Виталий Потопахин

Язык C

ОСВОЙ НА ПРИМЕРАХ

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.068+800.92C
ББК 32.973.26-018.1
П64

Потопахин В. В.

П64 Язык С. Освой на примерах. — СПб.: БХВ-Петербург, 2006. — 320 с.: ил.

ISBN 5-94157-950-0

Книга содержит детальное описание языка С, сопровождаемое большим количеством законченных примеров. Рассмотрены указатели и представление структур данных с использованием механизма ссылок. Показано, как с помощью указателей в С создаются строки и такие конструкции данных, как связанные списки и деревья. Рассмотрены работа с файлами, операции ввода-вывода, графические возможности языка и многое другое. В приложении приведены примеры решения задач различной степени сложности.

Для начинающих программистов

УДК 681.3.068+800.92C
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.07.06.
Формат 60×90 1/16. Печать офсетная. Усл. печ. л. 20.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-950-0

© Потопахин В. В., 2006
© Оформление, издательство "БХВ-Петербург", 2006

Оглавление

Введение **1**

О технологии чтения 6

Глава 1. Кратко о языке C **9**

Простые примеры 10
Условные циклы 19
Указатели 21
Заключение 23

Глава 2. Величины **25**

Типы данных 25
Расположение объявлений 29
Инициализация переменных 30
Более подробно о времени жизни и области видимости 36
Классы памяти 39
  Объявления внутреннего уровня 40
  Объявления внешнего уровня 42
О преобразованиях типов 43
Объявления вида *typedef* 47
Заключение 48

Глава 3. Ввод/вывод **51**

Форматный ввод/вывод 51
Потоковый ввод/вывод 54
  Интересный пример вывода 55

Интересный пример ввода ... 55
Несколько полезных функций ... 56
Функции *putchar()* и *getchar()* ... 56
Функции *textcolor()* и *textbkcolor()* ... 57
Функция *gotoxy()* ... 58
Команда перевода курсора на следующую строку ... 58

Глава 4. Операции в С ... 61

Список операций ... 61
Арифметические операции ... 64
Операция *%* ... 65
Операции сдвига ... 66
Битовая логика ... 68
Логические операции ... 69
Операция следования ... 70
Операции индексации и построения выражений ... 71
Операция определения размера объекта *sizeof* ... 72
Арифметическое условие ... 73
Старшинство и порядок вычисления ... 74
Еще несколько замечаний о порядке выполнения операций ... 75

Глава 5. Операторы языка С ... 77

Общие сведения ... 77
Циклы ... 77
Цикл с предусловием ... 80
Цикл с постусловием ... 81
О структуре цикла ... 83
Замечания о цикле с постусловием ... 84
Конструкции выбора ... 86
Условный оператор ... 86
Оператор-переключатель ... 89
Оператор *break* ... 91
Составной оператор ... 92
Оператор продолжения *continue* ... 93
Оператор-выражение ... 94
Оператор *goto* ... 95

Пустой оператор *;* 96
Оператор возврата *return* 96

Глава 6. Константы 99

Простые константы 99
Перечисления 104

Глава 7. Массивы 107

Общие правила работы с массивами 107
Инициализация массива символов 115
Заключение 116

Глава 8. Структуры и объединения 117

Общие сведения 117
Структуры 118
Замечание об инициализации 120
Объединения 121
Замечание об инициализации объединений 125
Поля и экономия памяти 126
Заключение 129

Глава 9. Указатели 131

Объявление и создание указателей 131
Доступ к элементам структуры 136
Выделение и освобождение памяти 142
Арифметические операции над указателями 143
Замечание о шаге указателя 148
Указатель в никуда 151
Заключение 151

Глава 10. Строки 153

Нультерминальные строки C 153
О вложении присваивания 161
Пример упорядочения строк 162

Полезные функции 163
Функция *strstr()* 164
Функции *strlow()* и *strupr()* 164
Функция *strset()* 165
Функция *strcmp()* 166
Заключение 166

Глава 11. Связные списки и деревья 167

Построение связного списка 167
Задача 1. Обход связного списка 168
Задача 2. Обход связного списка неизвестной длины 169
Задача 3. Обход связного списка в двух направлениях 171
Несколько замечаний о методах обхода 172
Задача 4. Вставка элемента в связный список 173
Задача 5. Вставка связного списка 175
Задача 6. Сортировка по возрастанию массива, организованного в виде связного списка 176
Деревья 178
Задача 7. Построение двоичного дерева 178
Задача 8. Обмен местами значения левой и правой ветви двоичного дерева 179
Заключение 182

Глава 12. Файлы данных 183

Неформатный ввод/вывод 183
Форматный файловый ввод/вывод 189
Полезные функции 191
Функция *fseek()* 191
Функция *fgetpos()* 192
Функция *rewind()* 192
Функция *creat()* 193
Заключение 194

Глава 13. Построение сложной программы 195

Общие сведения 195
Передача одномерного массива 198
Пример 1. Расчет максимального в массиве 198

Пример 2. Расчет максимального с передачей массива, как параметра ... 200
Пример 3. Расчет максимального с передачей массива, как указателя ... 201
Передача многомерных массивов ... 203
Передача параметров по ссылке ... 204
Параметр по умолчанию ... 205
О возвращаемых значениях ... 206
Перегрузка функций ... 209
Указатель на функцию ... 212
Комментарии ... 215
Заключение ... 216

Глава 14. Графика языка С ... 217

Общая информация о графических режимах ... 217
Некоторые функции ... 219
Функция *arc()* ... 219
Функция *bar()* ... 219
Функция *bar3d()* ... 220
Функция *circle()* ... 220
Функция *cleardevice()* ... 220
Функция *closegraph()* ... 220
Функция *drawpoly()* ... 221
Функция *ellipse()* ... 221
Функция *floodfill()* ... 222
Функция *getbkcolor()* ... 222
Функция *getcolor()* ... 222
Функции *getmaxx()* и *getmaxy()* ... 222
Функция *getpixel()* ... 222
Функции *getx()* и *gety()* ... 223
Функция *linerel()* ... 223
Функция *lineto()* ... 223
Функция *moverel()* ... 223
Функция *moveto()* ... 223
Функция *outtext()* ... 224
Функция *outtextxy()* ... 224
Функция *putpixel()* ... 224

Функция *rectangle()* 224
Функция *sector()* 225
Функция *setbkcolor()* 225
Функция *setcolor()* 225
Функция *setfillstyle()* 225

Глава 15. Директивы 227

Директива *include* 227
Директива *define* 229
Строкообразующий оператор # 231
Замечание о расположении макросов 232
Заключение 233

ПРИЛОЖЕНИЯ 235

Приложение 1. Терминология 237

Приложение 2. Примеры задач 245

Предметный указатель 299

Введение

Язык C, созданный Денисом Ритчи в начале 70-х годов прошлого века в Bell Laboratory американской корпорации AT&T, считается языком системного программирования и, как говорят, был создан для тех, кто хочет заниматься системным программированием, но не хочет изучать ассемблер. Хотя, безусловно, C удобен и при написании прикладных программ. Первое описание языка дано в книге Б. Кернигана и Д. Ритчи, которая была переведена на русский язык[1]. Долгое время это описание являлось стандартом, однако ряд моментов допускали неоднозначное толкование, которое породило множество трактовок языка. Для исправления этой ситуации при Американском национальном институте стандартов (ANSI) был образован комитет по стандартизации языка C. И в 1983 году утвержден стандарт, получивший название ANSI C.

Язык уже достаточно давно получил всеобщее признание и стал чуть ли не эталоном. Несколько позже с появлением объектно-ориентированного подхода был разработан C++ или C с классами. И за давностью использования постепенно начинает забываться, что язык C и язык C++ — не совсем одно и то же. Наверное, если взять 100 молодых программистов, владеющих C/C++, и спросить, на чем они пишут, то ответ, скорее всего, будет таков: C++, вне зависимости от того, как в действительности выглядят их программы. Между тем, язык C++ — это язык объектно-ориентированного программирования, а далеко не всегда необходима именно объектная программа, и далеко не всегда программы

[1] Брайан У. Керниган, Денис М. Ритчи. Язык программирования C. — М.: Вильямс, 2005.

пишутся как объектные. Автор этой книги уверен, что подавляющая часть разумно написанных на C/C++ программ — это "обычные" программы, т. е. программы без классов. Слово "обычные" взято, правда, в кавычки, т. к. сегодня, наверное, уже спорно, что более обычно: объектная программа или не объектная. Так вот, разница между C и C++ именно в том, что второй нацелен на поддержку объектно-ориентированного программирования, а первый ничего про классы не знает.

Концепция объектного программирования достаточно сложна и для тех, кто не имеет хорошего программистского опыта или просто не хочет слишком уж углубляться в методологию программирования, а таких, кому необходимо быстро создавать небольшие программы, большинство, в этой концепции им нет необходимости. Это означает, что для такого человека необходимо изложить язык C, а не C++, а книга, которая лежит перед вами, представляет именно такое изложение.

Необходимо, правда, сразу заметить, что и сам по себе язык C достаточно сложен по сравнению, например, с языком Паскаль, который изначально создавался как язык для обучения, а уже потом был перехвачен профессионалами, понявшими его изящество и мощность и по сравнению с языком Бейсик, единственная идея которого — это максимальная простота. Язык C создавался сразу как язык для профессионального программирования, причем системного. Поэтому глубокое изучение языка чревато большими сложностями. Но идти до конца, конечно, не обязательно, большинство прикладных программ не нуждается во всех возможностях, прелестях и нюансах этого языка, а, следовательно, бояться его не стоит.

Первый важный вопрос, который должен встать перед изучающим любой язык, — это: чем данный язык выделяется среди остальных. Ответ таков: C дает максимальную гибкость по отношению к аппаратуре компьютера, большую гибкость имеет только ассемблер и при этом C — все-таки язык высокого уровня, располагающий всем привычным набором операторов.

Гибкость достается особым отношением к типам данных. Они в языке C сильно похожи на те данные, которые непосредственно

понимает процессор. И, несмотря на то, что называются они поразному, фактически каждый тип для компилятора — не более чем последовательность байтов, забота о смысле которой возлагается на программиста. Компилятор С позволит целому присвоить символьное или сложить два символьных значения, т. к. с байтами данных эти операции вполне законны. Если, с точки зрения программиста, такие операции не имеют смысла, то это проблема его и только его.

Язык С дает программисту инструменты для работы с битами данных, что еще более приближает программиста к нижнему уровню (только необходимо сказать правду: по своим возможностям обработки битовых данных С — все-таки далеко не ассемблер). Но за возможности необходимо платить. И плата достаточно серьезная. Очень многие вещи, которые компилятор, например языка Паскаль, не пропустит как ошибочные, для компилятора С будут безразличны, следовательно, ответственность программиста за действия его программы существенно возрастает.

В отношении данных С допускает большую вольность не только в контроле типов. Данные в С-программе можно объявлять где угодно, вследствие чего понятие локальной переменой заметно усложняется. Для С локальная переменная — это переменная, объявленная в составном операторе. Таким образом, понятие локальной переменной опускается с уровня программного модуля, такого как подпрограмма, функция или процедура, до минимального блока, каким является составной оператор. А для полного понимания, что такое данные в С, придется разобраться еще с довольно нетривиальными понятиями времени жизни, области видимости и класса памяти.

Существенные отличия ожидают новичка не только в структурах данных. Для того, чьим первым языком был Бейсик, Фортран или Паскаль, наверное вызовет некоторое удивление оператор `for`. И в Бейсике, и во многих других языках этот оператор является конструкцией цикла, но С предоставляет программисту совершенно необычную свободу действий с компонентами этого оператора. В языке достаточно большой набор способов организации присваивания значений, и не все из них нуждаются в операторе,

обозначаемом привычным знаком равенства. Без них можно обойтись, традиционное присваивание тоже есть, но использование иных форм при достаточном опыте позволит сэкономить немного места.

Наверное, некоторую сложность вызовет нетребовательность языка к расположению функций, которые можно раскидать по программе как угодно, даже разместить в различных файлах.

В общем, язык интересный, а некоторая его сложность вполне будет компенсирована возможностью написания более изящного и эффективного кода.

Эта книга описывает язык довольно подробно, с большим количеством примеров, которыми иллюстрируется практически все, что может вызвать сложность понимания. Более того, если примеров, приведенных в главах, вам будет недостаточно, то в *приложении 2* вы сможете найти еще 20 достаточно интересных примеров.

Глава 1 написана для обеспечения легкости усвоения материала теми, для кого С — первый язык программирования, хотя, может быть, эта глава окажется полезной и более искушенным читателям. По сути, *глава 1* — это очень короткое описание языка. Его достаточно для того, чтобы написать свою первую простую программу и затем, отталкиваясь от этого важного успеха, уже без особых затруднений двигаться по более сложным разделам книги.

Особое значение среди языковых понятий и конструкций имеют указатели. Этому понятию в различной степени посвящены самые разные главы. Есть глава, которая так и называется "Указатели", это *глава 9*. Но едва ли меньшее значение для понимания этого термина имеют *главы 10* и *11*. В них говорится о строках, связных списках и деревьях. Это структуры данных, построенные с помощью указателей. Необходимо отметить, что в языке С указатели среди видов данных занимают особое место. Практически любые данные так или иначе связаны с указателями или могут быть связаны с пользой для программиста.

Глава 11 немного выбивается из общей логики. Здесь, собственно, рассказ идет не о языке, а специальном классе задач, и сама

глава построена как перечень решенных задач, поэтому ее наравне с *приложением 2* можно рассматривать, как практическое приложение, однако в связи с особой важностью темы этот материал оформлен не как приложение, а как глава основного материала.

Технология ввода/вывода на С имеет два решения. Оба решения одинаково интересны и имеют свои преимущества. Поэтому операциям ввода/вывода посвящена отдельная глава — *глава 3*. Было бы логично эту главу совместить с рассказом о файлах, но все же она стоит отдельно по тем соображениям, что ввод с клавиатуры и вывод на экран монитора все же встречается чаще, чем файловый ввод/вывод. Но если вы внимательно прочитаете *главу 12* о файлах, то сможете заметить, что в С есть общее понятие потока данных, а куда этот поток направлен — это второй, если не третий вопрос.

Обработке файловых данных также посвящена отдельная глава, и это тоже глава не о базовых возможностях языка. Сам по себе язык С не знает, что такое файлы. Надо сказать, что в стандарте языка вообще нет средств ввода/вывода, в том числе и средств ввода/вывода с использованием файлов данных. Изучая ввод/вывод, вы фактически изучаете дополнительные библиотеки, разработанные для большего удобства прикладных программистов. Вообще, в книге будет много материала, который выходит за рамки собственно языка и является описанием каких-то дополнительных библиотек. Но без этого материала книга станет тонкой и неинформативной. Говоря о библиотеках, необходимо заметить: то, что выходит за рамки стандарта, не обязано поддерживаться любым компилятором, поэтому не факт, что любой наш пример будет успешно откомпилирован и выполнен в любой среде. Но утешает тот факт, что в России люди, начинающие работать с языком С, все-таки чаще всего обращаются к тому же компилятору, на который опирается и материал этой книги. То есть к компилятору, разработанному компанией Borland для среды MS-DOS

И конечно, важнейшее значение имеют приложения. Их два. *Приложение 1*, описывающее терминологию, наверное, наименее важно. Оно потребуется только в том случае, если у вас возник-

нет желание поглубже разобраться в понятийном аппарате языка и выстроить для себя систему теоретических знаний, но для практического владения языком это, пожалуй, не так уж важно.

Приложение 2, безусловно, имеет очень большую практическую значимость. В нем 20 работающих программ. Первые две программы логически просты. Следующие 18 невелики по тексту, но имеют достаточно сложную логику. И все 20 программ имеют ярко выраженный прикладной характер. Логически сложные задачи представлены в укороченном варианте: условие, кратко идея решения и текст программы, снабженный комментариями. Задачи взяты из книги В. Потопахина "Turbo-Pascal. Решение сложных задач"[2]. Системщику от этих задач толку будет, наверное, немного. В них не показаны особенности работы с регистровой памятью или битовыми операциями, но для тех, кто хочет писать прикладные программы, эти примеры окажут хорошую услугу. Такой подбор примеров обусловлен тем, что книга в основном рассчитана на прикладников, а не на системщиков. Кроме того, если С — ваш первый язык, не стоит сразу увлекаться системными задачами, это может оказаться слишком сложным делом. Более разумно начать с задач прикладных. А если интерес к системному программированию останется, то в этой книге для вас будет достаточно много информации и в этой области.

О технологии чтения

Конечно, любую книгу желательно читать с самого начала и делать это последовательно. Но что касается языка программирования, то здесь есть небольшая, но очень существенная сложность. Если язык излагать строго так, как это принято в официальных документах, описывающих стандарт языка, то вполне можно добиться жесткой последовательности и полной непротиворечивости. Но это повлечет за собой потерю прозрачности языковых конструкций.

[2] Потопахин В. Turbo-Pascal. Решение сложных задач. — СПб.: "БХВ-Петербург", 2006.

Автор в построении книги исходит из того, что необходимо изучать не формальные определения, а функциональные возможности. Но тогда очень трудно определить, что за чем должно следовать. Например, имя массива в языке С является указателем на область памяти, в которой данный массив хранится. С другой стороны, понятие указателя предоставляет возможность организации массивов. Оба понятия функционально взаимосвязаны настолько сильно, что разорвать их практически невозможно. А разрывать все равно приходится, т. к. какой-то порядок изложения все-таки необходим. Поэтому иногда придется оставлять некоторые вопросы в уме, в надежде получить ответы позже, а иногда возвращаться назад, чтобы лучше понять, о чем идет речь, достаточно часто материал повторяется в связи с введением новых понятий.

Отчасти *глава 1* снимает эту трудность, т. к. уже с первых страниц дает общий обзор языка, такую общую, хотя и очень приблизительную картину. Но главный инструмент в борьбе с трудностями чтения — это примеры. Почти все, что нужно проиллюстрировать, иллюстрируется примерами, которые сразу работают. Но конечно, все наши усилия не решат проблемы восприятия языка без вашей активной работы с компилятором. Это, впрочем, верно в отношении абсолютно любого языка. Язык программирования изучается в практической деятельности, и этого правила еще никто не отменял.

И последнее замечание. Часть примеров оформлена в виде фрагментов, которые необходимо дополнить прежде, чем они станут программой, а часть примеров — это действительно работающие программы, которые были тщательно отлажены прежде, чем попасть в текст. Поэтому если вы наберете наш пример и он откажется работать, то все-таки посмотрите внимательно, верно ли он набран. Помните, в нашем деле мелочей нет, и один неверно введенный символ может кардинально изменить смысл программы.

Глава 1

Кратко о языке С

Если язык С — не первый ваш язык и вы не считаете себя начинающим программистом, то эту главу можно пропустить, но если вы с языка С начинаете свое программистское образование, то данный текст необходимо прочитать максимально внимательно. Далее, на очень простых примерах мы рассмотрим организацию ввода/вывода в С, самые основные конструкции языка и общую структуру программы. Затем это все будет рассмотрено еще раз и более детально, но, может быть, первый раз лучше обойтись без многочисленных деталей, а увидеть все главное и сразу.

Несмотря на то, что в этой главе подробно разбираются достаточно содержательные примеры и показаны все основные конструкции, не рассчитывайте, что после прочтения вы сможете делать что-то серьезное самостоятельно. Цель главы очень проста — только составить самое общее представление о том, что такое программа, написанная на языке С. Для того чтобы составить быстрое и в то же время достаточно качественное представление об изучаемом языке, мы не будем тщательно изучать типы данных, правила их объявления и преобразования, не станем подробно говорить о свойствах различных библиотечных функций и операций. Мы подойдем к языку с прикладной точки зрения. А именно попробуем решить несколько несложных задач и для каждой из них посмотрим, что необходимо для записи соответствующего алгоритма.

