
35:   printf("Starting number of hours....? ");
36:   scanf("%lf", &startHours);
37:   printf("Hourly increment.....? ");
38:   scanf("%lf", &hourlyIncrement);
39:   printf("Starting number of KWatts...? ");
40:   scanf("%lf", &startWatts);
41:   printf("KWatts increment.....? ");
42:   scanf("%lf", &wattsIncrement);
43:   printf("Cost per kilowatt hour (KWH)? ");
44:   scanf("%lf", &costPerKwh);
45: }
46:
47: double Cost(double time, double power, double rate)
48: {
49:   return (rate * power * time) * 0.001;
50: }
51:
52: void PrintTable(void)
53: {
54:   int row, col;
55:   double hours, watts;
56:
57:   /* Печать верхней строки таблицы */
58:   printf("\nHrs/KWatts");
59:   watts = startWatts;
60:   for (col = 1; col <= MAXCOL; col++) {
61:     printf("%10.0lf", watts);
62:     watts+= wattsIncrement;
63:   }
64:   /* Печать строк таблицы */
65:   hours = startHours;
66:   for (row = 1; row <= MAXROW; row++) {
67:     printf("\n%7.1lf - ", hours);
68:     watts = startWatts;
69:     for (col = 1; col <= MAXCOL; col++) {
70:       printf("%10.2lf", Cost(hours, watts, costPerKwh));
71:       watts+= wattsIncrement;
72:     }
73:     hours += hourlyIncrement;
74:   }
75:   printf("\nCost of electricity %.2lf per KWH\n",costPerKwh);
76: }
```

```
77:
78: int Finished(void)
79: {
80:   int answer;
81:
82:   printf("\nAnother table (y/n) ?");
83:   answer = getch();
84:   putchar(answer);
85:   putchar('\n');
86:   return (toupper(answer) != 'Y');
87: }
```

Скомпилируйте и запустите эту программу. В ответ на приглашение введите начальное значение количества часов (100 – хорошее число) и приращение по времени (например 24), начальное значение количества киловатт (попробуйте 2000) и приращение по киловаттам (возьмите число 2), а также стоимость одного киловатта (например 47.5).

Electric – это самая сложная программа среди тех, что вы встречали до сих пор. Но несмотря на свой размер, она построена в том же ключе, что и предыдущие примеры. Строки 1-3 содержат заголовочные файлы, объявляющие прототипы стандартных функций и другие необходимые для программы элементы. В строках 5-6 объявляются две символические константы – количество строк и столбцов в программе. Чтобы изменить формат таблицы, достаточно изменить эти константы, вместо того чтобы искать по всей программе операторы, связанные с выводом таблицы.

Строки 10-13 объявляют четыре прототипа функции. Хорошо подобранные имена проясняют назначение функций. Функция **Initialize()** выполняет различные действия в начале построения каждой таблицы. С функцией **Cost()** вы уже знакомы. Функция **PrintTable()** отображает результат работы программы (выводит таблицу). И, наконец, функция **Finished()** возвращает значение «истина», если пользователь не требует вывода другой таблицы.

В строках 17-21 объявляются глобальные переменные, которые доступны в любом месте программы.

Замечание. *Часто бывает трудно принять решение, какой должна быть переменная – глобальной или локальной. Переменные, содержащиеся в строках 17-21, – глобальные по своей природе, так как они влияют на конечный результат. Переменные, имеющие более узкое назначение, например те, которые управляют циклами* ***for*** *или* ***while****, не следует делать глобальными.*

Функция **main()** в этой программе коротка и опрятна – хороший пример того, как функции могут упрощать сложные программы. В строках 25-28 выполняется цикл **do-while**, пока функция **Finished()** не вернет

истинное значение. В цикле вызываются функции **Initialize()** и **PrintTable()** (строки 26-27).

Функция **Initialize()** (строки 32-45) выводит название программы и предлагает пользователям ввести значения, запоминаемые в глобальных переменных. Функция **PrintTable()** (строки 52-76) использует цикл **for** для вывода строк и столбцов таблицы. Оператор в строке 67

```
printf("\n%7.1lf - ", hours);
```

выводит значение времени в формате с плавающей запятой, занимающее семь позиций, причем одна позиция отводится для десятичного знака после запятой. Оператор в строке 70

```
printf("%10.2lf", Cost(hours, watts, costPerKwh));
```

вызывает функцию **Cost()**, передавая локальные переменные **hours**, **watts** и глобальную переменную **costPerKwh** в качестве аргументов. Оператор **printf()** отображает возвращаемое функцией **Cost()** значение в десяти позициях с двумя десятичными знаками после запятой.

Последняя функция в программе, **Finished()**, занимающая строки 78-87, предлагает ответить на вопрос «быть или не быть» следующей таблице («**Another table (y/n) ?**»). В строке 83 вызывается функция **getch()**, принимающая символ ответа пользователя, который затем отображается на экране операторами:

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

Функция **putchar()** полезна для отображения одиночных символов. Вторая строка выводит символ '**\n**', вызывающий переход на новую строку.

Безымянные параметры

Прототип функции **Cost()** из предыдущего раздела мог бы быть объявлен с использованием безымянных параметров:

```
double Cost(double, double, double);
```

Сравните это объявление с прототипом, используемым в строке 11 листинга 3.6:

```
double Cost(double time, double power, double rate);
```

Эти две формы записи эквивалентны, поскольку компилятор игнорирует имена параметров **time**, **power**, **rate** в объявлении функции. Чтобы сгенерировать вызов функции **Cost()**, компилятору достаточно знать только типы данных параметров, а имена присутствуют лишь для вашего удобства и удобства тех, кто будет читать программу.

Однако в отличие от прототипа определение функции должно содержать имена своих параметров, чтобы операторы функции могли к ним обращаться:

```
double Cost(double time, double power, double rate);
{
```

```
  return (rate * power * time) * 0.001;
}
```

Упражнение. *Добавьте параметры в функции в листинге 3.1. В вашей программе оператор* ***CountUp(10)*** *должен отображать числа от 1 до 10; оператор* ***CountDown(100)*** *должен отображать числа от 100 до 1 и т.д.*

Рекурсия: то что сворачивается, должно разворачиваться

Слово *рекурсия* буквально означает возврат. Когда вы смотрите в зеркало, имея позади себя другое зеркало, то видите бесконечно повторяющуюся серию изображений. Каждое изображение является отражением (рекурсией) света от предыдущего.

В программировании говорят о рекурсии, когда функция вызывает саму себя несколько раз. Адреса возврата таких вызовов запоминаются в стеке подобно отражениям в зеркалах, пока какое-нибудь событие не остановит этот процесс.

Некоторые алгоритмы рекурсивны по своей природе. Например, факториал числа равен произведению предшествующих ему последовательных чисел. Так, факториал числа 5 равен 1 * 2 * 3 * 4 * 5 = 120. В общем случае факториал n равен произведению n на факториал n – 1. Учитывая этот факт, вы можете написать рекурсивную функцию факториала следующим образом:

```
double Factorial(int number)
{
  if (number > 1)
    return number * Factorial(number - 1);
  return 1;
}
```

Функция **Factorial()** возвращает значение типа **double** и объявляет единственный параметр **number** типа **int**. Вначале оператор **if** проверяет значение параметра **number**. Если **number** больше единицы, функция возвращает значение **number**, умноженное на факториал **number – 1**. Таким образом, функция **Factorial()** вызывает саму себя со значением аргумента меньшим на единицу. Когда **number** станет равен единице, выполнится оператор **return 1** (факториал единицы равен 1), и рекурсия начнет «разворачиваться» (рис. 3.2).

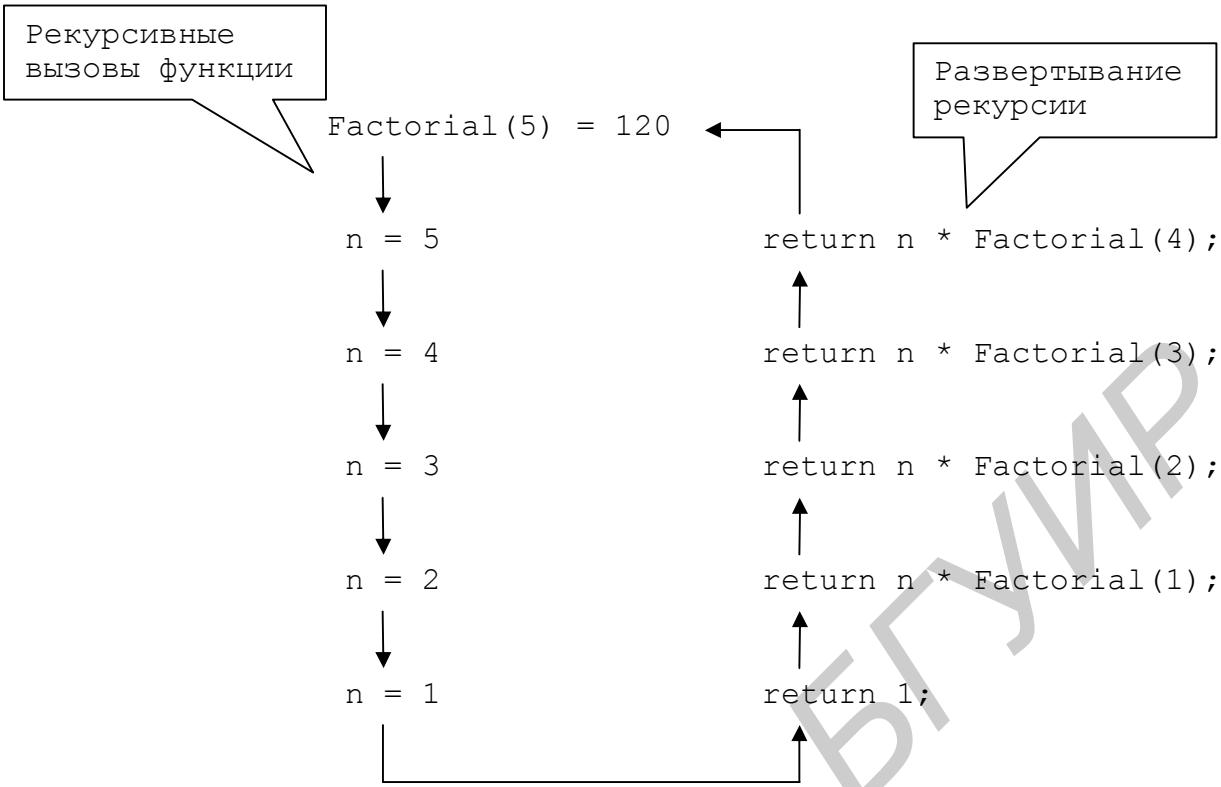

Рис. 3.2. Рекурсивные вызовы функции для выражения Factorial(5)

Замечание. *Используйте пошаговый режим отладчика, чтобы лучше понять, как работает рекурсия.*

Следующая простая программа демонстрирует ключевые особенности рекурсии. Введите листинг 3.7, скомпилируйте и запустите его. На экране вы увидите счет значений от 1 до 10. Прежде чем продолжить чтение, постарайтесь понять, как программа recount использует рекурсию, чтобы справиться с этой задачей.

Листинг 3.7. recount.c (счет от 1 до 10 с использованием рекурсии)

```
 1: #include <stdio.h>
 2:
 3: void Recount(int top);
 4:
 5: main()
 6: {
 7:   Recount(10);
 8:   return 0;
 9: }
10:
11: void Recount(int top)
12: {
13:   if (top > 1)
14:     Recount(top - 1);
15:   printf("%4d", top);
16: }
```

В строке 7 вызывается функция **Recount()**, которая использует рекурсию, чтобы посчитать от 1 до 10 (никто не станет спорить, это нелегкий путь для решения простой задачи).

В строке 13 проверяется значение параметра **top**. Если оно больше единицы, функция рекурсивно вызывает саму себя в строке 14, передавая **top – 1** в качестве нового аргумента. Когда значение **top** уменьшится до нуля, строка 15 отобразит значение этой переменной.

Заметьте, что на экране отображается десять значений, стало быть, оператор **printf()** выполнился десять раз. Это показывает, что каждый рекурсивный вызов завершается возвратом к точке вызова, выполняя оператор **printf()** один раз для каждого вызова функции **Recount()**.

Если рекурсивная функция объявляет какие-нибудь локальные переменные, то эти переменные повторно создаются на каждом уровне рекурсии. Например, если функция **A()** объявляет локальную переменную **i**, то, когда **A()** рекурсивно вызывает саму себя, в стеке создается совершенно новая переменная **i**.

Замечание. *Все рекурсивные функции могут быть записаны без рекурсии, хотя не всегда так просто. Рекурсия – это дорогое удовольствие. Слишком большое количество вызовов функций может истощить стековую память, вызвав ошибку переполнения. Кроме того, вызовы функций отнимают много времени, поэтому нерекурсивные алгоритмы обычно работают быстрее.*

Упражнение. *Напишите нерекурсивную версию функции вычисления факториала.*

Резюме

- Нисходящий метод – естественный способ решения больших проблем путем разбиения их на более мелкие, которые легче решить. Применяйте этот метод в программировании, разделяя задачи на подзадачи, а те, в свою очередь, представляя в виде узкоспециализированных функций.
- Функции могут возвращать значения любого типа. Функции типа **void** не возвращают значений.
- Переменные, объявленные внутри функций, имеют локальную область видимости и доступны операторам только внутри тех же самых функций.
- Локальные переменные запоминаются в стеке, и им отводится память каждый раз при вызове функции. Локальные переменные не сохраняют своих значений между вызовами функций, в которых они объявлены.
- Глобальные переменные объявляются вне функций и предварительно инициализируются нулевыми значениями.
- Глобальные переменные «видимы» для всех операторов в программе. Они существуют, пока выполняется программа.

- Функции могут объявлять параметры. Операторы передают параметрам значения в виде аргументов, обеспечивая функциям входные данные.
- Рекурсией называется процесс, при котором функция вызывает саму себя. Некоторое событие (условие) должно обязательно останавливать следующие друг за другом рекурсивные вызовы, иначе будет вызвана ошибка переполнения стека.
- Некоторые самоссылочные алгоритмы (например вычисление факториала) удобно программировать с помощью рекурсии. Однако рекурсивные функции обрабатываются медленнее и занимают больше стековой памяти, чем их нерекурсивные эквиваленты.

Обзор функций

Таблица 3.1

Функции работы с терминалом в текстовом режиме

Функция	**Прототип и краткое описание**
clreol()	`void clreol(void);` Стирает символы на экране от позиции курсора до конца строки.
clrscr()	`void clrscr(void);` Очищает экран.
getch()	`void getch(void);` Считывает один символ с клавиатуры без отображения на экране. Нажатие <Enter> не требуется.
getche()	`void getche(void);` Считывает один символ с клавиатуры, отображая его на экране. Нажатие <Enter> не требуется.

Замечание. *Функции из табл. 3.1 поддерживаются только в среде MS-DOS.*

4. Массивы

Представьте себе программу обработки базы данных, которая запоминает всю информацию в переменных. Какая была бы путаница! Такие переменные, как **alexAddress** или **iraSalary**, засорили бы всю программу, сделав выполнение операций типа сортировки и печати архисложными.

К счастью, в языке С есть *массивы*, которые помогут организовать данные быстро и эффективно.

Введение в массивы

В массивах объединяются элементы одного и того же типа. У вас может быть массив целых чисел, массив значений с плавающей запятой или массив символов. Массив может состоять из одного элемента, а может запомнить столько, сколько позволяет объем оперативной памяти.

Объявление

```
int scores[100];
```

определяет массив с именем **scores**, способный хранить 100 значений типа **int**. Если **scores** – глобальная переменная, то элементы массива с именем **scores** инициализируются нулями. Если массив локален (т.е. объявляется внутри какой-либо функции), то его значения не инициализируются и содержат случайные значения.

Одно только имя массива **scores** (без скобок) представляет весь массив как единое целое. Чтобы получить доступ к конкретному элементу, после имени массива добавьте индекс в квадратных скобках. Например, оператор

```
scores[5] = 89;
```

присваивает значение **89** шестому элементу массива **scores**. Почему шестому? Потому что первый элемент массива имеет индекс 0. Оператор

```
scores[0] = 1000;
```

присваивает первому элементу массива круглую сумму.

Замечание. *Поскольку первый индекс массива равен нулю, то для любого объявления вида **T name[N]**, (где **T** – тип данных) допустимые значения индексов находятся в диапазоне от 0 до N – 1. Использование индекса вне этого диапазона (например **name[-2]**) приведет к ссылке на область памяти, не принадлежащую массиву, что может вызвать серьезные ошибки.*

Элементы массива запоминаются в памяти последовательно. Как показано на рис. 4.1, массив напоминает штабель ящиков. Выражение **scores[0]** представляет первый ящик. Сразу за ним в памяти располагается элемент **scores[1]**. Затем идет **scores[2]**, **scores[3]** и т.д. Последним элементом является **scores[99]**.

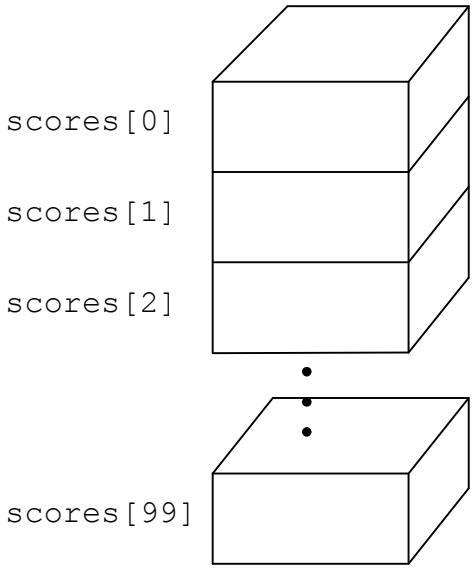

Рис. 4.1. Массив похож на штабель ящиков

Индексами массивов могут служить любые целые выражения – константы, переменные, результаты функций и т.д. Если у вас объявлена целая переменная **i**, то можно использовать следующие операторы для отображения шестого элемента массива **scores**:

```
i = 5;
printf("score = %d\n", scores[i]);
```

Чтобы обработать все элементы массива обычно используется цикл **for**. Перед вами один из способов отображения значений всех элементов массива **scores**:

```
for (i = 0; i < 100; i++)
  printf("scores[%d] = %d\n",
     i, scores[i]);
```

Инициализация массивов

Чтобы проинициализировать массив начальными значениями, объявите массив, как обычно, но после объявления в фигурных скобках напишите список значений через запятую. Например, объявление

```
int digits[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

создает 10-элементный массив целых чисел с именем **digits** и последовательно присваивает значения от 0 до 9 каждому элементу массива. Результат действия этого объявления аналогичен результату работы следующих операторов:

```
int i, digits[10];          /* индекс i и массив из 10 целых */
for (i = 0; i < 10; i++)
  digits[i] = i;
```

Вы даже можете заставить компилятор автоматически вычислять размер массива. Объявление

```
int digits[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

создает массив из 10 целых, как и раньше. Но в этом случае пустые скобки заставляют компилятор выделить ровно столько памяти, сколько необходимо для хранения перечисленных элементов.

При явном задании размера массива в квадратных скобках совершенно не обязательно задавать значение для каждого элемента массива. Например, объявление

```
int digits[10] = {0, 1, 2, 3, 4};
```

создает массив из 10 целых, но инициализирует только первые пять.

Если количество значений в фигурных скобках больше размера массива, будет сгенерирована ошибка. Например, если компилятор «накормить» таким объявлением

```
int digits[5] = {0, 1, 2, 3, 4, 5, 6};
```

то он «поперхнется» и выдаст: «Too many initializers» («Слишком много инициализаторов»).

Листинг 4.1 имитирует бросание пары игральных костей и запоминает результаты в массиве. Затем программа использует стандартный статический метод хи-квадрат, в качестве эталонного теста генератора случайных чисел. Скомпилируйте и запустите программу. По приглашению сообщите программе, сколько «бросков» сымитировать – лучше между 10,000 и 50,000.

Листинг 4.1. dice.c (оценка генератора случайных чисел)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <time.h>
 4:
 5: void Initialize(void);
 6: long GetNumThrows(void);
 7: void ThrowDice(long numThrows);
 8: double sqr(double r);
 9: double ChiSquare(long numThrows);
10: void DisplayResult(long numThrows);
11:
12: #define MAX 13 /*индексы от 0 до 12; 0 и 1 не используются*/
13:
14: double count[MAX];
15: double prob[MAX] = {
16:   0.0, 0.0, 1.0 / 36, 1.0 / 18, 1.0 / 12, 1.0 / 9, 5.0 / 36,
17:   1.0 / 6, 5.0 / 36, 1.0 / 9, 1.0 / 12, 1.0 / 18, 1.0 / 36
18: };
19:
20: main()
21: {
22:   long numThrows;
23:
24:   printf("\nDice - A random number benchmark\n\n");
25:   numThrows = GetNumThrows();
26:   if (numThrows > 0) {
27:     ThrowDice(numThrows);
28:     DisplayResult(numThrows);
29:   }
30:   return 0;
31: }
32:
33: long GetNumThrows(void)
34: {
35:   long answer;
36:
```

```
37:   printf("How many throws? ");
38:   scanf("%ld", &answer);
39:   return answer;
40: }
41:
42: void ThrowDice(long numThrows)
43: {
44:   long i;
45:   int k;
46:
47:   randomize();
48:   for (i = 1; i <= numThrows; i++) {
49:     k = (1 + random(6)) + (1 + random(6));
50:     count[k]++;
51:   }
52: }
53:
54: double sqr(double r)
55: {
56:   return r * r;
57: }
58:
59: double ChiSquare(long numThrows)
60: {
61:   double v = 0.0;
62:   int i;
63:
64:   for (i = 2; i < MAX; i++)
65:     v += (sqr(count[i])) / prob[i];
66:   return ((1.0 / numThrows) * v) - numThrows;
67: }
68:
69: void DisplayResult(long numThrows)
70: {
71:   int i;
72:
73:   printf("\n Dice    Proba-      Expected   Actual  \n");
74:   printf(" Value   bility       Count     Count  \n");
75:   printf("=========================================\n");
76:   for (i = 2; i < MAX; i++) {
77:     printf("%5d%10.3lf%12.0lf%10.0lf\n",
78:       i, prob[i], prob[i] * numThrows, count[i]);
79:   }
80:   printf("\nChi square = %lf\n", ChiSquare(numThrows));
81: }
```

Строки 14 и 15-18 объявляют два массива, **count** и **prob**, каждый из которых может хранить 13 значений типа **double**. В строке 12 определяется символическая константа **MAX**, поэтому другие операторы могут легко ограничить индексы диапазоном от 0 до **MAX – 1**. (Поскольку значения,

даваемые двумя игральными костями, находятся в диапазоне от 2 до 12, позиции 0 и 1 в массивах не используются.)

Строки 15-18 объявляют и предварительно инициализируют массив **prob** вероятностями для каждой возможной комбинации костей. Существует три варианта выпадения значения 4: 3 + 1, 2 + 2 и 1 + 3. Поскольку существует 36 (6 * 6) различных пар выпадения костей, вероятность выпадения 4 равна 3/36 (3 комбинации из 36) или 1/12. Выражения в строках 16-17 вычисляют подобные вероятности. Индекс означает число очков. Так, элемент **prob[4]** содержит вероятность выпадения четырех очков.

Функция **ThrowDice()** (строки 42-52) имитирует выбрасывание костей. Чтобы задать начальное значение для генератора случайных чисел, в строке 47 вызывается функция **randomize()**, прототип которой объявлен в файле time.h. Функция использует значение текущего времени для начальной установки генератора, что гарантирует различную последовательность значений при каждом запуске программы. Если вы хотите для каждого запуска программы получать одну и ту же последовательность случайных чисел, закомментируйте функцию **randomize()** – это полезный способ стабилизации результатов программы, который может помочь при отладке программ, в которых используются случайные числа.

Каждая итерация цикла **for** в строках 48-51 имитирует встряску и выбрасывание пары костей. В строке 49 переменной **k** присваивается выражение

```
(1 + random(6)) + (1 + random(6))
```

Выражение **random(N)** возвращает случайное целое число в пределах от 0 до N – 1. Таким образом, выражение **random(6)** возвращает значение между 0 и 5. Прибавление единицы (**1 + random(6)**) даст значение в диапазоне от 1 до 6, что соответствует количеству очков при выбрасывании одной игральной кости. В этой программе функция **random()** вызывается дважды, чтобы дать значения в диапазоне от 2 до 12. Выражение **2** + **random(11)** также сгенерировало бы значения в этом диапазоне, однако все они стали бы равновероятны.

В строке 50 выполняется выражение **count[k]++**, которое инкрементирует значение **k**-го элемента массива **count**. Индекс **k** равен количеству очков при текущем выбрасывании костей. После окончания цикла **for** массив **count** будет хранить количество выпадений для каждого возможного случая.

При выполнении программы будут отображены три значения: вероятность, ожидаемое количество выпадений и реальное количество из массива **count**. На рис. 4.2 показаны результаты работы программы для 50,000 выбрасываний костей – эквивалент одной недели беспрерывной игры в казино (или двух недель, если при этом еще и спать).

```
How many throws? 50000

  Dice       Proba-      Expected      Actual
 Value       bility       Count        Count
=========================================
    2         0.028         1389         1310
    3         0.056         2778         2773
    4         0.083         4167         4264
    5         0.111         5556         5513
    6         0.139         6944         7015
    7         0.167         8333         8315
    8         0.139         6944         6878
    9         0.111         5556         5582
   10         0.083         4167         4183
   11         0.056         2778         2787
   12         0.028         1389         1380
Chi square = 8.759116
```

Рис. 4.2. Результаты работы программы dice

Как видно из рис. 4.2, а может быть, и из вашего личного опыта, реальные числа слегка отличаются от ожидаемых значений. Это не должно вызывать удивления – в конце концов, если бы игра в кости была полностью предсказуемой, вряд ли многочисленные казино получали прибыль.

Чтобы проанализировать результат работы генератора случайных чисел и определить, попадают ли полученные числа в пределы допустимых отклонений от ожидаемых значений, применяется критерий хи-квадрат. Для получения численного выражения критерия используется следующая формула:

$$V = \frac{1}{n}\sum_{i=1}^{K}\left(\frac{\sigma_i^2}{p_i}\right) - n\,.$$

Эта формула определяет хи-квадрат как единицу, деленную на число независимо полученных выборок (n), умноженную на сумму по i от 1 до K квадратов полученных выборок (σ_i^2), отнесенных к ожидаемым вероятностям (p_i), минус число выборок.

Для числа выбрасываний костей n, диапазона значений i, реальных подсчетов σ и вероятностей p функция **ChiSquare()** (строки 59-67) вернула число 8.759116 (см. рис. 4.2). Чтобы посмотреть, как формула переводится на язык C, сравните с ней операторы этой функции.

Для того чтобы воспользоваться результатом, возвращаемым функцией, необходима таблица распределения хи-квадрат, которую можно найти в большинстве книг по статистике (в табл. 4.1 дан только фрагмент таблицы). Строки таблицы (m) представляют число степеней свободы

исходных данных, уменьшенное на единицу. Для случая выбрасывания костей в диапазоне возможных значений от 2 до 12 существует 11 категорий. 11 – 1 = 10, поэтому наша строка: v = 10.

Таблица 4.1

Распределение хи-квадрат

m	99%	95%	75%	50%	25%	5%	1%
9	2.088	3.325	5.899	8.343	11.39	16.92	21.67
10	2.558	3.940	6.737	9.342	12.55	18.31	23.21
11	3.053	4.575	7.584	10.34	13.70	19.68	24.73

Интерпретировать информацию в табл. 4.1 можно следующим образом: значения хи-квадрат между 6.737 и 12.55 должны выпадать примерно в 50% случаев. Значение, большее чем 23.21, должно встречаться не чаще чем один раз на 100 запусков программы. Значение, меньшее чем 3.940, будет означать неслучайность. (Замена реальных значений на ожидаемые даст в результате вычисления формулы хи-квадрат 0.000, что означает неправдоподобно хорошие данные.) Результат работы программы (число 8.759116) попадает где-то в середину табличного диапазона, что дает возможность оценить неплохое качество работы генератора случайных чисел.

Замечание. *Лучше всего запускать программу dice, по крайней мере, три раза. Плохое значение хи-квадрат маловероятно, но этот факт не обязательно означает неудовлетворительную работу генератора случайных чисел.*

Результаты работы программы dice выводятся в виде сформатированной таблицы. В строках 76-79 выполняется цикл **for**, который с помощью оператора **printf()** построчно выводит содержимое таблицы. Строка

```
"%5d%10.3lf%12.0lf%10.0lf\n"
```

форматирует четыре переменные – одно десятичное целое и три значения с плавающей запятой. Целое число отображается в пяти позициях, значения с плавающей запятой занимают 10, 12 и 10 позиций соответственно, причем первое число может иметь три знака после запятой, а два других – нуль.

Такой сложный оператор **printf()** легче записать и отладить в виде отдельных операторов:

```
printf("%5d", i);
printf("%10.3lf", prob[i]);
printf("%12.0lf", prob[i] * numThrows);
printf("%10.0lf", count[i]);
printf("\n");
```

После отладки отдельных операторов объедините их в один, как показано в листинге.

Использование *sizeof* с массивами

Если **anyArray** – это имя какого-то массива, то выражение **sizeof**(**anyArray**) равно числу байтов, занимаемых массивом в памяти, а **sizeof**(**anyArray[0]**) – числу байтов, занимаемых одним элементом.

Во время инициализации массива можно использовать **sizeof()**, чтобы точно определить число элементов. Вместо строк

```
#define MAX 5;
int anyArray[MAX] = {0, 1, 2, 3, 4};
```

вы можете написать

```
int anyArray[] = {0, 1, 2, 3, 4};
#define MAX (sizeof(anyArray) / sizeof(anyArray[0]));
```

В первой строке объявляется массив неопределенного размера, и элементам присваиваются начальные значения. Поскольку в скобках не указан размер массива, компилятор выделяет ровно столько памяти, сколько необходимо для запоминания перечисленных значений. Вторая строка определяет символическую константу **MAX**, равную размеру массива в байтах, деленному на размер одного элемента. Поскольку элементы массива запоминаются в памяти последовательно, то константа **MAX** теперь равна числу элементов в массиве.

Использование этого метода поможет избежать ошибки, вызванной заданием слишком малого числа элементов массива. Например, компилятор не станет жаловаться на объявление следующего вида:

```
#define MAX 5;
int anyArray[MAX] = {0, 1, 2, 3};
```

Здесь элементу **anyArray[4]** значение в явном виде не присваивается (этот факт легко не заметить), и в дальнейшем может быть допущена трудноуловимая ошибка.

Использование массивов констант

Чтобы не допустить изменений в элементах массива, предваряйте его объявление ключевым словом **const**. Это особенно важно для программ, которые сопровождают несколько человек. Объявление

```
const int anyArray[] = {0, 1, 2, 3, 4};
```

создает пятиэлементный массив с именем **anyArray**. Благодаря модификатору **const** любой оператор, который пытается изменить значение элементов массива, не будет компилироваться:

```
anyArray[4]++;      /* ??? */
```

Символьные массивы

Строки представляют собой массивы значений типа **char**. Вы уже встречали много примеров строк, но теперь подумайте о них как о символьных массивах. Объявление

```
char name[128];
```

задает символьный массив **name**, содержащий 128 элементов типа **char**. Аналогично тому, как это делается в других массивах, вы можете использовать индексное выражение для доступа к конкретному элементу, в данном случае, к символу. Если **a** является переменной типа **char**, то оператор

```
a = name[3];
```

присваивает четвертый символ массива **name** переменной **a**.

Чтобы инициализировать символьный массив, присвойте ему литерную строку, заключенную в кавычки:

```
char composer[] = "Peter Tchaikovsky";
```

Как и в других объявлениях массивов, пустые скобки заставляют компилятор вычислять объем памяти, необходимый для запоминания инициализирующего значения. Не забывайте, что все строки заканчиваются невидимым нулевым байтом, поэтому строка **composer** в этом примере имеет длину 12 байтов, а не 11.

Многомерные массивы

Такой массив, как **count[100]**, является одномерным – его значения построены в одну колонку, и для получения доступа к любому элементу нужен только один индекс.

Массивы могут обладать двумя и более измерениями. Двухмерный массив, например, можно представить в виде матрицы, а трехмерный – в виде куба. На самом деле, независимо от количества измерений, элементы массивов хранятся в памяти последовательно, один за другим.

Двухмерные массивы

Вы можете объявить двухмерный массив следующим образом:

```
int matrix[5][8];
```

matrix можно назвать массивом массивов. В данном случае матрица состоит из 5 строк и 8 столбцов. Выражение **matrix[0]** обозначает первую строку, выражение **matrix[1]** – вторую и т.д. Чтобы получить доступ к элементам массива, используйте две пары квадратных скобок. Оператор

```
matrix[4][2] = 5;
```

присвоит значение **5** третьему элементу пятой строки матрицы (рис. 4.3).

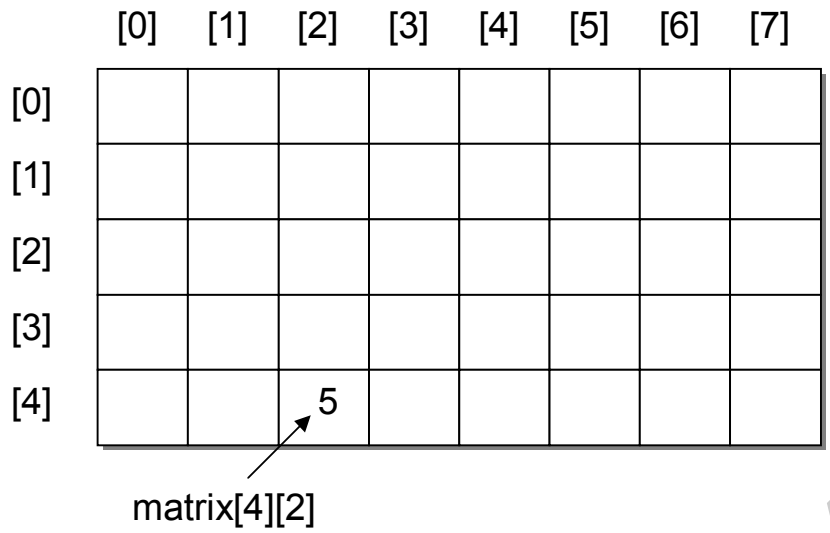

Рис. 4.3. Двухмерный массив хранится в памяти последовательно, но его удобно рассматривать в виде матрицы

Трехмерные массивы

Следующая запись

```
int cubic[10][20][4];
```

объявляет трехмерный массив целых чисел **cubic**. Можно сказать, что **cubic** является массивом массивов массивов. Концептуально он является трехмерной структурой (т.е. имеет высоту 10, ширину 20 и глубину 4).