Простые примеры

Рассмотрим следующую задачу. Пусть дано множество чисел. Необходимо найти среди них наибольшее.

Предположим, что наше множество пронумеровано. Это естественное предположение, потому что если даже это и не так, то пронумеровать элементы множества несложно. Можно любой выбранный элемент объявить первым, затем любой из оставшихся — вторым, потом любой из оставшихся — третьим и т. д.

Итак, нумерация проблемы не составляет, поэтому будем считать, что числа множества пронумерованы. Договоримся об обозначениях. Пусть множество обозначено буквой A. Тогда A_1 — первый элемент, A_2 — второй и т. д., т. е. A_i — i-ый элемент. Количество элементов множества будем обозначать буквой N. С учетом принятых договоренностей можно написать алгоритм ввода такого множества чисел. Текст ниже:

```
Ввести N
Для i от 1 до N с шагом 1 делать: Ввести Ai
```

Команда `Ввести` говорит о том, что для ее исполнения необходимо ввести число с клавиатуры компьютера. Вторая команда называется *конструкцией цикла*. Она состоит из двух составляющих: заголовка цикла и набора команд, называемого телом цикла. В нашем примере тело цикла состоит из одной команды.

В заголовке сказано, что величина `i`, имея в начале выполнения цикла значение 1, изменяется на каждом шаге работы цикла на единицу, и это происходит до тех пор, пока ее значение не достигнет значения `N`. Как только значение `i` достигнет `N`, выполнение цикла прекратится. В тексте шаг цикла состоит в выполнении команды `Ввести Ai` — эта команда является телом цикла. В нашем примере в теле цикла только одна команда, но их может быть несколько.

Из сказанного, наверное, уже понятно, что смысл конструкции цикла в том, чтобы многократно выполнить много команд, не повторяя их записи. В нашем цикле на первом шаге будет введено

число с номером 1, на втором — число с номером 2 и т. д. до номера `N`.

Конечно, если вы имеете хотя бы небольшой опыт программирования на любом другом языке, то сказанное ранее должно быть вполне понятно, ничего нового мы здесь не сообщили. Поэтому далее просто запишем, как текст этого алгоритма выглядит на языке С.

```
cin >> N;
for (int i=1;i<=N;i++) cin >> A[i];
```

Нужно пояснить эти строки.

Команда `cin` — это команда ввода значения с клавиатуры. В нашем фрагменте сначала вводится число `N`, а затем в теле цикла многократно выполняется ввод элементов массива.

Массив — это сложная структура данных, которая содержит в себе много чисел (или данных другого типа), имеющих одинаковое имя. В нашем случае — общее имя элементов массива `A`, в квадратных скобках проставляется номер элемента, и только номерами они друг от друга и отличаются.

`for` — это ключевое слово, обозначающее цикл, точнее один из типов циклов, существующих в С (есть еще два, но о них позже). После него в круглых скобках записано то, что было записано в заголовке цикла алгоритма: `i=1` — параметр цикла имеет начальным значением единицу. `i<=N` — условие продолжения цикла (цикл работает, пока параметр не больше `N`). И, наконец, запись `i++` говорит о том, что `i` изменяется с шагом 1.

Ключевое слово `int` утверждает, что величина `i` — целая. Это называется *объявлением типа*. Выполнить операцию объявления типа надо бы для всех используемых переменных. То есть в полностью завершенной программе должно быть объявление типа для массива `A` и величины `N`.

Учтите, это не текст законченной программы. Чтобы этот фрагмент реально начал работать, необходимо сделать еще достаточно много. Но мы пока вернемся к задаче, решим ее и запишем алгоритм на обычном русском языке. Идея алгоритма такова: объявим первое число наибольшим, а затем для каждого из ос-

тавшихся проверим, не больше ли оно уже найденного наибольшего, и если больше, то поменяем значение наибольшего. Далее представлен алгоритм:

```
Наибольшее = первому
Для всех чисел, начиная от второго и до N–го, делать
    Если Очередное число больше Наибольшего
      То Наибольшее = Очередному
Вывести значение Наибольшего
```

А теперь то же самое запишем на языке С.

```
int max=A[1];
for ( int i =2; i<=N;i++)
  if (max<A[i]) max=A[i];
cout << max;
```

Смысл нового ключевого слова `cout`, наверное, уже ясен: это команда вывода значения на экран монитора. И появилось еще одно ключевое слово — `if`. Но его смысл также должен быть понятен из текста алгоритма. Эта команда проверяет условие (условие обязательно записывается в скобках), записанное после ключевого слова `if`, и если условие верно, то выполняется команда, записанная после условия. Это самая простая форма *условного оператора*, подробности мы сейчас рассматривать не будем. Необходимо пояснить, что в теле цикла многократно выполняется только команда выбора `if`, т. к. цикл `for` считает своим телом только одну команду, записанную после заголовка. Правда, эта единственная команда может быть составным оператором, но об этом позже, сейчас же скажем, что команда `cout` выполняется уже за пределами цикла и, следовательно, выполняется только один раз (листинг 1.1).

Листинг 1.1

```
#include <iostream.h>
#include <conio.h>
void main()
{ int N,A[100];
```

```
  cin >> N;
  for (int i=1;i<=N;i++) cin >> A[i];
  int max=A[1];
  for (i=2;i<=N;i++)
    if (max<A[i]) max=A[i];
  cout << max;
  getch();
}
```

Начинается текст программы с записи `#include <iostream.h>`. Это не команда языка. Это директива компилятору, директивам в нашей книге посвящена *глава 15*, в которой достаточно подробно описано, зачем они нужны и какие директивы существуют. Директивы в отличие от команд выполняются только на этапе компиляции, а в ходе исполнения программы их в тексте кода уже нет. Данная директива подключает к тексту программы так называемый *файл заголовков*. Этот файл состоит из имени и расширения h и содержит объявления различных функций, хранящихся в библиотеках (не все возможности языка известны непосредственно компилятору, часть из них хранится отдельно и подключается к работе по мере необходимости).

Далее записан исполняемый текст. Он представлен одной функцией, состоящей из заголовка и текста, заключенного в фигурные скобки. Заголовок несет в себе информацию о том, данные какого типа функция возвращает по завершении своей работы, тип указывается перед именем. Затем записывается имя функции. Имя — это почти любой набор допустимых символов, но одна функция обязательно должна называться `main()`, с нее начинается выполнение программы. В нашей программе функция только одна, поэтому она в обязательном порядке имеет имя `main()`.

И наконец, в круглых скобках после имени (эти скобки обязательны в любом случае) записываются имена переменных, используемых функцией для получения значений извне. Наша программа состоит только из одной функции, она ничего не получает, и круглые скобки пусты.

Текст программы мы уже анализировали ранее, поэтому скажем только об одной команде:

```
int N,A[100];
```

В ней, как и было обещано, объявлены недостающие типы для переменной `N` и массива, а для массива еще указан его максимально допустимый размер.

И самое последнее. Функция `getch()` нужна только для того, чтобы окно с выполняемой программой не закрылось с завершением выполнения, и мы успели что-либо увидеть. Эта функция требует ввода символа, и пока ни одна клавиша не нажата, программа будет ждать, выполнение не будет завершено, и мы увидим результаты работы программы.

Все! Программа полностью готова, ее можно откомпилировать и запустить на выполнение.

Наша первая программа состоит из одной функции, но может состоять и из нескольких. Поэтому даже для первого знакомства необходимо показать, как строится такая программа. Для примера возьмем ту же самую задачу. Ввод и вывод результата оставим главной функции, а расчеты перенесем в другую. Для того чтобы показать процесс передачи данных из функции в функцию, передадим в дополнительную функцию величину `N`. Конечно, надо было бы передать и массив `A`. Но так как наша цель — только знакомство, не будем усложнять себе задачу и ограничимся передачей лишь одной величины. Текст программы записан в листинге 1.2.

Листинг 1.2

```
#include <iostream.h>
#include <conio.h>
int maximum(int);
int A[100];
void main()
{ int N;
  cin >> N;
  for (int i=1;i<=N;i++) cin >> A[i];
```

```
  int max=maximum(N);
  cout << max;
  getch();
}
int maximum(int m)
{
  int max=A[1];
  for (int i=2;i<=m;i++)
    if (max<A[i]) max=A[i];
  return max;
}
```

Примечание

В тексте программы появилось одно, на первый взгляд, несущественное изменение. А именно объявление массива сделано до функции `main()`. На самом деле, это очень важный момент, связанный с понятиями локальной и глобальной переменных.

Глобальной называется переменная, которой можно пользоваться в любом месте программы. *Локальная переменная* наоборот доступна только в той структурной единице, в которой она объявлена. Эти два понятия присутствуют почти в любом языке программирования, за исключением, быть может, некоторых версий языка Бейсик. Отличие между языками заключается в том, что понимается под структурной единицей. Например, в языке Паскаль, структурными единицами являются процедура и функция. В языке С структурной единицей считается составной оператор, т. е. группа команд, записанная между фигурными скобками. Проблемы с объявлениями переменных мы еще рассмотрим подробнее, сейчас же будет достаточно сказать, что функция также является составным оператором, поэтому массив, объявленный в предыдущем решении, известен только функции `main()`, т. к. объявлен внутри нее. Если все так и оставить, то на этапе компиляции получим сообщение, что массив функции `maximum()` не известен. Мы же вынесли объявление массива за пределы всех существующих фигурных скобок, что как раз и обеспечивает доступность массива нашей вспомогательной функции, он становится глобальным.

Еще одно важное дополнение — `int maximum(int)`. Это так называемый *"прототип"* или *"заголовок"*. Появление прототипа есть следствие важного принципа, который также присутствует в любом языке. Звучит принцип так: все, что используется в программе, должно быть объявлено до первого своего использования. В нашем же примере функция `maximum()` используется в функции `main()`, а описана ниже ее, следовательно, на момент своего первого использования (т. е. в функции `main()`) функция `maximum()` неизвестна. Для того чтобы выйти из затруднительной ситуации, и описывается прототип, т. е. краткая информация о том, что представляет собой функция `maximum()`. В нашем примере указано, что она возвращает целое значение, на входе получает одну целую переменную и имеет имя `maximum`. После того как записан прототип, саму функцию можно описывать где угодно в тексте программы. Единственный твердый запрет языка C — это запрет на вложенные описания, т. е. функцию нельзя записывать внутри другой функции. Без прототипа можно обойтись, если функцию `maximum()` разместить до функции `main()`, но все же принято размещать функции так, как удобно разработчику, а в начале файла программы указывать все их прототипы.

В самой функции `maximum()` появилась новая команда `return`. Это команда, возвращающая значение, т. е. то значение, которое функцией `maximum()` будет передано переменной `max`.

В рассмотренном примере мы воспользовались целочисленным типом `int`. Конечно, это не единственный тип данных, поддерживаемый языком C. Существуют два типа для чисел с десятичной точкой. Это тип `float` (одинарная точность) и тип `double` (двойная точность). Для представления символьных данных предназначен тип `char`.

Следующий пример (листинг 1.3) показывает использование типа `double`.

Листинг 1.3

```
#include <iostream.h>
#include <math.h>
void main()
```

```
{ double result[100][3];
  int i=0;
  for (float x=0; x<=3.14; x+=0.1)
    { result[i][0]=x;
      result[i][1]=sin(x);
      result[i][2]=cos(x);
      i++;
    }
  for (int k=0;k<i;k++)
   cout <<result[k][0]<<" "<<result[k][1]<<" "<<re-
sult[k][2]<<"\n";
}
```

Обратите внимание, что в заголовке цикла величина `x`, играющая роль параметра цикла, имеет тип `float`, т. е. является действительным (вещественным) числом. Во многих языках программирования параметр цикла — величина обязательно целая. Язык С свободен от такого ограничения. Оператор

```
x+=0.1
```

это еще одна форма *оператора присваивания*. То же самое можно записать универсальным присваиванием:

```
x=x+0.1
```

Но С предлагает нам целый набор специализированных операторов, которые в некоторых случаях помогают выполнить запись короче. Язык С для каждой арифметической операции предлагает дополнительную форму присвоения: `+=`, `-=`, `*=`, `/=`. Это звучит так: вычислить и прибавить, вычислить и отнять и т. д.

Еще два важных замечания о массивах

Замечание первое. Массив, объявленный в нашем примере, назвать двумерным массивом будет не совсем точно. Более правильно будет сказать, что мы объявили 100 массивов длиной по 3 элемента каждый. В большинстве случаев разницы между многомерным массивом и массивом массивов нет, но все равно запомните этот факт. Хорошо изучив указатели и усвоив связь между указателями и массивами, вы поймете, что это не вполне одно и то же.

Замечание второе. Нумерация элементов массива всегда начинается с нуля. Объявить массив с произвольным интервалом изменения индекса нельзя. В некоторых языках такого ограничения нет. В C это ограничение связано с идеей приближения языка к языками низкого уровня и обеспечения максимальной гибкости и переносимости программ.

Еще два важных понятия, необходимых для хорошего понимания механизма работы C с величинами: это *область видимости* и время жизни величины. Эти два понятия описывают, где, в какой части программы объявленной переменной можно пользоваться, и сколько времени она существует. Язык C разрешает объявлять переменные где угодно, поэтому и появилась необходимость в правилах для определения времени жизни и области видимости. Для простых случаев эти два понятия сложностей не составляют. *Область видимости* для переменной — это тот составной оператор, в котором она объявлена. *Время жизни* — с момента объявления до момента завершения выполнения составного оператора, в котором величина объявлена.

Но в C все не так просто. Прежде всего, объявление может быть сделано на внешнем уровне, т. е. за пределами всех возможных блоков, тогда эта переменная глобальная. И переменная может быть объявлена внутри оператора — это иная ситуация, для которой наше определение полностью бы подходило, если бы не еще один важный нюанс. В C существуют *классы памяти*, определяющие способ хранения переменных и в силу этого влияющие на область видимости и время жизни. Наши обычные переменные, такие, которые мы использовали в приведенных примерах, — это переменные класса `auto`. Для них определение области видимости и времени жизни остаются в силе. Но это не единственный класс. Существуют еще три. О них подробнее позже. Сейчас только упомянем о классе `static`. Область видимости для переменных этого класса определяется так же, а вот время жизни становится глобальным. Это означает, что выход за пределы блока, в котором переменная была объявлена, не приводит к потере значения. И это независимо от того, чем именно является блок — просто сложным оператором или функцией.

Рассмотрим следующий пример (листинг 1.4).

Листинг 1.4

```
#include <iostream.h>
void main()
{ int s;
  for (int i=1; i<=5; i++)
    for (int k=1; k<=5; k++)
    { static int sum=0;
      sum+=k;
      s=sum;
    }
  cout << s;
}
```

Результатом работы будет число 75. Это означает, что величина `sum` накапливается на всех шагах двух циклов и не инициализируется нулем при повторном вхождении в составной оператор, следующий после заголовка второго цикла. Уберите ключевое слово `static`, и вы увидите совершенно иной результат.

Условные циклы

Предположим, есть необходимость вычислять некоторую величину до тех пор, пока не будет выполнено некоторое условие. Пусть, например, необходимо посчитать сумму элементов ряда 1 + 1/2 + 1/3 + ... до тех пор, пока элемент ряда не станет меньше некоторого малого числа. Ясно, что нужен цикл, но сейчас нам неизвестно, сколько будет выполнено операций. Эту задачу можно решить тремя способами. Вполне сгодится уже известный нам цикл `for` в силу его огромной гибкости, но С для таких ситуаций предоставляет еще две формы цикла: цикл с предусловием вида:

`while (`*`условие`*`)` *`оператор`*

и цикл с постусловием, т. е. цикл вида

`do`

{ *`перечень операторов`*

```
}
while (условие)
```

Решение нашей задачи с помощью цикла `for` представлено в листинге 1.5.

Листинг 1.5

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  float sum=0;
  for (float i=1;1/i>=0.001;i++) sum+=1/i;
  cout <<sum;
}
```

Эта же задача для цикла `while` представлена в листинге 1.6.

Листинг 1.6

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  float sum=0;
  float i=1;
  while (1/i>=0.001)
   {sum+=1/i;
    i++;
   }
  cout <<sum;
}
```

И эта же задача для цикла `do...while` решена с помощью программы из листинга 1.7.

Листинг 1.7

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  float sum=0;
  float i=1;
  do
   {sum+=1/i;
    i++;
   }
  while (1/i>=0.001);
  cout <<sum;
}
```

Указатели

В языке С много любопытных конструкций и объектов. Конечно, в кратком изложении мы не будем говорить обо всем даже в двух словах, но, по крайней мере, об указателях сказать совершенно необходимо. Внешне *указатель* — очень простая вещь. Это переменная, содержащая адрес, по которому находится числовое или какое-либо иное значение. Объявить указатель просто, вот так:

```
int *a
```

Это объявление указателя на целую величину. Указатели полезны во многих отношениях, например, когда необходимо передать величину из одной функции в другую, далее мы поговорим об этом более подробно. Но самое важное в них, с точки зрения автора этой книги, — это то, что указатели представляют собой универсальный механизм выделения памяти под любые переменные и структуры. Указатель может ссылаться как на одиночную переменную, так и на массив, и в обоих случаях он работает одинаково.

Для того чтобы обратиться к адресу, переменная, объявленная как указатель, используется по своему имени, а чтобы обратиться

к значению, на которое ссылается указатель, к его имени достаточно добавить звездочку, вот так:

- `int *a;` — объявлен указатель на целое;
- `*a=8;` — в ячейку памяти, на которую указывает a, помещено число 8;
- `a++;` — выполнен переход на следующую ячейку.

Для указателей в C доступно достаточно много мощных операций, что делает их важным и интересным инструментом. А сейчас перепишем программу поиска максимального с использованием указателей (листинг 1.8).

Листинг 1.8

```
#include <iostream.h>
#include <conio.h>
int maximum(int,int *);
void main()
{ int *a,*b;
  clrscr();
  int n;
  cin >>n;
  a=new int[n+1];
  b=a;
  for (int i=0;i<=n;i++)
    {cin >>*a;
     a++;
    }
  int max=maximum(n,b);
  cout << max;
}
int maximum(int n, int *a)
{ int max=*a;
  for (int i=0;i<=n;i++)
   { if (*a>max) max=*a;
```

```
    a++;
   }
  return max;
}
```

Во-первых, сразу обратите внимание на тот факт, что мы избавились от глобальных переменных. Это хорошо. Во-вторых, обратите внимание на оператор

```
a=new int[n+1];
```

Здесь мы выделили память под массив именно такого размера, какой нам нужен, и ни одного байта больше. Объявить массив как