Все многомерные массивы располагаются в памяти в таком порядке, что медленнее всего изменяется крайний левый индекс, а крайний справа – быстрее всего. Другими словами, чтобы отобразить значения массива **cubic**, соблюдая порядок расположения в памяти, вы можете использовать вложенные циклы **for** таким образом:

```
for (i = 0; i < 10; i++)
  for (j = 0; j < 20; j++)
    for(k = 0; k < 4; k++)
      printf("%d\n", cubic[i][j][k]);
```

Замечание. *Многомерные массивы растут очень быстро. Массив двухбайтовых целых чисел с размерностью 10x10x8 занимает 1,600 байт. Добавление четвертого измерения (10x10x10x8) потребует уже 16,000 байт.*

Инициализация многомерных массивов

Иногда инициализация многомерных массивов бывает очень полезной. Листинг 4.2 демонстрирует типичное использование инициализированного многомерного массива для запоминания названий месяцев.

Листинг 4.2. months.c (запоминание названий месяцев в массиве)

```
 1: #include <stdio.h>
 2:
 3: #define NUMMONTHS 12
 4:
 5: char months[NUMMONTHS][4] = {
 6:   "Jan",  "Feb", "Mar", "Apr",
 7:   "May",  "Jun", "Jul", "Aug",
 8:   "Sep",  "Oct", "Nov", "Dec"
 9: };
10:
11: main()
12: {
13:   int month;
14:
15:   for(month = 0; month < NUMMONTHS; month++)
16:     printf("%s\n", months[month]);
17:   return 0;
18: }
```

Объявленный в строке 5 массив **months** (месяцы) запоминает 12 трехсимвольных строк, оканчивающихся нулевым байтом. Выражение **months[0]** относится к **Jan**, **months[1]** – **Feb** и т.д. Такие выражения наталкивают нас на восприятие массива **months** как одномерного. Но на самом деле он имеет два измерения. Например, выражение **months[1][2]** ссылается на символ **b** строки “**Feb**”.

Листинг 4.3 использует многомерные массивы для отображения шахматной доски и фигур. (К сожалению, программа умеет только перемещать фигуры на доске и на самом деле не играет в шахматы.) Кроме того, программа chess демонстрирует, как ключевое слово **typedef** помогает сделать программу удобной для чтения.

Листинг 4.3. chess.c (пример шахматной доски в виде многомерного массива)

```
 1: #include <stdio.h>
 2:
 3: /* Символы шахматных фигур. Запоминаются в массиве board */
 4: #define NUMPIECES 13
 5: typedef enum piece {
 6:   EMPTY, WPAWN, WROOK, WKNIGHT, WBISHOP, WQUEEN, WKING,
 7:          BPAWN, BROOK, BKNIGHT, BBISHOP, BQUEEN, BKING
 8: } Piece;
```

```
 9:
10: /* Новое имя для типа int */
11: #define NUMRANKS 8
12: typedef int Ranks;
13:
14: /* Имена файлов */
15: #define NUMFILES 8
16: typedef enum files {
17:   A, B, C, D, E, F, G, H
18: } Files;
19:
20: /*Доска 8x8 с начальным расположением фигур*/
21: Piece board[NUMRANKS][NUMFILES] =
22: {
23:  {WROOK,WKNIGHT,WBISHOP,WQUEEN,WKING,WBISHOP,WKNIGHT,WROOK},
24:  {WPAWN, WPAWN, WPAWN, WPAWN, WPAWN, WPAWN, WPAWN, WPAWN},
25:  {EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY},
26:  {EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY},
27:  {EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY},
28:  {EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY, EMPTY},
29:  {BPAWN, BPAWN, BPAWN, BPAWN, BPAWN, BPAWN, BPAWN, BPAWN},
30:  {BROOK,BKNIGHT,BBISHOP,BQUEEN,BKING,BBISHOP,BKNIGHT,BROOK}
31: };
32:
33: /* Строковый массив сокращенных названий шахматных фигур */
34: char pieceNames[NUMPIECES][3] =
35: {
36:   "..", "wP", "wR", "wN", "wB", "wQ", "wK",
37:         "bP", "bR", "bN", "bB", "bQ", "bK"
38: };
39:
40: /* Прототипы функций */
41: void MovePiece(Files f1, Ranks r1, Files f2, Ranks r2);
42: void DisplayBoard(void);
43:
44: main()
45: {
46:   DisplayBoard();   /* перед перемещением фигуры */
47:   MovePiece(D, 2, D, 4);
48:   printf("\nPawn to D4 (d2-d4)\n");
49:   DisplayBoard();   /* после перемещения фигуры */
50:   return 0;
51: }
52:
53: void MovePiece(Files f1, Ranks r1, Files f2, Ranks r2)
54: {
55:   board[r2-1][f2] = board[r1 - 1][f1];
56:   board[r1-1][f1] = EMPTY;
57: }
58:
59: void DisplayBoard(void)
60: {
```

```
61:   Ranks r;
62:   Files f;
63:
64:   for (r = NUMRANKS; r > 0; r--) {
65:     printf("\n%d: ", r);
66:     for (f = A; f <= H; f++)
67:       printf("  %s", pieceNames[board[r-1][f]]);
68:   }
69:   printf("\n     a  b  c  d  e  f  g  h\n");
70: }
```

Даже без детального изучения программы chess вы можете легко заметить двухмерный массив шахматной доски **board**. Вместо таинственных чисел в строках 21-31 записаны имеющие смысл слова. Для белых фигур – **WROOK** (ладья), **WKNIGHT** (конь), **WBISHOP** (слон), **WQUEEN** (ферзь), **WKING** (король), **WPAWN** (пешка). Для черных – соответственно **BROOK**, **BKNIGHT**, **BBISHOP**, **BQUEEN**, **BKING** и **BPAWN**. Слово **EMPTY** обозначает пустую клетку. Порядок следования символов соответствует начальному расположению фигур в шахматах.

Старайтесь добиться того же уровня удобочитаемости в ваших собственных программах. Месяц, а тем более год спустя, когда вам понадобится изменить программу, вы оцените свою заботу о том, чтобы программа была понятной.

В строке 12 программы chess **typedef** используется для того, чтобы получить новое имя для типа **int**:

```
typedef int Ranks;
```

Поскольку символ **Ranks** (ряды) эквивалентен **int**, его можно использовать везде, где используется **int**. Например, в прототипе функции в строке 41

```
void MovePiece(Ranks r1, Files f1, Ranks r2, Files f2);
```

параметры **r1** и **r2** являются номерами ряда шахматной доски. В действительности они имеют тип **int**, но новое имя типа данных проясняет их назначение.

Вы также можете использовать **typedef** с более сложным типом данных. На строках 5-8 объявляется псевдоним **Piece** для перечислимых символов, используемых для идентификации пустых клеток шахматной доски и играющих фигур. Объявление использует тот факт, что в языке C различаются строчные и прописные буквы. Запись

```
typedef enum piece {
. . .
} Piece;
```

объявляет псевдоним **Piece** для перечисления **enum piece** {...}. Имена **Piece** и **piece** являются здесь совершенно разными идентификаторами.

Теперь, используя **typedef**, вы можете объявить переменную с именем **anyPiece** типа **Piece**:

```
Piece anyPiece;
```

Она будет обозначать то же самое, что и

```
enum piece anyPiece;
```

Строки 16-18 точно так же объявляют псевдоним **Files** для вертикалей шахматной доски от A до H. В этой программе буквы от **A** до **H** не являются символами; они представляют собой односимвольные перечислимые константы. Во внутреннем представлении **A** равно 1, **B** равно 2 и т.д.

Вооруженные этими символами строки 21-31 объявляют двухмерный массив **Piece board**, используя символические константы **NUMRANKS** (количество строк) и **NUMFILES** (количество столбцов). Это объявление также демонстрирует возможность инициализации двухмерного массива с помощью вложенных скобок.

Еще одна встреча с многомерным массивом происходит в строках 34-38, где объявляется строковый массив **pieceNames** (названия шахматных фигур). Строка ".." представляет пустую клетку. Порядок элементов в массиве **pieceNames** совпадает с порядком названий фигур в перечислении **piece**. Таким образом, **pieceNames[WROOK]** – название белой ладьи ("**wR**"); а **pieceNames[BQUEEN]** – название черной королевы ("**bQ**").

Эти взаимосвязи между элементами массива и индексами делают программу несложной для написания и сопровождения. Например, функция **MovePiece()** (строки 53-57), используя операторы присваивания, «передвигает» шахматную фигуру с одной клетки на другую.

Функция **DisplayBoard()** отображает игровую поверхность шахматной доски с помощью двух циклов **for** в строках 64 и 66. Строка 67 демонстрирует пример вложенного индекса массива. Выражение **pieceNames[board[r–1][f]]** передает строковое имя шахматной фигуры на клетке доски в строке **r – 1** и столбце **f**.

Замечание. *Многие опытные программисты предлагают не вводить в программу никаких литеральных числовых значений за исключением **0** и **1**. Все другие литеральные значения должны быть заменены символическими константами.*

Передача массивов функциям

Вы можете передавать массивы функциям. Предположим, вам необходимо просуммировать значения, запомненные в массиве.

```
#define MAX 100
double data[MAX];
```

Прототип функции, которая принимает в качестве параметра массив значений типа **double**, можно записать следующим образом:

```
double SumOfData(double data[MAX]);
```

Лучше оставить квадратные скобки пустыми и добавить второй параметр, означающий размер массива:

```
double SumOfData(double data[], int n);
```

Функцию **SumOfData()** написать нетрудно. Простой цикл **while** суммирует элементы массива, а оператор **return** возвращает результат:

```
double SumOfData(double data[], int n)
{
  double sum = 0;
  while (n > 0)
    sum += data[--n];
  return sum;
}
```

Замечание. *Для экономии стековой памяти С передает функциям не содержимое массива, а лишь его адрес. Элементы массива остаются на своих местах. Такой способ экономит память стека, но приводит к тому, что изменение значений элементов массива в функции затрагивает исходные данные.*

Передача многомерных массивов функциям

При передаче многомерных массивов функциям вы должны также передать дополнительную информацию о размерности массива, чтобы избежать неправильной адресации. В случае двухмерного массива компилятору нужно знать количество столбцов, чтобы вычислять адреса элементов в начале каждой строки.

Этот факт имеет важные последствия при объявлении многомерных массивов в качестве параметров функции. Определив символические константы **ROWS** и **COLS** как количества строк и столбцов в двухмерном массиве, вы можете объявить функцию с двухмерным массивом в качестве параметра следующим образом:

```
void AnyFn(int data[ROWS][COLS]);
```

Вы также можете задать только количество столбцов:

```
void AnyFn(int data[][COLS]);
```

Иногда в функцию бывает удобно передать отдельным параметром количество строк:

```
void AnyFn(int data[][COLS], int numRows);
```

Параметр **numRows** сообщает функции, сколько строк в массиве **data**, реализуя тем самым способ, позволяющий передавать в одну и ту же функцию массивы с разным количеством строк. Конечно, было бы идеально передать параметрам функции **AnyFn()** обе размерности массива:

```
void AnyFn(int data[][], int numRows, int numCols);  /* ??? */
```

но, к сожалению, язык С не обладает такой возможностью. При передаче функциям двухмерных массивов вы должны задать, по крайней мере, количество столбцов, иначе компилятор не сможет правильно вычислить

адреса элементов массива. Приведенное объявление функции не будет скомпилировано.

Листинг 4.4 демонстрирует использование многомерных массивов в качестве параметров функции. После запуска программы вы увидите две таблицы с различным количеством строк, что подтверждает способность программных функций обрабатывать массивы разной длины.

Листинг 4.4. multipar.c (передача функции многомерных массивов)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <time.h>
 4:
 5: #define COLS 8
 6:
 7: void FillArray(int data[][COLS], int numRows);
 8: void DisplayTable(int data[][COLS], int numRows);
 9:
10: int data1[7][COLS];
11: int data2[4][COLS];
12:
13: main()
14: {
15:   randomize();
16:   FillArray(data1, 7);
17:   DisplayTable(data1, 7);
18:   FillArray(data2, 4);
19:   DisplayTable(data2, 4);
20:   return 0;
21: }
22:
23: void FillArray(int data[][COLS], int numRows)
24: {
25:   int r, c;
26:
27:   for (r = 0; r < numRows; r++)
28:     for (c = 0; c < COLS; c++)
29:       data[r][c] = rand();
30: }
31:
32: void DisplayTable(int data[][COLS], int numRows)
33: {
34:   int r, c;
35:
36:   for (r = 0; r < numRows; r++) {
37:     printf("\n");
38:     for (c = 0; c < COLS; c++)
39:       printf("%8d", data[r][c]);
40:   }
41:   printf("\n");
42: }
```

Функции **FillArray()** и **DisplayTable()** задают только число столбцов в параметре **data**. Реальное число строк передается отдельным параметром **numRows** типа **int**. Этот способ позволяет обеим функциям принимать массивы переменной длины при условии, что число столбцов остается фиксированным.

Указатели

Вы, наверное, будете удивлены, узнав, что массивы на самом деле являются замаскированными указателями. Настало время поближе познакомиться с этими загадочными переменными.

Программы используют указатели, чтобы найти данные, подобно тому как на почте используют почтовые адреса, чтобы найти необходимых людей. В С-программе указатель содержит *адрес* объекта (переменной или функции), хранящегося в памяти.

Несмотря на кажущуюся простоту, указатели – одно их самых гибких средств языка. С их помощью программы могут выполнять чтение и запись данных в любое место памяти. С другой стороны, это мощнейшее средство при неправильном обращении может разрушить данные в незащищенном месте памяти так быстро, что вы и глазом не успеете моргнуть. Но не следует из-за этой потенциальной опасности отказывать себе в удовольствии пользоваться указателями. Преодолев все трудности общения с ними, вы будете удивляться, как вообще можно было обойтись без них.

Введение в указатели

Указатель – это обычная переменная, подобная переменным типа **int** или **float**. Чтобы объявить указатель, добавьте звездочку перед именем переменной при ее объявлении. Например, **int p** объявляет **p** как целую переменную, а **int *p** объявляет **p** как *указатель* на целую переменную. (Расположение звездочки не является строгим правилом: можно записать **int* p** или **int *p**.)

Как и все переменные, указатели требуют инициализации перед своим использованием. При инициализации указателя вы даете ему адрес, который указывает на определенное место в памяти. Никогда не используйте неинициализированные указатели. Они, подобно стрелам, выпущенным из лука наугад, часто вызывают трудно обнаруживаемые ошибки.

В этой главе описано несколько способов инициализации указателей. Изучите эти методы и неотступно следуйте им, чтобы предотвратить возможные ошибки.

Объявление и разыменование указателей

Если вы объявите целую переменную и указатель как

```
int i;      /* целая переменная i */
int *p;     /* указатель p на целое значение */
```

то сможете выполнить присваивание переменной **p** адреса переменной **i**:

```
p = &i;
```

Унарный оператор взятия адреса **&** возвращает адрес объекта в памяти. После присваивания указатель **p** указывает место в памяти, где хранится значение **i**. Чтобы отобразить значение переменной **i**, вы можете записать

```
printf("%d", i);
```

Чтобы сделать то же самое, но с помощью указателя, запишите

```
printf("%d", *p);
```

Использование указателя для получения доступа к адресуемому им данному называется *разыменованием* указателя. Выражение ***p** разыменовывает указатель **p**, возвращая целое значение, на которое он ссылается.

Из объявления указателя компилятор знает, какой тип данных адресует указатель. Поскольку указатель **p** объявлен как **int *p**, компилятор «понимает», что выражение ***p** возвращает значение типа **int**. Выражение ***p** может использоваться всюду, где разрешено применять целые переменные.

Указатели в качестве псевдонимов

Листинг 4.5 демонстрирует принципы объявления, инициализации и разыменования указателей.

Листинг 4.5. alias.c (объявление, инициализация и разыменование указателя)

```
 1: #include <stdio.h>
 2:
 3: char c;    /* символьная переменная */
 4:
 5: main()
 6: {
 7:   char *pc; /* указатель на символьную переменную */
 8:
 9:   pc = &c;
10:   for (c = 'A'; c <= 'Z'; c++)
11:     printf("%c", *pc);
12:   return 0;
13: }
```

Программа alias выводит алфавит. Единственный оператор вывода находится в строке 11, причем он напрямую не связан с глобальной символьной переменной **c**, которой в цикле **for** в строке 10 присваиваются

символы от ‘**A**’ до ‘**Z**’. Если оператор **printf()** не использует символьную переменную, то как же он может отображать буквы алфавита?

Взгляните вначале на строку 7. Объявление **char *pc** сообщает компилятору, что **pc** – указатель на значение типа **char**. Строка 9 присваивает переменной **pc** адрес переменной **c** (рис. 4.4). В операторе **printf()** (строка 11) выражение ***pc** разыменовывает указатель, возвращая значение переменной, на которую он указывает (символ, хранящийся в переменной **c**).

Когда указатель адресует область памяти, где хранится значение другой переменной, то он называется *псевдонимом*. Подобно маске, псевдоним указателя скрывает истинное лицо объекта.

Нулевые указатели

Нулевой указатель никуда не указывает. Он подобен стреле без наконечника или дорожному знаку, который упал на землю. В языке С нулевой указатель – это указатель, который, по крайней мере в данный момент, не адресует никакого допустимого значения в памяти.

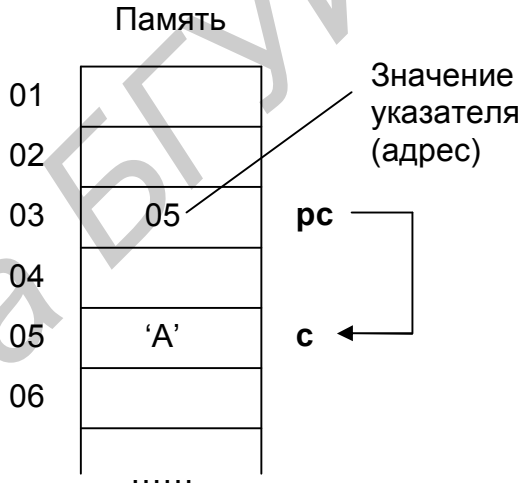

Рис. 4.4. Переменная **c** и указатель **pc** ссылаются на одно и то же значение

Значение нулевого указателя равно нулю – единственный адрес, до которого указатель не может «дотянуться». Вместо того чтобы записать литеральную цифру **0**, где-нибудь в заголовочном файле можно определить символ **NULL**:

```
#define NULL 0
```

Еще лучше включить один из таких заголовочных файлов, как stddef.h, stdio.h, stdlib.h или string.h, которые выполнят эту работу за вас.

Поскольку указатели являются числами, им можно присваивать целые значения. После объявления вида **double *fp** следующие операторы компилируются без ошибки:

```
fp = NULL;
fp = 0;
fp = 12345; /* ??? */
```

Первые две строки дают один и тот же результат – присваивание переменной **fp** значения нуля. Но первая строка является самой безопасной и гарантирует корректную работу во всех моделях памяти. Последняя строка скомпилируется, но вызовет предупреждение компилятора: «Nonportable pointer conversion» («Недопустимое преобразование указателя»). Никогда не присваивайте указателям значения таким образом.

Присвоив указателю значение **NULL**, программа может проверить достоверность этого факта:

```
if (fp != NULL)
  оператор;
```

Если указатель равен **NULL**, следует предположить, что он не адресует достоверные данные; это простой, но эффективный метод защиты от ошибок.

Подобно всем глобальным переменным, глобальные указатели инициализируются равными нулю. Если вы объявите указатель **fp** как глобальную переменную

```
float *fp;  /* fp == NULL */
main()
{
  ...
}
```

то **fp** будет равен значению **NULL** в начале работы программы. Но если вы объявите **fp** внутри функции, то, подобно всем локальным переменным, указатель будет иметь непредсказуемое значение. Для защиты от использования неинициализированных указателей лучше присвоить локальному указателю значение **NULL**:

```
void f(void)
{
  float *fp = NULL;
  . . .
}
```

Указатели типа void

Указатель типа **void** указывает на неопределенный тип данных. Объявите указатель этого типа следующим образом:

```
void *nowhereLand;
```

Указатель **nowhereLand** может адресовать любое место в памяти и не ограничивается никаким определенным типом данных: он может адресовать переменную типа **double**, символ или произвольную область памяти, которая принадлежит операционной системе.

Замечание. *Не путайте нулевой указатель с указателем типа **void**. Нулевой указатель не адресует достоверных данных, а указатель на **void** адресует данные неопределенного типа и также может быть нулевым.*

При знакомстве с указателями типа **void** часто возникает вопрос: каким образом использовать данные, которые адресуются таким указателем? Один из способов – приведение типов. Для этого нужно предварить указатель объявлением типа данных, взятым в круглые скобки. Приведение типа заставляет компилятор рассматривать элемент одного типа (в данном случае указатель на **void**) как элемент другого типа (например указатель на **double**).

Для примера рассмотрим один из способов построения буферов данных, при котором выделяется некоторая область памяти и создается указатель для адресации этого буфера:

```
char buffer[1024]; /* 1024-байтовый буфер */
void *bp;          /* пока ни на что не указывает */
bp = &buffer;      /* указателю bp присвоен адрес buffer */
```

Если **c** – переменная типа **char**, то для присваивания ей первого символа из символьного массива **buffer**, адресуемого указателем **bp** типа **void**, вы можете записать:

```
c = *(char *)bp;
```

Выражение (**char ***)**bp** заставляет компилятор временно обращаться с **bp** как с указателем на **char**. Разыменование выражения вида ***(char ***)**bp** позволяет получить значение переменной типа **char**, которую адресует указатель **bp** (т.е. первый символ массива **buffer**).

Если буфер состоит из значений типа **int** и если **i** – переменная типа **int**, то с помощью аналогичного выражения приведения типов можно присвоить целое число из буфера переменной **i**:

```
i = *(int *)bp;
```

Указатели и функции

Функции могут возвращать указатели, и вы можете передавать указатели в качестве аргументов параметрам функции.

Если функция объявляет параметр как указатель на некоторую переменную в памяти, то значение переменной может быть модифицировано внутри функции, т.к. в этом случае в стек копируется не значение переменной, а ее адрес.

Листинг 4.6. вызывает функцию, которая выполняет распространенную задачу, встречающуюся при сортировке данных – обменивает значения двух переменных.

Листинг 4.6. swapptr.c (обмен значениями внутри функции)

```
 1: #include <stdio.h>
 2:
 3: void PrintValues(void);
 4: void Swap(int *a, int *b);
 5: int a = 1, b = 0;
 6:
 7: main()
 8: {
 9:   PrintValues();
10:   Swap(&a, &b);
11:   PrintValues();
12:   return 0;
13: }
14:
```

```
15: void PrintValues(void)
16: {
17:   printf("\na = %d, b = %d", a, b);
18: }
19:
20: void Swap(int *a, int *b)
21: {
22:   int temp;
23:
24:   temp = *a;
25:   *a = *b;
26:   *b = temp;
27: }
```

Программа swapptr объявляет в строке 5 две глобальные переменные: **a** и **b**. Функция **PrintValues()** (строки 15-18) выводит на экран их значения. В программе функция вызывается дважды: до и после функции **Swap()**, чтобы проверить результаты работы последней.

Функция **Swap()** объявляет два параметра-указателя: **a** и **b**. В строке 10 функции передаются адреса глобальных переменных, благодаря чему становится возможным их изменение внутри функции.

Внутри функции **Swap()** переменные обмениваются своими значениями. Так как параметры функции объявлены как указатели, в выражениях используется операция разыменования ***.

Задание. *Реализуйте функцию **Swap()** из листинга 4.6. без использования указателей:*

```
void Swap(int a, int b);
```

*Запустите программу на исполнение. Почему значения переменных **a** и **b** остаются без изменений?*

Указатели и динамические переменные

До сих пор в программных примерах мы сохраняли значения в глобальных или локальных переменных. Глобальные переменные находятся в фиксированных областях сегмента данных программы, локальные – сохраняются в стеке и существуют, пока активны функции, в которых они объявлены.

Оба вида переменных имеют одну общую черту – вы объявляете их прямо в тексте программы. Другой способ выделяет память для переменных *во время* выполнения программы. Такие переменные называются динамическими – они создаются в тот момент, когда это необходимо, и запоминаются в блоках памяти переменного размера, которые программисты называют *кучей*.

Кучу можно себе представить в виде просторной области памяти, которая может обеспечить гораздо больше места, чем глобальные или

локальные переменные, и, таким образом, является прекрасным местом для запоминания больших буферов и структур данных, способных расширяться и сужаться в зависимости от размеров принимаемой информации.

Днамические переменные также называются *переменными, адресуемыми указателями*, т.к. только с помощью указателей можно использовать память в куче.

Резервирование памяти в куче

Существует несколько способов выделения памяти в куче для динамических переменных. Самый распространенный использует библиотечную функцию с именем **malloc()**, являющимся сокращением от *memory allocation* (выделение памяти). Функция **malloc()** выделяет в куче заданное количество байтов. Функция возвращает адрес выделенного блока, который обычно присваивается указателю.

Чтобы воспользоваться данной функцией, включите заголовочный файл alloc.h и объявите указатель

```
#include <alloc.h>
...
double *v;     /* v – указатель на тип double */
```

Затем вызовите функцию **malloc()**, задавая в круглых скобках число байтов для резервирования, и присвойте указателю **v** значение, возвращаемое функцией. Но сейчас перед вами загадка: сколько байтов вам нужно зарезервировать? Вы можете поискать размер элемента типа **double** в справочнике, но в целях лучшей переносимости программы используйте оператор **sizeof**:

```
v = (double *)malloc(sizeof(double));
```

Указатель **v** сейчас адресует блок памяти в куче, точно соответствующий размеру одного значения **double**. Функция **malloc()** может отказать, если куча уже заполнена или оставшаяся память не в состоянии вместить требуемое количество байтов. В этом случае **malloc()** возвращает нуль.

Теперь вы можете разыменовать указатель **v**, и использовать его как обычную переменную типа **double**:

```
*v = 3.14159;
```

А вот вывод значения, запомненного в куче:

```
printf("Value = %lf", *v);
```

Для резервирования памяти в куче вместо функции **malloc()** можно вызвать аналогичную функцию **calloc()**, прототип которой также объявлен в файле alloc.h. Эта функция работает подобно функции **malloc()**, но требует два аргумента – количество объектов, которые вы желаете разместить, и размер одного объекта. Используйте **calloc()** следующим образом:

```
long *lp;
```

```
lp = (long *)calloc(1, sizeof(long));
```

При этом резервируется память на одно значение **long**. Чтобы выделить память для 10 значений, вы должны записать:

```
lp = (long *)calloc(10, sizeof(long));
```

Кроме выделения памяти, функция **calloc()** устанавливает каждый зарезервированный байт равным нулю.

Удаление памяти в куче

По окончании работы с выделенным блоком памяти в куче вы должны освободить его, чтобы **malloc()**, **calloc()** и другие функции выделения памяти могли снова воспользоваться этой памятью. Если вы не освободите зарезервированную память, которая вам больше не нужна, ее нельзя будет использовать до конца работы программы.

Чтобы освободить блок памяти, вызовите функцию **free()**. Допустим, указатель **p** адресует некоторый блок памяти в куче:

```
int *p;
p = (int *)malloc(sizeof(int));
```

После работы с этой памятью освободите ее, выполнив следующий оператор:

```
free(p);
```

В этом операторе вам не нужно задавать размер освобождаемой памяти – он запоминается в нескольких специальных байтах, примыкающих к каждому зарезервированному блоку.

Предупреждение. *После освобождения памяти, адресуемой указателем **p**, ни при каких обстоятельствах не используйте этот указатель для выполнения чтения или записи значений в эту теперь незащищенную память. Некоторые программисты присваивают **NULL** указателям на освободившуюся память, что помогает предотвратить случайное использование незащищенных блоков памяти.*

Указатели и массивы

Все идентификаторы массивов на самом деле являются указателями, и все указатели могут адресовать массивы. Звучит странно? Чтобы разобраться, рассмотрим природу массива. Как вы уже знаете, массив объединяет «под одной крышей» набор переменных одного типа. Если идентификатор **collection** означает массив, то выражение **collection[0]** – его первый элемент. В известном смысле выражение **collection[0]** является аналогией разыменованного указателя.

Дело в том, что идентификатор **collection** в действительности является *указателем-константой*. Он указывает на первый элемент массива и не может быть переадресован. В остальном он ничем не отличается от обычных

указателей. Таким образом, выражение ***collection** обозначает то же самое, что и **collection[0]** – значение первого элемента массива.

Добавление единицы к указателю увеличивает его значение на величину того типа данного, которое он адресует. Если массив **collection** имеет тип **int**, то прибавление к нему единицы (**collection + 1**) увеличивает получаемое значение на 2 (т.к. размер типа **int** – 2 байта). Если бы массив имел значение **long**, то значение увеличилось бы на 4 и т.д. В любом случае выражение (**collection + 1**) является адресом второго элемента массива. Получаются следующие равенства:

```
*(collection + 1) == collection[1]  /* то же значение */
collection + 1 == &collection[1]    /* тот же адрес */
```

Не перепутайте ***(collection + 1)** и ***collection + 1.** Косвенная операция * имеет более высокий приоритет, чем операция +, поэтому последняя запись эквивалентна записи **(*collection) + 1**.

```
*(collection + 1) /* значение второго элемента массива */
*collection + 1   /* 1 добавляется к значению 1-го элемента */
```

Стандарт С описывает систему обозначений массива в терминах указателей, т.е. выражение **collection[n]** при компиляции программы переводится в ***(collection + n)**. Квадратные скобки – лишь для удобства программистов.

Примечание. *Выражение ***(collection + n)** можно понимать следующим образом: «Перейти к ячейке памяти с обозначением **collection**, переместиться на **n** единиц и осуществить здесь выборку значения».*

Хотя обычно все эти подробности, касающиеся адресации массивов, оставляют компилятору, иногда бывает полезно взять на себя эту работу и использовать вместо индексов массивов указатели в явном виде. В качестве такого примера скомпилируйте и запустите листинг 4.7.

Листинг 4.7. ptrarray.c (адресация элементов массивов указателями)

```
 1: #include <stdio.h>
 2:
 3: #define MAX 10     /* размер массива */
 4:
 5: void showFirst(void);
 6:
 7: int array[MAX];    /* глобальный массив из MAX целых */
 8:
 9: main()
10: {
11:   int *p = array;  /* p - указатель на массив целых */
12:   int i;
13:
14:   for (i = 0; i < MAX; i++)
15:     array[i] = i;
16:   for (i = 0; i < MAX; i++)
17:     printf("%d\n", *p++);
```

```
18:    p = &array[5];
19:    printf("array[5] = %d\n", array[5]);
20:    printf("*p..... = %d\n", *p);
21:    return 0;
22: }
```

Программа ptrarray показывает, как использовать указатели для получения доступа к элементам массива. В строке 11 объявляется указатель **p** типа **int**, которому присваивается адрес массива **array** того же типа. Если для вычисления адреса массива вы попытались бы использовать оператор взятия адреса вида **&array**, то это было бы ошибкой, которая вызвала бы предупреждение компилятора «Suspicious pointer conversion» (Подозрительное преобразование указателя). Помните, что **array** – это указатель, и вы можете прямо присваивать его значение другому указателю того же типа.

В строке 15 используется обычное индексное выражение **array[i]** для заполнения массива значениями от 0 до **MAX – 1**. Затем второй цикл **for** в строках 16-17 отображает содержимое массива. Но на этот раз при обращении к элементам массива программа использует указатель **p**.

Посмотрите внимательно на выражение ***p++** в строке 17. Оно содержит два действия. Разыменование ***p** дает значение, адресуемое указателем **p**. Затем оператор инкремента продвигает указатель **p** к следующему целому значению в массиве, заставляя цикл отобразить все его элементы.

Динамические массивы

Массивы фиксированного размера могут понапрасну тратить память. Например, если вы, желая подстраховаться, объявили 100-элементный массив типа **double** подобно следующему:

```
double myArray[100];
```

но используете только 70 элементов, то в этом случае память, отведенная еще для 30 значений, пропадает зря. Поскольку размер одного элемента типа **double** – 8 байтов, значит у других операций «отобрано» 240 байтов. В больших программах с множеством таких массивов объем зря потраченной памяти может достигать ошеломляющих размеров.

Если *до начала* выполнения программы определить размер массива невозможно, используйте динамические массивы, которые могут изменять свой размер *во время* выполнения программы. Так как массивы и указатели суть одно и то же, для этого вы можете использовать уже знакомую вам технику.

Сначала объявите указатель требуемого типа данных:

```
double *myArrayP;
```

Затем где-нибудь в программе вызовите функцию **malloc()**, чтобы выделить память из кучи. Например, если программа попросила пользователя ввести количество элементов массива и он ввел число 70, используйте оператор:

```
myArrayP = (double *)malloc(70 * sizeof(double));
```

Аналогичным образом вы можете также вызвать функцию **calloc()**:

```
myArrayP = (double *)calloc(70, sizeof(double));
```

В данном случае все выделенные значения равны нулю.

В случае неудачного вызова функции **malloc()** и **calloc()** возвращают нуль, поэтому лучше всего после их вызова проверять, не равен ли указатель нулю:

```
myArrayP = (double *)calloc(70, sizeof(double));
if (myArrayP) /* если myArrayP не нуль, */
  оператор;   /* то выполняется оператор */
```

Теперь вы можете использовать идентификатор **myArrayP** так же, как и обычный массив. Следующий оператор запоминает число **2.8** в 12-м элементе массива:

```
myArrayP[11] = 2.8; /* квадратные скобки разыменовывают myArrayP*/
```

После использования массива освободите выделенную для него память:

```
free(myArrayP);
```

Затем вы можете вызвать функцию **calloc()** снова, чтобы выделить в куче другой объем памяти для другого массива типа **double** и присвоить новый адрес указателю **myArrayP**.

Резюме

- Массивы содержат переменные одного и того же типа.
- Массив объявляется с помощью квадратных скобок. Например, запись **int scores[100]** объявляет массив **scores** для запоминания 100 значений типа **int**.
- Чтобы обратиться к конкретному элементу массива, используйте индексное выражение вида **scores[9]**, которое возвращает десятый элемент массива **scores**. Первым индексом массива является нуль, таким образом, **scores[0]** – это первый элемент массива, **scores[1]** – второй и т.д.
- Строки являются массивами значений типа **char**. Вы можете использовать индексные выражения для доступа к отдельным символам, запомненным в строках.
- Двухмерные массивы можно рассматривать как прямоугольные матрицы, но они, как и все массивы, запоминаются в памяти последовательно. Для доступа к элементам двухмерного массива используйте выражения с двойными скобками, например **point[x][y]**.

- Многомерные массивы могут иметь любое количество индексов. Однако их число редко превышает 3. Трехмерный массив можно объявить следующим образом: **int cubic[10][20][4];**
- Для экономии памяти в языке С массивы передаются параметрам функции по адресу. В функциях, которые объявляют массивы в качестве параметров, любые операторы, изменяющие элементы массивов, также изменяют и исходное содержимое массивов.
- Указатели являются переменными, которые содержат адреса других переменных и объектов в памяти.
- Объявляйте указатели с помощью оператора *****. Объявление **int p** объявляет **p** как целую переменную; **int *p** объявляет **p** как указатель на переменную типа **int**.
- Чтобы получить значение, на которое ссылается указатель, нужно выполнить операцию разыменования (*****). Чтобы получить адрес переменной, используйте оператор взятия адреса (**&**).
- Нулевой указатель, имеющий значение 0, не адресует достоверных данных. Никогда не используйте нулевые указатели для чтения или записи информации в память.
- Указатель типа **void**, объявляемый как **void *p**, является указателем общего назначения, способным адресовать значения любого типа и в любом месте памяти. Не путайте нулевые указатели с указателями типа **void**. Нулевые указатели не адресуют допустимых данных. Указатели типа **void** адресуют данные неопределенного типа.
- Чтобы сообщить компилятору о типе данных, которые будет адресовать указатель типа **void**, используйте переопределение типов. Если указатель **p** имеет тип **void**, то выражение **(int *)p** заставит компилятор временно обращаться с указателем **p** как с указателем на значение типа **int**.
- Параметры функций, являющиеся указателями, позволяют изменять значение соответствующих аргументов. Указатели также могут быть возвращены в качестве результатов работы функций.
- Программы могут запоминать динамические переменные в обширной области памяти, называемой кучей. При завершении работы с динамическими переменными не забудьте освободить неиспользуемый блок памяти. Никогда не используйте освобожденный указатель для операций чтения или записи в память.
- Имя массива на самом деле является *указателем-константой*, который указывает на первый элемент массива и не может быть переадресован. В отличие от него обычная переменная типа указатель может принимать разные значения и указывать на различные объекты в памяти. За исключением этой особенности указатели и массивы являются эквивалентами. Указатель, подобно массиву, может быть индексирован.

- Динамические массивы могут изменять свой размер в процессе выполнения программы. Используйте динамические массивы, когда заранее неизвестно, сколько значений нужно хранить в массиве.

Обзор функций

Таблица 4.2

Функции для работы со случайными числами (stdlib.h)

Функция	Прототип и краткое описание
rand()	`int rand(void);` Функция возвращает псевдослучайное число в диапазоне от 0 до 32767.
random()	`int random(int num);` Возвращает псевдослучайное число между 0 и **num – 1**.
randomize()	`void randomize(void);` Инициализирует генератор случайных чисел случайным значением, определяемым по текущему времени.

Таблица 4.3

Функции для выделения и освобождения памяти (alloc.h)

Функция	Прототип и краткое описание
calloc()	`void *calloc(unsigned n, unsigned m);` Возвращает указатель на начало области динамически выделенной памяти для размещения **n** элементов по **m** байтов каждый. При неудачном завершении возвращает **NULL**.
free()	`void free(void *b1);` Освобождает блок динамически выделенной памяти с адресом первого байта **b1**.
malloc()	`void *malloc(unsigned n);` Возвращает указатель на блок динамически выделенной памяти длиной **n** байтов. При неудачном завершении возвращает значение **NULL**.
realloc()	`void *realloc(void *b1, unsigned n);` Сохраняя содержимое блока динамической памяти с адресом первого байта **b1**, изменяет его размер до **n** байтов. Если **b1** равен **NULL**, то функция выполняется как **malloc()**. При неудачном завершении возвращает значение **NULL**.

5. Строки

Благодаря строкам программы обладают даром речи. Почти каждая программа содержит одну или несколько строк. Настало время изучить эту тему глубже. В следующих разделах вы узнаете, как различные строки запоминаются в памяти, а также поближе познакомитесь с некоторыми строковыми функциями, которые могут разбивать строки, склеивать их вместе, выполнять поиск символов и т.п.

Что такое строка

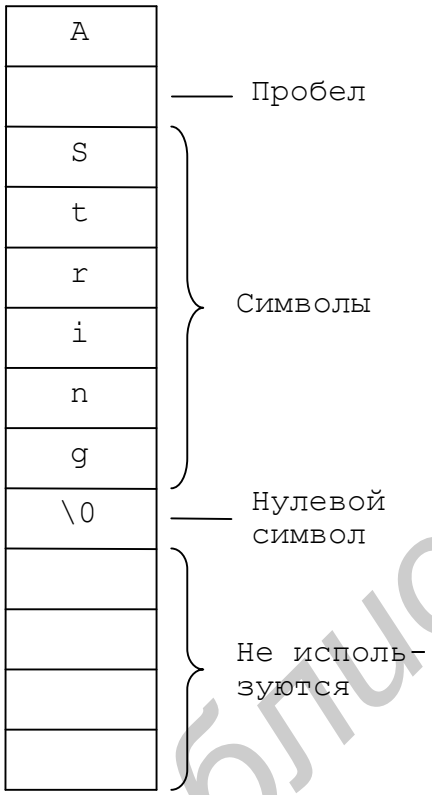

Рис. 5.1. Размещение строки в памяти

Как говорилось в предыдущей главе, строка представляет собой массив значений типа **char**, завершающийся нулевым байтом. Каждый символ в строке – это на самом деле целое число, представляющее собой код ASCII.

Как показано на рис. 5.1, символы строки запоминаются в памяти последовательно, друг за другом. Нулевой символ (на рисунке он показан как **\0**) следует сразу за последним значащим символом. Если выделенная для строки память не до конца заполнена символами, то байты, расположенные после нулевого символа, могут содержать произвольные значения.

Символы ASCII из стандартного набора имеют значения в диапазоне от 0 до 127. Стандартный набор включает управляющие последовательности, цифры и буквы английского алфавита (см. прил. 1). Символы ASCII из расширенного набора имеют значения от 128 до 255. В этой части таблицы запоминаются, в частности, буквы русского алфавита. В строках ANSI C вы можете запоминать любые символы из стандартного или расширенного набора.

Управляющие последовательности – специальные символы ASCII, которые нельзя ввести в текстовых редакторах. В табл. 5.1 перечислены некоторые из них.

Таблица 5.1

Строковые управляющие последовательности

Код	Значение	Десятичное	Шестнадца-теричное	Символ
'\a'	Звонок ("будильник")	7	0x07	BEL
'\b'	Шаг назад	8	0x08	BS
'\n'	Новая строка	10	0x0a	LF
'\r'	Возврат каретки	13	0x0d	CR
'\t'	Горизонтальная табуляция	9	0x09	HT
'\v'	Вертикальная табуляция	11	0x0b	VT
'\\'	Обратная косая черта	92	0x5c	\
'\''	Апостроф	39	0x27	'
'\"'	Двойная кавычка	34	0x22	"
'\?'	Знак вопроса	63	0x3f	?

Каждая из управляющих последовательностей в табл. 5.1 представляет собой одиночный символ, запоминаемый в памяти как целое число и состоящий из обратной косой черты, за которой следует буква или знак препинания.

Управляющие последовательности допускаются всюду, где могут быть печатные символы. Символ '**\n**', например, вам уже знаком. Чтобы закончить строку символом перехода на новую строку, нужно записать:

```
"Эта строка заканчивается кодом новой строки\n";
```

Компилятор заменяет символ новой строки '**\n'** на управляющий код конца строки, имеющий значение 10 (в шестнадцатеричном выражении 0x0a).

Поскольку управляющая последовательность начинается с обратной косой черты, для ввода самого символа обратной косой черты нужно ввести два этих символа подряд. Например, чтобы ввести в строку путь к файлу, нужно задать с клавиатуры маршрут, аналогичный следующему:

```
"c:\\by\\test\\fun.c";
```

Если же вы забудете удвоить символ обратной косой, то строка

```
"c:\by\test\fun.c";  /* ??? */
```

при отображении может повести себя очень странно. Компилятор будет интерпретировать **\b** – как звонок, **\t** – как горизонтальную табуляцию и **\f** – как символ подачи бланка. Если строка вашего маршрута не работает, проверьте, нет ли там «одинокой» обратной косой черты.

Замечание. *Есть одно исключение из правила относительно обратной косой черты. При разделении имен каталогов в директивах* ***#include*** *их не следует удваивать. Вот пример:*

```
#include "c:\anydir\anyfile.h"
```

Для ввода длинных литеральных строк ведите один символ обратной косой черты, чтобы сообщить компилятору о продолжении на следующей строке. Например, чтобы присвоить длинную строку указателю **p**, вы можете написать:

```
char *p = "Эту длинную строку я \
ввел на нескольких строках, \
которые заканчиваются символами обратной косой черты.";
```

Строки обычно запоминаются одним из трех способов:

- Как литеральные строки, введенные непосредственно в текст программы.
- Как переменные, имеющие фиксированный размер в памяти.
- Как указатели, которые адресуют массивы символов, располагающиеся в динамической памяти.

Очень важно понять природу этих способов запоминания, чтобы знать, как использовать строки, а чуть позже вы увидите, как использовать и строковые функции.

Строковые литералы

Литеральные строки вводятся непосредственно в текст программы. Они должны быть заключены в кавычки.

```
#define TITLE "My program"
```

Символическая константа **TITLE** ассоциируется с литеральной строкой “**My program**”, которая запоминается в сегменте глобальных данных программы. Всего эта строка будет занимать 11 байтов, включая как невидимый завершающий нулевой байт, добавляемый после символа ‘**m**’, так и пробел между двумя словами.

В выражениях компилятор обращается с литеральной строкой, подобной “**My program**”, как с адресом первого символа строки. Поэтому следующий оператор будет корректным:

```
char *stringPtr = "My program";
```

Переменная **stringPtr** объявляется как указатель на значение **char**. Данное присваивание инициализирует указатель **stringPtr** адресом символа ‘**M**’. Этот оператор, несмотря на свой внешний вид, не копирует символы из одной области памяти в другую. Указателю **stringPtr** присваивается адрес, по которому запоминаются символы.

Поскольку массивы и указатели эквивалентны, то следующее выражение также будет приемлемым для компилятора:

```
char stringVar[] = "My program";
```

но в этом случае литеральные символы из строки “**My program**” копируются в область памяти, зарезервированную для массива **stringVar**.

Есть еще одно различие. При сделанном объявлении

```
char *stringPtr = "Литеральная строка";
```

какой-нибудь оператор может позже переприсвоить указателю **stringPtr** адрес другой литеральной строки:

```
stringPtr = "Новая литеральная строка";
```

Однако если вы объявляете строку следующим образом:

```
char stringVar[] = "Литеральная строка";
```

то позже уже не сможете присвоить **stringVar** другую строку

```
stringVar = "Новая литеральная строка";  /* ??? */
```

так как **stringVar** является указателем-константой.

Строковые переменные

Строковая переменная занимает фиксированный объем памяти. Поскольку строки являются массивами, при их объявлении используются квадратные скобки, внутри которых находится целое значение, определяющее длину строки:

```
char stringVar[128];
```

Объявленная таким образом переменная **stringVar** может хранить от 0 до 127 символов плюс завершающий нуль. Если объявление находится вне какой бы то ни было функции, то переменная **stringVar** глобальна, и ею могут пользоваться любые операторы. Все байты в глобальных строках (как и во всех глобальных переменных) устанавливаются равными нулю в начале выполнения программы. Если переменная **stringVar** была объявлена внутри функции, то ее могут использовать операторы только этой функции. Локальные строки (как и другие локальные переменные) временно запоминаются в стеке и не обнуляются.

Строковые указатели

Строковые указатели являются не строками, а указателями, которые определяют местонахождение в памяти первого символа строки. Строковые указатели объявляются как **char *** или, чтобы операторы не могли изменить адресуемые ими данные, как **const char ***.

В строковых указателях нет ничего особенного – они просто указывают на массив значений типа **char** и ведут себя аналогично другим указателям (см. главу 4). Существует много функций, которые обрабатывают строки, адресуемые указателями, и объявления **char *** очень распространены.

Чтобы выделить память в куче для строкового указателя, вызовите функцию **malloc()** и задайте размер, в который не забудьте включить один дополнительный байт для завершающего нуля. Оператор

```
stringPtr = (char *)malloc(81);
```

резервирует 81 байт памяти в куче и присваивает указателю **stringPtr** адрес первого байта. Строка может содержать до 80 символов плюс завершающий нуль. По окончании работы со строкой вызовите функцию **free()**, чтобы вернуть зарезервированную память обратно в кучу, сделав ее доступной для будущих вызовов функции **malloc()**:

```
free(stringPtr);
```

Чтобы обратиться к отдельным символам строки, вы можете использовать квадратные скобки. Например, чтобы отобразить третий символ в строке, адресуемой указателем **stringPtr**, можно написать:

```
printf("%c", stringPtr[2]);
```

При подобном индексировании строк лишь на вас лежит ответственность за то, что в указанном месте находится допустимый символ. Если адресуемая строка содержит меньше трех значащих символов, то этот оператор выведет некорректную информацию.

Нулевые строки и нулевые символы

В программировании на С нуль имеет много значений, и важно понимать различные значения нуля.

- Нулевой символ имеет ASCII-значение, равное нулю, и обычно в программах представляется символической константой **NULL**.
- Строка с завершающим нулем представляет собой массив с нулевым символом после последнего значащего символа в строке. Все строки должны иметь одну дополнительную позицию для завершающего нулевого символа.
- Нулевая строка – это строка, которая начинается с нулевого символа. Длина нулевой строки равна нулю, но ее размер в памяти может занимать больше одного байта. Литеральная нулевая строка записывается как “”.
- Нулевой указатель на строку не адресует никаких достоверных данных – он не является эквивалентом нулевой строки. Чтобы создать нулевой указатель на строку, присвойте указателю значение **NULL**. Чтобы создать нулевую строку, присвойте **NULL** первому символу строки.
- Наконец, символ ‘**0**’ является обычным символом, содержащимся в таблице ASCII, и имеющим десятичный код 48.

Строковые функции

Существует богатая библиотека строковых функций, и все они начинаются с букв **str**, что позволяет легко находить их в описании библиотек. Включите в начало вашего модуля заголовочный файл string.h:

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

чтобы использовать строковые функции.

Отображение строк

Чтобы отобразить строку на экране, передайте ее функции **printf()**. Например, следующие две строки создают, инициализируют и отображают строку с именем **company**:

```
char company[] = "Intel Corporation";
printf(company);
```

Вообще говоря, громоздкая функция **printf()** предназначена для решения более сложных задач форматированного вывода. Для того чтобы отобразить текст и начать новую строку, проще вызвать небольшую функцию **puts()** из файла stdio.h:

```
char processor[] = "Pentium IV";
puts(processor);
```

А вот для вывода форматированных строк **printf()** незаменима:

```
printf("Как жаль, что у моего компьютера нет %s!", processor);
```

Чтение строк

Кроме отображения строк, большинству программ нужно также вводить символы с клавиатуры, дисковых файлов и других источников. Чтобы прочитать строку символов, выполните оператор типа:

```
scanf("%80s", string);
```

Спецификатор `"%80s"` указывает на то, что будет прочитано не более 80 символов, что позволяет избежать переполнения строки. Затем вы можете отобразить результат:

```
puts(string);
```

Если вы захотите испытать этот метод ввода строк, вы можете столкнуться с одной проблемой. Функция **scanf()** завершит ввод на первом же пробеле или символе табуляции, или при нажатии клавиши <Enter>. Например, если вы введете “**Это строка**”, в строке запомнится только “**Это**”. Чтобы ввести строки, содержащие пробелы, используйте библиотечную функцию **gets()**:

```
puts("Введите строку: ");
gets(string);
```

К сожалению, функция **gets()** не защищает пользователей от ввода большего количества символов, чем может вместить строка. Например, объявим такую строку:

```
char string[10];
```

Тогда при вводе 11-го и последующих символов функция **gets()** будет принимать их, записывая за пределами строки поверх данных, которые, к своему несчастью, оказались в этой области. Чтобы предотвратить подобные

ошибки, используйте буферы ввода большого размера (обычно берется размер 128 байтов):

```
char string[128];
```

Преобразование строк в значения

Символьные строки часто используют для ввода значений в программу с клавиатуры. Если вводятся числа, то их необходимо преобразовать в целые значения или значения с плавающей запятой.

Используйте функцию **atof()** для преобразования строк ASCII в значения с плавающей запятой типа **double** (несмотря на то, что функция называется **atof()**, что означает «из ASCII во float», результат будет иметь тип **double**). Для преобразования строк в целые значения воспользуйтесь функцией **atoi()** («из ASCII в integer»). А функция **atol()** поможет в преобразовании строк в значения типа **long**. Все три функции описаны в файле stdlib.h.

Два листинга демонстрируют использование этих функций. Листинг 5.1 преобразует введенную строку в целые значения в различных форматах. Скомпилируйте и запустите эту программу, а затем введите целое число.

Листинг 5.1 convert.c (преобразование строки в целые значения)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3:
 4: main()
 5: {
 6:   int value;
 7:   char string[128];
 8:
 9:   printf("Enter value: ");
10:   gets(string);
11:   value = atoi(string);
12:   printf("Value in decimal = %d\n", value);
13:   printf("Value in hex = %#x\n", value);
14:   printf("Value in octal = %o\n", value);
15:   return 0;
16: }
```

Чтобы поместить введенный с клавиатуры текст в строку, программа пользуется функцией **gets()**. В свою очередь, функция **atoi()** преобразует эту строку в целое значение, присваивая его переменной **value** (строка 11). Затем три оператора **printf()** отображают значение переменной **value** в десятичном (**%d**), шестнадцатеричном(**%#x**) и восьмеричном(**%o**) форматах.

Чтобы преобразовать строки в значения типа **long int**, используйте функцию **atol()**. Например, вы могли бы заменить объявление в строке 6 на **long value**, а затем в строке 11 выполнить оператор **value** = **atol**(**string**).

Следующая программа показывает, как преобразовать строки в значения с плавающей запятой. Листинг 5.2 предлагает ввести значение в милях и отображает эквивалентное расстояние в километрах.

Листинг 5.2 kilo.c (преобразование миль в километры)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3:
 4: main()
 5: {
 6:   double miles;
 7:   char string[128];
 8:
 9:   printf("Convert miles to kilometers\n");
10:   printf("How many miles? ");
11:   gets(string);
12:   miles = atof(string);
13:   printf("kilometers = %lf\n", miles * 1.609344);
14:   return 0;
15: }
```

В строке 12 функция **atof()** преобразует введенный текст в значение с плавающей запятой типа **double**, присваивая его переменной **miles**. Затем оператор **printf()** отображает значение **miles**, умноженное на приблизительное число километров в одной миле (1.609344).

Определение длины строк

Длина строки определяется просто. Для этого нужно передать указатель на строку функции **strlen()**, которая возвратит длину строки в символах. В следующем примере

```
char *s = "Любая строка";
int len = strlen(s);
```

переменная **len** устанавливается равной длине строки, адресуемой указателем **s**. Листинг 5.3 показывает, как использовать функцию **strlen()**.

Листинг 5.3. length.c (использование функции **strlen()**)

```
 1: #include <stdio.h>
 2: #include <string.h>
 3:
 4: #define MAXLEN 128
 5:
 6: main()
 7: {
 8:   char string[MAXLEN];   /* место для 255 символов */
 9:
10:   printf ("\nEnter a string: ");
11:   gets(string);
```

```
12:   puts("");              /* начать новую строку */
13:   puts(string);
14:   printf("Length = %d characters\n", strlen(string));
15:   return 0;
16: }
```

Строка 8 определяет строковую переменную **string**, которая принимает ввод. После того как вы введете строку, программа передаст переменную **string** функции **strlen()**, которая вычислит длину строки в символах (строка 14). Оператор **printf()** в этой же строке листинга отобразит вычисленное значение.

Копирование строк

Оператор присваивания для строк не определен. Если **sl** и **s2** – символьные массивы, вы не сможете скопировать один в другой следующим образом:

```
s1 = s2;   /* ??? */
```

Данный оператор не компилируется. Но если **s1** и **s2** объявить как указатели типа **char ***, компилятор согласится с этим оператором, хотя вряд ли вы получите ожидаемый результат. Вместо копирования символов оператор **s1** = **s2** скопирует значение *указателя* **s2** в *указатель* **s1**. Указатель **s1**, таким образом, будет указывать на то же место в памяти, что и **s2**, а информация, которую он адресовал до этого, может быть утеряна.

Чтобы корректно скопировать одну строку в другую, вызовите функцию **strcpy()**. Для двух указателей **s1** и **s2** типа **char *** оператор

```
strcpy(s1, s2);
```

копирует символы, адресуемые указателем **s2**, в память, адресуемую указателем **s1**, включая завершающие нули. Ответственность за то, что принимающая строка будет иметь достаточно места для хранения новой, лежит на вас.

Функция **strncpy()** аналогична по действию функции **strcpy()**, однако она позволяет ограничивать количество копируемых символов. Оператор

```
strncpy(s1, s2, 10);
```

скопирует 10 символов из строки, адресуемой указателем **s2**, в область памяти, адресуемую указателем **s1**. Если строка **s2** имеет больше 10 символов, то результат усекается. Если же меньше – неиспользуемые байты устанавливаются равными нулю.

Дублирование строк

В больших программах с множеством строковых переменных готовить специальные буферы для функции **gets()** слишком утомительно. Хорошо бы иметь такую функцию, которая позволяла бы вводить строку с клавиатуры и

запоминать ее в куче, чтобы строка занимала ровно столько байтов, сколько требуется. Долой напрасные затраты памяти!

Листинги 5.4 и 5.5 образуют один небольшой модуль с единственной функцией **GetStringAt()**, которая удовлетворяет этим требованиям. Файл gets.h – заголовочный: он содержит только объявления и прототип функции. Файл gets.c – это отдельный модуль, в котором описана функция **GetStringAt()**. Ни один из этих файлов не является законченной программой.

Листинг 5.4. gets.h (заголовочный файл для gets.c)

```
1: /* gets.h - заголовочный файл для gets.c */
2:
3: #define MAXLEN 128 /* максимальный размер строки */
4:
5: char *GetStringAt(int size);
```

Листинг 5.5. gets.c (описание функции получения строки)

```
 1: #include <stdio.h>
 2: #include <string.h>
 3: #include <conio.h>
 4: #include "gets.h"
 5:
 6: /* Замечание: это не полная программа. Вы должны
 7:    скомпоновать этот модуль с главной программой. */
 8:
 9: char *GetStringAt(int size)
10: {
11:   char buffer[MAXLEN];   /* временный буфер ввода */
12:   int i = 0;             /* индекс буфера */
13:   char c;                /* принимает введенные символы */
14:
15:   if ((size > MAXLEN) || (size <= 0)) /* проверить размер */
16:     size = MAXLEN;
17:   while (--size > 0) {   /* ввод строки */
18:     c = getchar();
19:     if (c == '\n')
20:       size = 0;
21:     else
22:       buffer[i++] = c;
23:   }
24:   buffer[i] = '\0';      /* завершение строки нулем */
25:   return strdup(buffer); /* возвращение копии buffer */
26: }
```

Модуль в листинге 5.5 включает свой собственный заголовочный файл (строка 4), который определяет константу **MAXLEN** и объявляет прототип функции **GetStringAt()**.

Оператор **if** в строке 15 проверяет параметр **size**. Если его значение находится вне заданного диапазона, то ему присваивается значение

MAXLEN. В строке 17 цикл **while** вызывает функцию **getchar()** (строка 18), которая ожидает, пока вы введете символ. Программа присваивает этот символ переменной **c** и проверяет его на совпадение с управляющим символом '**\n**' («новая строка», означающая, что вы нажали <Enter>). Если это условие оказалось выполненным, программа устанавливает параметр **size** равным нулю, завершая тем самым цикл **while**. В противном случае программа присваивает значение переменной **c** элементу массива **buffer** с индексом **i**. Индекс инкрементируется оператором ++ для подготовки к вводу следующего символа

Замечание. *Выполните цикл **while** в пошаговом режиме отладчика, наблюдая при этом за значениями переменных **buffer**, **c** и **i**. Это хороший способ узнать, как цикл **while** вводит строку.*

В строке 24 добавляется завершающий нулевой символ, служащий признаком конца строки. И, наконец, строка 25 возвращает результат функции **GetStringAt()**. Этот оператор передает строку **buffer** функции **strdup()**, которая размещает копию строки в динамической памяти.

Новая строка занимает столько памяти, сколько необходимо. Если переменная **s** имеет тип **char ***, то оператор

```
s = strdup("Двойная тревога");
```

выделит ровно 16 байтов памяти кучи, скопирует в эту область памяти 15-символьную строку «Двойная тревога» плюс завершающий нуль и возвратит адрес этой области. По окончании работы со строкой следует освободить эту область памяти обычным способом:

```
free(s);
```

Предупреждение. *Вы можете модифицировать строки, создаваемые функцией **strdup()**, но при этом не должны расширять их за пределы отведенных им объемов памяти. Если все-таки это необходимо сделать, скопируйте вашу строку в новый, больший по размеру буфер, зарезервированный функцией **malloc()**, после чего освободите память, которую строка занимала ранее.*

Листинг 5.6 представляет собой основную программу, которая использует функцию **GetStringAt()**, объявленную в файле gets.h и описанную в модуле gets.c. Чтобы создать работающую программу, вы должны скомпоновать эти модули вместе. Для этого откройте новый файл проекта, добавьте в него модули duped.c и gets.c, а затем выберите команду **Run** в вашей среде программирования. Эта команда: 1) скомпилирует модуль gets.c, создавая объектный файл с именем gets.obj, 2) скомпилирует модуль duped.c и создаст объектный файл duped.obj, 3) скомпонует эти два объектных файла и 4) создаст файл duped.exe, который вы можете запускать на исполнение.

Листинг 5.6. duped.c (использование функции **GetStringAt()**)

```
1: #include <stdio.h>
2: #include <conio.h>
```

```
 3: #include <string.h>
 4: #include <alloc.n>
 5: #include "gets.h"
 6:
 7: #define PROMPT "String: " /* приглашение на ввод */
 8:
 9: main()
10: {
11:   char *s; /* указатель на результат функции GetStringAt()*/
12:
13:   clrscr();
14:   printf(PROMT);
15:   s = GetStringAt(MAXLEN);
16:   if (s) {
17:     puts("Your entry is: ");
18:     puts(s);
19:     printf("Length = %d characters\n", strlen(s));
20:     free(s);
21:   } else
22:     puts("Error duplicating string!");
23:   return 0;
24: }
```

Строка 15 вызывает функцию **GetStringAt()**, передавая ей в виде параметра максимальную длину строки. В строке 16 программа проверяет результат функции **GetStringAt()**. Если она возвращает нуль, это значит, что функция **strdup()** не смогла создать копию строки, возможно, из-за недостатка памяти в куче. Обратите внимание также на освобождение памяти в строке 20 после того, как строка оказывается больше не нужной.

Упражнение. *Создайте рекурсивную версию функции* ***GetStringAt()***.

Сравнение строк

С помощью функции **GetStringAt()** вы можете написать программу, которая предлагает ввести пароль. Чтобы определить, правильный ли вы ввели пароль, листинг 5.7 вызывает функцию **strcmp()**, которая сравнивает две строки.

Программа password вызывает функцию **GetStringAt()** из модуля gets.c. Чтобы создать исполняемый файл, вы должны скомпоновать программу с этим модулем. Для компиляции, компоновки и выполнения программы откройте файл проекта под именем password, добавьте в него модули password.c и gets.c, а затем выполните команду Run. (Если программа не запуститься, попробуйте команду Build All. Проверьте также имена маршрутов каталогов в проекте).

Чтобы программа password успешно завершилась, введите пароль “**Informatics**” с учетом прописных и строчных букв. Если вы введете пароль

неправильно, программа отобразит сообщение об ошибке и потребует начать сначала.

Листинг 5.7. password.c (предложение ввести пароль)

```
 1: #include <stdio.h>
 2: #include <conio.h>
 3: #include <string.h>
 4: #include <alloc.h>
 5: #include "gets.h"
 6:
 7: #define FALSE 0
 8: #define TRUE 1
 9: #define PASSWORD "Informatics"
10: #define PROMPT "Enter password: "
11:
12: main()
13: {
14:   char *s;
15:   int done = FALSE;
16:
17:   clrscr();
18:   while (!done) {
19:     printf(PROMPT);
20:     s = GetStringAt(MAXLEN);
21:     clrscr();
22:     if (!s) {
23:       puts("Error: Out of memory");
24:       return 1;
25:     } else {
26:       done = (strcmp(s, PASSWORD) == 0);
27:       if (!done)
28:         puts("Error in password! Type "PASSWORD" to quit");
29:       free(s); /* освобождаем память */
30:     }
31:   }
32:   clrscr();
33:   puts("Correct password given");
34:   return 0;
35: }
```

В строке 26 показано, как сравнивать две строки. Оператор

```
done = (strcmp(s, PASSWORD) == 0);
```

устанавливает переменную **done** равной значению «истина», если строка, адресуемая указателем **s**, равна строке “**Informatics**”, представленной константой **PASSWORD**.

Если **i** – переменная типа **int**, и если **s1** и **s2** – указатели на **char**, то оператор

```
i = strcmp(s1, s2);
```

установит **i** равной –1 или другому отрицательному числу, если строка, адресуемая указателем **s1**, в алфавитном порядке меньше строки, адресуемой указателем **s2**. Если строки в точности совпадают, функция возвратит нуль. Если строка **s1** в алфавитном порядке больше строки **s2**, она вернет +1 или другое положительное число.

Функция **strcmp()** чувствительна к регистру букв – при сравнении строчные буквы считаются б*о*льшими, чем их прописные эквиваленты (так как буквы нижнего регистра имеют б*о*льшие ASCII-значения, чем буквы верхнего регистра). Для сравнения двух строк без учета регистра вызовите функцию **stricmp()**. Буква **i** означает «ignore case» («игнорировать регистр»). Эта функция действует аналогично функции **strcmp()**, но перед сравнением преобразует все буквы в прописные. Например, если сравнение выполнять с помощью функции **strcmp()**, то строка “**Apple**” алфавитно окажется меньше строки “**apple**”. Если же для сравнения использовать функцию **stricmp()**, эти строки будут считаться идентичными.

Чтобы сравнить только часть двух строк, используйте функцию **strncmp()**. Например, оператор

```
i = strncmp(s1, s2, 2);
```

установит целую переменную **i** равной нулю только в том случае, если первые два символа строк, адресуемых указателями **s1** и **s2**, в точности совпадают. Для сравнения части строк без учета регистра вызывайте функцию **strnicmp()**.

Конкатенация строк

Конкатенация двух строк означает их соединение, при этом создается новая, более длинная строка. При объявлении строки

```
char original[128] = "Проверка ";
```

оператор

```
strcat(original, "связи");
```

превратит первоначальную строку **original** в “**Проверка связи**”.

При вызове функции **strcat()** убедитесь, что первый аргумент типа **char *** инициализирован и имеет достаточно места, чтобы запомнить результат, в противном случае может возникнуть серьезная ошибка.

Функция **strcat()** возвращает адрес результирующей строки (совпадающий с ее первым параметром) и может использоваться как каскад нескольких вызовов функций:

```
strcat(strcat(s1, s2), s3);
```

Этот оператор добавляет строку, адресуемую **s2**, и строку, адресуемую **s3**, к строке, адресуемой **sl**, что эквивалентно двум отдельным операторам:

```
strcat(s1, s2);
strcat(s1, s3);
```

Листинг 5.8 показывает, как использовать функцию **strcat()** для решения типичной проблемы – получения полного имени человека из фамилии, имени и отчества, запомненных по отдельности, например в базе данных. Скомпилируйте программу concat аналогично предыдущим примерам: откройте новый файл проекта, добавьте в него модули concat.c и gets.c и запустите программу на исполнение. Введите фамилию, имя и отчество. Программа соединит введенные вами строки и отобразит имя как одну строку.

Листинг 5.8. concat.c (конкатенация строк)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <conio.h>
 4: #include <string.h>
 5: #include <alloc.h>
 6: #include "gets.h"
 7:
 8: char *PromptFor(char *prompt);
 9:
10: main()
11: {
12:   char *firstName;
13:   char *middleName;
14:   char *lastName;
15:   char fullName[128];
16:
17:   clrscr();
18:   firstName = PromptFor("first name");
19:   middleName = PromptFor("middle name");
20:   lastName = PromptFor("last name");
21:   strcpy(fullName, firstName);
22:   strcat(fullName, " ");
23:   strcat(fullName, middleName);
24:   strcat(fullName, " ");
25:   strcat(fullName, lastName);
26:   printf ("Your name is: %s", fullName);
27:   free(firstName);   /* освобождает */
28:   free(middleName);  /* выделенную  */
29:   free(lastName);    /* память      */
30:   return 0;
31: }
32:
33: char *PromptFor(char *prompt)
34: {
35:   char *temp;  /* временный строковый указатель */
36:
37:   printf("Enter your %s: ", prompt);
38:   temp = GetStringAt(MAXLEN);
39:   if (!temp) {
40:     puts("\nError: Out of memory");
```

```
41:     exit(1);
42:   }
43:   return temp;
44: }
```

Строки 21-25 демонстрируют важный принцип конкатенации строк: всегда инициализируйте первый строковый аргумент. В данном случае символьный массив **fullName** инициализируется вызовом функции **strcpy()**, которая копирует символы **firstName** в **fullName**. После этого программа добавляет пробелы и две другие строки – **middleName** и **lastName**.

Функция **PromptFor** (строки 33-44) выводит приглашение и возвращает строку, введенную пользователем. Кроме того, функция **PromptFor** демонстрирует способ обработки исключительных ситуаций. Если функция **GetStringAt()** не сможет выделить память в куче и вернет **NULL**, то в строке 41 будет вызвана функция **exit()**, которая немедленно завершит программу и вернет операционной системе некоторое значение, позволяющее идентифицировать ошибку.

Если вы не уверены в том, что в строке достаточно места для присоединяемых подстрок, вызовите функцию **strncat()**, которая аналогична функции **strcat()**, но требует дополнительного аргумента, определяющего число копируемых символов. Для строк **s1** и **s2**, которые могут быть либо указателями типа **char ***, либо символьными массивами, оператор

```
strncat(s1, s2, 4);
```

присоединяет максимум четыре символа из **s2** в конец строки **s1**. Результат обязательно завершается нулевым символом.

Поиск элементов строк

Программам часто приходится выполнять поиск символов в строках. Распространенный пример – проверка имен файлов на заданное расширение. Например, после того как пользователю предложили ввести имя файла, проверяется, ввел ли он расширение “.txt”, и если это так, то выполняется заданное действие.

Поиск символов

Листинг 5.9 показывает, как использовать функцию **strchr()** для поиска отдельных символов в строке.

Листинг 5.9. ext1.c (проверка расширения файла, пример 1)

```
1: #include <stdio.h>
2: #include <string.h>
3:
4: main()
5: {
6:   char fileName[128];
```

```
 7:
 8:   printf("Enter file name: ");
 9:   gets(fileName);
10:   printf("As entered: %s\n", fileName);
11:   if (strchr(fileName, '.'))
12:     printf("File name is probably complete\n");
13:   else
14:     strcat(fileName, ".txt");
15:   printf("Final file name: %s\n", fileName);
16:   return 0;
17: }
```

Скомпилируйте и запустите программу ext1, затем введите имя файла с расширением. Например, если вы введете “**test.txt**”, программа отобразит:

```
Enter file name: test.txt
As entered: test.txt
File name is probably complete
Final file name: test.txt
```

Но если вы введете “**test**” без расширения, программа добавит к имени файла расширение:

```
Enter file name: test
As entered: test
Final file name: test.txt
```

Программа ext1 находит расширение в имени файла, выполняя поиск символа точки во введенной строке. Ключевым в этой программе является оператор **if-else** (строки 11-14):

```
if (strchr(fileName, '.'))
  printf("File name is probably complete\n");
else
  strcat(fileName, ".txt");
```

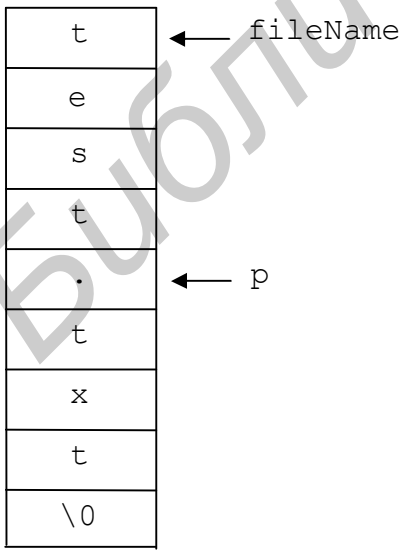

Рис. 5.2. Функция **strchr()** находит символ в строке

Выражение **strchr(fileName, ‘.’)** возвращает указатель на символ точки в строке **fileName**. Если такой символ не найден, функция **strchr()** возвращает нуль. Поскольку ненулевые значения означают «истину», вы можете использовать функцию **strchr()** в качестве логического выражения в операторах **if**.

Присвоим указателю **p** типа **char** значение, возвращаемое функцией **strchr()**:

```
p = strchr(fileName, '.');
```

Теперь на строку **fileName** указывают два указателя: **fileName**, адресующий полную строку, и **p**, адресующий подстроку “**.txt**” (рис. 5.2).

Строковые функции не заботятся о байтах, которые предшествуют их первому символу, поэтому оператор

```
puts(p);
```

отображает подстроку “**.txt**” так, будто она полная строковая переменная, а не часть другой строки. Здесь вы должны проявить осторожность. Если строка расположена в куче, то оператор

```
free(fileName);
```

корректно освобождает занимаемую строкой память, а оператор

```
free(p); /* ??? */
```

вызовет трудноуловимую ошибку. Никогда не используйте указатели на подстроки для освобождения памяти, занимаемой целой строкой.

Функция **strchr()** отыскивает первое появление символа в строке. Чтобы найти последнее появление, вызовите функцию **strrchr()**. Операторы

```
char s[] = "Abracadabra";
char p = strrchr(s, 'b');
```

установят указатель **p** равным адресу подстроки “**bra**” в конце строки “**Abracadabra**”.

Поиск подстрок

Кроме поиска символов в строке, вы можете «поохотиться» и за подстроками. Листинг 5.10 демонстрирует этот метод.

Листинг 5.10. ext2.c (проверка расширения файла, пример 2)

```
 1: #include <stdio.h>
 2: #include <string.h>
 3:
 4: main()
 5: {
 6:   char fileName[128];
 7:   char *p;
 8:
 9:   printf("Enter file name: ");
10:   gets(fileName);
11:   printf("As entered: %s\n", fileName);
12:   strlwr(fileName);
13:   p = strstr(fileName, ".txt");
14:   if (p)
15:     printf("File name is complete\n");
16:   else {
17:     p = strchr(fileName, '.');
18:     if (p)
19:       *p = NULL; /* удалить любое другое расширение */
20:     strcat(fileName, ".txt");
21:   }
22:   printf("Final file name: %s\n", fileName);
23:   return 0;
```

```
24: }
```

Новая программа создает имя файла, которое обязательно заканчивается расширением “.txt”. Запустите программу ext2 и введите “**test”** или “**test.txt**”, что даст результат, аналогичный программе ext1. Но если вы введете “**test.c**”, программа ext2 отобразит следующее:

```
Enter file name: test.c
As entered: test.c
Final file name: test.txt
```

Чтобы определить, есть ли в имени файла расширение “.txt”, программа выполняет в строке 13 оператор

```
p = strstr(fileName, ".txt");
```

Подобно **strchr()**, функция **strstr()** возвращает адрес искомой подстроки или нуль, если подстрока не найдена. Поскольку расширение может быть введено и прописными буквами, программа выполняет оператор

```
strlwr(fileName);
```

чтобы перед вызовом функции **strstr()** преобразовать буквы введенной строки в строчные. Используйте оператор

```
strupr(fileName);
```

чтобы преобразовать все буквы строки **fileName** в прописные.

Программа ext2 также демонстрирует способ усечения строки в позиции заданного символа. Строка 17 вызывает функцию **strchr()**, чтобы установить указатель **p** равным адресу первой точки в строке **fileName**. Если результат этого поиска не нулевой (строка 18), строка 19 выполнит оператор, который запишет вместо точки нулевой байт:

```
*p = NULL;
```

Теперь строка готова к добавлению нового расширения путем вызова функции **strcat()** в строке 20.

Упражнение. *Модифицируйте программу ext2, чтобы она не использовала функцию **strstr()** для поиска расширения файла. Другими словами, ваша программа должна заменить расширение файла на “.txt” без обязательного поиска этого расширения.*

Разложение строк на подстроки

Прикладное программирование часто требует нахождения компонентов строки или *лексем* – этот процесс называется синтаксическим анализом. Если составные части строки отделены друг от друга запятыми, пробелами или другими разделителями, вы можете использовать мощную функцию **strtok()** для разложения ее на несколько подстрок.

Рассмотрим следующие объявления:

```
char s[] = "Now I know my ABCs";
char *p1, *p2, *p3, *p4, *p5;
```

Строковая переменная **s** инициализирована строкой “**Now I know my ABCs**”. Пять указателей на тип **char** пока что не инициализированы – функция **strtok()** использует эти указатели, чтобы сделать разбор строки. Сначала вызовите функцию **strtok()**, передав в качестве первого аргумента указатель на строку, а в качестве второго – разделительный символ:

```
p1 = strtok(s, " ");
```

Функция **strtok()** возвращает адрес первого компонента строки, отделенного от следующего заданным разделителем (в данном примере – символом пробела) или **NULL**, если разделитель не обнаружен. Разделителей может быть несколько. Например, чтобы разложить строки на элементы, разделенные запятыми, точками с запятой или пробелами, используйте в качестве второго аргумента строку `",; "`.

Функция вставляет нулевой байт в позицию первого обнаруженного разделителя. Тем самым создается маленькая подстрока, адрес которой возвращается функцией **strtok()**. Кроме того, эта функция настраивает свой внутренний указатель на символ, следующий за концом сформированной подстроки, чтобы следующий вызов **strtok()** мог продолжить разложение строки.

Чтобы выделить другие компоненты строки **s**, нужно просто несколько раз вызвать функцию **strtok()** с первым аргументом, имеющим значение **NULL**:

```
p2 = strtok(NULL, " ");
p3 = strtok(NULL, " ");
p4 = strtok(NULL, " ");
p5 = strtok(NULL, " ");
```

На рис. 5.3 показана разобранная строка и ее указатели от **p1** до **p5**. Каждый указатель адресует завершающуюся нулем подстроку внутри первоначальной строки. Каждая из подстрок является отдельной строковой переменной, которую можно передавать строковым функциям, таким как **strlen()**, **strcpy()** и **strdup()**.

Замечание. *Функция **strtok()** модифицирует исходную строку, поэтому перед ее разбором убедитесь, что вы сделали копию на случай необходимости использования строки в первоначальном виде*.

В предыдущем примере предполагалось, что можно было заранее узнать, сколько компонентов имеет строка. В большинстве же случаев это невозможно, поэтому, чтобы выделить составляющие строки, более разумно использовать цикл:

```
p = strtok(string, ";");
while (p) {
  /* обработка подстроки, адресуемой указателем p */
  p = strtok(NULL, ";");
}
```

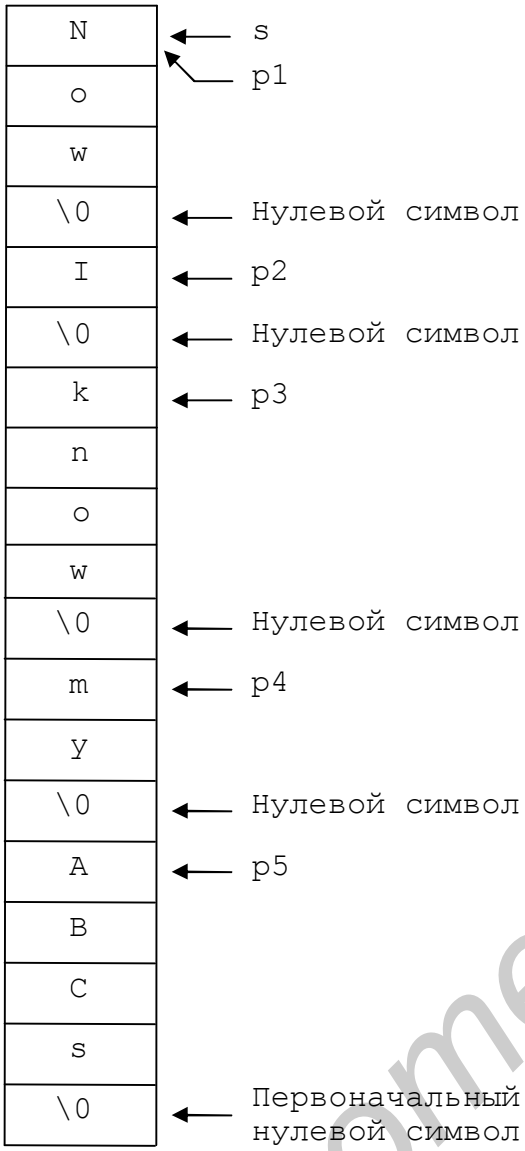

Рис. 5.3. Строка после разбора с помощью функции **strtok()**

Сначала указатель **p** устанавливается равным результату функции **strtok()**, которой были переданы указатель на исходную строку **string** и разделитель "**;**". Затем цикл **while** проверяет значение указателя **p** на равенство нулю. Если указатель имеет ненулевое значение, то он адресует очередную подстроку в строке **buffer**, и вы можете передавать его строковым функциям. После обработки найденной подстроки внутри цикла осуществляется попытка поиска других подстрок с помощью нового вызова функции **strtok()**, но теперь со значением **NULL** в качестве первого аргумента.

Листинг 5.11 показывает, как использовать функцию **strtok()** для выделения слов из строки. Скомпилируйте и запустите программу. После приглашения введите строку, состоящую из слов, разделенных пробелами. Операторы в строках 14-18 выполнят анализ введенного текста и отобразят его результаты на экране в виде отдельных слов.

Листинг 5.11. tokens.c (анализ строки слов)

```
 1: #include <stdio.h>
 2: #include <string.h>
 3:
 4: main()
 5: {
 6:   char buffer[128];
 7:   char *p;
 8:
 9:   printf("Enter a string of words separated by blanks.\n");
10:   printf(": ");
```

```
11:   gets(buffer);
12:   printf("As entered: %s\n", buffer);
13:   printf("As words (tokens):\n");
14:   p = strtok(buffer, " ");
15:   while (p) {
16:     puts(p);
17:     p = strtok(NULL, " ");
18:   }
19:   return 0;
20: }
```

Резюме

- Строка представляет собой массив значений типа **char**, заканчивающийся нулевым байтом. Если строка не заполняет полностью выделенную для нее память, то байты, находящиеся после завершающего нуля, остаются неиспользованными.
- Символы строки – это на самом деле целые числа, представляющие собой номера символов в таблице ASCII.
- Строки имеют одну из трех форм: литеральные строки, вводимые непосредственно в текст программы, массивы символов фиксированного размера и указатели типа **char ***, которые обычно адресуют строки, хранящиеся в куче.
- Нулевая строка не имеет никаких значащих символов, кроме нуля, запомненного в ее первом байте.
- Строки можно использовать для ввода в программу целых и дробных чисел.
- Функция **puts()** отображает строку на экрана. Используйте эту быструю функцию для простого вывода строки. Для более сложного форматированного вывода воспользуйтесь могущественной **printf()**.
- С помощью функции **scanf()** в строку можно ввести заданное количество символов. Для ввода строк, содержащих пробелы, используйте **gets()**.
- Файл string.h содержит богатую библиотеку строковых функций (см. следующий раздел). Многие строковые функции возвращают указатели типа **char ***, поэтому результат одной функции может передаваться другой в качестве аргумента.

Обзор функций

Таблица 5.2

Функции для работы со строками

Функция	Прототип и краткое описание
1	2
	Библиотека stdio.h
gets()	`char *gets(char *str);` Получает следующую строку со стандартного потока ввода и записывает ее в **str**; в случае успешного завершения функция возвращает **str**, в противном случае – **NULL**.
puts()	`int puts(const char *str);` Записывает строку в стандартный поток вывода. В случае ошибки функция возвращает значение **EOF**.
sprintf()	`int sprintf(char *str, const char *format, ...);` Копирует форматированную строку **format** в указанную строку. В случае успешного завершения функция возвращает количество записанных байтов информации (не считая завершающего нуля); в противном случае возвращается значение **EOF**.
sscanf()	`int sscanf(const char *str, const char *format, ...);` Считывает форматированный ввод из указанной строки (см. функцию **scanf()**).
	Библиотека stdlib.h
atof()	`double atof(const char *str);` Преобразует строку **str** в вещественное число типа **double**. Преобразование завершается по достижении первого символа, который не является частью числа. Возвращается нуль, если не было найдено ни одного числа.
atoi()	`int atoi(const char *str);` Преобразует строку **str** в целое число типа **int**. Преобразование завершается по достижении первого символа, который не является частью числа. Возвращается нуль, если не было найдено ни одного числа.
atol()	`long atol(const char *str);` Преобразует строку **str** в целое число типа **long**. Преобразование завершается по достижении первого символа, который не

Продолжение табл. 5.2

1	2
	является частью числа. Возвращается нуль, если не было найдено ни одного числа.
itoa()	`char *itoa(int value, char *str, int base);` Преобразует целое число **value** в строку **str**. Для преобразования числа используется основание **base** (2 ≤ **base** ≤ 36). Для десятеричной системы **base** равно 10.
ltoa()	`char *ltoa(long value, char *str, int base);` Преобразует переменную **value** типа **long** в строку **str**. Для преобразования числа используется основание **base** (2 ≤ **base** ≤ 36). Для десятеричной системы **base** равно 10.
ultoa()	`char *ultoa(unsigned long value, char *str, int base);` Преобразует беззнаковую переменную **value** типа **long** в строку **str**. При изображении числа используется основание **base** (2 ≤ **base** ≤ 36). Для десятеричной системы **base** равно 10.
	Библиотека string.h
strcat()	`char *strcat(char *str1, const char *str2);` Объединяет строки **str1** и **str2** (конкатенация строк); результат записывается в **str1**. Функция возвращает **str1**.
strchr()	`char *strchr(const char *str, int c);` Возвращает указатель на первое вхождение в строке **str** символа **c**. Если символ не найден, возвращает **NULL**.
strcmp()	`int strcmp(const char *str1, const char *str2);` Сравнивает строки **str1** и **str2**. Результат отрицателен, если строка **str1** алфавитно меньше строки **str2**; равен нулю, если строки равны; положителен, если строка **str1** алфавитно больше строки **str2**.
strcpy()	`char *strcpy(char *str1, const char *str2);` Копирует строку **str2** в строку **str1**. Возвращает **str1.**
strdup()	`char *strdup(const char *str);` Выделяет память в куче и копирует в нее строку **str**. В случае успешного завершения возвращает указатель на копию строки в куче, в противном случае возвращает **NULL**.

Продолжение табл. 5.2

1	2
stricmp()	`int stricmp(const char *str1, const char *str2);` Сравнивает строки **str1** и **str2**, не делая различия регистров. Результат отрицателен, если строка **str1** алфавитно меньше строки **str2**; равен нулю, если строки равны; положителен, если строка **str1** алфавитно больше строки **str2**.
strlen()	`unsigned strlen(const char *str);` Возвращает длину строки **str**, не считая завершающий нулевой символ '**\0**'.
strlwr()	`char *strlwr(char *str);` Преобразует буквы верхнего регистра в строке (A-Z) в соответствующие буквы нижнего регистра (a-z). Возвращает преобразованную строку.
strncat()	`char *strncat(char *str1, const char *str2, int n);` Дописывает **n** символов строки **str2** к строке **str1** (конкатенация строк). Возвращает **str1**.
strncmp()	`int strncmp(const char *str1, const char *str2,int n);` Сравнивает первые **n** символов строк **str1** и **str2**. Результат отрицателен, если строка **str1** алфавитно меньше строки **str2**; равен нулю, если строки равны; положителен, если строка **str1** алфавитно больше строки **str2**.
strncpy()	`char *strncpy(char *str1, const char *str2, int n);` Копирует **n** символов строки **str2** в строку **str1**. Возвращает **str1**.
strcspn()	`int strcspn(const char *str1, const char *str2);` Возвращает длину первого сегмента строки **str1**, содержащего символы, не входящие во множество символов строки **str2**.
strnicmp()	`int strnicmp(char *str1, const char *str2, int n);` Сравнивает **n** символов строки **str1** и **str2**, не делая различия регистров. Результат отрицателен, если строка **str1** алфавитно меньше строки **str2**; равен нулю, если строки равны; положителен, если строка **str1** алфавитно больше строки **str2**.
strnset()	`char *strnset(char *str, int c, int n);` Заменяет первые **n** символов строки **str** символом **c**. Возвращает модифицированную строку.

Окончание табл. 5.2

1	2
strpbrk()	`char *strpbrk(const char *str1, const char *str2);` Ищет в строке **str** первое появление любого из множества символов, входящих в строку **str2**. В случае успешного поиска возвращает указатель на найденный символ, в противном случае возвращает **NULL**.
strrchr()	`char *strrchr(const char *str, int c);` Возвращает указатель на последнее вхождение символа **c** в строку **str**. Если символ не найден, возвращает **NULL**.
strset()	`char *strset(char *str, int c);` Заменяет все символы строки (за исключением завершающего нуля) на заданный символ **c**. Возвращает модифицированную строку.
strstr()	`char *strstr(const char *str1, const char *str2);` Ищет в строке **str1** подстроку **str2**. Возвращает указатель на символ в строке **str1**, с которого начинается **str2**, или **NULL**, если подстрока не найдена.
strtok()	`char *strtok(char *str1, const char *str2);` Ищет в строке **str1** лексемы, разделенные символами из строки **str2**.
strupr()	`char *strupr(char *str);` Преобразует буквы нижнего регистра (a-z) в строке **str** в буквы верхнего регистра (A-Z) и возвращает модифицированную строку.

Кроме функций работы со строками библиотека string.h определяет несколько функций, которые оперируют памятью на общем уровне.

Таблица 5.3

Функции работы с памятью (string.h)

Функция	**Прототип и краткое описание**
1	2
memchr()	`void *memchr(const void *s, int c, size_t n);` Ищет первое вхождение символа **c** среди исходных **n** символов объекта, на который указывает **s**; возвращает указатель на первое появление символа либо **NULL**, если символ не найден.

Окончание табл. 5.3

1	2
memcmp()	`int memcmp(const void *s1, const void *s2, size_t n);` Сравнивает посимвольно две области памяти **s1** и **s2** длиной **n** байтов; интерпретирует каждое значение как **unsigned char**. Возвращает нуль, если объекты одинаковы; отрицательное значение, если первый объект численно меньше, чем второй; положительное значение, если первый объект больше, чем второй.
memcpy()	`void *memcpy(void *dest, const void *src, int n);` Копирует **n** байтов из области памяти, на которую указывает **src**, в область, указанную **dest**. Возвращает **dest**. Если указатели **dest** и **src** адресуют перекрывающиеся области памяти, результат непредсказуем.
memicmp()	`int memicmp(const void *s1, const void *s2, unsigned n);` Работает аналогично **memcmp()**, но не делает различия регистров.
memmove()	`void *memmove(void *dest, const void *src, int n);` Копирует **n** байтов из **src** в **dest**. Возвращает указатель **dest**. Функция аналогична **memcpy()** за тем исключением, что если указатели **dest** и **src** адресуют перекрывающиеся области памяти, копирование происходит корректно.
memset()	`void *memset(void *s, int c, unsigned n);` Записывает во все байты области **s** значение **c**. Длина области **s** равна **n** байтов. Возвращает указатель **s**.

Таблица 5.4

Функции, завершающие работу программы (stdio.h)

Функция	Прототип и краткое описание
exit()	`void exit(int status);` Немедленно прекращает выполнение программы. Значение **status**, равное нулю, обычно говорит о нормальном завершении программы. Ненулевое значение свидетельствует об ошибках.

6. Структуры

Структуры помогают упорядочить данные, объединяя несколько переменных «под одной крышей». В отличие от массивов, которые могут содержать значения только одного типа, структуры могут состоять из значений различных типов данных.

Чтобы объявить структуру, напишите ключевое слово **struct** и добавьте к нему идентификатор (имя структуры) со списком переменных в скобках (членами структуры). Объявление каждого члена и всей структуры в целом должны заканчиваться точкой с запятой.

```
struct coordinate {
  int x;
  int y;
};
```

Структура **coordinate** имеет два члена типа **int**: **x** и **y**. Такую структуру удобно использовать, например, для описания точек на карте или графике. Теперь объявим переменную структурного типа:

```
struct coordinate point;
```

Слово **struct** в таком объявлении нужно включить обязательно, несмотря на то, что вы уже объявили структурный тип ранее. Следующая запись не будет скомпилирована:

```
coordinate p;  /* ??? */
```

Чтобы избежать повторного набора слова **struct**, при объявлении структуры вы можете воспользоваться услугами оператора **typedef**:

```
typedef struct coordinate {
  int x;
  int y;
} Coordinate;
```

Как вы знаете, **typedef** не создает новый тип данных – он только связывает существующий тип данных с псевдонимом, который обычно пишется с прописной буквы. При объявлении структуры с помощью этого оператора вы можете опустить имя структуры и просто записать:

```
typedef struct {
  int x;
  int y;
} Coordinate;
```

Теперь с помощью нового псевдонима можно объявить структурную переменную:

```
Coordinate point;
```

Переменная **point** содержит два целых значения: **x** и **y**. Чтобы получить доступ к ним, используйте запись через точку. Например:

```
point.x = 5;
```

```
point.y = 6;
```

Имена членов должны быть уникальными внутри одной и той же структуры, но они не вступают в конфликт с именами других переменных. Например, не будет ошибкой, если в программе будет объявлена структура **Coordinate**, а также переменные **x** и **y**.

Структуры могут запоминать переменные любых типов, включая массивы и даже другие структуры. Вот образец сложной структуры**:**

```
typedef struct complexStruct {
  float aFloat;
  int anInt;
  char aString[8];
  char aChar;
  long aLong;
} ComplexStruct;
```

ComplexStruct представляет собой структуру, состоящую их пяти членов: четырех переменных и одного массива. Переменная **data**, объявленная как

```
ComplexStruct data;
```

запоминает эти значения в одной удобной «упаковке».

Сравнивание и присваивание структур

Вы можете присваивать значения одной структурной переменной другой при условии, что эти переменные имеют один и тот же тип. После объявления

```
Coordinate var1, var2;
```

Следующий оператор присваивает значения переменной **var2** переменной **var1**:

```
var1 = var2;
```

Непосредственно сравнивать две структуры нельзя. *Оператор*, подобный следующему, не будет скомпилирован:

```
if (var1 == var2)  /* ??? */
  оператор;
```

Но если вы не можете сравнивать две структуры, то никто не запрещает вам сравнивать их члены. Вот корректный способ выполнения *оператора*, если переменные **var1** и **var2** равны:

```
if ((var.x == var2.x) && (var1.y == var2.y))
  оператор;
```

Инициализация структур

Дата – это хороший пример для того, чтобы показать, как структуры помогают упорядочить данные. Три компоненты даты – день, месяц и год – удобно представлять целыми числами. Вы можете объявить структуру даты следующим образом:

```
typedef struct date {
  char day;
  char month;
  unsigned year;
} Date;
```

Члены структуры **day** и **month** имеют тип **char**, диапазон значений которого вполне достаточен, чтобы вместить значения дня (от 1 до 31) и месяца (от 1 до 12). Член структуры **year** имеет тип **unsigned int**. Листинг 6.1 демонстрирует использование структуры **Date**.

Листинг 6.1. date.c (запоминание даты в структуре)

```
 1: #include <stdio.h>
 2:
 3: typedef struct dateStruct {
 4:   char day;
 5:   char month;
 6:   unsigned year;
 7: } Date;
 8:
 9: main()
10: {
11:   Date date;
12:
13:   printf("Date test\n");
14:   date.day = 1;
15:   date.month = 9;
16:   date.year = 2004;
17:   printf("The date is: %02d.%02d.%04d\n",
18:     date.month, date.day, date.year);
19:   return 0;
20: }
```

В строках 14-16 происходит присваивание значений членам структуры **date**. В строках 17-18 оператор **printf()** выводит эти значения на экран, добавляя точки и ведущие нули для удобства отображения:

```
Date test
The date is: 01.09.2004
```

Вы можете присваивать начальные значения членам структуры прямо в объявлении. Используйте следующую форму записи:

```
struct name var = {элементы};
```

Здесь **name** – имя структуры, **var** – переменная данного структурного типа, *элементы* – константные выражения для присваивания членам структуры. Отделяйте константы запятыми, располагая их в соответствии с порядком объявленных в структуре членов. Например, следующая запись присвоит дату 1.09.2010 переменной **date**:

```
Date date = {1, 9, 2010};
```

Использование вложенных структур

Если членом структуры является другая структура, то в результате получаем вложенную структуру – одна структура внутри другой. Предположим, вы объявляете структуру для запоминания времени:

```
typedef struct time {
  char hour;       /* от 0 до 23 */
  char minute;     /* от 0 до 59 */
  char second;     /* от 0 до 59 */
} Time;
```

Используя структуру **Date**, описанную в предыдущем разделе, вы можете построить вложенную структуру **DateTime** следующим образом:

```
typedef struct dateTime {
  Date theDate;
  Time theTime;
} DateTime;
```

Структура **DateTime** объявляет два члена: **theDate** и **theTime**, причем каждый их них является структурой. После объявления переменной **dt**

```
DateTime dt;
```

выражение **dt**.**theDate** обозначает член **theDate** типа **Date** структуры **dt**, а **dt**.**theTime** – член **theTime** типа**Time**.

Замечание. *Структура не может объявлять в качестве члена структуру своего собственного типа. Другими словами, структура не может объявлять членом саму себя. Однако членом структуры может быть указатель на структуру того же типа.*

Для доступа к вложенным членам структур используйте необходимое число точек. Например, выражение **dt**.**theDate**.**month** ссылается на член **month**, принадлежащий структуре **theDate**, которая, в свою очередь, принадлежит структуре **dt**.

Листинг 6.2 расширяет возможности предыдущей программы. С помощью вложенных структур программа запоминает и отображает значения даты и времени.

Листинг 6.2. datetime.c (запоминание даты и времени в структурах)

```
 1: #include <stdio.h>
 2:
 3: typedef struct dateStruct {
 4:   char day;
 5:   char month;
 6:   unsigned year;
 7: } Date;
 8:
 9: typedef struct timeStruct {
10:   char hour;
11:   char minute;
12:   char second;
```

```
 13: } Time;
 14:
 15: typedef struct dateTime {
 16:   Date theDate;
 17:   Time theTime;
 18: } DateTime;
 19:
 20: main()
 21: {
 22:   DateTime dt;
 23:
 24:   printf("Date and time test\n");
 25:   dt.theDate.day = 1;
 26:   dt.theDate.month = 9;
 27:   dt.theDate.year = 2004;
 28:   dt.theTime.hour = 18;
 29:   dt.theTime.minute = 30;
 30:   dt.theTime.second = 0;
 31:   printf("The data is: %02d.%02d.%04d\n",
 32:     dt.theDate.day, dt.theDate.month, dt.theDate.year);
 33:   printf("The time is: %02d:%02d:%02d\n",
 34:     dt.theTime.hour, dt.theTime.minute, dt.theTime.second);
 35:   return 0;
 36: }
```

В строках 25-30 членам вложенных структур структуры **dt** присваиваются значения. Два оператора **printf()** в строках 31-34 отображают эти значения на экране:

```
Date and time test
The data is: 01.09.2004
The time is: 18:30:00
```

Структуры и функции

Члены структур могут быть любого типа данных, но они не могут быть функциями. Несмотря на это ограничение, структуры и функции могут активно взаимодействовать. Структуры могут передаваться параметрам функции, а функции могут возвращать структуры как результаты своей работы.

При разработке графических интерфейсов и заставок можно воспользоваться полезной структурой, определяющей прямоугольник по координатам пикселов на экране. Вот одна из возможных конструкций:

```
typedef struct rectangle {
  int left;
  int top;
  int right;
  int bottom;
} Rectangle;
```

Структура **Rectangle** имеет четыре члена, представляющих левую, верхнюю, правую и нижнюю координаты прямоугольника на дисплее. Программам, которые используют структурные переменные типа **Rectangle**, понадобится поддержка различных функций. Например, вместо явного присваивания переменной **Rectangle** четырех координат:

```
Rectangle r;
r.left = 5;
r.top = 8;
r.right = 60;
r.bottom = 18;
```

вы можете написать функцию (допустим с именем **RectAssign**), которая возвращает инициализированную структуру типа **Rectangle**:

```
Rectangle r;
r = RectAssign(5, 8, 60, 18);
```

Вот один из вариантов реализации такой функции:

```
Rectangle RectAssign(int l, int t, int r, int b)
{
  Rectangle rtemp;
  rtemp.left = l;
  rtemp.top = t;
  rtemp.right = r;
  rtemp.bottom = b;
  return rtemp;
}
```

Несмотря на то что переменная **rtemp** локальна в функции и не существует вне области ее действия, функция может возвращать значение переменной **rtemp** в качестве своего результата. При выполнении оператора **return** переменная **rtemp** временно копируется в стек, а затем оттуда – в переменную, которой присваивается результат функции. После этого данные из стека удаляются.

Вы также можете передавать структуры параметрам функции. Например, если вам нужно сравнивать две структуры типа **Rectangle**, вы можете написать такую функцию:

```
int RectsEqual(Rectangle r1, Rectangle r2)
{
  return ((r1.left   == r2.left  ) &&
          (r1.top    == r2.top   ) &&
          (r1.right  == r2. right) &&
          (r1.bottom == r2.bottom));
}
```

Совет. *В сложных выражениях выравнивайте скобки и операторы как показано выше, чтобы облегчить восприятие текста и отладку.*

Теперь вы можете вызывать функцию **RectsEqual()** в любом месте программы, где разрешены логические выражения. Например, оператор **if** выполнит *оператор*, если две структуры типа **Rectangle** – **var1** и **var2** – равны:

```
if (RectsEqual(var1, var2))
  оператор;
```

Структуры и массивы

Комбинация массивов и структур позволяет создавать мощные структуры данных, которые могут организовать информацию почти без всяких ограничений. Существуют две основные комбинации:

- массивы структур;
- структуры с членами, являющимися массивами.

Давайте рассмотрим подробно каждый из этих типов данных.

Массивы структур

Два или несколько массивов, содержащих связанные между собой данные, часто можно преобразовать в один массив структур, который, как правило, более эффективен. Пусть, к примеру, нам необходимо запомнить 100-элементный набор трехмерных координат. Вы могли бы объявить три массива:

```
double x[100];
double y[100];
double z[100];
```

Это решает задачу, однако, чтобы получить доступ к одной точке, потребуется выполнить три операции индексирования. Используя гипотетическую функцию **PlotPoint()** для вычерчивания одиннадцатой точки, вы должны были бы записать примерно следующее:

```
PlotPoint(x[10], y[10], z[10]);
```

Лучший вариант – объявление структуры с тремя членами типа **double**:

```
struct point3D {
  double x;
  double y;
  double z;
}
```

Использование структуры **point3D** позволяет обойтись только одним массивом для запоминания 100-элементного набора данных:

```
struct point3D points[100];
```

Такие выражения, как **points[9]** и **points[50]**, обозначают трехчленные структуры **point3D**, расположенные согласно их индексу. Теперь можно объявить функцию **PlotPoint()**, которая имеет только один параметр:

```
void PlotPoint(struct point3D p);
```

и передавать функции **PlotPoint()** структуры, организованные в массив:

```
PlotPoint(points[9]);
```

Структуры с членами, являющимися массивами

Структуры в качестве своих членов могут иметь массивы. Вот типичный пример структуры с несколькими символьными массивами:

```
struct person {
  char name[30];
  char address[40];
  char city[15];
  ...
};
```

Членами структур могут быть массивы и любых других типов данных, в том числе и массивы других структур.

Динамические структуры данных

Чтобы эффективно использовать память компьютера, большие структуры с множеством членов (особенно если некоторые из них являются массивами) лучше хранить в куче. Вначале объявите указатель на структурный тип, наприме:

```
Coordinate *p;
```

Затем вызовите функцию **malloc()**, чтобы выделить память в куче и присвоить адрес указателю **p**:

```
p = (Coordinate *)malloc(sizeof(Coordinate));
```

Указатель **p** сейчас адресует блок памяти, размер которого как раз такой, чтобы запомнить одну структуру типа **Coordinate**. Но в отличие от глобальных или локальных структурных переменных, вы не можете использовать точку для доступа к членам структуры. Следующие операторы не будут работать:

```
p.x = 123;  /* ??? */
p.y = 321;  /* ??? */
```

Указатель **p** не является структурной переменной. Это указатель на структуру, поэтому вы должны использовать оператор **->**, который похож на стрелку:

```
p->x = 123;
p->y = 321;
```

Как обычно, вы можете разыменовать указатель на структуру. Выражение ***p** представляет структуру, адресуемую **p**. Выражение **(*p).x** эквивалентно **p->x**. Листинг 6.3 демонстрирует, как использовать структуры, адресуемые указателями.