```
int a[n+1];
```

нельзя, а указатель помогает нам обойти это ограничение.

Заключение

Итак, еще раз перечислим основные объекты языка С, с которыми нам придется иметь дело, в двух словах опишем их функциональное назначение и на этом завершим наше краткое изложение и перейдем к подробному.

Итак. Переменные и константы являются основными объектами, которыми оперирует программа. Описания перечисляют переменные, которые будут использоваться, указывают их тип и, возможно, их начальные значения. Операции определяют, что с переменными и константами будет сделано. Выражения объединяют переменные и константы для получения новых значений. А оператор — это минимальная управляющая команда, используемая в программе.

Глава 2

Величины

Типы данных

Язык C, как впрочем и любой другой язык программирования, требует от программиста описания величины, на основании которого компилятор выделяет под нее память и определяет, какие операции с данной величиной являются допустимыми. Описание в C — это сложная конструкция, состоящая из описателей, имеющих различный смысл. В языке C для полного описания переменной указывается ее тип и класс памяти. Начнем с простого перечисления описателей (табл. 2.1).

Таблица 2.1. *Базовые типы языка C*

Тип	Длина в битах	Интервал значений
`unsigned char`	8	0 — 255
`char`	8	–128 — 127
`unsigned int`	16	0 — 65 535
`short`	16	–32 768 — 32 767
`int`	16	–32 768 — 32 767
`unsigned long`	32	0 — 4 294 967 295
`long`	32	–2 147 483 648 — 2 147 483 647
`float`	32	$3{,}4\times10^{-38}$ — $3{,}4\times10^{38}$
`double`	64	$1{,}7\times10^{-308}$ — $1{,}7\times10^{308}$
`long double`	80	$3{,}4\times10^{-4932}$ — $1{,}1\times10^{-4932}$

Сразу уточним, что величина типа `int` самым серьезным образом зависит от аппаратуры компьютера, от того, как на конкретном процессоре реализуется тип целый. Поэтому, несмотря на одинаковое описание типов `int` и `short` (это два байта вне зависимости от аппаратной реализации), их конкретная реализация может существенно отличаться. Однако это отличие вряд ли удастся обнаружить на современных компьютерах.

Из построенного перечня описателей хорошо видно особое значение описателя `unsigned`. Он указывает на то, что выбранный тип не содержит в себе знака. Таким образом, данный описатель исключает отрицательную часть и вдвое увеличивает положительную.

Еще один важный факт. Все типы имеют числовые значения. Это прямое следствие важнейшей идеи языка — как можно более полно соответствовать внутренним вычислительным процессам, которые реально выполняются процессором, а, как известно, на самом нижнем уровне нет ничего кроме чисел со знаком или без знака.

Конечно, можно сказать, что на самом нижнем уровне нет ничего кроме двоичных чисел, но ограничение типов только лишь различными по длине двоичными числами слишком сильно обеднило бы выразительную силу языка. Такой язык вряд ли смог бы стать привлекательным для прикладных программистов, т. к. в этом случае временны́е затраты на написание прикладных программ, выполняющих даже простую арифметику, окажутся слишком велики. Поэтому ограничение только числовыми типами — это компромисс, позволивший сделать язык очень гибким и в то же время достаточно выразительным.

Кроме того, даже среди указанных типов есть один с особыми свойствами. Это тип `char` (и конечно `unsigned char`). Это тип числовой, и с ним возможны арифметические операции, но значение переменной этого типа легко и естественно выводится в виде символа. Попробуйте, например, выполнить программу из листинга 2.1.

Листинг 2.1

```
#include <iostream.h>
void main()
{ char a='f';
  char b;
  b=a+123;
  cout <<b;
}
```

Результат, выведенный на экран, будет представлять собой символ, код которого содержит переменная `b`. Этот пример говорит о том, что типы `int` и `char` в значительной степени эквиваленты, и с этим обстоятельством мы еще не раз встретимся.

И еще одно важное замечание. Вы можете присваивать значения любого типа переменным любого типа. Например, компилятор не обнаружит ошибки и выполнит такую программу (листинг 2.2).

Листинг 2.2

```
#include <iostream.h>
void main()
{ double a=678688;
  int b=a;
  cout << b;
}
```

Программа, конечно, будет выполнена, но вы не должны ожидать, что значение длинной переменной сохранится. Программа выполнит преобразование типов, так чтобы между переменными не возникло конфликта по вопросу требуемой памяти, но это произойдет за счет значения. Лишние биты будут просто обрезаны. В некотором смысле можно утверждать, что тип `char` — основной в языке C. Во-первых, он не зависит от реализации языка на конкретной машине, во-вторых, он позволяет выполнять над собой все арифметические операции, в-третьих, он совместим с любыми другими типами (обратное неверно).

Еще один важный описатель, участвующий в создании типа — это описатель `static`. Это так называемый *описатель класса памяти*. В конце главы мы более подробно поговорим о том, какие еще бывают классы памяти и какая в них может быть потребность, но класс памяти `static` обладает особо важными свойствами, кардинально меняющими поведение переменных, поэтому остановимся на нем сейчас.

Чтобы понять его роль, необходимо рассмотреть понятие локальной и, соответственно, глобальной переменной. Их определения легки для понимания. *Локальной переменной* называется переменная, объявленная внутри составного оператора. Напомним, что *составной оператор* — это часть текста программы, заключенная между двумя фигурными скобками, открывающей и закрывающей, об этом уже говорилось в *главе 1*. *Глобальная переменная* — это переменная, объявленная до самой первой функции и, следовательно, известная всей программе.

Далее мы более подробно поговорим о таких важных свойствах локальных переменных, как время жизни и область видимости, сейчас лишь заметим, что значение локальной переменной сохраняется в стеке, который создается под составной оператор и уничтожается по завершении работы составного оператора.

Переменная, объявленная как переменная класса `static`, хранится в статической памяти (не стековой), и ее значения сохраняются и по завершении работы составного оператора. Для иллюстрации рассмотрим простой пример (листинг 2.3).

Листинг 2.3

```
#include <iostream.h>
#include <conio.h>
int f(int);
void main()
{ clrscr();
  cout << f(1)<< " ";
  cout << f(1);
}
```

```
int f(int t)
{ static int a=t;
  a++;
  return a;
}
```

В главной функции два совершенно одинаковых вызова функции `f`, и мы могли бы ожидать, что результат работы второго вызова не будет отличаться от результата первого. Но в действительности второй вызов выдаст результат, отличный от первого. Причина такого события в объявлении

```
static a=t;
```

Значение переменной a после первого вызова не погибнет, а продолжит свое существование в статической памяти и будет использовано при последующем вызове.

Необходимо отметить, что глобальные переменные являются переменными класса `static`, даже если они и не объявляются как таковые.

Расположение объявлений

Объявление переменной может быть сделано почти в любом месте, с соблюдением следующего условия: объявление не может зависеть от выполнения какого-либо условия. Это означает, что следующий фрагмент программы ошибочен:

```
if  (y==5) float a=8.1;
```

Но, с другой стороны, объявление возможно в теле составного оператора. Следовательно, следующий вариант объявления вполне допустим:

```
if (y==5) {float a=8.1;}
```

Данный нюанс предупреждает казус. Отсутствие объявления до первого использования есть синтаксическая ошибка. Но рассмотрим следующий фрагмент:

```
{ if (i!=7) int y=8;
```

```
  int sum+=y;
}
```

Если условие оператора выбора окажется невыполненным, оператор присваивания нельзя будет выполнить по причине ошибки, которую должен поймать компилятор. Но он ее отловить не сможет, т. к. компилятор не в состоянии проанализировать ход выполнения программы. Поэтому лучше такие объявления запретить. Это не слишком обеднит наши возможности. Например, этот же фрагмент можно без особых проблем переписать так:

```
{ int y;
  if (i!=7) y=8;
  int sum+=y;
}
```

Это естественный выход, т. к. для проверки условия существенно важно именно задать значение переменной, а не объявить ее.

Инициализация переменных

Операция *инициализации* — это задание первого значения переменной. В отличие от многих языков, в C почти не работает принцип умолчания, который гарантирует присвоение переменной какого-то значения, если программист не позаботился об этом специально.

В C-программе рассчитывать на принцип умолчания можно только в случае переменных класса `static` и внешних переменных, которые, впрочем, тоже принадлежат к классу `static`, даже если это не объявлено специальным образом. Их значение по умолчанию равно нулю, иные переменные (автоматические) без инициализации равны чему угодно.

> ***Примечание***
>
> О классе автоматических переменных будет сказано далее в этой главе, а т. к. этот термин применяется уже сейчас, то поясним, что любая переменная, чей класс не указан напрямую, явным образом является автоматической.

Инициализировать переменную возможно как в момент задания, так и после объявления. Для автоматических переменных инициализацию можно выполнить так:

```
int a;
a=8;
```

Это же самое возможно сделать так:

```
int a=8;
```

Впрочем, если подходить более строго, то в первом случае инициализации нет вовсе, а оператор `a=8` — это просто оператор присваивания. Однако здесь может возникнуть вопрос: если эти ситуации неразличимы, то зачем введен термин "инициализация"? Новый термин должен содержать в себе новый смысл. Новый смысл действительно есть. Заключается он в задании исходного значения переменной. Однако для автоматических переменных различие имеет косметический характер, просто инициализировать переменные в момент объявления удобно, это позволяет немного сократить текст.

Для статических переменных момент инициализации имеет более серьезное значение. Рассмотрим программу, в которой демонстрируется особенность класса `static`, с небольшим изменением (листинг 2.4).

Листинг 2.4

```
#include <iostream.h>
#include <conio.h>
int f(int);
void main()
{ clrscr();
  cout << f(1)<< "  ";
  cout << f(1);
}
int f(int t)
{ static int a;
```

```
  a=t;
  a++;
  return a;
}
```

Разница заключается в том, что переменная объявлена, а первое значение получено одним оператором позже. Различие принципиальное. Дело в том, что все операторы функции `f` выполняются при каждом ее вызове, а инициализация переменной класса `static` выполняется только один раз при вхождении в тело функции. Поэтому в измененном примере величина `a` после своего объявления каждый раз получает значение 1, и результат работы функции оба раза равен 2.

Еще один интересный пример представлен в листинге 2.5.

Листинг 2.5

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  for (int i=1;i<=10;i++)
   { static int sum=0;
     sum+=i;
     cout <<sum<<" ";
   }
}
```

Если бы в теле цикла переменная `sum` была объявлена как автоматическая, то на каждом шаге цикла ее значение инициализировалось бы нулем, и затем выполнялось присвоение очередного значения параметра цикла. В нашей же программе фактически считается сумма первых десяти чисел, т. к. инициализация суммы в теле цикла выполняется только один раз!

Переменные класса `static` могут быть очень полезны в задачах, требующих организации "параллельных вычислений". Термин

"параллельные вычисления" взят в кавычки вот почему: обычно, когда говорят о такой организации вычислений, имеется в виду сложная, многопроцессорная вычислительная система, для которой разрабатывается специальная технология разбиения единого вычислительного процесса на несколько независимых, каждый из которых выполняется на отдельном процессоре. Наша работа посвящена технологии программирования на обычном персональном компьютере, и значение термина будет принципиально иным.

Пусть для решения некоторой задачи требуются результаты работы двух вычислительных процессов, назовем их условно A и B. И пусть они устроены так, что процесс A может начаться независимо от процесса B, и на каждом шаге своего цикла он порождает данные для процесса B. Тогда общая схема вычислений может выглядеть например так:

Пока не получен конечный результат:

- ❒ выполнить шаг процесса A;
- ❒ выполнить вычисления процесса B.

Из этой схемы хорошо видно, что выполнение процесса A прерывается для выполнения B, но если A и B организованы в виде отдельных функций (что целесообразно, если A и B — крупные процессы с большим количеством операторов), то после прерывания A, следующий шаг его работы будет выполняться с точно такими же исходными данными, как и на предыдущем шаге. Тогда и B на каждом шаге будет делать то же самое, а, следовательно, циклическое выполнение общего вычислительного процесса практически потеряет смысл.

При использовании переменных класса `static` состояние процесса A можно изменять от шага к шагу, и такая вычислительная схема окажется осмысленной. Термин "параллельность" здесь используется для указания того факта, что процесс A не завершается перед началом процесса B, он просто останавливается и затем продолжается с точки останова. Корректность использования термина определяется тем, что процессы A и B при такой организации можно действительно распараллелить.

Рассмотренные примеры не исчерпывают всех вопросов инициализации. Отдельная тема для разговора — это инициализация сложных структур данных: массивов и записей. Но о приемах инициализации сложных структур мы будем говорить в последующих главах.

> ***Важное примечание***
>
> Ранее было сказано, что инициализация переменных класса `static` выполняется один раз при входе в блок, в котором переменная определяется. Это противоречит некоторым учебникам, в которых утверждается, что инициализация переменных класса `static` выполняется в момент запуска программы. Возможно, этот вопрос зависит от реализации языка, но компилятор, на который мы опираемся, делает это так, как было сказано ранее. Это можно доказать примером из листинга 2.6.

Листинг 2.6

```
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
void main(void)
{ clrscr();
  int n=0;
  for (int i=1;i<=10;i++)
    n+=i;
  static int r=n;
  cout << r;
}
```

В момент запуска этой программы значение автоматической переменной `n` еще не посчитано, и, следовательно, статическая переменная `r` проинициализирована быть не может. Однако инициализация осуществляется совершенно правильно, что возможно только в том случае, если она выполняется в момент объявления в ходе работы программы.

Различия в технологии инициализации автоматических и статических переменных очень велики по происходящим от этого по-

следствиям. Для подтверждения приведем еще один пример. Пусть требуется рассчитать сумму факториалов. В листинге 2.7 представлена программа, решающая эту задачу посредством автоматических переменных.

Листинг 2.7

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  int n;
  cin >> n;
  long sum=0,factorial=1;
  for (int i=1;i<=n;i++)
    sum+=factorial*=i;
  cout << sum;
}
```

Хорошая, достаточно простая программа. Но в функции `main()` переменные `sum` и `factorial` имеют одинаковые области видимости (часть программы, в которой эти переменные можно использовать), что не логично. Переменная, хранящая сумму, действительно может быть нужна далее, а вот переменная `factorial` имеет четко локальный смысл, за пределами расчетного цикла она не нужна, но в пределах класса автоматических переменных это положение не исправить, т. к. есть необходимость задать ее значение до начала вычислительного процесса. Класс `static` позволяет легко улучшить программу в части экономии переменных (листинг 2.8).

Листинг 2.8

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
```

```
  int n;
  cin >> n;
  long sum=0;
  for (int i=1;i<=n;i++)
   { static long factorial=1;
     sum+=factorial*=i;
   }
  cout << sum;
}
```

В новом варианте, переменная `factorial` имеет область видимости, ограниченную составным оператором, являющимся телом цикла. Следовательно, за пределами этого составного оператора переменная `factorial` не существует.

Более подробно о времени жизни и области видимости

Программа на C обладает сложной блочной структурой, единицей которой является составной оператор. Составные операторы могут включаться один в другой, каждый такой оператор может содержать объявления имен переменных, и эти имена, объявленные в разных операторах, могут совпадать. Поэтому важно разобраться, что делает программа, когда ей в разных местах встречаются одинаковые имена. Например, что будет напечатано при выполнении следующего программного блока:

```
{ int x=1;
  { int x=2;}
  cout << x;
}
```

Для ответа на поставленный вопрос язык C предлагает следующее правило: переменная видна (т. е. ее значение можно использовать) от точки, в которой она объявлена, и до закрывающей скобки того составного оператора, в котором она объявлена.

Переменные, объявленные внутри составного оператора, называются *локальными*. В момент объявления локальной переменной, переменная с таким же именем, но объявленная на верхнем уровне, скрывается, но ее значение не теряется. Значение скрытой переменной опять становится доступным при возвращении выполнения программы в тот составной оператор, в котором была объявлена скрытая переменная. Таким образом, в нашем примере будет напечатано значение 1, т. к. для команды `cout` переменная, объявленная во внутреннем операторе, не видна.

Игнорирование этого правила может привести к очень грубым ошибкам. Приведем примеры организации вложенного цикла (табл. 2.2).

Таблица 2.2. *Примеры организации вложенного цикла*

Блок без ошибки	**Блок с ошибкой**

`for (int i=1; i<=5; i++)`
`{ for (int i=1; i<=5; i++)`
`int x:=i*i;`
`}`
	`for (int i=1; i<=5; i++)`
	`{ for (i=1; i<=5; i++)`
	`int x:=i*i;`
	`}`

Ошибка заключается в том, что в заголовке внутреннего цикла параметр цикла не объявлен как локальный, и поэтому используется переменная `i`, объявленная в заголовке внешнего цикла. В результате по выполнении внутреннего цикла внешний также окажется выполненным. В примере, приведенном слева, переменная `i` объявляется как в заголовке внешнего цикла, так и в заголовке внутреннего цикла, и это разные локальные переменные.

Необходимо также не забывать, что составной оператор начинается фигурной скобкой, и никак иначе. Иногда можно встретиться с такой ошибкой:

```
for (int i=1;i<=5;i++)
  for (int i=1;i<=6;i++)
```

Объявление переменной `i` здесь осуществлено на внешнем уровне и действительно как для первого, так и для второго заголовка

цикла. Компилятор в этом случае объявит, что дублируется объявление переменной. Необходимо только заметить, что понятие составного оператора имеет некоторые нюансы, зависящие от реализации. Например, для компилятора Turbo-C компании Borland этот фрагмент ошибочный, т. к. заголовок цикла для данного компилятора не есть начало составного оператора, а для компилятора C++ версии 5 под Windows той же фирмы это же объявление не содержит ошибки, т. к. для этого компилятора заголовок цикла есть составной оператор. Мы в тексте книги ориентируемся на компилятор Turbo-C под MS-DOS.

Если переменная используется в блоке и в нем не объявлена, то работает объявление, сделанное на предыдущем уровне вложенности (листинг 2.9).

Листинг 2.9

```
#include <stdio.h>
void main()
{ int x=9;
  { int x=8;
     { printf("%d",x); }
  }
}
```

Эта программа распечатает значение `x`, равное 8.

Глубина вложения блоков может быть самой различной, и возможны объявления переменных на самом внешнем уровне. Это так называемые *глобальные переменные*. Глобальные переменные имеют особое положение. Для них существует *операция разрешения области видимости* (листинг 2.10).

Листинг 2.10

```
int x;
void main();
 { { int x=2;  // Присваивание локальному x //
```

```
        :: x=3; // Присваивание глобальному x //
    }
}
```

Операция разрешения области видимости допустима только для глобальных переменных. Поэтому следующая конструкция ошибочна:

```
{ static int x;
  { ::x=8;
  }
}
```

Причем она ошибочна, даже не глядя на то, что переменная `x` объявлена как переменная класса `static`. Этого не достаточно. Существенно важен именно глобальный характер области видимости, а переменная, объявленная внутри составного оператора, имеет все же локальную видимость.

Время жизни переменной — это временной интервал, в течение которого существует переменная. Переменная начинает свое существование в момент определения и заканчивает в момент вывода управления программой из составного оператора, в котором данная переменная была определена. Таким образом, для локальных переменных определяется локальное время жизни, глобальные переменные существуют до тех пор, пока выполняется программа.

Однако жизнь переменной можно продолжить и после выхода из блока, для этого ее достаточно определить как переменную класса `static`. Такую возможность мы уже обсуждали ранее.

Классы памяти

Язык С позволяет описывать для переменных не только имена и типы, но и так называемые *классы памяти*, определяющие самый общий способ хранения значений переменных. Ранее мы уже рассматривали свойства класса памяти `static`. Это не единствен-

ный класс памяти. Их в языке C целых четыре, и сейчас мы рассмотрим их подробнее:

- `auto`;
- `static`;
- `register`;
- `extern`.

Прежде чем описывать свойства разных классов памяти, заметим, что эти свойства сильно зависят от уровня объявления. Возможны два варианта: во-первых, объявление может быть сделано на внутреннем уровне, т. е. внутри составного оператора или функции (а функция — это тоже составной оператор), и, во-вторых, объявление может быть сделано на внешнем уровне, т. е. за пределами какого-либо составного оператора. Поэтому рассмотрим эти две ситуации отдельно.