Листинг 6.3. pstruct.c (адресация членов структур с помощью указателей)

```
1: #include <stdio.h>
2: #include <stdlib.h>
3:
4: typedef struct coordinate {
```

```
 5:   int x;
 6:   int y;
 7: } Coordinate;
 8:
 9: main()
10: {
11:   Coordinate* pixel; /* объявление указателя на структуру */
12:
13:   pixel = (Coordinate *)malloc(sizeof(Coordinate));
14:   if (pixel == NULL) {
15:     puts("Out of memory!");
16:     exit(1);
17:   }
18:   pixel->x = 10;  /* присваивание 10 члену x */
19:   pixel->y = 11;  /* присваивание 11 члену y */
20:   printf("x = %d\n", pixel->x);
21:   printf("y = %d\n", pixel->y);
22:   free(pixel);
23:   return 0;
24: }
```

Самоссылочные структуры

Структуры, ссылающие на себя, обычно располагаются в динамической памяти. Каждая структура содержат указатель, который ссылается на другую структуру того же типа. «Цепляясь» с помощью указателей друга за друга, подобно виноградной лозе, цепочки структур такого типа могут динамически расти при добавлении элементов и сокращаться при их удалении. Память в куче при этом расходуется очень экономно, т.к. выделяется ровно столько памяти, сколько нужно для хранения данных.

Самыми распространенными самоссылочными структурами являются списки, стеки, и очереди.

Стеки

Вы уже знаете, что стек – это специальная область памяти, которая хранит локальные переменные. Однако сами по себе стеки – весьма полезные структуры данных, которые пригодятся вам для хранения различной информации.

Стек – это набор связанных между собой элементов, размещаемых в динамической памяти. Стеки организованы по принципу: последним вошел – первым вышел. Данные добавляются в начало стека, называемое вершиной, и извлекаются также из начала стека. Представить себе стек позволяет пример: шарики закатываются в трубку, запаянную с одного конца.

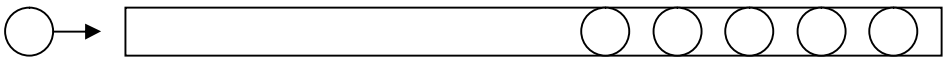

Рис.6.1. Модель стека

Чтобы организовать стек, объявите вначале следующую структуру:

```
struct item {
  int data;           /* любые члены */
  struct item *next; /* указатель на другую структуру типа item*/
};
```

Структура типа **struct item** имеет два члена: целую переменную, которая в этом примере представляет информацию для запоминания в структуре, и указатель **next** на «свой» тип структуры. Какое же это мощное средство! Несколько переменных типа **item** теперь могут формировать список, просто связываясь друг с другом с помощью указателя на следующий элемент.

Для упрощения программирования используйте оператор **typedef**:

```
typedef struct item {
  int data;
  struct item *next;
} Item;
```

Далее, объявите указатель типа **Item**, который будет служить вершиной стека:

```
Item *top = NULL;
```

Если указатель **top** равен нулю, значит стек пуст, поэтому хорошая идея – проинициализировать указатель прямо в его объявлении. Чтобы начать стек, проверьте его член **top**, и, если он окажется нулевым, выделяйте память для нового элемента типа **Item**:

```
if (top == NULL) {
  top = (Item *)malloc(sizeof(Item));
  top->data = 1;
  top->next = NULL;
}
```

Теперь в стеке есть один элемент, адресуемый указателем **top**. Указатель **next** указывает на **NULL**, что является признаком конца стека (рис. 6.2).

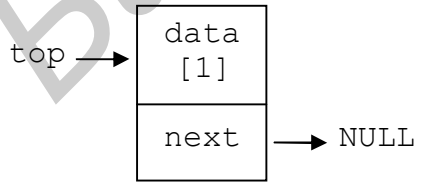

Рис. 6.2. Стек из 1 элемента

Из одного элемента стек состоит крайне редко. Рано или поздно вам понадобиться добавить в стек новую информацию. С помощью указателя **p** следующие операторы добавят в стек новый элемент:

```
Item* p;
p = (Item *)malloc(sizeof(Item));
p->data = 2;
p->next = top;
```

```
top = p;
```

Внимательно изучите эти операторы. После того как программа выделит память для элемента, адресуемого **p**, она присвоит переменной **data** значение **2**, а указателю **p–>next** – адрес вершины стека. После этого указателю **top** присваивается значение **p**, чтобы он вновь указывал на последний созданный элемент. Сейчас стек выглядит, как показано на рис. 6.3.

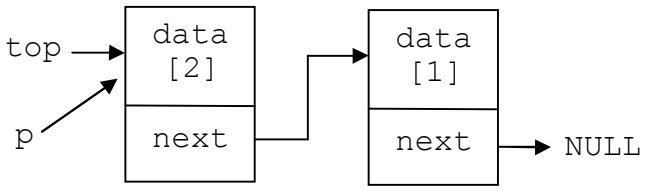

Рис. 6.3. Стек из 2 элементов

После добавления еще нескольких структур стек принимает вид, показанный на рис. 6.4. Каждая структура типа **Item** связана со следующей с помощью указателя **next**.

Путем разыменования указателя **top** можно получить доступ к любому элементу стека. Выражение **top–>data** относится к переменной **data** в последнем элементе стека. Чтобы получить доступ к другим членам, вы должны использовать операторы, подобные следующим (каждый из них присваивает переменной **i** значение члена **data** разных элементов стека):

```
i = top->data;                        /* i = 4 */
i = top->next->data;                  /* i = 3 */
i = top->next->next->data;            /* i = 2 */
i = top->next->next->next->data; /* i = 1 */
```

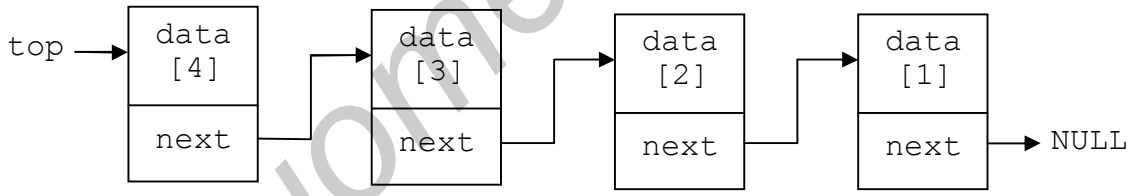

Рис. 6.4. Стек из нескольких элементов, связанных с помощью указателей на следующий элемент

Однако многоссылочные выражения слишком громоздки и в таком виде используются редко. Вместо этого, чтобы «пройтись» по элементам стека, большинство программистов используют цикл:

```
Item *p = top;
while (p != NULL) {
  printf("%d\n", p->data);
  p = p->next;
}
```

Временный указатель **p** типа **Item *** инициализируется адресом вершины стека. В цикле оператор **printf()** отображает значение члена структуры **data**, а указателю **p** присваивается адрес следующей структуры в стеке. Чтобы цикл мог успешно закончиться, последняя структура должна иметь член **next**, равный нулю.

Еще одна распространенная операция над стеками – удаление элементов. Для стека, изображенного на рис. 6.4, следующие операторы удаляют элемент, адресуемый указателем **top**:

```
Item *p = top;
top = top->next;
free(p);
```

Временному указателю **p** типа **Item *** присваивается значение **top**, после чего указателю **top** присваивается адрес следующей структуры (рис. 6.5). Последний оператор передает указатель **p** в функцию **free()**, удаляя отсоединенный элемент стека из кучи.

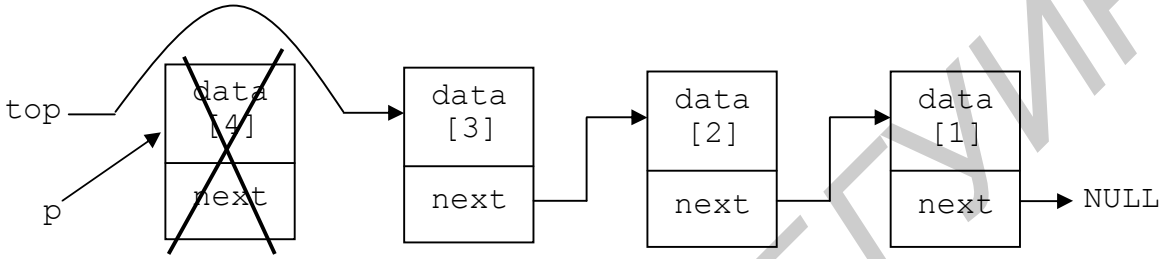

Рис. 6.5. Удаление элемента в стеке

Листинг 6.4 демонстрирует принципы добавления и удаления данных из стека. Скомпилируйте и запустите программу, затем нажмите <A> несколько раз, чтобы добавить в стек новые элементы. Нажмите <D>, чтобы удалить элементы. Если вы удалите все элементы, программа напечатает, что стек пуст.

Листинг 6.4. stack.c (запоминание значений в стеке)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <ctype.h>
 4: #include <conio.h>
 5:
 6: #define FALSE 0
 7: #define TRUE 1
 8:
 9: typedef struct item {
10:   int data;          /* данные для запоминания в стеке */
11:   struct item *next; /* указатель на следующий элемент */
12: } Item;
13:
14: void Push(void);
15: void Pop(void);
16: void Display(void);
17:
18: Item* top = NULL;    /* указатель на стек */
19:
20: main()
```

```
21: {
22:   int done = FALSE;  /* если TRUE, программа завершается */
23:   char c;            /* символ команды пользователя */
24:
25:   while (!done) {
26:     Display();
27:     printf("\n\nA)dd, D)elete, Q)uit ");
28:     c = getch();
29:     switch (toupper(c)) {
30:       case 'A':
31:         Push();
32:         break;
33:       case 'D':
34:         Pop();
35:         break;
36:       case 'Q';
37:         done = TRUE;
38:         break;
39:     }
40:   }
41:   return 0;
42: }
43:
44: /* Добавление элемента в стек */
45: void Push(void)
46: {
47:   Item* p;
48:
49:   p = (Item *)malloc(sizeof(Item)); /* создание элемента */
50:   p->data = rand();
51:   p->next = top;
52:   top = p;
53: }
54:
55: /* Удаление элемента из стека */
56: void Pop(void)
57: {
58:   Item* p;
59:
60:   if (top != NULL) {  /* если стек не пуст... */
61:     p = top;
62:     top = top->next;
63:     free(p);
64:   }
65: }
66:
67: /* Отобразить содержимое стека */
68: void Display(void)
69: {
70:   Item* p = top;
71:
72:   clrscr();
```

```
 73:   if (p == NULL)
 74:     printf("Stack is empty");
 75:   else
 76:     printf("Stack: ");
 77:   while (p != NULL) {
 78:     printf("\n%d", p->data);
 79:     p = p->next;   /* адресовать следующий элемент стека*/
 80:   }
 81: }
```

В строке 18 объявляется указатель на вершину стека **top**. Будучи глобальным, указатель автоматически инициализируется нулевым значением, однако здесь ему явно присваивается значение **NULL**, чтобы подчеркнуть, что в начале программы стек пуст.

Основные функции, используемые при работе со стеками – это **Push()** и **Pop()**. Функция **Push()** (строки 45-53) создает новый элемент и помещает его на вершину стека. В строках 49-50 с помощью указателя **p** происходит создание нового элемента и инициализация переменной **data** случайным числом. Оператор в строке 51:

```
p->next = top;
```

связывает новый элемент с предыдущими, образуя цепочку элементов.

Функция **Pop()** (строки 56-65) удаляет элемент, являющийся вершиной стека и освобождает выделенную память. Функция действует в полном соответствии с принципом «Последним вошел – первым вышел».

Функция **Display()** (строки 68-81) отображает содержимое стека. С помощью оператора в строке 79:

```
p = p->next;
```

цикл **while** последовательно перебирает элементы стека, до тех пор, пока не встретится указатель на **NULL**.

Очередь

Еще одной распространенной структурой данных является очередь. Очередь можно представить себе как трубку с двумя открытыми концами (рис. 6.6): данные добавляются в начало очереди, а извлекаются из ее конца. Другими словами, очередь работает по принципу: «Первый вошел – первый вышел».

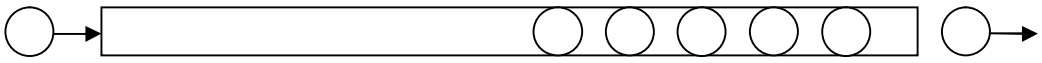

Рис.6.6. Модель очереди

Очереди находят в компьютерных системах многочисленные применения. Большинство компьютеров оборудовано только одним

процессором, поэтому в каждый конкретный момент времени может обсуживаться только один процесс или задача. Остальные задачи ставятся в очередь.

Информационным пакетам в сети Internet также приходится «стоять в очередях». Пакет в сети переправляется от одного сетевого узла к другому, пока не достигнет узла назначения. Сетевой узел способен послать в каждый момент всего один пакет, поэтому все остальные пакеты становятся в очередь, ожидая, когда пересылающий узел сможет их обработать.

Листинг 6.5 представляет собой пример работы с очередью. Программа queue предлагает выполнить следующие действия: поставить элемент в очередь, удалить его из очереди либо выйти из программы.

Листинг 6.5. queue.c (работа с очередью)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <ctype.h>
 4: #include <conio.h>
 5:
 6: #define FALSE 0
 7: #define TRUE 1
 8:
 9: typedef struct item {
10:   int data;
11:   struct item *next;
12: } Item;
13:
14: void Enqueue(void);
15: void Dequeue(void);
16: void Display(void);
17:
18: /* Указатели на начало (голову) и конец (хвост) очереди */
19: Item *head = NULL, *tail = NULL;
20:
21: main()
22: {
23:   int done = FALSE;
24:   char c;
25:
26:   while (!done) {
27:     Display();
28:     printf("\n\nA)dd, D)elete, Q)uit ");
29:     c = getch();
30:     switch (toupper(c)) {
31:       case 'A':
32:         Enqueue();
33:         break;
34:       case 'D':
35:         Dequeue();
36:         break;
```

```
37:       case 'Q':
38:         done = TRUE;
39:         break;
40:     }
41:   }
42:   return 0;
43: }
44:
45: /* Поставить элемент в очередь */
46: void Enqueue(void)
47: {
48:   Item* p;
49:
50:   p = (Item *)malloc(sizeof(Item)); /* создать элемент */
51:   p->data = rand(); /* присвоить data случайное число */
52:   p->next = NULL;   /* установить признак конца очереди */
53:   if (head == NULL) /* если очередь пуста... */
54:     head = p;       /* head указывает на созданный элемент*/
55:   else              /* иначе... */
56:     tail->next = p; /* поставить новый эл. в конец очереди*/
57:   tail = p;         /* tail указывает на конец очереди */
58: }
59:
60: /* Удалить элемент из очереди */
61: void Dequeue(void)
62: {
63:   Item* p = head;      /* p указывает на голову очереди */
64:
65:   if (head != NULL) {  /* если очередь не пуста... */
66:     head = head->next; /* head указывает на следующий эл. */
67:     if (head == NULL)  /* если очередь содержит 1 элемент */
68:       tail = NULL;     /* tail равен нулю */
69:     free(p);           /* удалить элемент */
70:   }
71: }
72:
73: /* Отобразить содержимое очереди */
74: void Display(void)
75: {
76:   Item* p = head;
77:
78:   clrscr();
79:   if (p == NULL)
80:     printf("Queue is empty");
81:   else
82:     printf("Queue: ");
83:   while (p != NULL) {
84:     printf("\n%d", p->data);
85:     p = p->next;
86:   }
87: }
```

За исключением нескольких важных особенностей листинг 6.5 похож на предыдущий. В строке 18 для работы с очередью объявляется два указателя: **head** («голова») и **tail** («хвост») очереди. Указателя два, т.к. в очереди необходимо отслеживать начальный и конечный элемент (рис. 6.7).

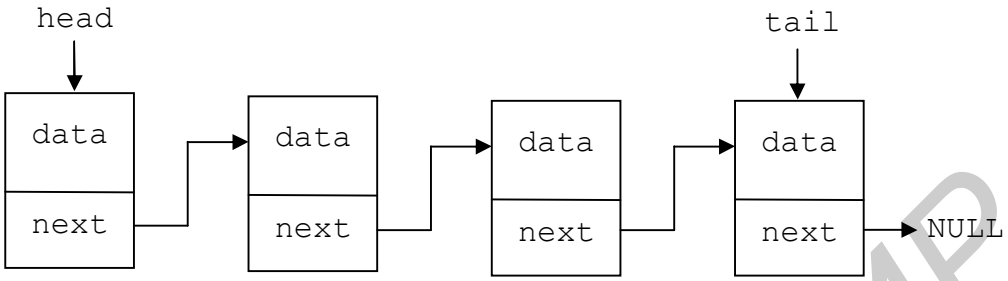

Рис. 6.7. Очередь, состоящая из 4 элементов

Функция **Enqueue()** («поставить в очередь»), описанная в строках 45-57, похожа на функцию **Push()** для работы со стеками. С помощью временного указателя **p** создается новый элемент, член **data** которого инициализируется случайным значением. Указатель **next** создаваемого элемента всегда должен быть нулевым, т.к. элементы могут добавляться только в конец очереди. Если очередь пуста, то и «голова», и «хвост» очереди будут указывать на создаваемый (первый) элемент. В противном случае на новый элемент будет указывать лишь **tail** («хвост»).

Функция **Dequeue()** («удалить из очереди»), описанная в строках 60-70, напоминает функцию **Pop()**. Вначале проверяется, есть ли в очереди элементы. Если это так, то указателю **head** присваивается адрес следующего элемента. После этого элемент, на который **head** указывал раньше, удаляется. Функция работает в полном соответствии с принципом «Первым вошел – первым вышел», так как удаляет элементы только из «головы» очереди.

Списки

Списки являются обобщением стеков и очередей. Вы можете добавлять элементы в любое место списка: в начало, середину или конец. Аналогично из списка можно удалить любой элемент, вне зависимости от его местоположения.

Листинг 6.6 демонстрирует работу со списками. Программа list расширяет возможности предыдущих программ, позволяя вставлять символы в список в алфавитном порядке.

Листинг 6.6. list.c (управление списком символов)

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <conio.h>
4: #include <ctype.h>
5:
```

```
 6: typedef struct item {
 7:  char data;
 8:   struct item *next;
 9: } Item;
10:
11: void Insert(char value);
12: void Delete(char value);
13: void Display(void);
14:
15: Item* list = NULL;
16: #define FALSE 0
17: #define TRUE 1
18:
19: main()
20: {
21:   int done = FALSE;
22:   char c, value;   /* символ команды пользователя и */
23:                    /* вводимое значение            */
24:   while (!done) {
25:     Display();
26:     printf("\n\nA)dd, D)elete, Q)uit ");
27:     c = getch();
28:     switch (toupper(c)) {
29:       case 'A':
30:         printf("\nEnter a character: ");
31:         value = getch();  /* получить символ */
32:         Insert(value);    /* вставить его в список */
33:         break;
34:       case 'D':
35:         if (list != NULL) { /* если список не пуст */
36:           printf("\nEnter character to be deleted: ");
37:           value = getch();  /* получить символ */
38:           Delete(value);    /* удалить его из списка */
39:         }
40:         break;
41:       case 'Q':
42:         done = TRUE;
43:         break;
44:     }
45:   }
46:   return 0;
47: }
48:
49: /* Вставить элемент в список */
50: void Insert(char value)
51: {
52:   Item *p, *cur = list, *prev = NULL;
53:
54:   p = (Item *)malloc(sizeof(Item));
55:   p->data = value;
56:   p->next = NULL;
57:   /* Цикл while находит место для вставки элемента */
```

```
  while (cur != NULL && value > cur->data) {
    prev = cur;
    cur = cur->next;
  }
  if (prev == NULL) { /* если новый элемент должен быть */
                      /* первым в списке... */
    p->next = list;   /* связать его со списком */
    list = p;         /* list указывает на первый элемент */
  } else {            /* иначе... */
    prev->next = p;   /* вставить элемент */
    p->next = cur;    /* в середину или конец списка */
  }
}

/* Удалить элемент из списка */
void Delete(char value)
{
  Item *prev = list, *cur = list->next, *p;

  if (value == list->data) { /* если нужно удалить */
                             /* начальный элемент списка */
    p = list;          /* p указывает на удаляемый элемент*/
    list = list->next; /* list указывает на след. элемент */
    free(p);           /* удалить элемент */
  } else {             /* иначе... */
    /* С помощью цикла while найти элемент */
    while (cur != NULL && cur->data != value) {
      prev = cur;
      cur = cur->next;
    }
    if (cur != NULL) { /* если элемент найден... */
      p = cur;         /* p указывает на удаляемый элемент*/
      prev->next = cur->next; /* перенастроить связи */
      free(p);                /* удалить элемент */
    }
  }
}

/* Отобразить список */
void Display(void)
{
  Item* p = list;

  clrscr();
  if (p == NULL)
    printf("List is empty");
  else {
    printf("List: ");
    while (p != NULL) {
      printf("%c -> ", p->data);
      p = p->next;
    }
```

```
110:     printf("NULL");
111:   }
112: }
```

Функция **Insert()** (строки 50-70) добавляет символ **value** в список в алфавитном порядке. Функция объявляет на один, а три временных указателя: **p** – рабочий указатель, с помощью которого в список добавляются элементы; **cur** (от англ. current) – «текущий» указатель и **prev** (от англ. **previous**) – «предыдущий» указатель, которые помогают определить место вставки элемента и перенастроить связи. Указатель **cur** инициализируется адресом первого элемента списка (или нулем, если список не создан), указатель **prev** инициализируется значением **NULL**.

После создания нового элемента, в строках 58-61 запускается цикл **while**:

```
while (cur != NULL && value > cur->data) {
  prev = cur;
  cur = cur->next;
}
```

Цикл последовательно «перебирает» все элементы списка до тех пор, пока не будет найден элемент, с большим по алфавиту значением члена **data** либо не будет достигнут конец списка. В теле цикла указатели **cur** и **prev** «ходят» друг за другом подобно влюбленной парочке: сначала указателю **prev** присваивается значение **cur**, потом **cur** «переходит» на следующий элемент списка. Таким образом, после завершения цикла указатель **cur** будет указывать на элемент, перед которым нужно делать вставку, а **prev** – на предыдущий элемент (рис. 6.8).

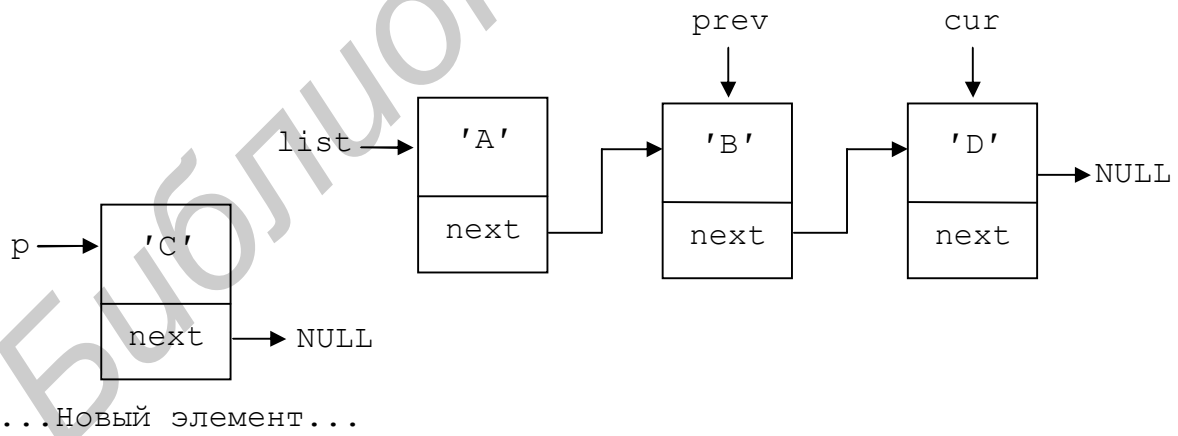

Рис. 6.8. Элемент ‘**C**’ должен быть вставлен между элементами ‘**B**’ и ‘**D**’

Если по окончании цикла указатель **prev** равен нулю (строка 62), то это значит, что либо список пуст, либо все его элементы имеют алфавитно большие значения, чем **value**. В обоих случаях элемент должен быть вставлен в начало списка, а его адрес присвоен указателю **list**.

Если указатель **prev** не равен нулю, то новый элемент придется вставлять в середину или конец списка. Для этого нужно перенастроить связи списка, как это показано в строках 67-68 (см. рис. 6.9).

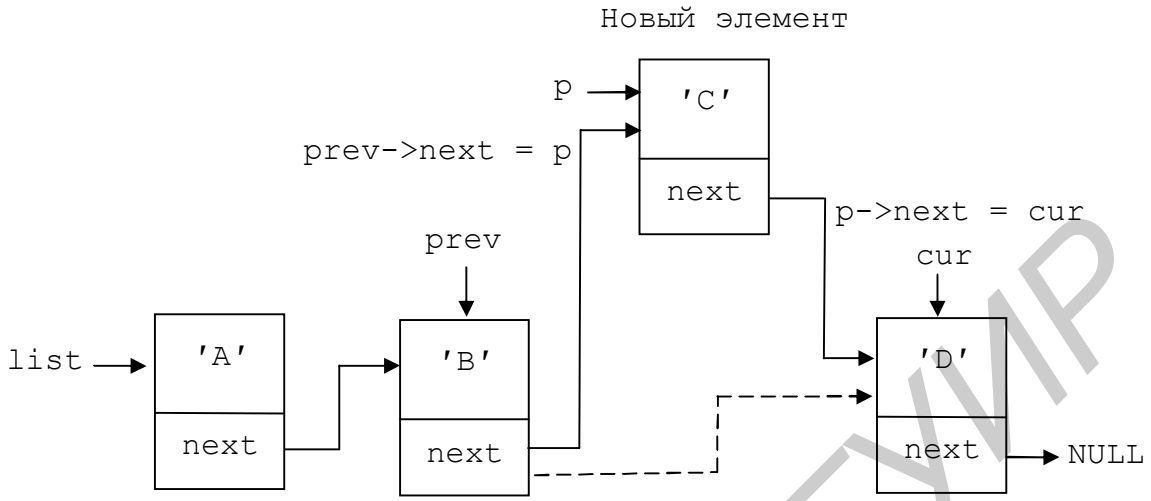

Рис. 6.9. Вставка элемента **'C**' в середину списка

Функция **Delete()**, описанная на строках 73-94, позволяет удалить из списка элемент, содержащий заданный символ. Функция вызывается, только если список не пуст (строки 35-39). Так же как и **Insert()**, функция **Delete()** объявляет три вспомогательных указателя **p**, **prev** и **cur**. Если удалить нужно начальный элемент списка, на который указывает **list** (строка 77), то используется тот же алгоритм, что и при удалении элемента из стека. В противном случае перед удалением происходит перенастройка связей: члену **next** предыдущего элемента присваивается адрес следующего за найденным элемента (строка 90).

Двунаправленные списки

Однонаправленные списки, которые мы рассматривали выше, отличаются тем, что каждый элемент знает о следующем, но ничего не знает о предыдущем элементе. Как вы уже заметили, в такой список легко добавлять элементы в начало, немного труднее в конец, а вот удаление или вставка элемента посередине вызывает определенные трудности. Существует еще один существенный недостаток: пройти однонаправленный список можно только от начала до конца, но невозможно от конца до начала.

Двунаправленные списки лишены этих недостатков, так как каждый элемент такого списка «знает» о своем предыдущем и следующем элементах.

```
struct item {
  int data;
  struct item *next; /* указатель на следующий элемент списка */
  struct item *prev; /* указатель на предыдущий элемент списка*/
};
```

Схематически двунаправленный список можно изобразить можно следующим образом:

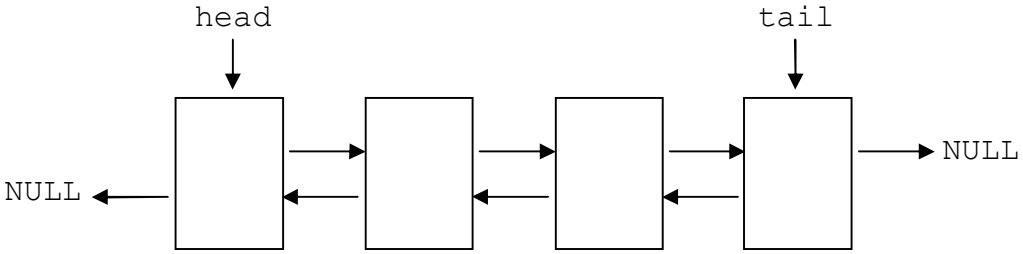

Рис. 6.10. Двусвязный список

Листинг 6.7 похож на предыдущий. Но для работы с элементами программа использует более «надежный» двунаправленный список.

Листинг 6.7. list2.c (работа с двунаправленным списком)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <conio.h>
 4: #include <ctype.h>
 5:
 6: typedef struct item {
 7:    char data;
 8:    struct item *next;
 9:    struct item *prev;
10: } Item;
11:
12: void Insert(char value);
13: void Delete(char value);
14: void Display(void);
15:
16: Item *head = NULL, *tail = NULL;
17: #define FALSE 0
18: #define TRUE 1
19:
20: main()
21: {
22:   int done = FALSE;
23:   char c, value;
24:
25:   while (!done) {
26:     Display();
27:     printf("\n\nA)dd, D)elete, Q)uit ");
28:     c = getch();
29:     switch (toupper(c)) {
30:       case 'A':
31:         printf("\n Enter a character: ");
32:         value = getch();
33:         Insert(value);
34:         break;
35:       case 'D':
```

```
36:         if (head != NULL) {
37:           printf("\n Enter the character to be deleted: ");
38:           value = getch();
39:           Delete(value);
40:         }
41:         break;
42:       case 'Q':
43:         done = TRUE;
44:         break;
45:     }
46:   }
47:   return 0;
48: }
49:
50: void Insert(char value)
51: {
52:   Item *p, *cur = head;
53:   /* Создание нового элемента */
54:   p = (Item *)malloc(sizeof(Item));
55:   p->data = value;
56:   p->next = p->prev = NULL;
57:
58:   if (head == NULL) {     /* если список пуст... */
59:     head = tail = p;      /* head и tail указывают на */
60:     return;               /* созданный элемент */
61:   }
62:   /* Поиск места для вставки */
63:   while (cur != NULL && value > cur->data)
64:     cur = cur->next;
65:
66:   if (cur == head) {      /* если элемент нужно вставить */
67:     p->next = head;       /* в начало...*/
68:     head->prev = p;
69:     head = p;
70:   }
71:   else if (cur == NULL) { /* если элемент нужно вставить */
72:     tail->next = p;       /* в конец...*/
73:     p->prev = tail;
74:     p->next = NULL;
75:     tail = p;
76:   }
77:   else { /* если элемент нужно вставить в середину...*/
78:     cur->prev->next = p;
79:     p->prev = cur->prev;
80:     cur->prev = p;
81:     p->next = cur;
82:   }
83: }
84:
85: void Delete(char value)
86: {
87:   Item *p = head;
```

```
 88:   /* Найти элемент, содержащий заданное значение */
 89:   while (p != NULL && p->data != value)
 90:     p = p->next;
 91:
 92:   if (p == NULL)         /* если элемент не найден */
 93:     return;              /* выйти из функции */
 94:   if (p == head) {       /* если нужно удалить первый эл. */
 95:     head = head->next;   /* перенастроить */
 96:     head->prev = NULL;   /* связи */
 97:   }
 98:   else if (p == tail) { /* ... удалить последний элемент */
 99:     tail = tail->prev;   /* перенастроить */
100:     tail->next = NULL;   /* связи */
101:   }
102:   else {  /* если нужно удалить элемент в середине списка */
103:     p->prev->next = p->next; /* перенастроить */
104:     p->next->prev = p->prev; /* связи */
105:   }
106:   free(p);               /* удалить элемент  */
107: }
108:
109: void Display(void)
110: {
111:    Item *p = head;
112:
113:    clrscr();
114:    if (p == NULL)
115:      puts("List is empty");
116:    else {
117:      printf("NULL <-> ");
118:      while (p != NULL) {
119:        printf("%c <-> ", p->data);
120:        p = p->next;
121:      }
122:      printf("NULL");
123:    }
124: }
```

Задание. *Внимательно рассмотрите функции* ***Insert()*** *и* ***Delete()****. С помощью карандаша и бумаги попробуйте самостоятельно разобраться, как происходит вставка и удаление элементов в начало, конец и середину двунаправленного списка.*

Упражнение. *Функция* ***Display()*** *выводит элементы двунаправленного списка в прямом порядке, реализуйте функцию* ***DisplayBack()****, которая выводила бы их в обратном порядке.*

Упражнение. *Какой общий недостаток имеют листинги 6.4, 6.5, 6.6 и 6.7? (Подумайте, прежде чем читать дальше). Правильно, при выходе из программы не освобождается память, занятая элементами динамических структур. Обычно это не вызывает проблем, т.к. при завершении*

*программы память освобождается автоматически, но все же лучше освобождать память явным образом. Реализуйте для названных листингов функцию **Clear()**, которая при выходе из программы удаляла бы из памяти все элементы динамических структур.*

Резюме

- Структуры могут объединять несколько переменных одного или разных типов «под одной крышей». Структуры помогают организовать данные так же, как папки помогают организовать документы на рабочем столе.
- Для объявления структуры используйте ключевое слово **struct**. Определение структуры создает новый тип данных, который можно использовать для объявления переменных.
- С помощью точки вы можете обратиться к конкретному члену структуры. Если у вас есть указатель, адресующий структуру, используйте оператор **->**.
- Структурам можно присваивать начальные значения, используя список инициализации. Для этого после имени переменной в объявлении структуры ставится знак равенства, за которым следует помещенный в фигурные скобки и разделенный запятыми список инициализаторов. Если инициализаторов в списке меньше, чем имеется членов структуры, оставшимся членам автоматически присваивается значение 0 (или **NULL**, если член структуры – указатель).
- Структуры можно присваивать, но нельзя сравнивать. Если вам нужно сравнить две структуры, используйте поэлементное сравнение их членов.
- Структуры можно передавать функциям в качестве аргументов, а функции могут возвращать структуры. Структурные аргументы и возвращаемые значения передаются через стек.
- В языке С вы можете объявлять массивы структур, а также массивы, являющиеся элементами структур.
- Чтобы эффективно использовать память, громоздкие структуры (или массивы структур) лучше хранить в куче. Такие структуры называются динамическими.
- Структуры ссылающиеся на себя содержат один или несколько указателей, которые адресуют структуры того же типа. Это позволяет самоссылочным структурам связываться между собой, образуя динамические «цепочки» структур.
- Связанный список – это линейный набор ссылающихся на себя структур. Длина списка может увеличиваться (при добавлении элементов) или уменьшаться (при их удалении). Добавлять и удалять элементы в список можно в произвольном порядке.

- Стеки и очереди являются разновидностями списков. Стек организован по принципу «Первый вошел – первый вышел», так как элементы добавляются и удаляются только из вершины стека.
- В очереди элементы удаляются с начала, а добавляются в конец. Благодаря этому их называют структурами данных типа «Первый вошел – первый вышел».
- Каждый элемент двунаправленного списка содержит указатели на предыдущий и следующий элемент в списке.
- Двунаправленный список легко пройти в прямом и обратном направлениях, вставить элемент в заданное место. Используйте двунаправленные списки для организации больших объемов данных, над которыми нужно производить операции вставки, удаления, перестановки элементов, сортировки и т.п.