Объявления внутреннего уровня

Переменная, объявленная как переменная класса `auto` — это переменная с локальным временем жизни. Такая переменная видна только в том блоке, в котором объявлена. Переменные данного класса автоматически не инициализируются, поэтому первоначальные значения им необходимо придать либо в момент объявления, либо в последующих операторах. При выходе из блока значения таких переменных теряются. Если для переменной класс не указан явным образом, то он по умолчанию `auto`. Переменные этого класса называются *автоматическими*, а их значения хранятся в стеке, что и позволяет забывать их значения по выходе из составного оператора, и этот же факт ограничивает наши возможности по созданию переменных размером стека.

Переменная, объявленная как переменная класса `static` — это переменная с глобальным временем жизни. Но ее область видимости ограничена тем составным оператором, в котором она объявлена. Глобальное время жизни означает, что значение переменной сохраняется при выходе из оператора. Это же самое означает, что операция инициализации таких переменных выполняется

только один раз, в момент первого вхождения в составной оператор, и повторной инициализации уже не происходит. Здесь ограничение по памяти определяется двумя существенными факторами. Во-первых, операционной системой, под которую создан компилятор, и, во-вторых, выбранной моделью памяти для компиляторов, работающих под управлением MS-DOS.

Переменная, объявленная как переменная класса `register`, должна храниться в регистровой памяти. Это означает, что переменные данного класса обрабатываются очень быстро. Но регистровая память очень мала по объему, следовательно, вполне возможна ситуация, в которой регистровой памяти для объявляемых переменных может просто не хватить. Ситуацию с нехваткой регистровой памяти легко создать, объявив достаточно много переменных класса `register`. В случае нехватки регистров переменные класса размещаются в оперативной памяти. Регистровая память выделяется для регистровых переменных в том порядке, в котором поступают их объявления. И последнее, регистровая память выделяется только под переменные типа `int` (и совместимые с ним) и указатели, т. к. они имеют такой же размер, как и объекты типа `int`, а только такие объекты и возможно разместить в одном регистре. В отношении области видимости и времени жизни переменные данного класса идентичны переменным класса `auto`.

Еще одно отличие переменных класса `register` от переменных любого другого класса заключается в том, что к этим переменным нельзя применить операцию получения адреса. То есть запись

```
register int a=1;
int *b=&a;
```

содержит ошибку.

Переменная, объявленная как переменная класса `extern`, если она объявлена на внутреннем уровне, т. е. внутри какого-либо составного оператора есть не более чем ссылка на переменную с тем же самым именем, но описанную на внешнем уровне в любом из исходных файлов данной программы. Внутреннее объявление

используется только для того, чтобы сделать переменную, объявленную на внешнем уровне, видимой внутри составного оператора. Если на внешнем уровне не окажется переменной с таким именем, то объявленная переменная будет видима только в своем составном операторе. Пример из листинга 2.11 это иллюстрирует.

Листинг 2.11

```
#include <conio.h>
#include <iostream.h>
int a=1;
void main()
{ clrscr();
  extern a;
  cout << a;
}
```

В этой программе переменная `a`, объявленная в теле функции `main()`, не инициализирована никаким значением, однако оператор вывода даст значение 1. Если же объявление в теле функции будет записано следующим образом:

```
extern a=2;
```

то компилятор сообщит об ошибке.

Объявления внешнего уровня

Переменные, объявляемые на внешнем уровне, могут быть только двух классов: `static` и `extern`. Если класс переменной не указан, то по умолчанию он `static`. Время жизни глобальных переменных равно времени выполнения программы, область видимости для переменной класса `static` — данный файл, для переменной класса `extern` — все файлы программы. Класс памяти для функций объявляется так же, как и для переменных. Отличия выражаются лишь тем обстоятельством, что для функции единица размещения есть файл, а для переменной единица размещения есть составной оператор.

О преобразованиях типов

Если тип величины задан, то изменить его уже не представляется возможным, поэтому, когда речь идет о преобразовании типов, имеется в виду согласование типов в ходе выполнения операций над величинами. Действительно, величина, требующая двух байтов, не может быть присвоена однобайтной величине, но следующий фрагмент:

```
int a=8;
char r=a;
```

вполне может иметь место. Для языка С такое присвоение вполне законно, поэтому чтобы обеспечить корректность выполнения присвоения, необходимо что-то сделать с лишним байтом целой величины. Вполне возможна и обратная операция:

```
char a='g';
int t =a;
```

В этом случае длины символьной величины недостаточно, чтобы заполнить целую, и здесь противоположная проблема: что делать с недостающим байтом? Решение этих двух проблем — где взять недостающие байты и что делать с лишними — и есть *преобразование типов*.

В общем, правило преобразования гласит, что если длинное присваивается короткому, то младшие биты длинного отбрасываются. Если же короткое присваивается длинному, то добавляются незначащие биты так, чтобы значение величины не изменилось. Таким образом, в случае присваивания длинного короткому мы теряем истинное значение числа, в случае присваивания короткого длинному значение числа сохраняется. Детально рассматривать все нюансы преобразования типов мы здесь не будем, т. к. это самым серьезным образом зависит от реализации языка и аппаратного обеспечения, и лучше для получения дополнительной информации обращаться к руководству по реализации или справочной системе.

Преобразование типов происходит не только в момент присвоения. Поводом для операции преобразования может стать арифме-

тическая операция, в которой участвуют два операнда разного типа. Действительно, рассмотрим следующий фрагмент:

```
float a=5.6;
int t=4;
int r=67*(a+t);
```

При выполнении сложения в скобке возникает законный вопрос о типе полученной суммы. Отложить этот вопрос до выполнения окончательного присваивания нельзя, т. к. сумму необходимо сохранить в памяти. Если тип суммы должен зависеть от типов операндов (а это разумно), то возникают две возможности. Можно тип суммы привести к типу левого операнда, и тогда сумма будет равна `9.6`, или тип суммы можно привести к типу правого операнда, тогда сумма будет равна `9`. Выходит, что, привязывая тип суммы к типу левого или правого операнда, мы можем менять значение суммы при перестановке слагаемых, что, конечно же, нелепо. Поэтому в суммах и разностях за тип результирующей величины принимается наибольший (по размеру используемой памяти) тип из участвующих в операции. То же самое можно сказать и о прочих операциях — неявное преобразование осуществляется в большую сторону, но тип результата выбирается из типов, участвующих в операции операндов. Это разумное правило может, впрочем, привести к существенным неприятностям. Рассмотрим пример:

```
int a=3;
cout << 1/a;
```

Тип операндов, участвующих в делении, целый. Для величины `a` это указано явно. Для единицы это предполагается. Это означает, что тип частного будет целый, что приведет, в свою очередь, к тому, что дробная часть частного будет отброшена, и частное окажется равно нулю. В *главе 4* мы еще раз столкнемся с этим парадоксом, и там же будет показан пример решения проблемы.

Итак, преобразование типов в C может выполняться неявным образом: при присвоении, в момент выполнения арифметических операций, при передаче величин в функции (о функциях — поз-

же), и преобразование типов можно выполнить явным образом. Пример:

```
float a=6.897;
cout << int(a);
```

В этом примере показан прием явного выполнения преобразования. Больше к этому добавить нечего. Единственно, что следует заметить — операция преобразования не работает на изменение типа самой величины. Это означает, что запись `int(a)`, не участвующая ни в каком арифметическом выражении или присвоении, не законна. То, что преобразование типов никак не влияет на исходное значение, вы можете увидеть, выполнив следующий фрагмент:

```
float a=6.897;
cout << int(a)<<"\n";
cout << a;
```

Второй оператор вывода даст значение `6.897`. То есть действительно, преобразование типов носит локальный характер и используется только в данной конкретной операции, никак не затрагивая собственно переменную.

Преобразование типов выполняется как явно, так и неявно в различного рода выражениях. Это означает, что преобразование выполняется в проверке различных условий, что может привести к самым неожиданным эффектам. Вот простой пример:

```
int a=3;
float b=3;
if (a==b) cout << "Два сравниваемые числа равны"
```

Этот фрагмент программы сообщит о равенстве двух чисел. Конечно, они не равны хотя бы потому, что занимают совершенно различный объем памяти, но в момент сравнения `int` будет преобразован до `float`, и в этом типе равенство действительно имеет место.

Мы потратили довольно много времени на исследование проблемы преобразования типов. Это время потрачено не зря. Преобразование типов — это одно из узких местах для изучающих язык C,

и потраченного времени жалеть не стоит. Но прежде чем перейти к следующему пункту, рассмотрим еще два интересных эпизода.

- Эпизод первый:

```
char c; cin >> c;
if (c>='0' && c<='9') cout << "Введен символ цифры"
```

В программном фрагменте объявлена символьная величина, но в условном операторе к ней применяются операции сравнения, предполагающие целые значения, но этот фрагмент вполне законный, в силу идентичности типов `int` и `char`. Язык С для обработки символов сопоставляет их с числами, называемыми *кодами символов*. И в этой таблице кодов символы цифр стоят рядом, и именно в таком порядке 0, 1, ..., 9, поэтому если окажется, что код введенного символа находится в интервале кодов нуля и девятки, то это, очевидно, цифра.

- Эпизод второй:

```
int i;
char c='g';
i=c;
c=i;
cout << c;
```

И опять-таки, в силу идентичности типов, значение символьной величины сохранится. Обратное неверно в силу того, что тип `int` на вашей машине, скорее всего, занимает более одного байта. Продемонстрируем сказанное следующим фрагментом:

```
int i=30000;
char a;
a=i;
i=a;
cout << i;
```

Здесь значение, выданное оператором вывода, будет самым существенным образом отличаться от того, что было задано в момент инициализации.

Объявления вида *typedef*

Язык С позволяет создавать объявления следующего вида:

```
typedef int example;
```

Такое объявление говорит о том, что декларатор `example` является синонимом спецификатора типа `int`. То есть если нам потребуется объявить переменную целого типа, мы сможем записать это так:

```
example n;
```

Конечно, смысла в такой записи немного. Но если речь идет о более сложных структурах, объявления которых требуются в самых разных частях программы, то выигрыш может оказаться заметным. Например:

```
typedef struct example
{ char a[100];
  int s;
  double sum;
};
```

Повторять многократно описание такого размера, конечно, будет накладно, и в этом случае декларатор `example` существенно сэкономит текст программы. В обычном объявлении структуры можно сразу объявить переменную такого вида. В объявлениях с использованием `typedef` этого сделать нельзя. Рассмотрим пример:

```
typedef struct example
{ char a[100];
  int s;
  double sum;
} newtype;
```

В обычном объявлении слово `newtype` было бы понято компилятором, как объявление переменной, имеющей тип описанной выше структуры. В данном случае это точно такой же декларатор, как и слово `example`, а, следовательно, возможно и следующее описание:

```
typedef struct
```

```
{ char a[100];
  int s;
  double sum;
} newtype;
```

Здесь опять только одно слово, являющееся декларатором. Это newtype, и именно оно и будет синонимом объявления структуры.

Декларатор, объявленные в typedef, играют роль спецификаторов типа, а это означает, что следующая запись является ошибкой:

```
typedef int mytype;
long mytype a;
```

Спецификатор типа не включает в себя объявление класса памяти. А это в свою очередь означает, что следующее объявление будет неверным:

```
typedef static int my;
my a;
```

И если есть потребность указать класс памяти, то это необходимо сделать следующим образом:

```
typedef int my;
static my a;
```

Заключение

Если говорить только о базовых типах, то язык С предоставляет не такой уж большой выбор для программиста. Фактически, на самом нижнем уровне можно говорить о существовании только числовых типов, но возможности комбинирования типов, в особенности возможности работы с различными указателями (о них специальная глава), с лихвой компенсируют бедность базовых типов.

Некоторую сложность для начинающего программиста может создать понятие классов памяти, правила определения времени жизни, области видимости и связанные с этим правила инициализации. Но здесь ничего не поделаешь, язык создавался для про-

фессионалов, а не для легкого обучения. Хорошее понимание этих правил придет с практикой и даст огромный выигрыш в возможности управлять памятью и тратить ее ровно столько, сколько необходимо на самом деле.

И последнее, в отношении величин язык C различает две различные сущности — это объекты, т. е. участок памяти, доступный для обработки, и так называемое L-значение, т. е. выражение, ссылающееся на объект. Самым простым примером L-значения является идентификатор. Некоторые операции могут в качестве своего результата выдавать L-значения. Например, если E — выражение указанного типа, то *E является выражением L-значения, ссылающимся на объект E.

Глава 3

Ввод/вывод

Форматный ввод/вывод

Прежде всего, необходимо сказать, что средства ввода/вывода не являются составной частью языка. Все, что мы будем рассматривать в этой главе — это операции, поддерживаемые специальными библиотеками, разработанными для конкретных реализаций языка С.

Язык С предлагает два способа организации ввода и вывода. Это форматный ввод/вывод и неформатный. Термин "формат" предполагает, что функции ввода/вывода передается не только список величин, но и описание внешнего вида результата операций ввода/вывода. В формат могут быть введены поясняющие строки, описатели формата данных, определяющие, сколько на данное предусмотрено символов, если же это число, то какого оно типа и, опять же, сколько в нем знаков, сколько знаков до десятичной точки, сколько после.

Функции *форматного ввода* и *вывода* имеют следующий синтаксис:

- функция вывода

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

- функция ввода

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

Обе функции требуют описания формата, что же касается аргументов, то функция вывода требует непосредственно имен переменных и значений величин, в то время как функция ввода — адресов переменных. Вопрос, почему так, смысла не имеет. Такова особен-

ность реализации. Ядро формата — это описатели, в табл. 3.1 приведен их список.

Таблица 3.1. *Описатели формата*

Символ	Ожидаемый ввод	Формат вывода
d	Целое	Целое десятичное со знаком
i	Целое	Целое десятичное со знаком
o	Целое	Беззнаковое восьмеричное целое
u	Целое	Беззнаковое десятичное целое
x	Целое	Беззнаковое шестнадцатеричное целое с цифрами (a, b, c, d, e, f)
X	Целое	Беззнаковое шестнадцатеричное целое с цифрами (A, B, C, D, E, F)
f	Действительное	Число со знаком в форме `[-]dddd.dddd`
e	Действительное	Число со знаком в форме `[-]d.dddd` или `e[+/-]ddd`
e	Действительное	То же, что и e. В экспоненциальной форме
c	Символ	Одиночный символ
s	Указатель на строку	Печатаются символы до тех пор, пока не будет достигнут терминальный ноль

В примере из листинга 3.1 вводятся две переменные, находится их сумма и выводится на экран. И для ввода и для вывода используется только один описатель — описатель целого числа.

Листинг 3.1

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

```
  scanf("%d%d",&a,&b);
  int sum=a+b;
  printf("Summa %d",sum);
}
```

Следующий пример демонстрирует, что значение числа не связано жестко с числовой системой (листинг 3.2). Число можно ввести как десятичное и использовать его как десятичное, а в выводе определить его формат как-то иначе.

Листинг 3.2

```
#include <stdio.h>
#include <conio.h>
void main(void)
{ clrscr();
  int r=29997;
  printf("Десятичное %d Восьмеричное %o Шестнадцатеричное %x",
          r,r,r);
}
```

Функция `printf` выводит одно и то же число в трех различных форматах: как десятичное число, как восьмеричное и как шестнадцатеричное. Необходимо, конечно, помнить, что формат вывода и формат представления числа — это не одно и то же. Само число представлено, как десятичное со знаком. Попробуйте заменить

```
int r=29997;
```

на

```
int r=-29997;
```

и вы увидите разницу.

Обратите также внимание, что в вывод допустимо кроме спецификаций формата включать символьные строки, что важно для понятного оформления результата. Если часть знаков выводимого числа не нужна, то спецификация формата позволяет, не меняя самого числа, вывести только необходимые знаки. Пример:

```
printf("%4.2f",a);
```

Число a выводится в формате числа с плавающей десятичной точкой, при этом общая длина числа — 4 знака, и после запятой мы увидим два знака.

Функции printf и scanf — универсальные функции. Кроме них язык C дает программисту неплохой набор специализированных функций, используя которые можно не тратить время на описание формата. Имена специализированных функций получаются от общего корня добавлением букв, обозначающих типы обрабатываемых данных. Подробное описание этих функций мы здесь приводить не будем, т. к. они хорошо описаны в справочной системе компилятора. А в качестве небольшого примера запишем вызов функции cprintf:

```
cprintf("Пример ввода и вывода на языке C");
```

Здесь просто выводится строка символов.

Потоковый ввод/вывод

Второй способ организации ввода/вывода в C называется *потоковым*. С точки зрения потоковых операций все данные можно рассматривать как поток байтов, без какой-либо структуры. Поток может быть входным, и байты из него поступают в переменные, и поток может быть выходным, и байты в него поступают из переменных. Бесструктурность потоков означает отсутствие потребности в описании формата. Все вводится и выводится совершенно единообразно.

Поток ввода управляется командой cin, поток вывода управляется командой cout. Примеры использования потоковых команд можно посмотреть в *главе 1*. Здесь заметим только, что количество вводимых структур для одного объявления ввода или вывода не ограничено. Поэтому вполне допустимы следующие записи:

```
cout << a << b << c << d;
cin  >> a >> b >> c >> d;
```

Интересный пример вывода

Программа из листинга 3.3 в нескольких вариантах выводит значение переменной, увеличенной на 1. Но как ни странно, результат работы операторов вывода существенно отличный. Первый вывод даст числа 49 и 50, второй вывод даст еще более интересный результат — 51.

Листинг 3.3

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  int a=49;
  cout << a++ << "  " << a+1 << "  ";
  cout << a;
}
```

Попробуем это объяснить. Прежде всего, отметим, что разница в результатах обусловлена различием в приоритетах операций увеличения и операции вывода. В процессе первой операции вывода сначала будет выполнен собственно вывод, выполнение операции ++ окажется отложенным. При этом вычисление выражения `a+1` не будет отложено. Это означает, что выражения `a++` и `a+1` несмотря на внешне одинаковый результат имеют принципиально различную природу. Первое есть операция, второе есть выражение. После вывода будет выполнено увеличение ++, и следующий оператор вывода даст уже значение 51.

Интересный пример ввода

В этом примере ввод организован настолько компактно, что все необходимые операции умещаются в выражении, играющем роль условия (листинг 3.4). Почему это возможно?

Листинг 3.4

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  int n=0,a[5];
  while (cin>>a[n++],n!=5);
  for (n=0;n<=4;n++)
    cout << a[n] << " ";
}
```

Первая причина успеха в том, что операция ввода имеет более высокий приоритет, чем увеличение справа, поэтому первым делом вводится значение элемента массива с индексом, равным текущему значению переменой `n`, и только затем значение `n` увеличивается на единицу. Вторая причина — в запятой. Выражение в скобках сложное, что вполне допускается языком C, но значение выражения может быть все равно только одно, поэтому простые выражения, перечисленные через запятую, вычисляются слева направо, и результатом всего сложного выражения считается крайнее правое. Поэтому операция ввода, несмотря на то, что реально выполняются все операции, на результат сложного выражения не оказывает влияния. Этот результат формируется выражением `n!=5`, поэтому пока `n` меньше 5, цикл продолжает работать и выполняет операцию ввода.

Несколько полезных функций

Функции *putchar()* и *getchar()*

Функции `putchar()` и `getchar()` — функции ввода/вывода низкого уровня. Функция `putchar()` занимается тем, что за один шаг своей работы отправляет на устройство вывода один символ,

функция `getchar()` наоборот забирает один символ из буфера ввода. Их точный синтаксис следующий:

```
void putchar(int)
int getchar()
```

Обратите внимание, что несмотря на название, говорящее о том, что функции работают с символами, обрабатываемые ими данные — числовые. Это отражает факт эквивалентности символов и чисел на нижнем уровне ввода/вывода. Для С визуальное представление символов символами не отражает их реального представления в недрах машины числом. Пример, приведенный в листинге 3.5, хорошо иллюстрирует сказанное.