7.Сортировка и поиск данных

Методы и алгоритмы сортировки

Сортировка данных является одним из наиболее важных применений компьютера. Телефонные компании сортируют списки своих клиентов по фамилиям, чтобы облегчить поиск телефонных номеров. Центры по обслуживанию и продаже автомобилей имеют специальные базы данных, отсортированные по названию модели либо году выпуска. Файлы, хранящиеся на вашем компьютере, тоже, скорее всего, отсортированы по какому-либо признаку: названию, расширению или дате создания. Поставщики вычислительных машин считают, что на сортировку в среднем по отрасли тратится более 25% машинного времени.

Цель любой сортировки – облегчить поиск элементов в некотором множестве (массиве). Действительно, если мы имеем дело с неотсортированным массивом, то будем вынуждены вести поиск с последовательным перебором, т.е. необходимо просматривать и проверять каждый элемент массива. Данный подход приемлем, если мы имеем дело с небольшими объемами информации, но что делать, если имеется массив данных, состоящий из нескольких миллионов или миллиардов элементов? В таких случаях просто не обойтись без эффективного метода сортировки.

Алгоритмы сортировки уже хорошо исследованы и изучены. Различные подходы к сортировке обладают своими достоинствами и недостатками. Хотя некоторые методы в среднем могут быть лучше других (например, это относится к быстрой сортировке), ни один из методов не будет идеальным для всех ситуаций, поэтому каждый программист должен иметь в своем арсенале несколько различных алгоритмов сортировки.

Для оценки эффективности сортировок можно использовать следующие параметры:

1. Число сравнений элементов множества (массива).
2. Число перестановок элементов.
3. Скорость сортировки в наилучшем случае.
4. Скорость сортировки в «среднем» случае.
5. Скорость сортировки в наихудшем случае.

Интенсивность работы алгоритма основана на количестве сравнений и перестановок, которые он выполняет. Наибольшая скорость сортировки, как правило, достигается, если массив уже упорядочен. Если массив не упорядочен, алгоритму приходится выполнять больший объем работы. И, наконец, наибольшие трудозатраты обычно требуются в тех случаях, когда массив отсортирован в обратном порядке.

Прямые сортировки

Прямые методы сортировки основаны на простых и очевидных алгоритмах. Общим у этих методов является то, что они решают поставленную задачу «в лоб», затрачивая на сортировку много времени.

Во всех методах сортировки, которые мы рассмотрим ниже, для перемены местами элементов используется уже знакомая вам функция **Swap()**:

```
void Swap(int *x, int *y)
{
  int t = *x; /* промежуточная переменная */
  /* Перемена данных местами */
  *x = *y;
  *y = t;
}
```

В конце раздела все алгоритмы будут собраны в рабочую программу.

Сортировка методом обмена

Сортировка методом обмена больше известна под названием *пузырьковой сортировки*, которое хорошо отражает суть метода. Будем рассматривать массив расположенным сверху вниз. Согласно методу «пузырька» «легкие» элементы массива должны «всплывать» наверх, а «тяжелые» – «тонуть». Алгоритмически это можно реализовать следующим образом. Будем просматривать весь массив снизу вверх и менять стоящие рядом элементы в том случае, если «нижний» элемент меньше («легче»), чем «верхний». Таким образом, «всплывет на поверхность» самый «легкий» элемент всего массива. Теперь повторим эту операцию для оставшихся неотсортированными n – 1 элементов, которые находятся «ниже» первого. «Тяжелые» элементы будут опускаться на дно, а «легкие», подобно пузырям, «всплывать» наверх, пока весь массив не окажется отсортирован.

а	б	в	г
1	1	1	**0**
3	3	**0**	1
0	**0**	3	3
2	2	2	2

Рис. 7.1. Пузырьковая сортировка: первый проход

На рис. 7.1 показан пример сортировки массива методом «пузырька». На рис. 7.1,а изображен исходный массив из четырех элементов. Будем просматривать его «снизу вверх». Так как элемент 2 «тяжелее» чем 0, элементы остаются на своих местах (рис. 7.1,б). На следующей итерации рассматриваются элементы 0 и 3. Поскольку 0 «легче», он «всплывает наверх», а элемент 3 «опускается вниз» (рис. 7.1,в). Наконец, сравниваются 0 и 1. После того, как элементы поменяются местами, элемент 0, как самый «легкий», «всплывет на поверхность», а элемент 1 «опустится вниз» (рис. 7.1,г). На этом первый проход завершен и элемент 0 занимает свое окончательное место. На следующих итерациях аналогичным образом будут отсортированы оставшиеся элементы: 1, 3 и 2.

Функция **BubbleSort()** реализует алгоритм сортировки методом «пузырька»:

```
void BubbleSort(int a[], int n) /* функции передается массив */
{                               /* и его размерность */
  int i, j;                     /* переменные цикла */

  for (i = 0; i < n; i++)
    for (j = n - 1; j > i; j--)
      if(a[j - 1] > a[j]) /* если элемент "тяжелее" следующего */
        Swap(&a[j - 1], &a[j]); /* поменять их местами */
}
```

Как видно, алгоритм достаточно прост, но, как иногда замечают, является непревзойденным по своей неэффективности. Для алгоритма пузырьковой сортировки количество сравнений всегда одинаково и равно $(n^2 - n) / 2$, где n - количество сортируемых значений.

В наилучшем случае (если массив уже отсортирован) количество перестановок равно нулю. Количество перестановок в среднем и в худшем случаях равны соответственно:

В среднем случае: $\frac{3}{4}(n^2 - n)$.

В худшем случае: $\frac{3}{2}(n^2 - n)$.

Пузырьковая сортировка относится к *n-квадратичным алгоритмам*; это значит, что время ее выполнения пропорционально квадрату количества элементов сортируемого массива. Данный алгоритм весьма неэффективен при работе с большими массивами.

Сортировка методом выбора

В основе этого метода лежит следующий алгоритм:

1. Находится наименьший элемент в массиве.
2. Найденный элемент меняется с первым элементом.
3. Процесс повторяется с оставшимися n – 1 элементами, n – 2 элементами и так далее, пока в конце не останется самый большой элемент, который перемещать уже не нужно.

Функция **MinSort()** сортирует передаваемый ей массив методом выбора:

```
void MinSort(int a[], int n)
{
  int i, j, k;

  for (i = 0; i < n - 1; i++) {
    for (k = i, j = i + 1; j < n; j++) /* находим в цикле */
      if(a[j] < a[k])   /* минимальный элемент */
        k = j;          /* и запоминаем его номер в k */
```

```
    Swap(&a[k], &a[i]); /* меняем местами минимальный и элем., */
  }                     /* с которого начинался цикл */
}
```

К сожалению, сортировка методом выбора тоже является n-квадратичным алгоритмом. Она требует $(n^2 - n) / 2$ сравнений, что сильно замедляет работу при большом количестве элементов. Количество перестановок соответственно равно:

В наилучшем случае: $3(n-1)$ (если массив уже упорядочен, требуется сравнить только $(n - 1)$ элемент, и каждое сравнение требует трех промежуточных шагов).

В среднем случае: $n(\ln n + y)$, где y – константа Эйлера, примерно равная 0.577216.

В наихудшем случае: $\frac{n^2}{4} + 3(n-1)$.

Хотя количество сравнений для пузырьковой сортировки и сортировки методом выбора одинаковы, количество перестановок в сортировке выбором намного лучше.

Сортировка методом вставки

Большинство картежников, сами того не сознавая, пользуются именно этим методом для упорядочения пришедших им карт. Когда игрок получает очередную карту, все предыдущие уже отсортированы, поэтому он просто вставляет ее в нужное место.

На первом шаге алгоритма методом вставки выполняется сортировка первых двух элементов массива. Далее к ним добавляется третий элемент, который занимает позицию среди элементов в соответствии со своим значением. Затем в этот список вставляется четвертый элемент и т.д. Процесс продолжается до тех пор, пока все элементы не будут отсортированы.

Функция **InsertSort()** реализует алгоритм сортировки методом вставки:

```
void InsertSort(int a[], int n)
{
  int i, j;

  for (i = 1; i <= n - 1; i++) {
    j = i;
    while (a[j] < a[j - 1] && j >= 1) {
      Swap(&a[j], &a[j - 1]);
      j--;
    }
  }
}
```

В отличие от пузырьковой сортировки и сортировки методом выбора количество сравнений при сортировке методом вставки, зависит от исходной

упорядоченности массива. В наилучшем, среднем и наихудшем случаях количество сравнений определяется формулами:

Наилучший случай: $2(n-1)$.

Средний случай: $\frac{1}{4}(n^2+n)$.

Наихудший случай: $\frac{1}{2}(n^2+n)$.

Количество перестановок в этом методе не превышает $\frac{n^2}{4}$.

Как мы видим, в наихудших случаях алгоритм вставки так же плох, как и пузырьковый метод или метод выбора; в среднем случае он лишь немногим лучше этих методов; однако с частично упорядоченными массивами этот метод работает хорошо. Тем не менее существуют еще более совершенные методы сортировки.

Усовершенствованные методы сортировки

Сортировка методом Шелла

Основная идея Дональда Шелла заключалась в том, чтобы вначале устранить массовый беспорядок в массиве, сравнивая далеко стоящие друг от друга элементы; затем сравнивать элементы, которые находятся ближе друг другу; и, наконец, сортировать смежные элементы.

Такой подход позволяет «разбить» массив на группы элементов, каждая из которых упорядочивается методом вставки. В процессе упорядочения размеры таких групп увеличиваются до тех пор, пока все элементы массива не войдут в упорядоченную группу. При этом данные перемещаются не на одну позицию, как в предыдущих методах, а большими скачками, что значительно ускоряет сортировку.

Но какие расстояния между сравниваемыми элементами выбрать? Точная последовательность расстояний может изменяться. Например, хорошо себя зарекомендовала последовательность 9, 5, 3, 2, 1, которая использована в примере ниже. Это значит, что в начале сортируются элементы, отстоящие друг от друга на 9 позиций, потом на 5, на 3, на 2 позиции, и, наконец, сортируются смежные элементы. В алгоритме можно задать и другую последовательность, главное, чтобы последний шаг равнялся единице.

Предупреждение. *Избегайте последовательностей степеней 2, поскольку математически доказано, что это снижает эффективность алгоритма.*

```
void ShellSort(int a[], int n)
{
  int i, j, k, temp;
  int gap;       /* текущее расстояние сравниваемых элементов */
```

```
  int d[] = {9, 5, 3, 2, 1}; /* последовательность расстояний */

  for (k = 0; k < 5; k++) {
    gap = d[k];
    for (i = gap; i < n; i++) {
      temp = a[i];
      for (j = i - gap; temp < a[j] && j >= 0; j -= gap)
        a[j + gap] = a[j];
      a[j + gap] = temp;
    }
  }
}
```

Время выполнения алгоритма Шелла пропорционально $n^{1.2}$ при сортировке n элементов.Число операций сравнения пропорционально $n\log n$. Это существенный прогресс по сравнению с n-квадратичными методами сортировки. Однако прежде, чем вы решите использовать сортировку Шелла, имейте в виду, что быстрая сортировка дает еще лучшие результаты.

Замечание. *Когда говорится, что время выполнения алгоритма пропорционально $n^{1.2}$, то имеется в виду не точное время исполнения, а то, что время (или какой-либо другой параметр) возрастает приблизительно с той же скоростью, что и функция $n^{1.2}$. Чтобы получить точное значение времени, нужно домножить это выражение на константу, которая зависит от конкретной реализации алгоритма, производительности компьютера и т.п. По этой же причине в формуле $n\log n$ не приводится основание логарифма. Домножив на заданную константу, мы можем перейти к любому основанию, поэтому записывать его не имеет смысла.*

Быстрая сортировка

Метод быстрой сортировки, разработанный и названный так его автором Чарльзом Хоаром, по своим показателям превосходит все остальные алгоритмы, рассмотренные в этом разделе. Метод считается наилучшим из разработанных на сегодняшний день универсальных алгоритмов сортировки.

Производительность метода быстрой сортировки хорошо изучена. Алгоритм подвергался математическому анализу, поэтому на этот счет существуют точные математические формулы. Среднее число выполняемых сравнений для метода быстрой сортировки равно $2n\log n$. Среднее количество перестановок равно $\frac{\log n}{6}$. Время работы алгоритма пропорционально $n\log n$. Как и в методе Шелла, ускорение достигается за счет сравнения далеко стоящих друг от друга элементов, потом – более близких и т.д.

Алгоритм быстрой сортировки выглядит следующим образом:

1) *Этап разбиения*. Возьмите первый элемент несортированного массива и определите его расположение в отсортированном массиве. Это положение будет найдено, если все значения слева от данного элемента будут меньше, а все значения справа от элемента – больше значения данного элемента. Теперь мы имеем один элемент, расположенный на своем месте в отсортированном массиве и два несортированных подмассива.
2) *Этап рекурсии*. Выполните шаг 1 на каждом из несортированных подмассивов. Каждый раз после выполнения шага 1 в подмассиве следующий элемент массива помещается на свое место в отсортированном массиве, и создаются два несортированных подмассива. Когда мы дойдем до подмассива, состоящего из одного элемента, этот элемент будет находиться на своем окончательном месте в упорядоченном массиве.

Это описание алгоритма в целом кажется достаточно ясным, но как нам определить окончательную позицию первого элемента каждого подмассива? В качестве примера рассмотрим следующий набор значений (элемент, выделенный жирным – элемент разбиения – должен быть помещен на свое окончательное место в массиве):

37 2 6 4 89 8 12 68 45

1) Начиная с правого элемента массива будем сравнивать каждый элемент с числом 37 до тех пор, пока не будет найден элемент меньший чем 37, после чего найденный элемент и 37 должны поменяться своими местами. Первым элементом, который меньше 37, является число 12, поэтому они меняются местами. Теперь массив выглядит так:

12 2 6 4 89 8 **37** 68 45

2) Далее начинаем движение с левой части массива, но начинаем со следующего элемента после 12, и сравниваем каждый элемент с 37, пока не обнаружим элемент больший чем 37, после чего меняем местами 37 и этот найденный элемент. В нашем случае первый элемент больший чем 37 – это 89, так что 37 и 89 меняются местами. Новый массив имеет вид

12 2 6 4 **37** 8 10 *89* 68 45

3) Затем начинаем справа, но с элемента, предшествующего 89, и сравниваем каждый элемент с 37 до тех пор, пока не найдем меньший чем 37, и опять поменяем местами 37 и этот элемент. Первый элемент, который меньше 37 – это 10, – меняем местами с 37. Теперь наш массив имеет вид

12 2 6 4 *10* 8 **37** 89 68 45

4) После этого начинаем движение с левой части массива, но начинаем с элемента, следующего за 10, и сравниваем каждый элемент с 37, пока не обнаружим элемент, больший чем 37. В нашем случае таких элементов не оказалось, и когда мы сравним 37 с самим собой, это будет означать, что процесс закончен и элемент 37 нашел свое окончательное место.

После завершения этой операции мы имеем два неупорядоченных подмассива. Подмассив со значениями меньше 37 содержит элементы 12, 2, 6, 4, 10 и 8. Подмассив со значениями большими 37 содержит 89, 68 и 45. Сортировка продолжается путем применения алгоритма разбиения к полученным подмассивам, как этот делалось с первоначальным массивам.

Как ни странно, рассмотренный алгоритм неэффективен при работе с уже упорядоченным массивом. Допустим, массив отсортирован по возрастанию. В этом случае, т.к. в качестве элемента разбиения мы берем первый элемент, элемент разбиения всегда будет минимальным. Это приведет к тому, что массив будет разделен наихудшим образом: в левом подмассиве окажется один элемент, а в правом останется n – 1 элементов. Таким образом, за «один проход» метод быстрой сортировки уменьшит длину сортируемого массива всего лишь на 1. В результате время работы алгоритма становится пропорциональным n^2. Один из способов решить эту проблему – выбрать в качестве элемента разбиения не первый, а средний либо случайный элемент массива.

```
void QuickSort(int a[], int n, int left, int right)
{
/*Инициализируем переменные левой и правой границами подмассива*/
  int i = left, j = right;
/*Выбираем в качестве элемента разбиения средний элемент массива*/
  int test = a[(left + right) / 2];

  do {
    while (a[i] < test)
      i++; /* находим элемент, больший элемента разбиения */
    while (a[j] > test)
      j--; /* находим элемент, меньший элемента разбиения */
    if (i <= j) {
      Swap(&a[i], &a[j]);
      i++;
      j--;
    }
  } while(i <= j);
/* рекурсивно вызываем алгоритм для правого и левого подмассива*/
  if (i < right)
    QuickSort(a, n, i, right);
  if (j > left)
    QuickSort(a, n, left, j);
}
```

Листинг 7.1 позволяет сравнить эффективность методов «пузырька», выбора, вставок, Шелла и быстрой сортировки. Каждому методу «предлагается» задание: отсортировать неупорядоченный целочисленный массив из 15 тысяч элементов. Программа определяет время выполнения тестового задания для каждого метода и выводит его на экран. Не пытайтесь перезагрузить компьютер, если на экране какое-то время ничего не будет

происходить: возможно, вам придется подождать, пока самые медленные алгоритмы справятся с заданием.

Листинг 7.1. sort.c (сравнение скорости работы пяти методов сортировки)

```
 1: #include <conio.h>
 2: #include <stdio.h>
 3: #include <stdlib.h>
 4: #include <time.h>
 5:
 6: #define N 15000
 7:
 8: void InitArray(int a[], int n);
 9: void CopyArrays(int a[], int b[], int n);
10: void Swap(int *x, int *y);
11: void BubbleSort(int a[], int n);
12: void SelectSort(int a[], int n);
13: void InsertSort(int a[], int n);
14: void ShellSort(int a[], int n);
15: void QuickSort(int a[], int n, int left, int right);
16:
17: void main()
18: {
19:   int source[N], test[N];
20:   long begin, end;
21:
22:   clrscr();
23:   printf("Sorting Methods Test\n\n");
24:   printf("Number of elements: %d\n\n", N);
25:   printf("Sorting methods     Time (sec)\n");
26:   InitArray(source, N);
27:   CopyArrays(test, source, N);
28:   begin = clock(); BubbleSort(test, N); end = clock();
29:   printf("\nBubble sort   %15f", (end - begin) / CLK_TCK);
30:   CopyArrays(test, source, N);
31:   begin = clock(); SelectSort(test, N); end = clock();
32:   printf("\nSelection sort%15f", (end - begin) / CLK_TCK);
33:   CopyArrays(test, source, N);
34:   begin = clock(); InsertSort(test, N); end = clock();
35:   printf("\nInsertion sort%15f", (end - begin) / CLK_TCK);
36:   CopyArrays(test, source, N);
37:   begin = clock(); ShellSort(test, N); end = clock();
38:   printf("\nShell sort    %15f", (end - begin) / CLK_TCK);
39:   CopyArrays(test, source, N);
40:   begin = clock(); QuickSort(test, N, 0, N-1); end = clock();
41:   printf("\nQuick sort    %15f", (end - begin) / CLK_TCK);
42: }
43:
44: void CopyArrays(int a[], int b[], int n)
45: {
46:   int i;
47:
```

```
48:   for (i = 0; i < n; i++)
49:     a[i] = b[i];
50: }
51:
52: void InitArray(int a[], int n)
53: {
54:   int i;
55:
56:   randomize();
57:   for (i = 0; i < n; i++)
58:     a[i] = rand();
59: }
60:
61: void Swap(int *x, int *y)
62: {
63:   int t = *x;
64:
65:   *x = *y;
66:   *y = t;
67: }
68:
69: void BubbleSort(int a[], int n)
70: {
71:   int i, j;
72:
73:   for (i = 0; i < n; i++)
74:     for (j = 0; j < n - i - 1; j++)
75:       if(a[j] > a[j + 1])
76:         Swap(&a[j], &a[j + 1]);
77: }
78:
79: void SelectSort(int a[], int n)
80: {
81:   int i, j, k;
82:
83:   for (i = 0; i < n - 1; i++) {
84:     for (k = i, j = i + 1; j < n; j++)
85:       if (a[j] < a[k])
86:         k = j;
87:     Swap(&a[k], &a[i]);
88:   }
89: }
90:
91: void InsertSort(int a[], int n)
92: {
93:   int i, j;
94:
95:   for (i = 1; i <= n - 1; i++) {
96:     j = i;
97:     while (a[j] < a[j-1] && j >= 1) {
98:       Swap(&a[j], &a[j - 1]);
99:       j--;
```

```
100:     }
101:   }
102: }
103:
104: void ShellSort(int a[], int n)
105: {
106:   int i, j, k, temp;
107:   int gap;
108:   int d[] = {9, 5, 3, 2, 1};
109:
110:   for (k = 0; k < 5; k++) {
111:     gap = d[k];
112:     for (i = gap; i < n; i++) {
113:       temp = a[i];
114:       for (j = i - gap; temp < a[j] && j >= 0; j -= gap)
115:         a[j + gap] = a[j];
116:       a[j + gap] = temp;
117:     }
118:   }
119: }
120:
121: void QuickSort(int a[], int n, int left, int right)
122: {
123:   int i = left, j = right;
124:   int test = a[(left + right) / 2];
125:
126:   do {
127:     while (a[i] < test)
128:       i++;
129:     while (a[j] > test)
130:       j--;
131:     if (i <= j) {
132:       Swap(&a[i], &a[j]);
133:       i++;
134:       j--;
135:     }
136:   } while(i <= j);
137:   if (i < right)
138:     QuickSort(a, n, i, right);
139:   if (j > left)
140:     QuickSort(a, n, left, j);
141: }
```

В строках 11-15 объявляются прототипы функций пяти методов сортировки. В строках 69-141 приведены уже знакомые вам описания. Все функции используют вспомогательную функцию **Swap()**, меняющую местами значение двух элементов (строки 61-67).

Символическая константа **N**, объявленная в строке 6, определяет количество сортируемых элементов. В строке 19 объявлено два массива: исходный (**source**) и тестовый (**test**) размерности **N**.

В функции **main()** в строке 26 вызывается функция **InitArray()**, которая инициализирует исходный массив случайными значениями. После этого с помощью функции **CopyArrays()** исходный массив копируется в тестовый.

Обратите внимание на строку 28:

```
begin = clock(); BubbleSort(test, N); end = clock();
```

Функция **clock()**, объявленная в заголовочном файле time.h, возвращает количество программных тактов (один такт примерно равен 1/18 секунды), прошедших со времени запуска программы. Таким образом, вызывая **clock()** до и после вызова функции сортировки мы можем засечь время выполнения функции в тактах. Чтобы определить время работы функции в секундах, используйте выражение (**end** – **begin**) / **CLK_TCK**, как это сделано в следующей строке.

После выполнения строки 28 массив **test** уже отсортирован, поэтому, прежде чем передать его следующей функции сортировки, в строке 30 мы вновь наполняем его случайными значениями из массива **source**.

После окончания работы программы вы увидите таблицу, похожую на ту, что изображена на рис. 7.2. В полном соответствии с теорией метод «пузырька» оказался самым медленным – 28 секунд. Метод выбора показал значительно лучшие результаты: в данном случае он работал в 4 раза быстрее «пузырька». Метод вставки затратил на выполнение задания примерно такое же время, что и метод «пузырька», однако в случае частично упорядоченных массивов этот метод работает гораздо быстрее. Усовершенствованные методы сортировки затратили на сортировку менее одной секунды. Быстрая сортировка оказалась в 514 раз быстрее «пузырька»!

```
Sorting Methods Test

Number of elements:  15000

Sorting methods      Time (sec)

Bubble sort          28.241758
Selection sort        7.637363
Insertion sort       22.967033
Shell sort            0.769231
Quick sort            0.054945
```

Рис. 7.2. Результаты работы программы sort

Упражнение. *Расширьте тестирующие возможности программы sort, измеряя скорость работы пяти методов при сортировке частично отсортированных, полностью отсортированных и отсортированных в обратном порядке массивов.*

Выбор метода сортировки

Опытные программисты, имея в своем арсенале несколько методов сортировки, выбирают наиболее подходящий из них для каждой конкретной ситуации. Руководствуйтесь при выборе следующими правилами:

- Обычно наилучшие результаты дает метод быстрой сортировки.
- При сортировке небольших массивов (менее 100 элементов) лучше использовать прямые методы, так как сложные алгоритмы имеют больший размер кода и при малом количестве сортируемых элементов неэффективны.
- Если сортируется связный список, то для него более эффективны методы вставок (метод обычных вставок или алгоритм Шелла).
- Если массив частично упорядочен, также лучше работают методы вставок.

Поиск данных

Пожалуй, одним из самых частых действий в программировании и повседневной жизни является поиск. Что только мы не ищем! Конспекты, аудитории, IP-адреса, телефоны, – одним словом, любой предмет среди ему подобных. Причем, как будет видно из этого раздела, человек использует один из самых эффективных методов поиска, даже не подозревая об этом.

Поиск – это процесс, результатом которого является указание точного места расположения искомого элемента (или, как его еще называют, *ключа*). Ниже рассмотрены несколько основных методов поиска.

Линейный поиск

Если никакой дополнительной информации о разыскиваемых данных нет (например, неизвестно, отсортирован массив или нет), то самый очевидный подход – это простой последовательный просмотр массива. Такой метод называется линейным поиском.

Условие окончания поиска:

1. Элемент найден, т.е. **array[i] == key**, либо
2. Весь массив просмотрен и совпадения не обнаружено.

Используя эти условия, реализуем алгоритм линейного поиска:

```
int LinearSearch(int array[], int n, int key)
{
  int i;

  for (i = 0; i < n; i++)
    if (array[i] == key)
      return i;
  return -1;
}
```

Функция **LinearSearch()** принимает в виде параметров массив, в котором будет производиться поиск, его размерность **n** и искомый элемент **key**. Обратите внимание, что если элемент найден, то определяется только

его первое вхождение в массив. Функция возвращает –1, если совпадения не существует.

Поскольку массив не упорядочен, вероятность нахождения требуемого значения в первом и последнем элементах массива одинакова. Таким образом, в среднем программа должна будет сравнить искомый элемент с половиной элементов массива, т.е. среднее число сравнений в данном методе равно n / 2. Очевидно, что метод линейного поиска в больших массивах применять нецелесообразно.

Бинарный поиск

Алгоритм линейного поиска неэффективен по своей природе, и его улучшение – неблагодарная задача. Значит, задачу поиска нужно решать другим путем. А что если массив отсортировать? Этот путь очень эффективен, так как сортировка занимает немного времени, а пользу приносит многократную. Действительно, интенсивный поиск в неотсортированном массиве даже представить трудно: кто же будет пользоваться телефонным справочником, в котором фамилии не отсортированы по алфавиту?

Алгоритм поиска делением пополам (так иногда называют бинарный поиск) заключается в следующем. Для определенности предположим, что элементы в массиве упорядочены по возрастанию. Выберем случайно некоторый элемент, например **array[m]**, и сравним его с искомым элементом **key**. Если он равен **key**, то поиск заканчивается, если он меньше **key**, то делаем вывод, что все элементы с индексами, меньшими или равными m, можно исключить из дальнейшего поиска; если же он больше **key**, то исключаются индексы большие и равные m.

Выбор m совершенно не влияет на корректность алгоритма, но влияет на его эффективность. Очевидно, что чем большее количество элементов исключается на каждом шаге алгоритма, тем этот алгоритм эффективнее. Оптимальным решением будет выбор среднего элемента, так как при этом в любом случае будет исключаться половина массива.

```
int BinarySearch (int array[], int key, int i, int j)
{
  int middle;

  while (i <= j) {
    middle = (i + j) / 2;
    if (key == array[middle])
      return middle;
    else if (key < array[middle])
      j = middle - 1;
    else
      i = middle + 1;
   }
   return -1;
```

```
}
```

Приведенный алгоритм, как и в случае линейного поиска, находит первый совпадающий элемент в массиве. Максимальное число сравнений равно $\log_2 n$, время работы пропорционально $\log n$.

Чтобы оценить эти цифры, представьте, что вам необходимо найти информацию в 1000-элементном массиве. Поскольку $2^{10} > 1000$, то, используя метод бинарного поиска, для нахождения заданного элемента вам нужно сделать не более десяти сравнений! Для поиска в массиве, содержащем миллион элементов, потребуется не более двадцати сравнений, поскольку $2^{20} > 1\ 000\ 000$. Чтобы найти элемент среди миллиарда подобных потребуется только около 30 сравнений, что значительно(!) меньше, чем порядка 500 000 000 сравнений при использовании линейного поиска.

Листинг 7.2. ищет ключ в массиве с помощью линейного и бинарного алгоритмов поиска. Программа позволяет сравнить эффективность двух методов, измеряя среднее количество сравнений.

Листинг 7.2. search.c (эффективность линейного и бинарного поиска)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <conio.h>
 4: #include <time.h>
 5:
 6: #define N 3000
 7: #define M 50
 8:
 9: void InitArray(int a[], int n);
10: int LinearSearch(int a[], int n, int key);
11: int BinarySearch(int a[], int key, int i, int j);
12: void ShellSort(int a[], int n);
13:
14: unsigned long counter1 = 0, counter2 = 0;
15:
16: main()
17: {
18:   int array[N];
19:   int i, key;
20:
21:   clrscr();
22:   for (i = 0; i < M; i++) {
23:     InitArray(array, N);
24:     key = array[random(N)];
25:     LinearSearch(array, N, key);
26:     ShellSort(array, N);
27:     BinarySearch(array, key, 0, N);
28:   }
29:   counter1 /= M;
30:   counter2 /= M;
31:   printf("Linear Search...\n");
```

```
32:   printf("Average amount of operations of matching: %ld\n",
33:     counter1);
34:   printf("\nBinary Search...\n");
35:   printf("Average amount of operations of matching: %ld\n",
36:     counter2);
37:   printf("\nNumber of elements: %d\n", N);
38:   return 0;
39: }
40:
41: void InitArray(int a[], int n)
42: {
43:   int i;
44:
45:   randomize();
46:   for (i = 0; i < n; i++)
47:     a[i] = rand();
48: }
49:
50: int LinearSearch(int array[], int n, int key)
51: {
52:   int i;
53:
54:   for (i = 0; i < n; i++) {
55:     counter1++;
56:     if (array[i] == key)
57:       return i;
58:   }
59:   return -1;
60: }
61:
62: int BinarySearch(int a[], int key, int i, int j)
63: {
64:   int middle;
65:
66:   while (i <= j) {
67:     middle = (j + i) / 2;
68:     if (key == a[middle]) {
69:       counter2++;
70:       return middle;
71:     }
72:     else if (key < a[middle]) {
73:      counter2++;
74:      j = middle - 1;
75:     }
76:     else
77:       i = middle + 1;
78:   }
79:   return -1;
80: }
81:
82: void ShellSort(int a[], int n)
83: {
```

```
 84:   int i, j, k, temp;
 85:   int gap;
 86:   int d[] = {9, 5, 3, 2, 1};
 87:
 88:   for (k = 0; k < 5; k++) {
 89:     gap = d[k];
 90:     for (i = gap; i < n; i++) {
 91:       temp = a[i];
 92:       for (j = i - gap; temp < a[j] && j >= 0; j -= gap)
 93:         a[j + gap] = a[j];
 94:       a[j + gap] = temp;
 95:     }
 96:   }
 97: }
```

Функции **InitArray()** (описана в строках 41-48) и **ShellSort** (строки 82-97) уже вам знакомы. Символическая константа **N** определяет количество элементов в массиве. Глобальная переменная **counter1** используется, чтобы подсчитать количество сравнений в алгоритме линейного поиска, **counter2** – в алгоритме бинарного поиска.

Для того чтобы получить усредненное количество сравнений, поиск заданного значения выполняется в цикле (строки 22-28) **M** раз. На каждой итерации массив инициализируется случайными значениями (строка 23) и переменной **key** присваивается значение случайно выбранного элемента массива. После этого данное значение ищется методом линейного поиска (строка 25) и, после сортировки, методом бинарного поиска (строка 27). После окончания цикла значение переменных **counter1** и **counter2** делится на **M**, что позволяет получить усредненное значение.

Упражнение. *В листинге 7.2 приведены нерекурсивные варианты функций линейного и бинарного поиска. Измените листинг, реализовав эти функции с помощью рекурсии.*

Поиск строки в тексте

Прямой поиск строки

Часто приходится сталкиваться со специфическим видом поиска, при котором необходимо найти первое вхождение слова (строки) в указанном тексте. Это действие типично для любых систем обработки текстов, поэтому для разработки эффективных алгоритмов, решающих эту задачу, было приложено немало усилий.

Рассмотрим функцию, реализующую алгоритм прямого поиска строки:

```
int DirectSearch(char *text, int n, char *str, int m)
{
  int i = -1, j;
  do {
```

```
    i++;
    j = 0;
    while (j < m && text[i + j] == str[j])
      j++;
  } while (! ((j == m) || (i == n - m)) );
  if (j == m)
    return i;
  return -1;
}
```

В функцию передаются в виде параметров текст (**text**), количество символов в тексте (**n**), искомое слово или строка (**str**) и ее длина (**m**). Вложенный цикл **while** начинает выполняться тогда, когда первый символ искомой строки совпадает с очередным, *i*-м, символом текста. Цикл завершается при исчерпании символов строки (перестает выполняться условие **j < m**) или при несовпадении очередных символов **text** и **str** (не выполнятся условие **text[i + j] == str[j]**).

Количество совпадений подсчитывается с использованием переменной **j**. Если совпадение произошло со всеми символами **str** (т.е. строка найдена), то выполняется условие **j == m**, и алгоритм завершается. В противном случае поиск продолжается до тех пор, пока не останется непросмотренной часть текста, которая содержит меньше символов, чем содержится в строке (т.е. этот остаток уже не может совпасть с **str**).

В худшем случае метод прямого поиска требует порядка $m \times n$ сравнений. Алгоритм работает достаточно эффективно в ситуациях, когда несовпадение символов обнаруживается достаточно рано, после небольшого числа сравнений во внутреннем цикле. Для текстов большой длины его использование, скорее всего, будет неэффективным.

Алгоритм Боуера–Мура

В 1975 г. Р. Боуер и Д. Мур изобрели более эффективный алгоритм. Его основным отличием от алгоритма прямого поиска является то, что сдвиг строки (слова) на каждом шаге алгоритма осуществляется не на один символ, а на некоторое количество символов, как правило, большее единицы.

В алгоритме Боуера-Мура сравнение символов начинается не с начала слова, а с конца. Перед началом поиска слово трансформируется в специальную таблицу. Для каждого символа **x** из алфавита рассчитывается величина **dx** – расстояние от самого правого в слове вхождения **x** до правого конца строки. Пусть, например, мы ищем строку “**abcd**”. Посмотрим на четвертую букву текста: если, к примеру, это буква **e**, то нет никакой необходимости читать первые три буквы. (В самом деле, в слове буквы **e** нет, поэтому оно может начаться не раньше пятой буквы.) Следовательно, мы можем сделать сдвиг вправо на всю длину строки. Если же, например, это буква **b**, то необходимо сделать сдвиг на величину **dx = 2**, равное расстоянию символа **b** до конца строки.

Функция **BMSearch()** реализует упрощенную схему алгоритма Боуера–Мура.