Листинг 3.5

```
#include <conio.h>
#include <stdio.h>
void main(void)
{ clrscr();   int c;
   while ((c = getchar()) != '\n')
      printf("%c", c);
}
```

Функции *textcolor()* и *textbkcolor()*

Назначение этих функций — определение цвета символов и цвета фона. Пример приведен в листинге 3.6.

Листинг 3.6

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  for (int i=1;i<=15;i++)
   { textcolor(i);textbackground(i+1);
```

```
      cout << "example string" << "\n";
      cprintf("example string");
    }
}
```

Обратите внимание, что задание цветов никоим образом не воздействует на потоковый вывод. Вывод функцией `cprintf` по существу делает то же самое, что и потоковый, но определение цветов воздействует на результат его работы.

Функция *gotoxy()*

Функция `gotoxy` определяет позицию курсора на экране. Функция требует два аргумента: координату `x` и координату `y`. В листинге 3.7 в качестве иллюстрации представлена программа, печатающая таблицу умножения.

Листинг 3.7

```
#include <conio.h>
#include <stdio.h>
void main()
{ clrscr();
  for (int i=1;i<=9;i++)
   for (int j=1;j<=9;j++)
    {gotoxy(i*3,j*2);printf("%d",i*j);}
}
```

Команда перевода курсора на следующую строку

В операциях ввода/вывода как потоковых, так и форматных часто используются Esc-последовательности, иногда их еще называют *управляющими символами*. Наиболее часто применяемая коман-

да — это `\n`, команда перевода на следующую строку. Пример ниже демонстрирует ее использование:

```
printf("Вывод очередного значения %d\n",a);
cout << a <<"\n";
```

Этим мы завершим главу. Конечно, ввод/вывод не ограничивается клавиатурой, как устройством ввода, и экраном монитора, как устройством вывода. Роль таких устройств могут играть файлы, но файловым операциям посвящена отдельная глава *(см. главу 12)*.

Глава 4

Операции в C

Список операций

Язык C предлагает большой набор арифметических операций. Его основа, конечно, обычные арифметические операции:

- `*` — умножение;
- `/` — деление;
- `+` — сложение;
- `-` — вычитание.

Арифметические выражения строятся по обычным математическим правилам. Умножение и деление имеют более высокий приоритет, чем сложение и вычитание, операции в скобках имеют более высокий приоритет, чем за скобками.

Существует универсальная операция присваивания `=`, которой принципиально достаточно для построения любого присвоения. Но для удобства C предлагает целый набор дополнительных операций, использование которых делает выражение несколько короче. Полный перечень всех операций C приведен в табл. 4.1.

Таблица 4.1. *Операции в C*

Операция	Действие
`=`	Простое присваивание
`+=`	Вычисление выражения, записанного в правой части с последующим суммированием

Таблица 4.1 *(продолжение)*

Операция	Действие
`-=`	Вычисление выражения, записанного в правой части с последующим вычитанием
`*=`	Вычисление выражения, записанного в правой части с последующим умножением
`/=`	Вычисление выражения, записанного в правой части с последующим делением
`%=`	Вычислить остаток и присвоить
`<<=`	Сдвинуть влево и присвоить
`>>=`	Сдвинуть вправо и присвоить
`&=`	Выполнить операцию И, а затем присвоить
`|=`	Выполнить включающее ИЛИ, а затем присвоить
`^=`	Выполнить исключающее ИЛИ, а затем присвоить
`++`	Приращение до или приращение после
`--`	Уменьшение до или уменьшение после
`+`	Унарный плюс (также сложение)
`-`	Унарный минус (также вычитание)
`*`	Умножение
`/`	Деление
`%`	Вычисление остатка
`<`	Меньше
`<=`	Меньше или равно
`>`	Больше
`>=`	Больше или равно
`==`	Равно
`!=`	Не равно
`&`	Побитовое И
`^`	Побитовое исключающее ИЛИ
`|`	Побитовое включающее ИЛИ

Таблица 4.1 *(окончание)*

Операция	Действие
`&&`	Логическое И
`||`	Логическое включающее ИЛИ
`!`	Логическое отрицание
`?:`	Арифметический `if`
`::`	Разрешение области видимости
`.`	Выбор члена
`->`	Выбор члена
`.*`	Выбор члена
`->*`	Выбор члена
`[]`	Индексация
`()`	Построение выражения или вызов функции
`Sizeof`	Размер объекта или типа
`&`	Адрес объекта
`*`	Разыменование
`New`	Создание, размещение объекта
`Delete`	Уничтожение объекта (освобождение области памяти)
`delete[]`	Уничтожение массива
`()`	Преобразование типа
`<<`	Сдвиг влево
`>>`	Сдвиг вправо
`,`	Следование

Операции, связанные с динамической памятью, т. е. имеющие смысл выбора члена, создание или уничтожения объекта, мы достаточно подробно рассмотрим далее, в *главе 9*, посвященной указателям. Действие операции `::` разрешения видимости рассмотрено ранее в *главе 2*, посвященной величинам. Арифметикой мы тоже уже занимались во всех предыдущих главах. Сейчас только выясним, зачем языку С так много арифметических операций и

какая между ними разница? Затем более детально поговорим о битовых и логических операциях.

Арифметические операции

Арифметические операции отличаются степенью специализации. Это означает, что в ряде случаев они эквивалентны между собой. К примеру, операция `s=s/2` эквивалентна операции `s/=2`. Операция `a=a+sin(b)` эквивалентна операции `a+=sin(b)`. Операция `a=a+1` эквивалентна `a++`. Как уже было сказано ранее, любую арифметическую операцию можно выполнить универсальным присвоением. Но специализированные операции могут сэкономить место и ресурсы, поэтому пренебрегать ими не следует.

Операции приращения и уменьшения обозначены как операции до или операции после. Это означает, что их можно выполнять разными способами. Можно записать `a++` и можно записать `++a`, обе записи законны и дадут одинаковый результат, если речь идет только о приращении. Но если операция приращения будет комбинироваться с другими операциями, то результат может зависеть от их последовательности. Программа, приведенная в листинге 4.1, показывает некоторые особенности.

Листинг 4.1

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int a=2;
/* Можно было бы ожидать, что величина b окажется равна 3,
но этого не произойдет, так как первая слева операция -
это простое присваивание. Поэтому величине b будет присвоено
исходное значение величины a, и лишь затем величина a будет
увеличена на единицу. */
  int b=a++;
  int c=5;
```

```
/* Здесь опять-таки первая операция - это операция приращения.
Поэтому величина c будет увеличена на 1 и лишь затем умножена
на 3. */
  ++c*=a;
  cout<<b<<" "<<a<<" "<<c;
}
```

Операция %

Нахождение остатка можно выполнить операцией div (целочисленное деление). Функция div позволяет найти сразу и остаток, и целую часть. В примерах нашей книги div встречается достаточно часто. Ее синтаксис

```
div_t имя_переменной = div(делимое, делитель)
```

Возвращаемый результат имеет структурный тип, содержащий два поля. Синтаксис структуры таков:

```
tyedef struct
 { long int rem;
   long int quot;
 } div_t;
```

Пример

```
div_t a=div(11,3);
cout <<a.rem<<" "<<a.quot;
```

Используя div, мы убиваем двух зайцев и сразу получаем и целую часть от деления, и остаток. Но если есть необходимость только в остатке, то операция % выглядит проще. Пример:

```
cout << 11%3;
```

Обычные правила арифметики несколько нарушаются операциями преобразования типов. Проблема преобразования типов заключается в ответе на следующий вопрос: предположим, выполняется арифметическая операция, аргументы которой величины разных типов. Какого типа будет результат?

Ответ на данный вопрос не безразличен для результата. В тексте программы, приведенной в листинге 4.2, трижды вычисляется

одна и та же сумма. В расчете суммы `s1` единица делится на целое число. Результат также является целым числом, что для всех `i`, больших 1, дает значение выражения `1/i`, равным нулю. В расчете суммы `s2` перед выполнением деления к величине `i` прибавляется ноль, но это действительный ноль, в результате чего выражение `(i+0.0)` не меняет значения, но меняет тип с целого на действительный, частное также становится действительным числом, что меняет значение суммы. В расчете суммы `s3` величина `t` равна единице, но это действительная единица. Действительная единица, деленная на целое число, также дает действительное число.

Листинг 4.2

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  float s1=0,s2=0,s3=0,t=1;
  for (int i=1;i<=10;i++)
   { s1=s1+1/i;
     s2=s2+1/(i+0.0);
     s3=s3+t/i;
   }
  cout << s1<<"\n";
  cout << s2<<"\n";
  cout << s3;
}
```

Операции сдвига

Операций сдвига в нашем распоряжении две: сдвиг вправо и сдвиг влево. Чтобы понять механизм действия этих команд, прежде всего, необходимо вспомнить, что где-то в глубине машины числа имеют двоичное представление, где каждый разряд — это бит. Битовые операции позволяют что-то делать с битовым пред-

ставлением числа. Две рассматриваемые сейчас операции изменяют двоичное представление, смещая биты слева направо или наоборот справа налево и одновременно добавляя ноль. Конечно, применяются операции к числам, могущим иметь различное представление, и результат будет выражен в той же системе счисления, в которой представлено исходное число, но выполняются операции над двоичным представлением.

Рассмотрим пример из листинга 4.3. В программе применяется битовый сдвиг числа a на количество битов, равное числу b (т. е. на один бит). В трех командах вывода печатается сначала исходное число, равное 100, затем то, что получится в результате сдвига влево, а потом то, что получится в результате сдвига вправо. Число a в двоичном представлении равно 1100100. После сдвига влево получится 11001000, что равно 200. А после сдвига вправо — 110010, что равно 50. Заметьте, что сдвиг равнозначен делению или умножению на 2 (при условии, что не теряются значащие единицы). Это означает, что часть операций деления может быть выражена более быстро операциями сдвига.

Листинг 4.3

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int a=100,b=1;
  cout <<a<<" ";
  cout <<(a<<b)<<" ";
  cout <<(a>>b);
}
```

В приведенном примере значение, полученное после операции сдвига, не присваивается, а сразу уходит в поток вывода. Но в таблице операций есть и операции сдвига с присвоением. Мы их рассматривать не будем, т. к. делают они, в сущности, то же самое. Обратите также внимание, что операция сдвига записывается

теми же знаками, что и операция вывода. Различие между ними видно из примера, но ошибиться легко. Если, например, убрать скобки, то обе операции будут распознаны компилятором, как операции вывода.

Битовая логика

Три операции, рассмотренные далее, есть *операции логические*, выполняемые по правилам математической логики, но по битам (см. таблицу значений логических операций — табл. 4.2). Поэтому, чтобы получить результат, необходимо десятичные числа перевести в двоичную систему счисления и выполнить операцию с каждой парой соответствующих разрядов. В примере, рассмотренном далее, операции выполняются с числами 123 и 156. Их двоичное представление: 123 = 1111011; 156 = 10011100. Выполним битовые операции с двоичным представлением.

Таблица 4.2. *Битовые операции*

Описание	Биты							
Число a	0	1	1	1	1	0	1	1
Число b	1	0	0	1	1	1	0	0
&	0	0	0	1	1	0	0	0
|	1	1	1	1	1	1	1	1
^	1	1	1	0	0	1	1	1

Результат битовых операций в десятичном представлении будет следующим:

```
123 & 156 = 11000 = 24
123 | 156 = 11111111 = 255
123 ^ 156 = 11100111 = 231
```

И именно эти значения и будут результатом работы программы (листинг 4.4).

Листинг 4.4

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int a=123,b=156;
  cout <<a<<" ";
  cout <<(a&b)<<" ";
  cout <<(a|b)<<" ";
  cout <<(a^b);
}
```

Логические операции

Операции `!`, `&&`, `||` отличаются от битовых тем, что они рассматривают свои аргументы как логические. Но поскольку в языке С нет логического типа, как например в языке Паскаль, то логика С в качестве аргументов использует обычные арифметические выражения, при этом ложным значением считается то, в выражении которого все разряды нули, в противном случае это истина. Таким образом `123 && 156 = 1`, т. к. после побитового И в выражении-результате стоят 2 единицы. Точно так же `123 || 156 = 1`, т. к. в выражении-результате есть единицы. Существенно изменяет значение, пожалуй, только операция отрицания: `!0 = 1` и наоборот `!1 = 0`.

Различие между логическими и битовыми операциями хорошо видно на следующем примере:

```
int a=1,b=2;
int c=a&b,d=a&&b;
cout <<c<<"  "<<d;
```

В результате выполнения этого фрагмента величина `c` получит значение ноль, а величина `d` — значение единица. Совокупно все логические операции можно показать в одной таблице. Будем обозначать истинное значение 1, а ложное 0 (табл. 4.3).

Таблица 4.3. *Логические операции*

a	b	A&&b	A||b	!a
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Операция следования

В программах C *операция следования* применяется в самых различных случаях, когда необходимо перечислить что-либо. Простейший пример применения следования — это объявления. Например:

```
char t='g',k;
```

Еще один любопытный пример следования приведен в программе из листинга 4.5.

Листинг 4.5

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  int a=0,b=45,c=10;
  a++,b++,c+=a+b;
  cout <<c;
}
```

Здесь перечислены три операции. Так как выполняются они последовательно, то переменная c будет суммирована со значениями переменных a и b, к которым уже была применена операция увеличения на единицу.

Если с помощью операции следования построено сложное выражение, результат которого должен быть присвоен переменной

оператором присваивания, то конечным результатом сложного выражения будет самое левое. Например, результатом присваивания `a=1,2,3` будет единица.

Следование вполне может применяться и в различных операторах. Например, следующая запись синтаксически вполне верна:

```
if (y>1, y<10) y=0;
```

Конечно, смысл такой записи с учетом того, что результатом сложного выражения будет результат последнего простого, не ясен, но компилятор против такого выражения не возразит, и в процессе работы программы это выражение будет обработано. Ниже приведен более осмысленный пример, показывающий, как из правила следования можно извлечь пользу:

```
if (t=y>1,y<10) y=0;
```

В этом примере в первой части условия вычисляется значение переменной, которое можно использовать в дальнейшем. Вычисляется оно вне зависимости от того, чем закончится результат проверки условия.

Операции индексации и построения выражений

Операция индексации часто связывается с понятием массива. Это не совсем точно. Массив — лишь частный случай использования индексации. Общий случай применения индексации можно описать так: пусть выделена некоторая область памяти, и в ней сохранена линейная последовательность объектов одинаковой структуры; тогда операция индексации позволяет обратиться к элементу этой последовательности по номеру. Это определение для тех, кто имеет опыт программирования, например, на языке Паскаль, также выглядит как определение массива, но прочитав главу об указателях *(см. главу 9)*, вы, наверное, согласитесь, что массив — это более узкое понятие.

Операция построения выражения используется для выделения выражения, в котором операции выполняются в том порядке, в каком следуют в соответствии с их приоритетами.

Операция определения размера объекта *sizeof*

Любой объект, размещенный в памяти, имеет определенный размер, и этот размер можно определить, воспользовавшись операцией `sizeof`. Далее приведены несколько примеров. Первый из них — в листинге 4.6.

Листинг 4.6

```
#include <conio.h>
#include <iostream.h>
void main(void)
{ clrscr();
  int a;
  float b;
double c;
  int e[100];
  cout << sizeof(a) << " " << sizeof(b) << " " << sizeof(c) <<
          " " << sizeof(e);
}
```

На выходе будет напечатано

```
2  4  8  200
```

что соответствует конкретным величинам. Операцией определения размера вполне можно воспользоваться для определения размера типа, даже без указания конкретной величины. Это можно сделать вот так:

```
int n=sizeof(double)
```

В результате величина `n` будет равна 8, что соответствует длине типа `double` в байтах. То, что `sizeof` понимает под объектом, следует понимать буквально. Рассмотрим для примера следующий фрагмент:

```
int *w;
w=new int[200];
```

```
cout <<sizeof(w);
```

В этом фрагменте командой

```
w=new int[200];
```

определен массив длиной в 200 символов, но если вы ожидаете в качестве результата увидеть размер массива, то вы самым серьезным образом ошибаетесь, т. к. объектом, размещенным в памяти, в этом случае будет не массив, а указатель на массив, а размер указателя и размер массива, на который показывает указатель, — это совсем не одно и то же. И `sizeof` не может определить размер функции. Хотя функция C в значительной степени обладает свойствами объекта, и ее адрес можно, например, передать указателю, на `sizeof` это не распространяется.

Арифметическое условие

Синтаксис:

<*выражение_1*> ? <*выражение_2*> : <*выражение_3*>

Здесь <*выражение_1*> должно иметь целочисленный тип, тип с плавающей точкой или быть указателем. Это выражение анализируется на предмет равенства его нулю. Вычисление выполняется следующим образом:

- если <*выражение_1*> имеет ненулевое значение, то вычисляется значение <*выражение_2*>, и это значение является значением условной операции;
- если <*выражение_1*> имеет нулевое значение, то вычисляется значение <*выражение_3*>, и это значение является значением условной операции.

В примере из листинга 4.7 организовано вычисление знакопеременного ряда вида: 1 – 2 + 3 – ...

Листинг 4.7

```
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
void main(void)
```

```
{ clrscr();
  int sum=0;
  for (int i=1;i<=3;i++)
    sum+=(div(i,2).rem)? i:-i;
  cout <<sum;
}
```

Требования на выражения, составляющие арифметическое условие, не очень строги, поэтому в записи вполне могут присутствовать функции и логические выражения, которые в C имеют также целый тип результата.

Старшинство и порядок вычисления

В приводимой табл. 4.4 сведены правила старшинства и ассоциативности всех операций, включая и те, которые мы еще не обсуждали. Операции, расположенные в одной строке, имеют один и тот же уровень старшинства; строки расположены в порядке убывания старшинства. Так, например, операции `*`, `/` и `%` имеют одинаковый уровень старшинства, который выше, чем уровень операций `+` и `-`.

Таблица 4.4. *Приоритет операций*

Операции	Порядок вычисления
`()`, `[]`, `->`, `.`	Слева направо
`!`, `\`, `^`, `++`, `--`, `-`, `(type)`, `*`, `&`, `sizeof`	Справа налево
`*`, `/`, `%`	Слева направо
`+`, `-`	Слева направо
`<<`, `>>`	Слева направо
`<`, `<=`, `>`, `>=`	Слева направо
`==`, `!=`	Слева направо
`&`	Слева направо

Таблица 4.4 *(окончание)*

Операции	Порядок вычисления
^	Слева направо
&&	Слева направо
?:	Справа налево
=, +=, -=	Справа налево
,	Слева направо

Еще несколько замечаний о порядке выполнения операций

Если для вас язык программирования С не первый, то ваше внимание следует обратить на следующую особенность — в список операций попал знак присваивания. Для многих языков, например Фортран, Бейсик, Паскаль, да и многих других присваивание — это оператор. В С это операция. Из чего следует, например, что такое выражение, как записанное ниже

```
a=s=d=f=9;
```

совершенно нормально с точки зрения С. В этом выражении четырем величинами будет присвоено значение 9.

Все, что мы рассмотрели ранее, — это равноправные по отношению друг к другу операции, которые можно комбинировать практически произвольным образом, при этом забота о смысле записанной длинной операции целиком возлагается на программиста. Бездумное комбинирование может привести к весьма неожиданным побочным эффектами. Рассмотрим следующий пример:

```
A[I]=I++;
```

Выполнение данного фрагмента в принципе возможно двумя способами:

- значение `I` может быть увеличено, затем присвоено элементу массива;

- значение `I` может быть сначала присвоено элементу массива и лишь затем увеличено.

Кроме того, порядок выполнения присвоения и увеличения изменяет не только присваиваемую величину, но и элемент массива, к которому применяется присвоение.

В каком именно порядке будут выполнены операции, самым серьезным образом зависит от реализации компилятора, поэтому использование такого вида записей резко ухудшает возможность переноса программ с одной реализации на другую. Такого же рода проблема возникает при следующем вызове функции:

```
Example(a++)
```

Совершенно неясно, передастся ли функции увеличенное значение или первым шагом будет осуществлен вызов функции, и лишь затем увеличено значение переменной. По всей видимости, этот вопрос также решается лично компилятором и, следовательно, самым существенным образом зависит от реализации. Проблема порядка вычислений в C существенно осложняется способностью различных операций вкладываться и сочетаться друг с другом. Это создает как интересные возможности для более краткой записи, так и побочные эффекты, вызывающие трудно обнаруживаемые ошибки. Поэтому если вы строите сложное выражение, будьте предельно внимательны, возможность краткой записи может оказаться медвежьей услугой неопытному программисту.

Глава 5

Операторы языка C

Общие сведения

Оператор — это конструкция, управляющая выполнением программы. Собственно, именно благодаря операторам программа может иметь нелинейный вид и не сводиться к перечню простых операций. Типов операторов в языке C, как и в любом другом, только три: операторы цикла, операторы выбора и линейные операторы. Впрочем, может быть, есть смысл выделить отдельную группу вспомогательных операторов, таких, которые с натяжкой можно отнести к одной из трех групп, но можно выделить и отдельно. Такими операторами в C является оператор прерывания `break`, оператор продолжения `continue` и оператор перехода `goto`. Их вполне можно рассматривать как средства организации выбора или цикла, но такая разноплановая функциональность наводит на мысль, что более точно их следовало бы назвать вспомогательными.

Циклы

Язык C, как и любой другой достаточно развитый язык, предлагает несколько способов организации циклов. Конструкции циклов по своим основным свойствам можно характеризовать как циклы с параметром или циклы по условию, а также как циклы с предусловием (т. е. такие, в которых условие цикла проверяется перед выполнением тела цикла) и как циклы с постусловием (т. е. условие цикла проверяется после выполнения тела цикла).

Однако необходимо заметить, что название "цикл с параметром" — не совсем точное название для той конструкции, которую дает в наше распоряжение С. Те, кто имеет опыт программирования на языках Бейсик или Паскаль, очень быстро заметят разницу. Классический цикл с параметром — скорее, частный случай того, что предлагает язык С. Этот частный случай мы уже рассматривали в *главе 1*. Напомним этот пример:

```
for ( int i=1;i<=N;i++) cin >> A[i];
```

Здесь есть тело цикла, состоящее из одной команды ввода, и заголовок, в котором записано начальное значение параметра, его конечное значение и шаг изменения. Это действительно не более, чем частный случай, потому как трем частям заголовка присвоен конкретный смысл, привязанный к термину "параметр". Чтобы лучше понять сказанное, рассмотрим синтаксис:

```
for ([<выражение1>]; [<выражение2>]; [<выражение3>])
  <команда>
```

Здесь <*команда*> выполняется до тех пор, пока <*выражение2*> не примет значение 0 (в смысле логического значения).

<*выражение1*> вычисляется перед выполнением первого шага цикла. Обычно оно используется для инициализации цикла, но это лишь частный случай.

После каждого шага цикла вычисляется <*выражение3*>. Обычно это выражение используется для изменения параметра цикла, но и это не обязательно.

Обратите внимание, что все три выражения заключены в квадратные скобки `[]`. Это означает, что все три выражения не являются обязательными и даже цикл `for (;;;)` с полностью пустым заголовком будет синтаксически правильным. Такое обобщенное понимание компонентов цикла дает большую свободу. Например, выражения, отвечающие за инициализацию и изменения в ходе работы цикла, могут быть сложными. Например, такими:

```
for (i=0, t=50; i < 40 && t>0; i++, t--)
```

Как видите, здесь две переменные играют роль параметров, и условие продолжения работы выглядит достаточно сложным.

Конечно, в большинстве реальных ситуаций указанные усложнения конструкции не нужны, и вполне можно обойтись частной конструкцией цикла с параметром. Но иногда свобода в построении конструкции позволяет немного сэкономить на командах и сделать программу немного более компактной.

Рассмотрим пример из листинга 5.1.

Листинг 5.1

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int n;
  cin >>n;
  for (int i=1,s=0;i<=n;s+=i,i++);
  cout << s;
}
```

Показанный пример считает сумму чисел. При этом все вычислительные операции скрыты в заголовке цикла. Это, конечно, не уменьшает количество необходимых команд, но делает программу чуточку изящнее. В более сложном примере выгода, наверное, была бы более видна.

> ***Примечание об инициализации***
>
> В синтаксисе оператора сказано, что первое из выражений вычисляется один раз до выполнения первого шага цикла, хотя это выражение не обязательно константа. Оно вполне может содержать переменные величины, чьи значения, возможно, изменяются в теле цикла. Но изменение таких переменных на инициализирующее выражение не оказывает никакого влияния.

Пример:

```
int n=1;
for (int i=n;i<=100;i++)
  { sum+=i;
     n++;
  }
```

Данный цикл будет выполняться 100 раз, несмотря на то, что переменная n, участвующая в инициализации выражения, на каждом шаге увеличивается. Напротив, переменные, участвующие в иных выражениях заголовка цикла, оказывают на его работу самое существенное влияние. К примеру, следующий цикл при выполнении зависнет:

```
int n=100;
for (int i=1;i<=n;i++)
  { sum+=i;
     n++;
  }
```

т. к. величина n, являющаяся верхней границей параметра, на каждом шаге цикла увеличивается одновременно с параметром, вследствие чего значение параметра никогда не достигнет значения n, и, соответственно, условие никогда не станет ложным.

Цикл с предусловием

Синтаксис:

```
while <условие>  <команда>
```

Здесь <*команда*> может быть как одной командой, так и составным оператором. <*условие*> — любое арифметическое выражение, значение которого воспринимается как логическое. Как уже было сказано в начале главы, тело цикла этой формы выполняется после проверки условия, следовательно, если перед первым шагом цикла условие окажется ложным, тело цикла не будет выполнено ни одного раза. Цикл работает до тех пор, пока условие истинно. Напомним, что условие (так, как его понимает С) — это арифметическое выражение, следовательно, условием может быть почти все, что угодно, в том числе и сложное выражение, состоящее из нескольких элементарных, перечисленных через запятую (об операторе следования см. далее в этой главе).

Цикл с постусловием

Синтаксис:

```
do
{ <последовательность_команд>
}
while <условие>
```

Цикл с постусловием также работает, пока истинно условие. Но тело цикла выполняется до проверки условия. Следовательно, данная форма цикла предпочтительна, если тело необходимо выполнить хотя бы один раз независимо от результата проверки условия, или если проверять условие до отработки команд цикла нет смысла. Так бывает, если, например, входящие в условие величины формируются в теле цикла. Но в значительной степени все три формы организации цикла решают одни и те же задачи и, следовательно, вполне взаимозаменяемы. Предпочтение же одной формы перед другой — это вопрос небольшого удобства и, зачастую, личного стиля.

Для демонстрации сказанного приведем программу уже решенной задачи — расчета суммы *N* чисел, но решим ее в трех вариантах (листинг 5.2).

Листинг 5.2

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int n;
  cin >>n;

  int s=0;
  for (int i=1;i<=n;i++) s=s+i;
  cout << s<< " ";

  s=0;i=1;
```

```
  while (i<=n)
   { s=s+i;
     i++;
   }
  cout << s<<"  ";

  s=0;i=1;
  do
   { s=s+i;
     i++;
   }
  while (i<=n);
  cout <<s;
}
```

Обратите внимание, как при замене цикла `for` на условные циклы расписываются части его заголовка. Инициализация переменной, играющей роль параметра цикла, проводится до заголовка. Увеличение этой переменной выполняется в теле цикла по завершении всех прочих операций, и только условие продолжения остается на старом месте, в заголовке.

Из сказанного следует, что принципиально три формы цикла не нужны. Любая из них может решить все проблемы, связанные с организацией циклов, но иногда использование циклов по условию делает текст немного изящнее.

Пусть, к примеру, есть некоторый цикл, выполнение которого требует проверки сложного условия, организованного в отдельной функции. Тогда следующий текст:

```
while (proverka()) тело_цикла
```

будет выглядеть немного лучше, чем такой:

```
for (;proverka();) тело_цикла
```

Во втором варианте есть лишние знаки, но, конечно, это в общем вопрос вкуса и стиля.

О структуре цикла

Тело любой конструкции цикла — это составной оператор. На перечень команд составного оператора язык С не накладывает практически никаких ограничений, а следовательно, циклы могут вкладываться друг в друга. В листинге 5.3 приведен пример организации двух циклов, распечатывающих таблицу умножения. Этот пример мы уже использовали в *главе 3* для демонстрации ввода/вывода.

Листинг 5.3

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int a,b;
  for (a=1;a<=9;a++)
   for (b=1;b<=9;b++)
    { gotoxy(3*a,3*b);
      cout << a*b;
    }
}
```

Две переменные, играющие роль параметров цикла, одновременно являются множителями и координатами в таблице умножения. Составной оператор в программе создан только для второго, внутреннего цикла. Это правильно, т. к. два оператора, организующие вывод произведения на экран, являются телом именно второго цикла, а не первого. Тело первого цикла состоит только из заголовка внутреннего цикла и, следовательно, в организации составного оператора не нуждается.

Частая ошибка, совершаемая при организации вложенных циклов, заключается в определении одной и той же переменной в качестве параметра. Поясним это на примере цикла `for`.

```
for (a=1;a<=9;a++)
```

```
for (a=1;a<=9;a++)
 {
 }
```

Если программист ожидает, что данная конструкция будет выполняться 81 раз, то он самым серьезным образом ошибается. Оба цикла — и внешний, и внутренний — завершают свою работу тогда, когда переменная a достигает своего максимального значения. Отсюда следует, что оба цикла завершат свою работу после первого же шага внешнего цикла. Действительно, на первом шаге внешнего цикла внутренний отработает до конца, и, следовательно, переменная a достигнет максимума, а это условие завершения не только внутреннего, но и внешнего цикла. Далее приведен еще один пример, где также одна и та же переменная используется как для внешнего, так и для внутреннего циклов, но указанной проблемы уже нет.

```
for (a=1;a<=9;a++)
 { int a;
   for (a=1;a<=9;a++)
    {
    }
 }
```

В данной программе есть одно принципиальное отличие. Имена параметров у внешнего и внутреннего циклов опять одинаковые, но сами переменные уже различны. Для внутреннего цикла объявлена новая переменная, являющаяся локальной для составного оператора, в каковом заключен внутренний цикл. Такой прием действительно решает проблему, но проще все же для каждого из вложенных циклов использовать переменную с новым, неповторяющимся именем.

Замечания о цикле с постусловием

Фигурные скобки для цикла с постусловием также не обязательны, как и для цикла с предусловием. После ключевого слова do вполне можно записать только один оператор. Пример — в листинге 5.4.

Листинг 5.4

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  int i=0;;
  do
    cout <<i++<<" ";
  while (i<10);
}
```

Первое замечание. В программе в одном операторе сразу и выводится текущее значение переменной, и выполняется увеличение этого значения. Но в отличие от цикла с предусловием игнорирование фигурных скобок приведет не к смысловой ошибке, а к ошибке синтаксиса, которая будет замечена компилятором. Таким ошибочным не компилируемым фрагментом будет нижеследующий:

```
do
  cout <<i<<" ";
  i++;
while (i<10);
```

Второе замечание. Компилятор для ключевого слова `do` ищет соответствующее ключевое слово `while` там, где завершается составной оператор, содержащий тело цикла. Рассмотрим следующую конструкцию:

```
do
{  ........
   ........
   while (условие);
}
while (<условие>);
```

Внутреннее ключевое слово `while` не является завершением цикла, начатого ключевым словом `do`, даже несмотря на то, что у него нет собственного тела.

Конструкции выбора

Условный оператор

Синтаксис:

```
if (<условие>) <оператор1>; [else <оператор2>;]
```

В случае истинности проверяемого условия выполняется <*оператор1*>, в случае его ложности выполняется <*оператор2*>. Ключевое слово `else` не обязательно. Вполне допустима конструкция

```
if (<условие>) <оператор1>;
```

Простые примеры применения конструкции `if` были приведены в *главе 1*. Сейчас рассмотрим более сложную ситуацию. Предположим, необходимо присвоить некоей переменной (пусть ее имя `b`) значение 2, но только в том случае, если другая переменная (пусть ее имя `d`) равна 1, и в то же время третья переменная (например, `r`) равна 5. Это условие возможно записать так:

```
if (d==1)
  if (r==5) b=2;
```

Приведенный пример показывает, что оператор, следующий за условием, вполне может оказаться еще одним оператором выбора. И цепочка таких вложений операторов выбора может оказаться практически сколь угодно длинной. Правда, необходимо заметить, что в данном случае построение вложенной конструкции неразумно. Совершенно такой же результат даст нижеследующая конструкция:

```
if (d==1 && r==5) b=2;
```

Но объединение условий посредством логических связок необходимо использовать осторожно. Не всегда это работает так просто. С появлением <*оператора2*>, выполняемого в случае ложности условия, ситуация существенно усложняется. Рассмотрим для иллюстрации следующую конструкцию:

```
 if (d==1)
  if (r==5) b=2;
  else b=3;
```

О чем говорит эта конструкция? О том, что для определения значения b величина d обязательно должна иметь значение 1. В противном случае наша конструкция из двух вложенных условий вообще не окажет никакого влияния на вычисление величины b. А если переменная d все-таки равна 1, то тогда выбор между двумя значениями осуществляется в зависимости от значения величины r. Если же мы вложенный выбор заменим одним со сложным условием, то смысл проверяемого условия существенно изменится.

```
if (d==1 && r==5) b=2;  else b=3;
```

Новое выражение говорит о том, что выбор между двумя значениями будет сделан обязательно в зависимости от того, ложно или истинно сложное условие. Здесь величина d не имеет особенной роли, она равноправна с величиной r. В то время, как в предыдущей конструкции при величине d, не равной единице, значение r не имеет никакого значения.

Вероятно, уже понятно, что ключевое слово else принадлежит ближайшей конструкции if. Далее для иллюстрации приведем чуть более сложный пример:

```
if (d==1)
   if (r==5) b=2;
   else b=3;
else b=4;
```

Здесь отступами показана сопринадлежность if и else. В этом примере значение величины b определяется так: если d не равно единице, то b=4. Если же равно, то значение b определяется из сравнения r==5. Если r равно 5, то выполняется b=2. Если же нет, то выполняется b=3.

Поставим теперь такую задачу. Пусть переменная d также играет определяющее значение и таким же образом участвует в определении величины b. Пусть также необходимо, чтобы при r=5 переменная b принимала значение 2. А при r<>5 величина b пусть имеет то же самое значение, которое она имела до входа в конст-

рукцию выбора. Иными словами, необходимо выбросить `else`, входящий во внутреннюю конструкцию. То есть сделать так:

```
if (d==1)
   if (r==5) b=2;
else b=4;
```

Но это совсем не то, что нам нужно. В таком варианте запись

```
else b=4;
```

автоматически встает на место уничтоженной и становится частью внутреннего условного оператора, а нам необходимо, чтобы эта запись продолжала оставаться частью внешнего оператора. Выход из положения таков:

```
if  (d==1)
  { if (r==5) b=2; }
else b=4;
```

Внутренняя конструкция условного оператора превращается в составной оператор, а запись `else b=4;` остается составной частью внешней конструкции выбора.

И последнее, и <*оператор1*>, и <*оператор2*>, указанные в синтаксисе, — это один, но возможно составной оператор.

А прежде чем мы перейдем к следующей конструкции, рассмотрим интересный пример совмещения условного оператора и оператора присваивания:

```
if (t=(y>1 && y<10)) y=0;
```

Здесь сразу выполняются два действия. Во-первых, проверяется условие, и в зависимости от результата выполняется или не выполняется присваивание после условия, и, во-вторых, непосредственно в условии вычисляется значение переменной `t`, которое можно использовать в дальнейшем. Если бы присвоение нельзя было использовать в условии, то тот же программный фрагмент был бы записан так:

```
t=(y>1 && y<10);
if (t) y=0;
```

Оператор-переключатель

Далее рассмотрим еще один оператор организации выбора, называемый оператором-переключателем. Начнем с примера работающей программы (листинг 5.5).

Листинг 5.5

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int n;
  cin >>n;
  switch (n)
  { case 1: cout << "Первый";
    case 2: cout << "Второй";
    case 3: cout << "Третий";
  }
}
```

Значение переменной n, указанной в круглых скобках после ключевого слова switch, определяет точку передачи управления. Ключевые слова case с указанным числовым значением — это точки, в которые управление передается. Работает конструкция так: оператор определяет значение величины n, находит точку, значение которой равно значению n, и передает управление на найденную точку. Далее выполняются все операторы от точки, на которую было передано управление, и до закрывающей скобки оператора-переключателя, если выполнение не прерывается каким-либо образом. Следовательно, при n=1 программа выполнит все три оператора вывода. При n=2 два последних, а при n=3 только один последний. Сам оператор-переключатель не дает возможности выполнить только группу операторов, непосредственно определенных за точкой передачи управления. Но это

можно сделать, воспользовавшись оператором прерывания. Вот так:

```
switch (n)
  { case 1: cout << "Первый";break;
    case 2: cout << "Второй";break;
    case 3: cout << "Третий";
  }
```

Новая конструкция case сработает несколько иначе. А именно при n=1 отработает только первый оператор вывода, при n=2 — только второй, и, соответственно, при n=3 в сравнении с предыдущей программой ничего не изменится. После третьего оператора вывода оператор break не записан, т. к. на этом этапе работа конструкции case завершается.

Два рассмотренных выше примера должны были дать достаточное представление о сущности конструкции выбора. Рассмотрим точное описание ее синтаксиса.

```
switch (<выражение>)
{
  case <константа>: <последовательность_команд>
  [default: <последовательность_команд>]
}
```

Здесь <*выражение*> — это арифметическое выражение, значение которого сравнивается с константами. Ключевое слово default необходимо для пометки последовательности команд, выполняемой в том случае, когда значение выражения не совпало ни с одной константой. default не является необходимой частью конструкции, и если оно будет опущено, то при отсутствии подходящей константы в конструкции case не будет выполнено никаких команд. Для иллюстрации напишем программу, выясняющую, был ли введен символ-цифра или какой-либо иной (листинг 5.6).

Листинг 5.6

```
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
void main(void)
{ clrscr();
  char c=getch();
  switch (c)
  { case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
    case '0': cout << "Цифра";break;
    default:  cout << "Символ";
  }
}
```

Конструкция `case` принципиально отличается от конструкции `if`. Если `if` — это действительно выбор из двух альтернатив, то `case` не сводится к увеличению количества альтернатив. `Case` позволяет организовать как выбор альтернатив, используя оператор `break`, так и их объединение.

Оператор *break*

Использование данного оператора уже рассматривалось ранее, но посвятим ему еще немного времени.

Оператор прерывает выполнение самого внутреннего из объемлющих его операторов `do`, `for`, `switch`, `while`. Управление пере-

дается оператору, следующему за тем оператором, который располагается за прерванным.

Рассмотрим пример программы, представленной в листинге 5.7.

Листинг 5.7

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  for (int i=1;i<=10;i++)
   { for (int j=1;j<=10;j++)
      { cout <<j<<" ";
        if (i==j) break;
      }
     cout <<"\n";
   }
}
```

В данном примере оператор `break` прерывает только внутренний цикл, поэтому внешний цикл отрабатывает все 10 положенных ему шагов, а время работы внутреннего цикла удлиняется с увеличением величины `i`. Наиболее часто данный оператор применяется в операторе-переключателе `switch` для того, чтобы обеспечить выполнение только одной альтернативы.