```
int BMSearch(char *text, int n, char *str, int m)
{
  int i = m, j = m, k, i1, D[128];
  /* Таблица расстояний */
  for (i1 = 0; i1 < n; i1++)
    D[i1] = m;
  for (i1 = 0; i1 < m - 1; i1++)
    D[(int)str[i1]] = m - i1 - 1;
  /* Поиск заданной строки в тесте */
  while(j > 0 && i <= n) {
    j = m;
    k = i;
    while(j > 0 && text[k - 1] == str[j - 1]) {
      k--;
      j--;
    }
    i += D[(int)text[i - 1]]; /* сдвиг слова */
  }
  if (m > 0 && n > 0 && j == 0)
    return k; /* возвращается позиция 1-го элемента строки */
  return -1;  /* или -1 в случае неудачи */
}
```

В подавляющем большинстве случаев БМ-поиск требует значительно меньше n сравнений. В лучшем случае число сравнений равно n / m.

Листинг 7.3 осуществляет поиск строки в тексте методом Буера–Мура и прямого поиска строки. Как и в предыдущем примере, программа оценивает эффективность алгоритмов по количеству сравнений, необходимых для поиска нужного элемента.

Листинг 7.3. tsearch.c (метод прямого поиска строки и БМ-поиск)

```
  1: #include <stdio.h>
  2: #include <string.h>
  3: #include <stdlib.h>
  4: #include <math.h>
  5:
  6: #define N 128
  7: int counter1 = 0, counter2 = 0;
  8: int DirectSearch(char *text, int n, char *str, int m);
  9: int BMSearch(char *text, int n, char *str, int m);
 10:
 11: void main()
 12: {
 13:   char str[N], text[N];
 14:   int index, n, m;
 15:
 16:   printf("Input the text: ");
 17:   gets(text);
```

```
18:   n = strlen(text);
19:   printf("Input the string to find: ");
20:   gets(str);
21:   m = strlen(str);
22:   printf("\nText: %s\n", text);
23:   printf("Text length: %d\n\n", n);
24:   printf("Direct Search...\n");
25:   index = DirectSearch(text, n, str, m);
26:   printf("Position in text: %d\n", index);
27:   printf("Amount of operations of matching: %d\n",counter1);
28:   printf("\nBM-Search...\n");
29:   index = BMSearch(text, n, str, m);
30:   printf("Position in text: %d\n", index);
31:   printf("Amount of operations of matching: %d\n",counter2);
32: }
33:
34: int DirectSearch(char *text, int n, char *str, int m)
35: {
36:   int i = -1, j;
37:   do {
38:     i++;
39:     j = 0;
40:     counter1++;
41:     while (j < m && text[i + j] == str[j]) {
42:       j++;
43:       counter1++;
44:     }
45:   } while (! ((j == m) || (i == n - m)));
46:   if (j == m)
47:     return i;
48:   return -1;
49: }
50:
51: int BMSearch(char *text, int n, char *str, int m)
52: {
53:   int i = m, j = m, k, i1, D[N];
54:
55:   for (i1 = 0; i1 < N; i1++)
56:     D[i1] = m;
57:   for (i1 = 0; i1 < m - 1; i1++)
58:     D[(int)str[i1]] = m - i1 - 1;
59:
60:   while(j > 0 && i <= n) {
61:     j = m;
62:     k = i;
63:     counter2++;
64:     while(j > 0 && text[k - 1] == str[j - 1]) {
65:       k--;
66:       j--;
67:       counter2++;
68:     }
69:     i += D[(int)text[i - 1]];
```

```
70:   }
71:   if (m > 0 && n > 0 && j == 0)
72:     return k;
73:   return -1;
74: }
```

Замечание. *В данном упрощенном примере текст представлен 128-байтовой строкой. Следующая глава расскажет, как обрабатывать большие объемы текстовой информации, хранящейся в файлах.*

Какой метод выбрать?

Как и в случае с сортировками, идеального метода поиска не существует. Для поиска среди 20 элементов нет разницы, какой метод использовать (скорее тот, который меньше набирать), но если речь идет о поиске информации в массиве данных с сотнями тысяч элементов, метод бинарного поиска будет значительно эффективнее.

Используйте простой алгоритм прямого поиска строки в небольших текстах. При работе с большими объемами текстовой информации выберите метод Буера–Мура.

Резюме

- Сортировка необходима, чтобы быстро найти нужную информацию в большом массиве данных. Сортировка и поиск являются одними из самых распространенных задач программирования.
- Прямые методы сортировки просты в реализации, но работают медленно. Наиболее известны метод «пузырька», выбора и вставок. Эти методы идеально подходят для сортировки массивов, содержащих не более 100 элементов. Усовершенствованные методы сортировки более сложны, но работают гораздо быстрее. Их лучше использовать при сортировке больших массивов данных.
- Запомните следующие формулы: время, необходимое для сортировки прямыми методами, пропорционально n^2, методом Шелла – $n^{1.2}$, методом быстрой сортировки – $n \log n$, где n – количество сортируемых элементов.
- При линейном поиске происходит сравнение каждого элемента массива с ключом. Среднее число сравнений в данном методе равно n / 2. Метод хорошо работает в массивах небольшого размера, а также (за неимением лучшего) в массивах, которые по каким-то причинам нельзя упорядочить.
- В бинарном поиске после каждого сравнения исключается половина элементов отсортированного массива. Среднее число сравнений в данном методе не превышает $\log_2 n$. При поиске информации в крупных массивах он оказывается наиболее эффективен.

- Метод прямого поиска строки просматривает текст последовательно, символ за символом, пытаясь установить соответствие между заданной строкой и частью текста. Метод работает хорошо, когда в тексте мало похожих слов, и расхождение обнаруживается на первых же итерациях. В худшем случае количество сравнений в данном методе равно произведению длины текста и слова: $n \times m$.
- В алгоритме поиска Боуера-Мура сравнение начинается с конца слова. Символы в тексте просматриваются не последовательно, а скачками, что значительно ускоряет алгоритм. Кроме специально подобранных случаев, БМ-поиск требует количества сравнений, намного меньшего n, поэтому метод целесообразно применять для поиска в текстах большой длины.
- Идеальных методов сортировки и поиска данных не существует. Опытные программисты владеют несколькими методами и выбирают наиболее эффективный из них для каждого конкретного случая.

Обзор функций

Таблица 7.1

Функции работы со временем (time.h)

Функция	**Прототип и краткое описание**
1	2
clock()	`clock_t clock(void);` Возвращает приближенное время процессора, прошедшее с момента запуска программы; чтобы получить время в секундах, необходимо разделить эту величину на значение **CLK_TCK**; возвращает –1, если информация о времени недоступна или непредставима.
ctime()	`char *ctime(const time_t *tp);` Преобразует календарное время **tp** в 26-символьную строку в форме **`Thu Mar 18 13:09:23 2004\n\0`**, и возвращает указатель на эту строку.
difftime()	`double difftime(time_t t2, time_t t1);` Вычисляет разницу (**t2** – **t1**) между двумя отметками календарного времени, выражает результат в секундах и возвращает его как число типа **double**.

Окончание табл. 7.1

1	2
stime()	`int stime(time_t *tp);` Устанавливает системное время и дату, выраженных как

количество секунд, прошедших с первой секунды 1970 года. **tp** указывает на новое значение времени.

time() `time_t time(time_t *tp);`

Возвращает текущее календарное время и также помещает его по адресу, указанному **tp**, если **tp** не равен **NULL**. Возвращает –1, если календарное время недоступно.

Примечание. *Тип **clock_t**, который используют функции из табл. 7.1, представляет собой определяемый тип, соответствующий типу **long**. Календарное время, которое он представляет, понимается как количество секунд, прошедших с первой секунды 1970 года.*

8. Файлы

Файлы являются существенной частью современных компьютерных систем. Они используются для хранения программ, документов, данных, корреспонденции, форм, графиков и многих других типов информации.

Программы, которые мы разрабатывали до сих пор, обладали одним общим недостатком: данные, хранящиеся в переменных и массивах, после завершения программы утрачивались. Техника работы с файлами позволит сохранить данные даже после выключения компьютера, что позволит вам создавать более сложные и совершенные программы.

Что такое файл?

Файл – это именованный раздел (обычно на диске) для сохранения информации. Язык С рассматривает файл как последовательность байтов, каждый из которых считывается в индивидуальном порядке (рис. 8.1). Каждый файл имеет внутренний указатель. Этот указатель обозначает позицию, с которой начнется следующая операция чтения либо записи новых данных. Как только вы прочитали или записали данные, внутренний указатель файла передвигается, подобно лодке, вверх или вниз по течению.

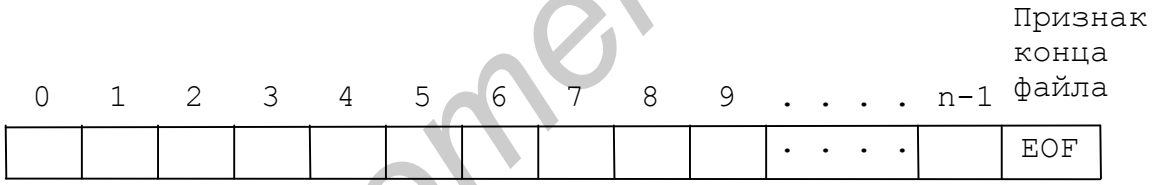

Рис. 8.1. Вид файла из n байтов

Последний байт файла имеет значение **EOF** (End Of File). Используйте эту константу в ваших программах, чтобы определить конец файла. Например,

```
do {                    /* выполнять */
...                     /* операторы */
} while (c != EOF)      /* пока не встретится "конец файла" */
```

Язык С имеет богатую библиотеку функций чтения, записи файлов и выполнения других операций (см. раздел «Обзор функций»). Большинство функций для работы с файлами начинается на букву **f**. Примите к сведению следующие замечания:

- Перед тем как использовать файлы на диске, вы должны их открыть. При открытии вы должны задать специальный режим доступа, чтобы было понятно, с каким типом файлов вы собираетесь работать и что именно вы собираетесь делать: читать или записывать данные.

- После того как файл открыт, вы можете начинать работу. При помощи текущего указателя файла вы можете выполнить чтение или запись данных в любую позицию файла.
- После окончания работы с файлом его нужно закрыть. Закрытие файла переносит в файл все данные, буферизованные в памяти. Когда программа завершается, все открытые файлы автоматически закрываются. Но все же лучше всегда закрывать файлы явным образом.

Текстовые файлы

Существует два основных способа обработки текстовых файлов: по символам или по строкам. Выбор обычно диктуют потребности вашей программы.

Чтение в посимвольном режиме

Открытие файла в режиме посимвольного чтения дает вам возможность проверить каждый символ файла. Листинг 8.1 демонстрирует этот метод. Откройте новый проект и скомпилируйте вместе модули gets.c из главы 5 и rchar.c. Перед запуском программы создайте текстовый файл в том же каталоге, где находятся модули. После этого запустите программу на выполнение и введите имя текстового файла.

Листинг 8.1. rchar.c (посимвольное чтение текстового файла)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <conio.h>
 4: #include <string.h>
 5: #include <alloc.h>
 6: #include "gets.h"
 7:
 8: #define SIZE 128 /* макс. размер строки для имени файла */
 9: void Error(const char *message);
10:
11: main()
12: {
13:   FILE *fp;
14:   char c, *filename;
15:
16:   printf("File name? ");
17:   filename = GetStringAt(SIZE);
18:   fp = fopen(filename, "r");
19:   if (!fp)
20:     Error("Opening file");
21:   while ((c = fgetc(fp)) != EOF)
22:     putchar(c);
23:   fclose(fp);
24:   free(filename);
```

```
25:   return 0;
26: }
27:
28: void Error(const char *message)
29: {
30:   printf("\n\nError: %s\n\n", message);
31:   exit(1);
32: }
```

Функция **main()** использует три переменные. Строка 13 объявляет указатель типа **FILE ***. Файловые функции в дальнейшем используют этот указатель для доступа к файлам. Две другие переменные предназначены для хранения символов, прочитанных из файла (**char c**), и адресации строки, содержащей имя файла (**char *filename**).

Оператор

```
filename = GetStringAt(SIZE);
```

устанавливает значение **filename** равным адресу строки, возвращаемой уже знакомой вам функцией **GetStringAt()**. После того как программа получит имя файла, программа в строке 18 вызывает функцию **fopen()**, для того чтобы открыть файл:

```
fp = fopen(filename, "r");
```

Первым параметром функции является имя файла, который необходимо открыть, второй параметр, “**r**”, определяет режим доступа к файлу. Этот параметр может быть одной из строк, перечисленных в табл. 8.1.

Таблица 8.1.

Режимы доступа к файлам для функции fopen()

Строка	Описание
"r"	Открывает файл только для чтения. Модификации файла не разрешены
"w"	Создает новый файл только для записи. Перезаписывает любой существующий файл с тем же именем. Чтение информации из файла не разрешено
"a"	Открывает файл в режиме только для записи с добавлением новой информации в конец файла. Если файла не существует, он создается, и любой существующий файл с тем же именем перезаписывается. Чтение информации из файла не разрешено
"r+"	Открывает существующий файл для чтения и записи
"w+"	Создает новый файл для чтения и записи. Перезаписывает любой существующий файл с тем же именем
"a+"	Открывает файл в режиме чтения и записи для добавления новой информации в конец файла. Если файла не существует, он создается, и любой существующий файл с тем же именем перезаписывается

Функция **fopen()** возвращает указатель файла типа **FILE ***, который вы должны запомнить в переменной, как показано в строке 18. Если возвращаемое значение равно **NULL**, значит, файл не может быть открыт (возможно, файла с указанным именем нет в директории либо он поврежден). Если функция **fopen()** возвращает ненулевое значение, значит, файл открыт и готов к работе.

Строки 21-22 демонстрируют процесс посимвольной обработки текстового файла. Цикл **while** вызывает функцию **fgetc()**, передавая ей в качестве аргумента указатель на файл **fp** и присваивая результат функции переменной **c**. Функция **fgetc()** аналогична функции **getchar()**, но в отличие от нее требует аргумент типа **FILE *** и, таким образом, может читать символы из любого файла, открытого функцией **fopen()**. Строка 22 отображает каждый символ, возвращаемый функцией **fgetc()**.

Цикл **while** завершится, когда функция **fgetc()** возвратит значение **EOF**, означающее, что программа прочитала последний символ в файле. Отличие этого значения от любого допустимого данного, которое может вернуть функция **fgetc()** (и другие функции чтения из файлов), гарантируется. После этого строка 23 закрывает файл путем вызова функции **fclose()**.

Чтение в построчном режиме

Большинство текстовых файлов организовано в виде строк, которые заканчиваются управляющими кодами возврата каретки и перевода строки. Вы можете обрабатывать эти файлы построчно, что повышает скорость работы программы за счет уменьшения количества циклов и вызовов функций. Однако построчное чтение имеет один серьезный недостаток: вы должны определить заранее максимальную длину строки и обеспечить соответствующий буфер.

Листинг 8.2 показывает, как следует выполнять построчное чтение текстового файла. Скомпилируйте и запустите программу аналогично предыдущему примеру, но вместо модуля rchar.c включите rline.c. Текстовые файлы, которые вы будете читать с помощью этой программы, не должны содержать строки длиной более 128 символов.

Листинг 8.2. rline.c (построчное чтение текстового файла)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <string.h>
 4: #include <alloc.h>
 5: #include "gets.h"
 6:
 7: #define SIZE 128 /* максимальный размер строки имени файла*/
 8: void Error(const char *message);
 9:
10: main()
11: {
```

```
12:   FILE *fp;
13:   char buffer[128], *filename;
14:
15:   printf("File name? ");
16:   filename = GetStringAt(SIZE);
17:   fp = fopen(filename, "r");
18:   if (!fp)
19:     Error("Opening file");
20:   while (fgets(buffer, 128, fp) != NULL)
21:     puts(buffer);
22:   fclose(fp);
23:   return 0;
24: }
25:
26: void Error(const char *message)
27: {
28:   printf("\n\nError: %s\n\n", message);
29:   exit(1);
30: }
```

Программа rline имеет одно существенное отличие от программы rchar. Ее цикл **while** в строках 20-21 вызывает функцию **fgets()**, чтобы прочитать строку из файла в буфер, объявленный в строке 13. Функция **fgets()** требует наличия трех аргументов: строковой переменной для запоминания строки текста, максимального числа читаемых символов и переменной типа **FILE ***, инициализированной с помощью функции **fopen()**.

При чтении строки из открытого файла функция **fgets()** добавляет в строку завершающий нуль. Если функция встречает символ новой строки (‘**\n**’), он также заносится в буфер. Убедитесь, что вы подготовили достаточно большой буфер для работы функции **fgets()**: любые строки, длина которых превышает размер буфера, усекаются.

Упражнение. *Напишите программу, которая читает строки текстового файла и выводит их на экран в алфавитном порядке.*

Посимвольная запись

Чтобы создать новый текстовый файл, объявите переменную типа **FILE ***, например, с именем **fp** и откройте его следующим образом:

```
fp = fopen("newfile.txt", "w");
```

Режим “**w**” создает новый файл, перезаписывая существующий с таким же именем. Убедитесь, что это как раз то, что вы хотели сделать! Чтобы открыть существующий файл для чтения и записи, используйте режим “**r+**”. В этом случае существующий файл не будет перезаписан.

После вызова функции **fopen()**, если **fp** не равно нулю, файл будет открыт и готов к приему символов. Например, записать в файл символы **A**, **B** и **C** можно следующим образом:

```
fputc('A', fp);
fputc('B', fp);
fputc('C', fp);
```

После работы с файлом закройте его с помощью функции **fclose()**:

```
fclose(fp);
```

Упражнение. *Напишите программу, которая преобразует все строчные буквы текстового файла в прописные.*

Построчная запись

Запись строк текста в файлы весьма похожа на посимвольную запись. Используйте те же операции открытия, пересылки данных и закрытия файлов. После открытия файла вы можете записать строку на диск путем вызова функции **fputs()**:

```
fputs("Запишите меня на диск, пожалуйста!", fp);
```

В отличие от **puts()** функция **fputs()** не добавляет символ перехода на новую строку в результирующую строку. Поэтому после **fputs()** вы должны вызывать функцию **fputc()** с символом новой строки в качестве первого аргумента:

```
fputc('\n', fp);
```

Замечание. *При записи в файл символ '**\n**' трансформируется в символы возврата каретки и перевода строки*.

Упражнение. *Напишите программу, которая объединяет два текстовых файла в один большой файл.*

Функция *printf()* и родственные ей функции

Почти в каждом листинге данного пособия есть функция **printf()**. Эта и родственные ей функции пригодятся вам и при работе с файлами.

С помощью функций семейства **printf()** можно записывать сформатированный текст прямо в файлы на диске. Для этого откройте файл путем вызова функции **fopen()**, используя один из режимов записи, приведенных в табл. 8.1. Затем вызовите функцию **fprintf()** аналогично тому, как вы вызывали функцию **printf()**, но в качестве первого аргумента задайте переменную типа **FILE ***.

Например, следующий оператор запишет в файл переменную **d** типа **double**, сформатированную в восьми позициях с двумя знаками после запятой:

```
fprintf(fp, "%8.2lf", d);
```

Особенно полезна функция **sprintf()** – строковая версия **printf()**. Объявите строковый буфер для хранения результата, а затем вызовите ее следующим образом:

```
char buffer[128]; /* буфер для хранения результата */
sprintf(buffer, "%8.2lf", d);
```

Если **d** – переменная типа **double**, то этот оператор поместит в буфер строковое представление переменной **d**, оканчивающееся нулевым байтом.

Листинг 8.3 демонстрирует практическое применение функции **fprintf()** – добавление номеров строк в текстовые файлы (как в листингах, напечатанных в этом пособии). Скомпилируйте вместе модули number.c и gets.c. Затем запустите программу и введите имя текстового файла и имя выходного файла, в который будут записаны строки с номерами.

Листинг 8.3. number.c (добавление номеров строк в текстовый файл)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <conio.h>
 4: #include <string.h>
 5: #include <alloc.h>
 6: #include "gets.h"
 7:
 8: #define SIZE 128 /* максимальный размер для имен файлов */
 9: void Error(const char *message);
10:
11: main()
12: {
13:   char *inpfname, *outfname; /* имена файлов */
14:   FILE *inpf, *outf; /* указатели на входной и вых. файл */
15:   char buffer[256];  /* запоминает прочитанные строки */
16:   int lineNumber = 0;/* текущий номер строки */
17:
18:   /* Предложить ввести имя входного и выходного файлов */
19:   printf("Input file name? ");
20:   inpfname = GetStringAt(SIZE);
21:   printf("Output file name? ");
22:   outfname = GetStringAt(SIZE);
23:
24:   /* Открыть файлы */
25:   inpf = fopen(inpfname, "r");
26:   if (!inpf)
27:     Error(inpfname);
28:   outf = fopen(outfname, "w");
29:   if (!outf)
30:     Error(outfname);
31:
32:   /* Добавить номера строк в выходной файл */
33:   printf("Numbering %s\n", outfname);
34:   while (fgets(buffer, 255, inpf) != NULL) {
35:     fprintf(outf, "%4d: %s", ++lineNumber, buffer);
36:     putchar('.'); /* отобразить на экране обратную связь */
37:   }
38:
39:   /* Закрыть файлы и освободить память */
40:   fclose(inpf);
41:   fclose(outf);
```

```
42:   printf("\nLine numbers added to %s\n", outfname);
43:   free(inpfname);
44:   free(outfname);
45:   return 0;
46: }
47:
48: void Error(const char *message)
49: {
50:   if (errno == 0)
51:     printf("Internal error: %s", message);
52:   else
53:     perror(message); /* распечатать системную ошибку */
54:   exit(1);
55: }
```

Программа number – хорошая иллюстрация принципов обработки файлов, описанных ранее. Кроме того, программа демонстрирует несколько новых приемов.

В строке 13 объявляются две переменных типа **char ***: **inpfname** (имя входного файла) и **outfname** (имя выходного файла). Этим переменным присваивается адреса строк, которые вы введете в ответ на приглашение.

После создания имен файлов, а также открытия входного и выходного файлов (строки 25-30) программа number выполняет цикл **while** (строки 34-37). В цикле вызывается функция **fgets()** для чтения строк из входного файла, а также функция **fprintf()**, которая добавляет номер в начало каждой строки и записывает их в выходной файл.

Строка 36 отображает на экране точку для каждой обработанной строки – хороший метод обратной связи при длительном выполнении задачи, который обеспечивает уверенность в том, что программа работает.

Функция **perror()** (строка 53) демонстрирует удобный способ отображения сообщений об ошибках для различных функций ввода-вывода. Большинство функций устанавливает внутреннюю переменную **errno** равной значению, представляющему одну или несколько ошибок. Вызов функции **perror()** с любым строковым аргументом отобразит эту строку с описанием аварийной ситуации.

Функция *scanf()* и родственные ей функции

Функция **scanf()** и родственные ей функции **cscanf()**, **fscanf()**, **sscanf()**, выполняют ввод двоичных значений из текстовых источников.

Листинг 8.4 показывает, как использовать **fscanf()** для чтения нескольких значений с плавающей запятой, сохраненных в текстовом файле. Перед запуском программы создайте файл с помощью любого текстового редактора по приведенному ниже образцу. Сохраните файл под именем test.dat в том же каталоге, где находится файл rarray.c. Первая строка в файле

по договоренности представляет собой целое число, равное количеству значений в файле.

```
10
3.14159
79.86
100.0
85.3
24.329
7.0
66.32
89.99
12.31
9.99
```

Теперь скомпилируйте и запустите программу rarray. Программа загрузит содержимое файла test.dat в массив значений типа **double** и отобразит его на экране.

Листинг 8.4. rarray.c (чтение массива чисел из текстового файла)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3:
 4: main()
 5: {
 6:   FILE *inpf;
 7:   int i, count;
 8:   double *array;
 9:
10:   /* Открыть файл */
11:   inpf = fopen("test.dat", "r");
12:   if (!inpf) {
13:     puts("Can't open test.dat");
14:     exit(1);
15:   }
16:
17:   /* Прочитать количество значений в файле */
18:   fscanf(inpf, "%d", &count);
19:
20:   /* Создать массив и прочитать значения из файла */
21:   printf("\nCreating array of %d values\n", count);
22:   array = (double *)malloc(count * sizeof(double));
23:   for (i = 0; i < count; i++)
24:     fscanf(inpf, "%lf", &array[i]);
25:   fclose(inpf);
26:
27:   /* Отобразить массив значений */
28:   for (i = 0; i < count; i++)
29:     printf("array[%d] = %lf\n", i, array[i]);
30:   free(array); /* освободить память */
31:   return 0;
32: }
```

В строке 18 показано, как использовать функцию **fscanf()**. В качестве первого аргумента передайте указатель открытого файла (**inpf**), в качестве второго – строку формата, которая сообщит функции, значения какого вида ей следует ожидать (**%d**), в качестве третьего – адрес переменной, в которой будет сохранено значение (**count**).

Строки из файла загружаются в динамический массив типа **double**, память для которого выделяется с помощью функции **malloc()** (строка 22). Цикл **for** в строках 27-29 снова вызывает функцию **fscanf()**, которая на этот раз задает строку формата "**%lf**" – значение с плавающей запятой двойной точности. После каждого вызова функции **fscanf()** внутренний указатель файла автоматически перемещается на следующую строку данных, что позволяет последовательно сохранить значения из файла в элементах массива.

Двоичные файлы

Текстовые файлы, несмотря на свое широкое распространение, являются только одним из видов файлов, которые можно хранить на дисках компьютера. Вместо того чтобы запоминать числа в текстовой форме, вы можете записывать двоичные данные типа **int** и **double** прямо в файлы, а затем считывать их значения в переменные вашей программы.

Одно из преимуществ запоминания двоичных данных в файлах – скорость. Преобразование из текстового формата в двоичный и обратно требует времени. Другое преимущество – память. Значение типа **double** обычно занимает 8 байтов. То же самое значение в текстовой форме может занимать гораздо больше байтов, особенно, если под каждое значение выделяется отдельная строка с управляющими символами возврата каретки и перевода строки в конце.

Обработка двоичных файлов

Язык С позволяет довольно легко работать с двоичными файлами. Методы обработки почти такие же, как и для текстовых файлов.

При открытии файлов для двоичной обработки, как и раньше, вызывайте функцию **fopen()**, но добавьте строчную букву **b** ко всем режимам доступа из табл. 8.1. Например, если **fp** - переменная типа **FILE ***, оператор

```
fp = fopen("myfile.dat", "r+b");
```

откроет файл myfile.dat для чтения и записи в двоичном режиме. Другие режимы доступа к двоичным файлам имеют вид: "**rb**" (только чтение), "**wb**" (создание нового файла), "**ab**" (добавление в конец файла) и т.д.

Существует два основных способа чтения и записи двоичных файлов – последовательный и произвольный. Давайте поближе познакомимся с этими двумя способами.

Файлы с последовательным доступом

Последовательная обработка полезна для быстрого запоминания и считывания значений в файлах. Обычно ее применяют, когда над всеми данными в файле выполняются одинаковые действия.

Простой пример демонстрирует принципы последовательной обработки данных. Листинг 8.5 записывает в двоичный файл целочисленный массив. Скомпилируйте и запустите программу wint, которая создаст файл int.dat в текущем каталоге.

Листинг 8.5. wint.c (запись целочисленного массива в двоичный файл)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3:
 4: main()
 5: {
 6:   FILE *fp;
 7:   int i;
 8:
 9:   fp = fopen("int.dat", "wb");
10:   if (!fp)  {
11:     puts("Can't create int.dat!");
12:     exit(1);
13:   }
14:   puts("Writing 100 integer values to int.dat");
15:   for (i = 0; i < 100; i++)
16:     fwrite(&i, sizeof(int), 1, fp);
17:   fclose(fp);
18:   return 0;
19: }
```

Строка 9 показывает, как создавать новый двоичный файл, используя опцию "**wb**". Если функция **fopen()** возвращает допустимый файловый указатель, то оператор **for** в строках 15-16 вызывает функцию **fwrite()** для записи ста целых значений в файл int.dat. Поскольку переменная типа **int** обычно занимает два байта, то результирующий файл будет иметь размер ровно 200 байтов.

Функция **fwrite()** требует следующие четыре параметра:

1. Адрес переменной или массива, из которого байты копируются на диск.
2. Число байтов в одной переменной.
3. Число записываемых элементов.
4. Переменную типа **FILE ***, указывающую на файл, открытый в двоичном режиме для записи.

Записать весь массив на диск можно и с помощью одного вызова функции **fwrite()**. Например, если объявлен массив

```
int array[100];
```

следующий оператор запишет его на диск:

```
fwrite(array, sizeof(int), 100, fp);
```

Чтение двоичных данных из файла программируется аналогично записи. Скомпилируйте и запустите программу rint, которая считывает и отображает значения из файла int.dat, созданного программой wint.

Листинг 8.6. rint.c (чтение двоичных значений из файла int.dat)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3:
 4: main()
 5: {
 6:   FILE *fp;
 7:   int i, value;
 8:
 9:   fp = fopen("int.dat", "rb");
10:   if (!fp) {
11:     puts("Can't open int.dat");
12:     exit(1);
13:   }
14:   for (i = 0; i < 100; i++) {
15:     fread(&value, sizeof(int), 1, fp);
16:     printf("%8d", value);
17:   }
18:   fclose(fp);
19:   return 0;
20: }
```

Строка 9 открывает файл int.dat, используя опцию “**rb**” («только чтение в двоичном режиме»). Чтобы выполнить последовательную загрузку значений с диска, цикл **for** в строках 14-17 вызывает функцию **fread()**, которая требует те же аргументы, что и функция **fwrite()**.

Функция **fread()** также может загрузить больше одного значения с диска за один прием. Если добавить в программу rint объявление целочисленного массива:

```
int array[100];
```

то вы сможете заменить цикл **for** одним вызовом функции **fread()**:

```
fread(array, sizeof(int), 100, fp);
```

Этот оператор является самым быстрым способом чтения данных из файла в память.

Файлы с произвольным доступом

Когда-то очень давно, когда дисковые устройства еще только впервые появились на компьютерном рынке, они совершили революцию в области запоминания данных на внешних носителях. В то время самым

распространенным средством хранения данных были магнитные ленты, которые читали и писали данные только последовательно. Новая технология добавила компьютерам роскоши, позволив запоминать информацию в произвольно заданных позициях, т.е. появилась возможность извлекать и вставлять записи в середину файла.

Но к файлам с произвольным доступом предъявляется одно обязательное требование: все записи должны иметь одну и ту же длину. За исключением этого момента между файлами с произвольным и последовательным доступом нет никакой физической разницы.

Чтобы продемонстрировать концепцию метода произвольного доступа, листинг 8.7 выполняет чтение одиннадцатого целого значения, записанного в файл int.dat программой wint.

Листинг 8.7. rintr.c (чтение 11-го значения в файле int.dat)

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3:
 4: main()
 5: {
 6:   FILE *fp;
 7:   int value;
 8:
 9:   fp = fopen("int.dat", "rb");
10:   if (!fp) {
11:     puts("Can't open int.dat");
12:     exit(1);
13:   }
14:   fseek(fp, 10 * sizeof(int), SEEK_SET);
15:   fread(&value, sizeof(int), 1, fp);
16:   printf("Record #10 = %d\n", value);
17:   fclose(fp);
18:   return 0;
19: }
```

Каждое значение в файле с произвольным доступом имеет соответствующий номер записи, аналогичный индексу в массиве. Первой записи в файле соответствует номер нуль, второй – номер один и т.д. Чтобы прочитать запись в произвольно выбранной позиции, вызовите функцию **fseek()** (строка 14), которая передвигает внутренний указатель файла на первый байт желаемой записи.

Функция **fseek()** требует три аргумента:

1. Переменную типа **FILE ***, указывающую на файл, открытый для двоичного доступа.
2. Значение смещения в байтах.
3. Одну из трех констант: **SEEK_SET**, **SEEK_CUR** или **SEEK_END**.

Если третий аргумент равен **SEEK_SET**, смещение равно числу байтов, на которые внутренний указатель файла передвинется вперед от начала файла. Если аргументом является **SEEK_CUR**, то смещение равно числу байтов, на которое внутренний указатель передвинется вперед или назад относительно его текущей позиции. Если аргумент равен **SEEK_END**, то смещение представляет число байтов для перемещения указателя с конца файла по направлению к началу.

В программе rintr оператор**:**

```
fseek(fp, 10 * sizeof(int), SEEK_SET);
```

производит смещение на десять записей от начала файла, т.е. устанавливает внутренний указатель на первый байт одиннадцатого значения.

Функция **fseek()** возвращает нуль при успешном ее выполнении. Однако поиск несуществующей области может и не сгенерировать ошибку. Всегда отслеживайте максимальное число записей в файле и примите меры для предотвращения поиска за пределами границ файла.

Несколько примеров помогут продемонстрировать возможности функции **fseek()**. Если переменная **fp** типа **FILE *** указывает на открытый файл записей типа **T**, то оператор

```
fseek(fp, sizeof(T), SEEK_CUR);
```

переместит внутренний указатель файла с текущей записи на следующую. Оператор

```
fseek(f, -sizeof(T), SEEK_CUR);
```

переместит указатель файла на одну запись назад. Вызов функции **fread()** перемещает указатель файла вперед, поэтому после чтения одной записи с помощью функции **fread()** можно использовать предыдущий оператор для восстановления позиции файлового указателя, например, для того, чтобы перезаписать новую информацию в то же место.

Другой удобный оператор находит конец файла:

```
fseek(f, 0, SEEK_END);
```

Этот оператор вы можете использовать перед вызовом функции **fwrite()**, чтобы добавить новые записи в конец файла.

Листинг 8.8 показывает, как позиционировать указатель файла, чтобы записать одно значение в файл int.dat, не затрагивая значений, стоящих до и после заданного. Запустите эту программу, а затем еще раз выполните программы rint и rintr. Как видите, операторы программы находят одиннадцатую запись и меняют ее значение с 10 на 99. Другие значения не затрагиваются.

Листинг 8.8.wintr.c (запись значения в произвольную позицию файла int.dat)

```
1: #include <stdio.h>
2: #include <stdlib.h>
3:
4: main()
```

```
 5: {
 6:   FILE *fp;
 7:   int value = 99; /* новое значение записи */
 8:
 9:   fp = fopen("int.dat", "r+b");
10:   if (!fp) {
11:     puts("Can't open int.dat");
12:     exit(1);
13:   }
14:   printf("Writing %d to record #10\n", value);
15:   fseek(fp, 10 * sizeof(int), SEEK_SET);
16:   fwrite(&value, sizeof(int), 1, fp);
17:   fclose(fp);
18:   return 0;
19: }
```

Обратите внимание, что строка 9 открывает файл int.dat для чтения и записи в двоичном режиме – возможно, это самый распространенный способ открытия файла для обработки с произвольным доступом. Строка 15 вызывает функцию **fseek()** для позиционирования внутреннего указателя файла на одиннадцатое значение (запись с номером 10). Строка 16 вызывает функцию **fwrite()**, которая модифицирует указанное значение.