Составной оператор

О данном операторе уже говорилось, и примеры его работы много раз использовались. *Составной оператор* — это группа операторов, заключенных в фигурные скобки. Составной оператор — основная структурная единица, из которой строится большая программа. Функция также представляет собой составной оператор.

Как правило, составной оператор используется в качестве тела других операторов, например, таких как `do`, `for`, `if`. Независимое использование составного оператора возможно, но это вряд ли действительно может понадобиться.

Составные операторы могут вкладываться друг в друга, но не могут пересекаться. Конечно, говорить о пересечении собственно составных операторов нет смысла. Но надо помнить, что составной оператор, как правило, оказывается привязанным к тому или иному ключевому слову, образует вполне определенную конструкцию, и в этих ситуациях говорить о пересечении уже есть смысл. Ниже пример ошибочной конструкции:

```
do
{
   while(условие)
      {
}
while (условие);
      }
```

Здесь очевидное пересечение двух составных операторов, и этот фрагмент не получится даже откомпилировать.

Оператор продолжения *continue*

Данный оператор позволяет проигнорировать группу операторов, если в их выполнении уже нет необходимости. Используется внутри циклов `for`, `do`, `while`. Действие его заключается в передаче управления на следующий шаг цикла. Программа из листинга 5.8 иллюстрирует действие оператора.

Листинг 5.8

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  for (int i=1;i<=10;i++)
   { for (int j=1;j<=10;j++)
      { if (i!=j) continue;
        int s=i*j;
```

```
        cout <<s<<" ";
      }
   }
}
```

Действие `continue` в данной программе приводит к тому, что двумя циклами считаются десять квадратов натуральных чисел, т. к. `continue` не позволяет полностью выполнять составной оператор (тело внутреннего цикла), если параметры циклов не равны.

Необходимо заметить, что оператор используется только в теле цикла, запись его в ином месте будет воспринята компилятором как синтаксическая ошибка. Кроме того, прерывается выполнение всех команд, следующих от местоположения `continue` до завершения составного оператора, в котором `continue` записан. Рассмотрим следующий фрагмент:

```
for (i=1;i<=10;i++)
 { {continue;}
    cout <<i;
 }
```

Здесь оператор `continue` заключен в дополнительный составной оператор, но это не мешает ему прервать выполнение именно того составного оператора, который является телом цикла. Если в дополнительном составном операторе записаны какие-либо действия после `continue`, то они также будут проигнорированы, но только лишь потому, что составляют тело цикла.

Оператор-выражение

В языке С выражение означает немного больше, чем присваивание. *Выражениями* также являются вызовы функций. Выражение может быть записано и без присваивания, например, так:

```
void main()
{ int n;
  n+1;
}
```

Данная программа синтаксически правильна, но бессмысленна. Однако существуют ситуации, в которых вычисление выражения без присваивания имеет смысл. Такой ситуацией, например, будет передача значения выражения в вызове функции или расчет выражения при проверке условия.

Оператор *goto*

Это оператор, которым следует пользоваться очень аккуратно. Его чрезмерное участие в программе может сделать логику слишком запутанной и непрозрачной. Как правило, его можно убрать из программы, немного изменив ее структуру. С этим оператором тесным образом связано понятие *метки*. А именно оператор передает управление на оператор с меткой. В примере из листинга 5.9 показано, как это делается.

Листинг 5.9

```
#include <iostream.h>
void main()
{ int sum=0;
  for (int i=1;i<=10;i++)
   { sum+=i;
     if (i==5) goto metka;
   }
  cout <<"Вывод";
  metka:
  cout <<sum;
}
```

В результате действия оператора `goto` вычисление суммы прекращается несколько раньше, и выполнение передается на оператор, следующий за идентификатором `metka`. Оператор `goto` не может передавать управление в другую функцию. Передача управления внутрь сложного оператора вполне допустима, но если в сложном операторе есть инициализирующие объявления,

то такая передача может привести к плохо предсказуемым последствиям.

Оператор `goto` может быть использован для организации циклов. Рассмотрим, например следующую конструкцию цикла:

```
int sum=0,i=1;
do
 { sum+=i;
   i++;
 }
while (i<10);
```

Этот фрагмент с использованием оператора `goto` можно переписать следующим образом:

```
int sum=0,i=1;
metka:
sum+=i;
i++;
if (i<10) goto metka;
```

Оба фрагмента дадут один и тот же результат, но первый более верен с точки зрения стиля.

Пустой оператор *;*

Точка с запятой в языке C является оператором, который не выполняет никаких действий.

Оператор возврата *return*

Данный оператор завершает выполнение функции. Возврат осуществляется в момент выполнения `return`, при этом в точку вызова возвращается значение, указанное после `return`. Если никакое значение не указано, то возвращаемое значение является неопределенным. Операторов `return` в функции может быть несколько; если их нет, то возвращаемое значение будет неопределенным.

Каких-либо ограничений на размещение оператора возврата нет, но в следующем примере (листинг 5.10) компилятор выдаст предупреждение о возможной проблеме с возвратом. Компилятор посчитает, что размещение оператора не гарантирует возврат.

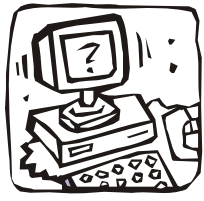
Листинг 5.10

```
#include <conio.h>
int example(int a)
{ if (a<10) return 1; else return 2;
}
void main()
{ clrscr();
  int r=example(5);
}
```

Конечно, на самом деле возврат гарантирован, это следует из элементарного логического анализа текста условия, но компилятор не занимается анализом хода выполнения программы, поэтому совершенно нормальная ситуация будет воспринята, как возможность ошибки. Если функцию записать вот так:

```
int example(int a)
{ if (a<10) return 1; else return 2;
  return 0;
}
```

что совершенно бессмысленно, то компилятор будет вполне удовлетворен.



Глава 6

Константы

Простые константы

Константы — это объекты, значение которых не может быть изменено в ходе работы программы. Константа — достаточно обычное понятие, хорошо известное из математики. Но в программировании термин "константа" понимается несколько шире, чем в математике. Именно поэтому ранее было сказано, что константа — это объект, а не просто число. Язык C предоставляет в распоряжение программиста числовые константы, а также символьные и строковые.

Говоря о константах, необходимо разделять два немного различных понятия: это *именованная константа*, т. е. имеющая имя и значение, и *константа-значение*, т. е. просто число, просто символ или просто строка. Просто число или символ может, например, появиться в выражении, неименованная строковая константа может появиться, например, в операции вывода.

```
cout << "Пример строковой константы";
```

В основном глава посвящена именованным константам. Числовые константы могут быть десятичными, восьмеричными и шестнадцатеричными. Десятичные числа в нашей книге появляются на каждом шагу, поэтому ограничимся примерами восьмеричных и шестнадцатеричных чисел.

Восьмеричная константа начинается цифрой ноль. Примеры: `0567`, `0433`, `01232`.

Шестнадцатеричная константа начинается символами `0x` или `0X`. Примеры: `0x5643`, `0x6757`, `0xab454f`, `0X6757`, `0XAB657F`.

Если шестнадцатеричная константа начинается с `0x`, то буквенные символы записываются в нижнем регистре, если же константа начинается с `0X`, то буквенные символы записываются в верхнем регистре. Заметим, что представление константы в той или иной системе счисления не мешает программе выполнять арифметические операции. Перевод из одного представления в другое выполняется без участия программиста. Пример:

```
cout <<12+03+0xa12;
```

Константы этого фрагмента заданы в трех системах счисления. Результат их сложения равен `2593` в десятичной системе счисления. Это число мы и увидим.

В листинге 6.1 приведен пример программы, в которой созданы 4 константы: целое число, действительное число, символ, строка символов.

Листинг 6.1

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  const int a=45;
  const float b=6.7;
  const char r='g';
  const char e[]="12345";
  cout <<a<<" "<<b<<" "<<r<<" "<<e;
}
```

Особое внимание в этом примере обратите на объявление строковой константы. Квадратные скобки обязательны. Строка символов в C — это фактически символьный массив, поэтому константа и объявляется как символьный массив. Указывать же длину этого массива как раз не обязательно. Так как это константа, то ее длина

вполне может быть определена на этапе компиляции. Указание типа является обязательным.

В этом фрагменте речь идет уже о специально объявленных константах. Как видно из примера, они получаются добавлением к обычному объявлению переменной ключевого слова `const`. Это небольшое добавление запрещает любые операции с константой, изменяющие ее значение. Поэтому следующий фрагмент не будет даже откомпилирован:

```
const int n=1;
n=2;
```

Значение константы определяется не в процессе работы программы, а в процессе компиляции, в том месте, где выполняется инициализация величины. Поэтому компилятор и определяет эту ситуацию как ошибочную. Он способен заметить, что величина определяется дважды. Два следующих примера также ошибочны:

```
const int n;
```

Компилятор заметит, что для константы не определено значение. Сделать это впоследствии также нельзя. Следующий фрагмент также ошибочен:

```
const int n;
n=1;
```

В данном фрагменте значение определено единожды, но не там, где это допустимо. Повторимся, точка инициализации — это единственное место, где возможно определить значение константы. Если в объявлении константы не указан тип, то тип по умолчанию будет `int`.

> ***Примечание***
>
> В примере из листинга 6.1 символьная константа заключена в апострофы, а строковая — в кавычки. Это принципиальная разница. Будьте внимательны с этими знаками. Вроде бы одна и та же запись `'A'` и `"A"` на самом деле означают разные вещи. Первая запись — это символ, а вторая — строка, с которой можно, например, выполнять операцию конкатенации или добавить терминальный ноль (о терминальном нуле *см. в главе 10*).

Рассмотрим пример из листинга 6.2.

Листинг 6.2

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  const a=7;
  const b=8.89;
  const c='f';
  cout <<a<<" "<<b<<" "<<c;
}
```

В программе объявлены три константы, и видимо по мысли программиста они имеют три различных типа: целый, действительный и символьный. Типы не указаны, но что имелось в виду, видно из определяемых значений. Однако компилятором будет выполнено преобразование типов, и мы увидим в выводе три целых числа.

Выражение, инициализирующее константу, вполне может быть сложным. Причем в формировании такого выражения могут участвовать как числовые неименованные константы, переменные, так и именованные константы. Пример из листинга 6.3 это демонстрирует.

Листинг 6.3

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  int a=2;
  const int r=6;
  const int w=2+a*r;
  cout <<w;
}
```

Можно сформулировать три правила описания констант:

- ❒ значение любого типа можно использовать как константу, если задать ему имя и добавить ключевое слово `const`;
- ❒ для описание множества целых констант допустимо использовать перечисление;
- ❒ имя массива является константой.

Последнее утверждение требует небольшого пояснения. Рассмотрим любое объявление массива, например такое: `int a[100]`. Невозможность изменения этого имени означает, что в квадратных скобках нельзя использовать переменную величину, т. е. нельзя написать так:

```
int a[n];
```

Строковые константы можно разрывать пропуском, переносом на другую строку или даже комментарием, но в процессе выполнения такие строки окажутся сцепленными. Например, результатом такого объявления:

```
const char a[]="123456"
"7890"
```

окажется строка `"1234567890"`.

В строку можно поместить и *терминальный ноль*, например, вот так:

```
const char a[]="123""\0""45677"
```

Но, скорее всего, ваша программа будет считать, что за нулем нет больше символов. Обратите внимание, как вставлен в строку-константу терминальный ноль. Если это сделать вот так:

```
const char a[]="123\045677"
```

то тот же самый терминальный ноль будет рассматриваться компилятором как самый обычный символ. Терминальный ноль, появившийся в строке константы, мешает использованию всей строки символов, но он не мешает определению. Все символы, указанные в определении, останутся в строке, и к ним можно обратиться по индексу. Пример из листинга 6.4 это демонстрирует.

Листинг 6.4

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  const char a[]="12345""\0""6789";
  cout <<a<<"\n";
  for (int i=0;i<10;i++)
   cout <<a[i];
}
```

Цикл вывода выдаст все символы, включенные в строку, не взирая на терминальный ноль.

Объявление массива-константы может выглядеть, например, так:

```
const int w[]={1,2,3,4,5}
```

В этом примере создан массив из пяти чисел. Еще раз отметим важную особенность констант: все они должны быть инициализированы, т. е. должны получить свое значение в момент объявления. Это также следствие запрета на изменение значения. Фактически ключевое слово `const` изменяет тип величины. Это означает, что при объявлении константы не только указывается размещение, но и ограничивается способ ее использования.

Перечисления

Перечисления можно рассматривать как способ объявления констант. Действие перечисления ограничено целыми числами, и существует две возможности объявления: во-первых, можно не указывать значения перечислителей, тогда по умолчанию первый перечислитель будет равен 0, второй — 1 и т. д. Во-вторых, значения перечислителей можно задать явным образом. В примере из листинга 6.5 показаны обе возможности.

Листинг 6.5

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  enum example1{a1,a2,a3};
  cout <<a1<<" "<<a2<<" "<<a3<<"\n";
  enum example2{a4=100,a5=200,a6=300};
  cout <<a4<<" "<<a5<<" "<<a6;
}
```

Константы, данные в перечислении, — это обязательно целые числа, поэтому перечисление следующего вида:

```
enum example{f=6,h=8.9};
```

будет незаконным. Константа h объявлена ошибочно.

Ранее уже было сказано, что константа перечисления — это обязательно целое число. Участие символьных констант в перечислении возможно по той причине, что символьный тип, т. е. тип char, и целый тип в языке C совместимы. Следующее перечисление для компилятора не содержит ошибок:

```
enum {a='g',b=9};
```

Однако первый перечислитель в момент использования все равно будет использован как целое. Целые константы перечисления — это целое со знаком. Следующее перечисление вполне законно:

```
enum {a=-1;b=4};
```

Как уже было упомянуто, по умолчанию значения перечислителей начинаются с нуля и изменяются с шагом 1. Если для какого-либо из перечислителей указано явное значение, то следующий получит значение на единицу больше. Пример:

```
enum example{a=5,b,c=10,d};
```

Значения неявных перечислителей следующие: b=6 и d=11.

Заметьте, что после ключевого слова enum может быть указано имя перечисления, в примерах, представленных ранее, мы ис-

пользовали для именования перечисления идентификатор example. Если перечисление объявлено так, то его имя становится именем нового типа. Но именовать перечисление совсем не обязательно. Даже без имени объявленные в перечислении константы вполне доступны для использования. Если же перечисление имеет имя, то это имя становится именем нового типа. Пример из листинга 6.6 иллюстрирует применение такого типа данных.

Листинг 6.6

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  enum example {a=56,b=12,c=78};
  example r=a;
  r++;
  cout<<r<<"\n";
  example t=9;
  cout <<t;
}
```

В программе создан тип-перечисление example. После чего объявлены две переменные этого типа. Переменная r объявлена корректно, и ее значение есть константа a с числовым значением, равным 56. Однако сама переменная r не является константой, и ее вполне можно изменять, что и сделано следующим оператором. Поэтому в выводе мы увидим не 56, а 57. Следующее объявление не вполне корректно. Но компилятор Turbo C не обнаружит ошибки, из-за которой стоило бы прервать компиляцию. Как максимум, вам будет выдано сообщение о том, что, возможно, не все здесь правильно, но программа будет откомпилирована и выполнена.

Изменять же значение самих перечислителей, конечно, нельзя. Это константы, и операции над ними необходимо проводить с выполнением правил, предписанных для констант.

Глава 7

Массивы

Общие правила работы с массивами

Данная глава посвящена массивам — важнейшему из составных типов данных. Новое понятие может быть получено из математического понятия множества, если добавить к нему нумерацию. То есть массив можно определить как множество пронумерованных элементов одинаковой природы. Одинаковость природы элементов массива — принципиальное требование любого языка программирования, в том числе и языка С. Нельзя, чтобы часть элементов имела тип целый, а часть — тип символьный. Или все целые, или все символьные, или все элементы числа с плавающей точкой и т. д. Это важнейшее условие создания и использования массивов.

Синтаксис объявления массива таков:

Тип <*имя_массива*>[*верхняя_граница_индекса*]

Индекс изменяется от нуля до верхней границы минус единица. Таким образом, если дано объявление

```
int a[3];
```

это означает, что задан массив целых (двухбайтных) чисел с номерами: 0, 1, 2. Элемент `a[3]` уже выходит за определенные границы.

Есть возможность определить *многомерный массив*. Для этого достаточно определить несколько индексов. Вот так: `float mas[5][12]` — это двумерный массив, в котором первый индекс изменяется до четырех и второй до 11.

В листинге 7.1 приведен пример программы, выполняющей следующие действия:

1. Вводится целочисленный массив.
2. Положительные числа переписываются в массив b.
3. Отрицательные числа переписываются в массив c.

Листинг 7.1

```
#include <conio.h>
#include <iostream.h>
void main()
{ int a[100],b[100],c[100],na,nb=-1,nc=-1;;
  clrscr();
  cin>>na;
  for (int i=0;i<na;i++) cin >>a[i];
  for (i=0;i<na;i++)
   if (a[i]>0) {b[++nb]=a[i];} else {c[++nc]=a[i];}
  cout<<"\n";
  for (i=0;i<=nb;i++) cout <<b[i]<<" ";
  cout <<"\n";
  for (i=0;i<=nc;i++) cout <<c[i]<<" ";
}
```

В программе заданы три массива целого типа и зарезервирована память на 100 элементов для каждого. Реально можно ввести и меньшее количество чисел. Больше нельзя. Но память, выделенная под 100 элементов, будет недоступна для других данных независимо от того, нужен этот массив или нет. Поэтому массив — очень полезная структура, но и очень затратная по памяти. Планируя использование массива, вы должны точно указать верхнюю границу, и это обязательно должна быть константа. Заметьте, что благодаря собственным индексам, введенным для вспомогательных массивов, память как бы экономится, т. к. массивы b и c используются частично. Но это кажущаяся экономия. Ведь неиспользованные элементы объявленных массивов вернуть в свободную

память нельзя. Для реальной экономии памяти надо было бы определить количество отрицательных и положительных чисел и уже потом объявлять массивы, но этого сделать нельзя. Попытка обойти это ограничение следующим образом:

```
int n;
n = реально необходимое число;
int mas[n];
```

будет отвергнута компилятором.

Такое ограничение может показаться лишним, но механизм определения статического массива позволяет часть проблем по проверке корректности текста программы переложить на компилятор, что для неопытного программиста очень полезно. К тому же есть возможность создания динамических массивов, свободных от указанного ограничения, но такую возможность мы рассмотрим в *главе 9*, посвященной указателям.

А сейчас еще один пример программы обработки одномерного массива (листинг 7.2), а затем поговорим немного о массивах многомерных.

Листинг 7.2

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int mas[100],n;
  cin >>n;
  for (int i=0;i<=n;i++) cin >>mas[i];
  for (i=0;i<n;i++)
   for (int k=0;k<n-i;k++)
     if (mas[k]>mas[k+1])
      { int c=mas[k];
        mas[k]=mas[k+1];
        mas[k+1]=c;
```

```
      }
  for (i=0;i<=n;i++) cout <<mas[i]<<"  ";
}
```

Приведенная программа реализует алгоритм пузырьковой сортировки. Это широко известный и наиболее простой алгоритм сортировки массива. Техника работы с многомерными массивами принципиально не отличается от техники работы с одномерным массивом, поэтому ограничимся одним содержательным примером (листинг 7.3).