Программирование баз данных

Вы уже познакомились со всеми основными способами чтения и записи текстовых и двоичных данных в файлы. Вооруженные этими знаниями, вы можете приступить к написанию программ обработки данных, которые могут запоминать, искать, сортировать и восстанавливать записи в дисковых файлах. Рамки этой главы (и даже всего пособия) слишком ограничены, чтобы представить полную систему управления базой данных. Но следующие несколько программ послужат толчком к созданию ваших собственных законченных проектов.

Базу данных удобно реализовать на С как файл структур. Каждая запись является структурой, члены которой содержат необходимую информацию. Все, что вы можете запомнить в структуре, можно записать и прочитать из двоичного файла с произвольным доступом.

Проектирование баз данных

Первым шагом в написании программы базы данных является разработка структуры данных. Вам также понадобится несколько функций для создания нового файла, чтения и запоминания записей, а также для ввода данных. Листинг 8.9 содержит необходимые объявления и прототипы функций, которые будут использованы другими программами этого раздела.

Листинг 8.9. db.h (заголовочный файл для базы данных)

```
 1: #include <stdio.h>
 2: #include <limits.h>
 3:
 4: #define FALSE 0
 5: #define TRUE 1
 6: #define NAMELEN 30
 7: #define ADDRLEN 50
 8: #define PHONELEN 13
 9:
10: typedef struct record {
11:   char name[NAMELEN];   /* имя клиента */
12:   char addr[ADDRLEN];   /* адрес клиента */
13:   char phone[PHONELEN]; /* телефон клиента */
14:   double balance;       /* текущий остаток на счете */
15:   long custnum;         /* номер клиента / число записей */
16: } Record;
17:
18: #define MAXREC (LONG_MAX / sizeof(Record))
19:
20: /* Создает новую базу данных. 0 = успех, -1 = неудача */
21: int CreateDB(const char *path);
22:
23: /* Открывает базу данных. Возвращает FILE * или NULL */
24: FILE *OpenDB(const char *path, Record *header);
25:
26: /*Читает запись с номером recnum в rec. Возвращает TRUE/FALSE*/
27: int ReadRecord(FILE *fp, long recnum, Record *rec);
28:
29: /* Записывает запись с номером recnum из rec. */
30: /* Возвращает TRUE/FALSE */
31: int WriteRecord(FILE *fp, long recnum, Record *rec);
32:
33: /* Добавляет запись в конец файла. Возвращает TRUE/FALSE */
34: int AppendRecord(FILE *fp, long *recnum, Record *rec);
35:
36: /*Выводит приглашение (label) и вводит значение типа long */
37: void InputLong(const char *label, long *lp);
38:
39: /* Выводит приглашение и вводит строку */
40: void InputChar(const char *label, char *cp, int len);
41:
42: /* Выводит приглашение и вводит значение типа double */
43: void InputDouble(const char *label, double *dp);
```

Строки 6-8 объявляют символические константы, которые используются в строках 11-13 для объявления символьных массивов. Не используйте поля типа **char *** в структурах – они только усложняют работу и требуют вызова функции **malloc()** для размещения каждого поля в памяти.

Лучше выделить память в куче для всей структуры и запомнить символьные поля там.

Обратите внимание на член структуры **custnum**. Он имеет двоякое назначение. Первая запись запоминает в этой переменной количество записей в файле базы данных. В остальных записях **custnum** используется «по назначению» – хранит номер клиента. Другими словами, первая запись в базе данных играет роль информационного заголовка, другие переменные кроме **custnum** в ней не используются; записи с номерами 1 и выше содержат значащую информацию – имя клиента, его адрес и другие данные. На рис. 8.2 показано, как организован такой файл.

Листинг 8.10 содержит описание функций, прототипы которых объявлены в файле db.h.

Листинг 8.10. db.c (реализация распространенных функций баз данных)

```
 1: #include <stdlib.h>
 2: #include <string.h>
 3: #include "db.h"
 4:
 5: int CreateDB(const char *path)
 6: {
 7:   FILE *fp;
 8:   Record rec;
 9:   int result;
10:
11:   fp = fopen(path, "wb");
12:   if (!fp)
13:     return FALSE;
14:   memset(&rec, 0, sizeof(Record));
15:   rec.custnum = 0;
16:   result = WriteRecord(fp, 0, &rec);
17:   fclose(fp);
18:   return result;
19: }
20:
21: FILE *OpenDB(const char *path, Record *header)
22: {
23:   FILE *fp = fopen(path, "r+b");
24:   if (fp)
25:     ReadRecord(fp, 0, header);
26:   return fp;
27: }
28:
29: int ReadRecord(FILE *fp, long recnum, Record *rec)
30: {
31:   if (recnum > MAXREC)
32:     return FALSE;
33:   if (fseek(fp, recnum * sizeof (Record), SEEK_SET) != 0)
34:     return FALSE;
```

```
35:   return (fread(rec, sizeof(Record), 1, fp) == 1);
36: }
37:
38: int WriteRecord(FILE *fp, long recnum, Record *rec)
39: {
40:   if (recnum > MAXREC)
41:     return FALSE;
42:   if (fseek(fp, recnum * sizeof(Record), SEEK_SET) != 0)
43:     return FALSE;
44:   return (fwrite(rec, sizeof(Record), 1, fp) == 1);
45: }
46:
47: int AppendRecord(FILE *fp, long *recnum, Record *rec)
48: {
49:   if (fseek(fp, 0, SEEK_END) != 0)
50:     return FALSE;
51:   *recnum = ftell(fp) / sizeof(Record);
52:   return WriteRecord(fp, *recnum, rec);
53: }
54:
55: void InputLong(const char *label, long *lp)
56: {
57:   char buffer[128];
58:
59:   printf(label);
60:   gets(buffer);
61:   *lp = atol(buffer);
62: }
63:
64: void InputChar(const char *label, char *cp, int len)
65: {
66:   char buffer[128];
67:
68:   printf(label);
69:   gets(buffer);
70:   strncpy(cp, buffer, len - 1);
71: }
72:
73: void InputDouble(const char *label, double *dp)
74: {
75:   char buffer[128];
76:
77:   printf(label);
78:   gets(buffer);
79:   *dp = atof(buffer);
80: }
```

Скорее всего, вам будет понятна большая часть модуля db.c. Функция **CreateDB()** (строки 5-19) вызывает функцию **fopen()** с опцией “**wb**”, чтобы создать новый файл базы данных. Вызов функции **memset()** с нулем в качестве второго параметра инициализирует каждый байт структуры **rec** нулевыми значениями. Поле **custnum** устанавливается равным нулю явным образом (строка 15), чтобы подчеркнуть, что на этой стадии база данных еще не содержит никаких записей.

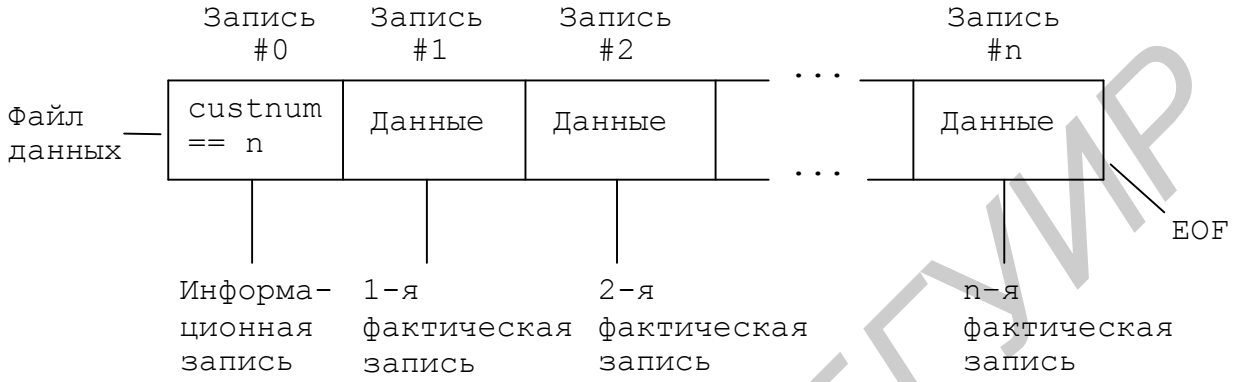

Рис 8.2. Пример организации файла базы данных

Функция **OpenDB()** (строки 21-27), которая возвращает значение типа **FILE ***, вызывает функцию **fopen()**, чтобы открыть существующий файл для чтения или записи. Функция также выполняет чтение информационной записи файла, позволяя программе установить количество содержащихся в файле записей.

Функция **ReadRecord()** (строки 29-36) выполняет чтение записи с номером **recnum** из файла **fp** в структуру, адресуемую указателем **rec**. Обратите внимание, как используется функция **fseek()** для перемещения внутреннего файлового указателя в позицию, начиная с которой функция **fread()** загружает байты записи с диска.

Функция **WriteRecord()** (строки 38-45) имеет те же параметры, что и функция **ReadRecord()**, но она записывает на диск структуру, адресуемую указателем **rec**. И снова функция **fseek()** позиционирует внутренний указатель файла. Затем функция **fwrite()** сохраняет подготовленную запись в этой позиции.

Функция **AppendRecord()** (строки 47-53) вызывает функцию **fseek()**, использующую опцию **SEEK_END**, для позиционирования внутреннего указателя как раз за концом файла. Строка 51 вызывает функцию **ftell()**, которая возвращает внутренний указатель файла. Деленное на размер одной записи, это значение равно номеру добавляемой записи. Строка 52 вызывает функцию **WriteRecord()**, чтобы присоединить новую запись к концу файла.

Функции **InputLong()**, **InputChar()** и **InputDouble()** (строки 55-80) являются простыми средствами для получения данных с клавиатуры. Эти функции упрощены до предела, чтобы не удлинять листинги в этом разделе.

Более сложные программы ведения баз данных содержат широкие возможности редактирования и проверки вводимых данных.

Создание файла базы данных

После разработки структуры базы данных и функций поддержки было бы неплохо научиться создавать файл базы данных (пусть даже и пустой вначале). Листинг 8.10 иллюстрирует один из способов решения этой задачи.

Чтобы создать законченную программу, вы должны скомпилировать модули makedb.c и db.c и скомпоновать их вместе. После того как программа начнет работать, введите имя создаваемого файла базы данных. Если такой файл уже существует, программа завершится, выдав сообщение об ошибке. Удалите этот файл и сделайте новую попытку либо введите другое имя файла.

Замечание. *Следующие три листинга – addrec.c, editrec.c и report.c – скомпилируйте аналогичным образом, как и makedb.c.*

Листинг 8.11. makedb.c (создание пустого файла базы данных)

```
 1: #include <stdlib.h>
 2: #include <string.h>
 3: #include "db.h"
 4:
 5: int FileExists(const char *path);
 6:
 7: main()
 8: {
 9:   char path[128];
10:
11:   puts("Create new database file");
12:   printf("File name? ");
13:   gets(path);
14:   if (strlen(path) == 0)
15:     exit(0);
16:   if (FileExists(path)) {
17:     printf("%s already exists.\n", path);
18:     puts("Delete file and try again.");
19:     exit(1);
20:   }
21:   if (!CreateDB(path))
22:     perror(path);
23:   else
24:     printf("%s created.\n", path);
25:   return 0;
26: }
27:
28: int FileExists(const char *path)
29: {
30:   FILE *fp = fopen(path, "r");
31:
```

```
32:   if (!fp)
33:     return FALSE;
34:   else {
35:     fclose(fp);
36:     return TRUE;
37:   }
38: }
```

В строке 21 вызывается функция **CreateDB()** модуля db.c, чтобы создать пустой файл базы данных, используя заданное вами имя. Если этот файл существует, программа откажется перезаписывать его, выдав сообщение об ошибке.

Строки 28-38 содержат удобную функцию **FileExists()**, которая возвращает «истину», если заданный файл, определяемый строкой **path**, существует. Вы можете скопировать ее в свою собственную библиотеку функций и использовать ее, чтобы узнать, существуют ли заданные файлы.

Добавление записей в базы данных

Вашей системе ведения базы данных необходима программа, которая могла бы добавлять новые записи. Листинг 8.12 демонстрирует одно из возможных решений. Скомпилируйте и запустите программу addrec, а затем введите запрашиваемые данные, чтобы добавить новую запись в конец файла.

Листинг 8.12. addrec.c (добавление записи в файл базы данных)

```
 1: #include <stdlib.n>
 2: #include <string.n>
 3: #include <mem.h>
 4: #include "db.h"
 5:
 6: void GetNewRecord(Record *rec);
 7:
 8: main()
 9: {
10:   char path[128];
11:   FILE *dbf;
12:   Record rec;
13:   long numrecs;
14:
15:   printf("Database file name? ");
16:   gets(path);
17:   dbf = OpenDB(path, &rec);
18:   if (!dbf) {
19:     printf("Can't open %s\n", path);
20:     exit(1);
21:   }
22:   numrecs = rec.custnum;
23:   printf("Number of records = %ld\n", numrecs);
```

```
24:   GetNewRecord(&rec);
25:   if (AppendRecord(dbf, &numrecs, &rec))
26:     printf("Record #%ld added to database\n", numrecs);
27:   memset(&rec, 0, sizeof(Record)); /* обнулить запись */
28:   rec.custnum = numrecs; /* обновление счетчика записей */
29:   WriteRecord(dbf, 0, &rec); /* запись заголовка */
30:   fclose(dbf);
31:   return 0;
32: }
33:
34: void GetNewRecord(Record *rec)
35: {
36:   memset(rec, 0, sizeof(Record));
37:   InputLong("Customer # : ", &rec->custnum);
38:   InputChar("Name : ", rec->name, NAMELEN);
39:   InputChar("Address : ", rec->addr, ADDRLEN);
40:   InputChar("Telephone : ", rec->phone, PHONELEN);
41:   InputDouble("Account balance : ", &rec->balance);
42: }
```

Упражнение. *Очевидно, что программа addrec была бы намного полезней, если бы могла добавлять больше, чем одну запись. Добавьте в программу цикл, и после ввода каждой записи спрашивайте пользователя, не хочет ли он добавить еще одну.*

Редактирование записей базы данных

Кроме ввода новых записей вам, вероятно, понадобится редактировать уже существующие. Используя методы произвольного доступа, описанные выше, листинг 8.13 предлагает ввести номер записи и позволяет обновить информацию, которую она содержит.

Листинг 8.13. editrec.c (редактирование записи в файле базы данных)

```
 1: #include <stdlib.h>
 2: #include "db.h"
 3:
 4: void EditRecord(FILE *f, long recnum, Record *rec);
 5:
 6: main()
 7: {
 8:   char path[128];
 9:   FILE *dbf;
10:   Record rec;
11:   long numrecs, recnum;
12:
13:   printf("Database file name? ");
14:   gets(path);
15:   dbf = OpenDB(path, &rec);
16:   if (!dbf) {
17:     printf("Can't open %s\n", path);
```

```
18:     exit(1);
19:   }
20:   numrecs = rec.custnum;
21:   printf("Number of records = %ld\n", numrecs);
22:   InputLong("Record number? ", &recnum);
23:   if ((recnum <= 0) || (recnum > numrecs)) {
24:     puts("Record number out of range");
25:     fclose(dbf);
26:     exit(1);
27:   }
28:   EditRecord(dbf, recnum, &rec);
29:   if (WriteRecord(dbf, recnum, &rec))
30:     printf("Record #%ld written to %s\n", recnum, path);
31:   else
32:     perror("Write record");
33:   fclose(dbf);
34:   return 0;
35: }
36:
37: void EditRecord(FILE *fp, long recnum, Record *rec)
38: {
39:   if (!ReadRecord(fp, recnum, rec)) {
40:     perror("Reading record");
41:     exit(1);
42:   }
43:   printf("Customer # : %ld\n", rec->custnum);
44:   InputLong("Customer # : ", &rec->custnum);
45:   printf("Name : %s\n", rec->name);
46:   InputChar("Name : ", rec->name, NAMELEN);
47:   printf("Address : %s\n", rec->addr);
48:   InputChar("Address : ", rec->addr, ADDRLEN);
49:   printf("Telephone : %s\n", rec->phone);
50:   InputChar("Telephone : ", rec->phone, PHONELEN);
51:   printf("Account balance : %.2lf\n", rec->balance);
52:   InputDouble("Account balance : ", &rec->balance);
53: }
```

Создание отчетов о содержимом базы данных

В качестве последней демонстрации различных видов программ, в которых нуждается система ведения баз данных, листинг 8.14 создает отчет о записях, содержащихся в базе данных. Скомпилируйте и запустите программу, затем введите имя файла базы данных. Программа отобразит содержимое полей всех записей базы данных.

Листинг 8.14. report.c (отчет по файлу базы данных)

```
1: #include <stdlib.h>
2: #include "db.h"
3:
4: main()
```

```
 5: {
 6:   char path[128];
 7:   FILE *dbf;
 8:   Record rec;
 9:   long numrecs, recnum;
10:
11:   printf("Database file name? ");
12:   gets(path);
13:   dbf = OpenDB(path, &rec);
14:   if (!dbf) {
15:     printf("Can't open %s\n", path);
16:     exit(1);
17:   }
18:   numrecs = rec.custnum;
19:   printf("\nNumber of records = %ld\n\n", numrecs);
20:   for (recnum = 1; recnum <= numrecs; recnum++)
21:     if (ReadRecord(dbf, recnum, &rec))
22:       printf("%4ld: #%ld, %s, Phone: %s, Balance: $%.2lf\n",
23:         recnum, rec.custnum, rec.name,rec.phone,rec.balance);
24:   fclose(dbf);
25:   return 0;
26: }
```

Упражнение. *На основе листингов этого раздела разработайте программу ведения базы данных. Реализуйте меню, которое предлагало бы пользователю следующие действия:*

- *создать новую базу данных;*
- *открыть существующую базу данных;*
- *добавить/удалить/изменить запись в открытой базе данных;*
- *создать отчет обо всех клиентах в базе данных, остатки счетов которых (поле **balance**) больше заданного значения;*
- *узнать сумму всех остатков счетов клиентов.*

Резюме

- Язык С рассматривает любой файл как последовательность байтов, завершающуюся признаком конца файла (**EOF**).
- Вне зависимости от режима работы обработка файлов происходит в три этапа: 1) открыть файл, 2) обработать данные и 3) закрыть файл.
- **FILE** является структурой, тип которой определен в заголовочном файле stdio.h. Для того чтобы работать с файлами, нет необходимости знать, как устроена эта структура. При открытии файла функция **fopen()** возвращает указатель на файл типа **FILE ***.
- Чтобы создать новый текстовый файл или уничтожить содержимое уже созданного файла, откройте файл для записи (“**w**”). Чтобы прочитать существующий файл, откройте файл в режиме «только для чтения» (“**r**”).

Чтобы добавить записи в конец существующего файла, откройте файл для добавления (“**a**”). Чтобы открыть файл для обновления (т.е. для одновременного выполнения операций чтения и записи), используйте один из режимов “**r+**”, “**w+**” или “**a+**”.

- В зависимости от потребностей вашей программы вы можете обрабатывать текстовые файлы в посимвольном режиме, в котором проверяется каждый символ файла, либо в более быстром построчном режиме.
- Двоичные файлы в большинстве случаев по размеру меньше текстовых, и программы обрабатывают их быстрее. Чтобы создать или обработать двоичный файл, необходимо добавить символ ‘**b**’ в строку, задающую режим открытия файла. Например: открыть двоичный файл для записи и чтения: “**r+b**”.
- Двоичные файлы могут обрабатываться с последовательным или произвольным доступом. Последовательная обработка применяется, когда со всеми записями файла производятся одинаковые действия. Произвольный доступ используется, чтобы прочитать или модифицировать отдельную запись в файле, не затрагивая остальные записи. Все записи в файле с произвольным доступом должны иметь одинаковую длину.
- База данных представляет собой группу связанных между собой файлов (либо, в простейшем случае, только один файл). Набор программ, разработанных для создания и управления базами данных, называется СУБД – Системой управления базами данных (DBMS – Data Base Management System).

Обзор функций

Таблица 8.2

Функции для работы с файлами (stdio.h)

Функция	**Прототип и краткое описание**
1	2
fclose()	`int fclose(FILE *fp);` Закрывает указанный файл. В случае успешного закрытия файла возвращает 0, в противном случае – **EOF**.
feof()	`int feof(FILE *fp);` Проверяет, достигнут ли конец файла. Возвращает ненулевое значение, если конец файла достигнут, в противном случае – 0 .

Продолжение табл. 8.2

1	2
fclose()	`int fclose(FILE *fp);`

Закрывает указанный файл. В случае успешного закрытия файла возвращает 0, в противном случае – **EOF**.

feof() `int feof(FILE *fp);`
Проверяет, достигнут ли конец файла. Возвращает ненулевое значение, если конец файла достигнут, в противном случае возвращает 0.

fgetc() `int fgetc(FILE *fp);`
Считывает символ из указанного файла. В случае успеха функция возвращает полученный символ, в противном случае возвращается значение **EOF**.

fgets() `char *fgets(char *str, int n, FILE *fp);`
Считывает не более **n – 1** символов из указанного файла в массив, адресуемый **str**. За последним символом, помещенным в массив, записывается ‘**\0**’. В случае успеха функция возвращает **str,** в противном случае – **NULL**.

fopen() `FILE *fopen(const char *fname, const char *mode);`
Открывает файл с именем **fname** в режиме, задаваемом строкой **mode**. Если файл успешно открыт, функция возвращает указатель на файл, иначе возвращается **NULL**.

fprintf() `int fprintf(FILE *pf, const char *format, ...);`
Записывает форматированный вывод в указанный файл. В случае успешного завершения функция возвращает количество записанных байтов информации; в противном случае возвращается значение **EOF**.

fputc() `int fputs(int c, FILE *fp);`
Записывает символ **c** в указанный файл. В случае успешного завершения функция возвращает записанный символ, в противном случае – значение **EOF**.

fputs() `int fputs(const char *str, FILE *fp);`
Записывает символьную строку, адресуемую **str**, в указанный файл. Символ конца строки ‘**\0**’ не записывается. В случае успешного завершения функция возвращает последний записанный символ, в противном случае – значение **EOF**.

Продолжение табл. 8.2

1	2
fread()	`size_t fread(void *p, size_t size, size_t n, FILE *fp);`

Считывает двоичные данные из файла в блок памяти, адресуемый **p**. Параметр **n** определяет количество считываемых элементов, параметр **size** – размер элементов. Общее количество считанных байтов равно **n * size**. В случае успешного завершения функция возвращает количество считанных элементов, в противном случае – 0.

fscanf() `int fscanf(FILE *fp, const char *format, ...);`
Функция форматированного ввода из указанного файла (см. функцию **scanf()**).

fseek() `int fseek(FILE *fp, long offset, int whence);`
Перемещает внутренний указатель файла, на который указывает **fp**, на **offset** байтов относительно точки отсчета, определенной значением **whence** (**SEEK_SET** – от начала файла, **SEEK_CUR** – от текущей позиции, **SEEK_END** – от конца файла). В случае успешного перемещения указателя, функция возвращает 0**,** в противном случае – ненулевое значение.

ftell() `long ftell(FILE *fp);`
Возвращает значение указателя текущей позиции в файле, на который указывает **fp**. В случае ошибки возвращает –1.

fwrite() `size_t fwrite(const void *ptr, size_t size, size_t n, FILE *fp);`
Записывает двоичные данные из блока, адресуемого **ptr**, в указанный файл. Параметр **n** определяет количество считываемых элементов, параметр **size** – размер элементов. Общее количество записанных байтов равно **n * size.**

rename() `int rename(const char *oldname, const char *newname);`
Переименовывает существующий файл. Директории, задаваемые в **oldname** и **newname** не обязательно должны совпадать, поэтому функцию можно использовать для перемещения файла из одной директории в другую. В случае успешного переименования возвращает 0; в противном случае возвращает –1.

Окончание табл. 8.2

1	2
unlink()	`int unlink(const char *fname);`

Удаляет файл с именем **fname**. Строка **fname** должна содержать полный маршрут к файлу. Перед удалением убедитесь, что указанный файл закрыт. При успешном удалении функция возвращает 0; в случае ошибки возвращает значение –1.

Таблица 8.3

Обработка ошибок (stdio.h)

Функция	**Прототип и краткое описание**
perror()	`void perror(const char *str);` Выводит сообщение о системных ошибках.

ПРИЛОЖЕНИЕ 1

Таблицы кодов ASCII *

Таблица П1.1

Коды управляющих символов (0–31)

Код	Обозначение	Клавиша	Значение	Отображаемый символ
1	2	3	4	5
0	nul	^@	Нуль	
1	soh	^A	Начало заголовка	☺
2	stx	^B	Начало текста	☻
3	etx	^C	Конец текста	♥
4	eot	^D	Конец передачи	♦
5	enq	^E	Запрос	♣
6	ack	^F	Подтверждение	♠
7	bel	^G	Сигнал (звонок)	•
8	bs	^H	Забой (шаг назад)	◘
9	ht	^I	Горизонтальная табуляция	○
10	lf	^J	Перевод строки	◙
11	vt	^K	Вертикальная табуляция	♂
12	ff	^L	Новая страница	♀
13	cr	^M	Возврат каретки	♪
14	so	^N	Выключить сдвиг	♫
15	si	^O	Включить сдвиг	☼
16	dle	^P	Ключ связи данных	►
17	dc1	^Q	Управление устройством 1	◄
18	dc2	^R	Управление устройством 2	↕
19	dc3	^S	Управление устройством 3	‼
20	dc4	^T	Управление устройством 4	¶
21	nak	^U	Отрицательное подтверждение	§
22	syn	^V	Синхронизация	▬
23	etb	^W	Конец передаваемого блока	↨
24	can	^X	Отказ	↑
25	em	^Y	Конец среды	↓
26	sub	^Z	Замена	→
27	esc	^[	Ключ	←
28	fs	^\	Разделитель файлов	∟

Окончание табл. П1.1

1	2	3	4	5
29	gs	^]	Разделитель группы	↔
30	rs	^^	Разделитель записей	▲
31	us	^_	Разделитель модулей	▼

Примечание. *В графе «Клавиши» обозначение ^ соответствует нажатию клавиши <Ctrl>, вместе с которой нажимается соответствующая «буквенная» клавиша, формируя код символа.*

Таблица П1.2

Символы с кодами 32–127

Код	Символ	Код	Символ	Код	Символ	Код	Символ
32	пробел	56	8	80	P	104	H
33	!	57	9	81	Q	105	I
34	"	58	:	82	R	106	J
35	#	59	;	83	S	107	K
36	$	60	<	84	T	108	L
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	A	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	del

Таблица П1.3

Символы с кодами 128–255 (Кодовая таблица 866 – MS-DOS)

Код	Символ	Код	Символ	Код	Символ	Код	Символ
128	А	160	а	192	└	224	р
129	Б	161	б	193	┴	225	с
130	В	162	в	194	┬	226	т
131	Г	163	г	195	├	227	у
132	Д	164	д	196	─	228	ф
133	Е	165	е	197	┼	229	х
134	Ж	166	ж	198	╞	230	ц
135	З	167	з	199	╟	231	ч
136	И	168	и	200	╚	232	ш
137	Й	169	й	201	╔	233	щ
138	К	170	к	202	╩	234	ъ
139	Л	171	л	203	╦	235	ы
140	М	172	м	204	╠	236	ь
141	Н	173	н	205	═	237	э
142	О	174	о	206	╬	238	ю
143	П	175	п	207	╧	239	я
144	Р	176	▒	208	╨	240	Ё
145	С	177	▓	209	╤	241	ё
146	Т	178	│	210	╥	242	Є
147	У	179	┤	211	╙	243	є
148	Ф	180	╡	212	╘	244	Ї
149	Х	181	╢	213	╒	245	ї
150	Ц	182	╖	214	╓	246	Ў
151	Ч	183	╕	215	╫	247	ў
152	Ш	184	╣	216	╪	248	°
153	Щ	185	║	217	┘	249	∙
154	Ъ	186	╗	218	┌	250	·
155	Ы	187	╝	219	█	251	√
156	Ь	188	╜	220	▄	252	№
157	Э	189	╛	221	▌	253	¤
158	Ю	190	┐	222	▐	254	■
159	Я	191	┐	223	▀	255	

Таблица П1.4

Символы с кодами 128–255 (Кодовая таблица 1251 – MS Windows)

Код	Символ	Код	Символ	Код	Символ	Код	Символ
128	Ђ	160		192	А	224	а
129	Ѓ	161	Ў	193	Б	225	б
130	‚	162	ў	194	В	226	в
131	ѓ	163	Ј	195	Г	227	г
132	„	164	¤	196	Д	228	д
133	…	165	Ґ	197	Е	229	е
134	†	166	¦	198	Ж	230	ж
135	‡	167	§	199	З	231	з
136	€	168	Ё	200	И	232	и
137	‰	169	©	201	Й	233	й
138	Љ	170	Є	202	К	234	к
139	‹	171	«	203	Л	235	л
140	Њ	172	¬	204	М	236	м
141	Ќ	173	-	205	Н	237	н
142	Ћ	174	®	206	О	238	о
143	Џ	175	Ї	207	П	239	п
144	ђ	176	°	208	Р	240	р
145	‘	177	±	209	С	241	с
146	’	178	І	210	Т	242	т
147	“	179	і	211	У	243	у
148	”	180	ґ	212	Ф	244	ф
149	•	181	µ	213	Х	245	х
150	–	182	¶	214	Ц	246	ц
151	—	183	·	215	Ч	247	ч
152		184	ё	216	Ш	248	ш
153	™	185	№	217	Щ	249	щ
154	љ	186	є	218	Ъ	250	ъ
155	›	187	»	219	Ы	251	ы
156	њ	188	ј	220	Ь	252	ь
157	ќ	189	Ѕ	221	Э	253	э
158	ћ	190	ѕ	222	Ю	254	ю
159	џ	191	ї	223	Я	255	я

***** ASCII (American Standard Code for Information Interchange – Стандартный американский код обмена информацией) – это код для представления

символов в виде чисел, в котором каждому символу сопоставлено число от 0 до 127. В большинстве компьютеров код ASCII используется для представления текста, что позволяет передавать данные от одного компьютера на другой.

Стандартный набор символов ASCII использует только 7 битов для каждого символа. Добавление 8-го разряда позволяет увеличить количество кодов таблицы ASCII до 255. Коды от 128 до 255 представляют собой расширение таблицы ASCII. Эти коды используются для кодирования символов национальных алфавитов, а также символов псевдографики, которые можно использовать, например, для оформления в тексте различных рамок и текстовых таблиц.

ПРИЛОЖЕНИЕ 2

Хороший стиль программирования

```
/* Пример кода, проявляющего характеристики стиля */
#include <stdio.h>
#define MAX 16000

int Sum(int n)
{
  int i, total = 0;
  for (i = 0; i < n; ++i) {
    total += i;
    if (total > MAX)
      printf("too large\n");
      exit(1);
    }
  }
  return total;
}
```

Вот некоторые негласные требования хорошего стиля программирования:

1. Один оператор на строку.
2. Пустая строка после начальных объявлений.
3. Фигурная скобка составного оператора ставится в той же самой строке, что и его управляющее выражение. Завершающая фигурная скобка выровнена по линии начального символа ключевого слова, начинающего оператор.
4. Все, что встречается после открытия (левой) фигурной скобки выравнивается на стандартное число пробелов; например, в этом примере и листингах пособия используются два пробела. Соответствующая закрывающая (правая) фигурная скобка ставится так, что все последующие операторы выравниваются по ней.
5. Для удобочитаемости пробел добавляется после каждой лексемы, за исключением точки с запятой.
6. Идентификаторы препроцессора печатаются заглавными буквами. Обычные идентификаторы – строчными.
7. Команды препроцессора и объявления на уровне файла записываются без отступов слева.

Часто встречающиеся разновидности этого стиля включают:

1. Выравнивание фигурных скобок всегда с новой строки.
2. Очень короткие операторы, которые концептуально связаны, могут находиться в одной и той же строке.

```
/* Модификации стиля */
int order(int& y, int& z)
{
  int temp;

  if (z < y)
  {
    temp = z; z = y; y = temp;
  }
  return z;
}
```

ПРИЛОЖЕНИЕ 3

Рекомендуемая литература

Керниган Б.В., Ритчи Ф.М. Язык программирования Си. 2-е изд. перераб. и доп. – М.: Финансы и статистика, 1992. – 272 с.

ISBN 5-279-00473-1 УДК 681.3.06.

Оригинал: Brian W.Kernighan, Dennis M.Ritchie. The C Programming Language AT&T Bell Lab. Murray Hill, New Jersey 1978, 1988.

ISBN 0-13-110370-9.

Дейтел П., Дейтел Х. Как программировать на С. 3-е изд. – М.: Бином, 2002. – 1168 с.

ISBN 5-9518-0002-1.

Прата С. Язык программирования С. Лекции и упражнения. Учебник – СПб.: ООО «ДиаСофтЮП», 2002. – 896 с.

ISBN 5-93772-049-0 УДК 681.3.06(075).

Скляров В.А. Программное и лингвистическое обеспечение персональных ЭВМ. Системы общего назначения: Справ. пособие. – Мн.: Выш. шк., 1992. – 462 с.

ISBN 5-339-00630-1.

Касаткин А.И. Профессиональное программирование на языке Си. Управление ресурсами: Справ. пособие. – Мн.: Выш. шк., 1992. – 432 с.

ISBN 5-339-06808-8 УДК 681.3.06(035.5).

Юлин В.А., Булатова И.Р. Приглашение к Си. – Мн.: Выш.шк., 1990. – 224 с. ISBN 5-339-00353-1 УДК 681.3.06:800.92.

Б. Керниган, Р. Пайк. Практика программирования. СПб.: «Невский Диалект», 2001г. – 384 с.

ISBN 5-7940-0058-9, 0-201-61586-X.

Св. план 2004, поз.56

Учебное издание

Мелещенко Александр Александрович

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С

Учебное пособие
для студентов специальности 31 03 04 “Информатика”
дневной формы обучения
по курсу «Конструирование программ и языки программирования»

Редактор Е.Н. Батурчик
Компьютерная верстка

Подписано в печать		Формат 60x84 1/16
Бумага офсетная.	Печать ризографическая	Усл.-печ.л.
Уч.-изд.л. 11,0	Тираж 150 экз.	Заказ

Издатель и полиграфическое исполнение:
Учреждение образования
"Белорусский государственный университет информатики и радиоэлектроники"
Лицензия ЛП № 156 от 05.02.2001.
Лицензия ЛВ № 509 от 03.08.2001.
220013, Минск, П.Бровки, 6.