Листинг 7.3

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int mas[10][10];
  for (int i=1;i<=9;i++)
   for (int k=1;k<=9;k++)
    mas[i][k]=i*k;
  for (i=1;i<=9;i++)
   for (k=1;k<=9;k++)
    { gotoxy(i*3,k*3);
      cout << mas[i][k];
    }
}
```

Написанная программа строит таблицу умножения (мы ее уже строили, но без использования массивов). Для небольшого удобства проигнорирован нулевой элемент, и построение матрицы начинается с первого. Видно, что никаких особенных языковых тонкостей с появлением дополнительных размерностей не появляется, но, конечно, задачи с использованием сложных массивов логически могут быть весьма непростыми.

Мы сейчас не ставим цели изучения логики программ, использующих массивы, наша цель проще — изучение языковых особенностей и, конечно, обойти особенности инициализации массивов нельзя. Программа из листинга 7.4 показывает технику инициализации одномерного и двумерного массивов.

Листинг 7.4

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int a[5]={1,2,3,4,5};
  for (int i=0;i<=4;i++)
   cout <<a[i]<<" ";
  cout <<"\n";
  int b[3][3]=
  { {1,2,3},
    {4,5,6},
    {7,8,9}
  };
  for (i=0;i<=2;i++)
   for (int j=0;j<=2;j++)
     cout <<b[i][j]<<" ";
}
```

Пример достаточно ясен, добавить к нему почти нечего, если обрабатываются числовые массивы. Если же массив символьный, то необходимо учесть тот факт, что символьные массивы используются в C для представления строк, а для строк есть небольшое соглашение, требующее завершать строку нультерминальным символом `\0`. Поэтому инициализация

```
char a[3]={'q','w','e'}
```

с точки зрения компилятора будет синтаксически правильной, но инициализировав массив таким образом, программист создает

себе проблемы в будущем. Значительно точнее было провести инициализацию так:

```
char a[4]={'q','w','e','\0'}
```

Для значений же числовых массивов ограничений нет. Величины, прописанные в фигурных скобках, могут быть как целыми, так и действительными, со знаком и без знака.

> ***Примечание***
>
> В примере для инициализации многомерного массива используются фигурные скобки. Необходимо заметить, что фигурные скобки — это часто не более, чем дань удобству и прозрачности процесса инициализации.

Без фигурных скобок вполне можно обойтись, и два следующих описания совершенно идентичны:

```
int a[2][2]=
{ {1, 2},
   {3, 4}
};
```

и

```
int a[2][2]=
{1, 2, 3, 4};
```

Но, конечно, первое описание немного предпочтительнее хотя бы потому, что оно более наглядно. Однако фигурные скобки в процессе инициализации играют совсем не косметическую роль. Например, следующая попытка инициализации ошибочна:

```
int a[2][2]=
{{1,2,3},
 {4}
};
```

Если бы скобки были нужны только для красоты, то это объявление ничем бы не отличалось от предыдущих. Однако компилятор выдаст сообщение, что инициализаторов слишком много. Это означает, что компилятор за счет инициализаторов первой скобки

пытается заполнить первый массив. Это не удается, т. к. есть лишние числа, и возникает состояние ошибки.

Многомерные массивы в языке С, если говорить точно, — это массивы массивов. То есть объявление `int a[5][5]` читать, как двумерный массив, будет не верно. Правильнее сказать, что объявлено пять массивов по пять элементов в каждом. Такая разница в прочтении — совсем не игра слов. Если мы говорим о многомерном массиве, то имеется в виду, что как единое целое выступает только один объект — это весь двумерный массив, состоящий из элементов. Объявляя массив массивов, мы объявляем не один целостный объект (многомерный массив), а массив целостных объектов (одномерные массивы), а это уже большая разница.

Разницу легко обнаружить, если попытаться воспользоваться операциями, предполагающими работу с целостными объектами, например, операцией `sizeof`. Напомним, что эта операция вычисляет объем памяти, занятый объектом. Поэтому аргументом этой операции может быть только целостный объект, а, следовательно, если одномерный массив, находящийся в составе многомерного, не является целостным объектом, то он не может быть аргументом для `sizeof`, а если он все же таков, то может. Программа, записанная в листинге 7.5, показывает, что целостным объектом является любой массив в составе массива большей размерности.

Листинг 7.5

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  int a[10][15][3];

  cout << sizeof(a)<<"\n";
  cout << sizeof(a[1])<<"\n";
```

```
  cout << sizeof(a[1][2])<<"\n";
}
```

На основании этого примера можно выдвинуть общее утверждение, что массив с N размерностями в языке С есть массив массивов размерности $N - 1$. Однако это не означает, что к массиву как объекту можно применять любые операции. Понимание целостности массива в языке С ограниченно. Например, по отношению к массивам не определена операция присваивания, следовательно, если a и b — два массива, то их присвоение a=b невозможно.

Все примеры демонстрировали свойства массивов простых типов. Но определение массива не ограничено простым типом. Язык С предоставляет возможность создать массив любого типа, в том числе и структурного. В листинге 7.6 показан пример объявления такого массива, и этот же пример демонстрирует технику инициализации.

Листинг 7.6

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  struct example{char a;int b;};
  example mass[2]=
   { {'q',1},
     {'h',2}
   };
  cout<<mass[1].a<<" "<<mass[1].b;
}
```

Первая фигурная скобка (значения, записанные в ней) инициализирует нулевую запись массива, а вторая фигурная скобка инициализирует, соответственно, вторую запись.

Инициализация массива символов

Для компактности иллюстраций будем рассматривать только двумерные массивы. Такой массив можно определить так:

```
char a[значение_границы1][значение_границы2]
```

и инициализировать его так же, как и все прочие массивы. Но можно использовать тот факт, что для C символьный массив — это строка. Вернитесь к *главе 6*: там можно увидеть объявления следующего вида:

```
const char a[]="строка символов"
```

Если убрать ключевое слово `const`, то оставшаяся часть будет не чем иным, как объявлением массива, и, как видите, его инициализацию можно выполнить целой строкой. Точно так же можно поступить и с двумерным массивом. Пример из листинга 7.7 это демонстрирует.

Листинг 7.7

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  char a[3][20]=
  { "Первая строка",
    "Вторая строка",
    "Третья строка"
  };
 for (int i=0;i<3;i++)
  cout<<a[i]<<"\n";
}
```

Этот же пример еще раз показывает, что многомерный массив в C — это массив массивов. Именно поэтому мы имеем возможность в цикле вывода обращаться к одномерным массивам, являющимся элементами двумерного, как отдельным объектам.

И более того, при этом реальная длина строк вполне может быть различной, что не доставит нам никаких дополнительных хлопот.

Заключение

Мы еще раз вернемся к понятию массива в *главе 9*, посвященной указателям, т. к. понятие массива и понятие указателя связаны самым тесным образом. Если говорить точно, то имя массива — это и есть указатель на область памяти, в которой записаны его значения. Поэтому рассказ о массивах и рассказ о указателях можно было бы и объединить, но т. к. с понятием указателя вообще в языке С связано очень много, а не только массивы, то указателям посвящена отдельная глава, в которой для массивов отведено особое место.

Глава 8

Структуры и объединения

Общие сведения

Программистам очень часто встречаются задачи, для решения которых было бы удобно объединять под одним именем данные различных типов. Необходимость этого следует из природы некоторых задач, поэтому любой язык, если только он претендует на профессионализм, должен предоставлять такие возможности. Язык С — это язык профессионального программирования, поэтому естественно, что он располагает средствами для такого объединения. А именно С предоставляет разработчику две конструкции с несколько различными свойствами. Во-первых, это структуры и, во-вторых, это объединения. И *структуры*, и *объединения* строятся на первый взгляд одинаково. Для их описания необходимо задать имя и перечень полей, являющихся структурами данных. Различие заключается в том, что все поля структуры равноправны, и к каждому из них можно получить доступ. Объединение же ставит своей целью не полноценное представление данных, а экономию памяти, и различные поля в объединении существуют только как несовместимые альтернативы. Задав объединение, мы получаем возможность обращения только к одному из полей.

А теперь подробнее.

Структуры

Рассмотрим пример (листинг 8.1).

Листинг 8.1

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  struct man{int value;char name;};
  man my;
  my.value=45;
  my.name='a';
  cout <<my.name<<"="<<my.value;
}
```

В этом примере переменная `my` содержит в себе две величины: величину целого типа с именем `value` и величину символьного типа с именем `name`. Для доступа к этим величинам необходимо указать общее имя переменной, под которым они объединены, в данном случае это имя `my`, а затем через точку — имя переменной, к которой необходим доступ.

Работе со структурой должно предшествовать ее описание. Описание начинается ключевым словом `struct`, после которого в фигурных скобках перечисляются *объявления полей* (так называются компоненты структуры). Описание обязательно завершается знаком `;`. Это один из немногих случаев, когда после фигурной скобки ставится точка с запятой.

Описание структуры равнозначно описанию нового типа, поэтому после завершения описания возможно создавать объявления переменных типа "структура". В приведенном примере (листинг 8.2) описание структуры не связано с последующим объявлением переменной, здесь описана структура и только затем объ-

явлена переменная. Но эти два действия можно выполнять в одной языковой конструкции. Например, вот так:

```
struct my_struct{int a; char b;} my;
```

Результатом выполнения этой конструкции будет объявление переменной `my` типа `my_struct`. Описания структур могут быть очень сложными. Они могут содержать в своем описании массивы и даже описания иных структур. В примере, приведенном в листинге 8.2, объявлена переменная `f`, которая является массивом структур, содержащих в качестве поля другую структуру, имеющую поле-массив. В примере показано, как добраться до ячейки этого глубоко лежащего массива `b`.

Листинг 8.2

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  struct my_struct{int a;char b[10];};
  struct new_struct{int b;my_struct c;};
  new_struct f[2];
  f[1].c.b[5]='g';
}
```

Структура может быть составной частью другой структуры, но объявляться она должна так, как показано в примере выше. Объявить структуру внутри структуры можно, но использовать ее не удастся. То есть, например, такое объявление, как показано ниже, будет принято компилятором:

```
struct my
 { int s;
   struct {char b;};
   struct example(int g;);
 };
```

В этом примере предпринята попытка объявить одну структуру внутри другой. Текст будет принят компилятором, как вполне законный, но использовать поля этих записей не получится.

Замечание об инициализации

Объявление структуры — это создание нового типа, поэтому конечно же в объявлении типа не может быть никаких действий по инициализации полей. Следующая запись ошибочна, что выяснится уже на этапе компиляции:

```
struct example
 { int a=1;
   char b='a';
 }
```

Инициализация величин, имеющих тип структуры, возможна точно так же, как и инициализация, например, массивов. Пример из листинга 8.3 показывает эту технику.

Листинг 8.3

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  struct name
  { char a;
    int b;
    char c[20];
  };
 name my={'a',1,"Пример строки"};
 cout <<my.a<<" "<<my.b<<" "<<my.c;
}
```

Инициализирующие выражения сопоставляются с полями попарно, при этом выполняется проверка типов, так что будьте внима-

тельны с порядком инициализации. Из сказанного следует ошибочность следующего примера:

```
struct name
  {
    char a[5];
    int b;
    char c[10];
  };
 name my={'a',1,"123456"};
```

Компилятор ожидает увидеть 5 значений для инициализации первого поля, являющегося символьным массивом. Полей окажется недостаточно, в этой ситуации компилятор попытается использовать последующие инициализирующие выражения и столкнется с проблемой соответствия типов. Следующий же пример ошибки уже не содержит:

```
struct name
  {
    char a[5];
    int b;
    char c[10];
  };
 name my={{'a'},1,"123456"};
```

В этом фрагменте компилятору с помощью фигурных скобок указано, что для массива есть только один инициализатор, последующие же выражения предназначены для инициализации прочих полей.

Объединения

Вполне может возникнуть потребность в создании структуры со взаимоисключающими полями, т. е. такими полями, потребность в которых есть, но эта потребность не может быть одновременной. Предположим, нам необходимы два поля, одно из которых

целое, а второе символьное. Соответствующее описание структуры будет выглядеть так:

```
struct example{ int a; char g;};
```

Из того, что заявленные поля не нужны одновременно, следует, что память, занятая структурой, будет использоваться неэффективно. Конечно, потеря одного-двух байтов не слишком отяготит современный компьютер, но необходимо понимать, что в реальной задаче структура может быть очень и очень велика, и тогда неэффективность распределения памяти окажется заметной. Можно выйти из положения, создав множество структур, в каждой из которых не будет взаимоисключающих полей. Но это плохой выход, т. к. в этом случае, сложности с определением типов мы перенесем на логику программы, которая станет длиннее и сложнее, что вряд ли является хорошей платой за экономию памяти. Кроме того, в этом случае экономию можно будет обеспечить только за счет создания динамических структур.

Относительно хорошее решение проблемы предлагает *объединение*. Это тоже структура, содержащая несколько полей, но на момент объявления переменной заданного типа объединение создает только одно поле из объявленных — именно то, которое нужно. Структура, записанная выше, будет выглядеть так:

```
union example { int a; char g;};
```

Компилятор, анализируя запись объединения, определяет наибольшее поле и выделяет памяти столько, сколько необходимо для размещения наибольшего. Реально же будет размещено то поле, которое потребуется. В листинге 8.4 приведен простой пример использования объединения.

Листинг 8.4

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  union my_struct{int a;char b[1];};
```

```
  my_struct my;
  my.b[0]='g';
  my.a=6;
  cout <<my.a<<" "<<my.b[0];
}
```

В программе предпринята попытка разместить в памяти оба поля, что принципиально делать нельзя. Компилятор отреагирует на это спокойно. Программа будет откомпилирована и выполнена, но в выводе символьного значения, присвоенного нулевому элементу массива `b`, мы не увидим. Это следствие использования общей памяти. Значение переменной `a` стерло присвоенное символьное значение. Если присвоения `y.b[0]='g';` и `my.a=6;` поменять местами, то увидим обратный эффект. Если же массив объявить побольше, то в выделенной памяти удастся размесить и целое поле, и часть массива. Но это, конечно, очень опасный способ объявления переменных.

Иногда объединения используются программистами для неявного преобразования типов. Действительно, если в выделенную область памяти положить значение одного типа, а затем использовать его, как значение другого типа, то произойдет преобразование. Немного изменим программу нашего примера (листинг 8.5).

Листинг 8.5

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  union my_struct{int a;char b;};
  my_struct my;
  my.a=132;
  cout <<my.b;
}
```

Здесь также объявлены два поля. Целое поле больше, следовательно, память компилятором будет выделена под него. Затем присвоением в эту выделенную память заносится целое значение 132. Но следующей командой из этой же памяти читается уже символьное значение. Символьное поле также было объявлено, поэтому здесь все законно, и компьютеру ничего не остается делать, как преобразовать целое число в символ. Обратное преобразование также будет правильным.

Объединение вполне может быть частью большей структуры. Пример из листинга 8.6 показывает достаточно сложную ситуацию.

Листинг 8.6

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  struct example
  {char a;int b;
   union {char sim;float d;};
   union {char c[5];float e[5];};
  };
  example MY;
  MY.d=4.45;
  MY.e[3]=5.7;
}
```

Во-первых, структура содержит два объединения, во-вторых, второе объединение состоит из двух массивов. Но такая конструкция совсем не усложняет доступа к элементам.

Еще одно существенное отличие объединений от структур заключается в том, что объединение, объявленное самостоятельно, а не как составная часть структуры, не может быть передано функции или возвращено ею.

Замечание об инициализации объединений

Поля объединения — это альтернативы, поэтому естественно, что С не позволяет провести инициализацию всех полей. В точке объявления объединения возможно выполнить инициализацию только первого поля. Поэтому следующий пример ошибочен:

```
union name
  {
    char a[5];
    int b;
    char c[10];
  };
 name my={5};
```

Компилятор ожидает инициализации не целочисленного поля, а символьного массива. Если же есть необходимость выполнить инициализацию все же целого поля, то в описании объединения это поле должно быть первым. Следующий пример уже не содержит ошибки:

```
union name
  {
    int b;
    char a[5];
    char c[10];
  };
 name my={5};
```

Однако инициализация не отменяет возможности использовать другие поля. И после инициализации вопросы доступа к полям объединения решаются так же, как и без нее. Это показано в программе из листинга 8.7.

Листинг 8.7

```
#include <conio.h>
#include <iostream.h>
void main()
```

```
{ clrscr();
  union name
  {
    int b;
    char a[5];
    char c[10];
  };
 name my={5};
 for (int i=0;i<5;i++) my.a[i]='1';
 cout <<my.a;
}
```

И последнее замечание. Данный механизм инициализации работает и в том случае, если объединение является частью сложной структуры.

Поля и экономия памяти

Достаточно часто в программистской практике встречаются ситуации, в которых даже минимального типа данных слишком много. Пусть, например, необходимо запомнить, случилось в процессе выполнения некоторого цикла некоторое событие или нет (например, нашли или нет ноль в массиве в процессе просмотра его элементов). Для решения такого технического вопроса вполне достаточно одного бита, но минимальная структура `char` — это целых восемь битов.

Конечно, в большинстве случаев потеря семи битов не играет никакой существенной роли, особенно сейчас, когда в распоряжении программиста сотни мегабайт. Кроме того, практика показывает, что работа с целыми типами логически проще и, проигрывая в объеме памяти, затраченной на структуры данных, мы существенно выигрываем в объеме памяти, затраченной на программный код. Но тем не менее, язык С предоставляет возможность работать с битовыми структурами. Полноценного, самостоятельного битового типа не существует, но есть возможность упаковки битовых значений внутри целой переменной. Пример из листинга 8.8 показывает технику решения этого вопроса.

Листинг 8.8

```
#include <conio.h>
#include <iostream.h>
void main(void)
{ clrscr();
 struct reg
 { unsigned a:3;
   unsigned b:2;
 } example;
 example.a=5;
 example.b=4567;
 cout <<example.a<<" "<<example.b;
}
```

Базовой конструкцией для битовой упаковки, как видите, является структура. В ней указываются битовые поля, и через двоеточие описывается количество полей, используемых структурой. Доступ к полям выполняется так же, как и для обычных компонентов записи. В записи примера для поля `b` отведены только два бита, поэтому при попытке втолкнуть в него большое значение оно будет урезано до двух битов, и в выводе мы увидим, что значение компонента `b` оказалось равно 3.

Максимальное количество битов, выделяемых на одно поле, равно 16 (т. е. два байта). При попытке выделить больше компилятор сообщит о том, что битовое поле слишком велико. Таким образом, тип целый есть максимально возможный тип для упаковки битовых полей. Не стоит думать, что, используя одно битовое поле, вам удастся получить экономию в семь битов. Потому ранее и говорилось, что речь идет не о самостоятельном типе, а об упаковке. Действительно, упаковав один бит в целую переменную, мы просто теряем семь битов. Рассмотрим в доказательство сказанного следующий пример (листинг 8.9).

Листинг 8.9

```
#include <conio.h>
#include <iostream.h>
void main(void)
```

```
{ clrscr();
 struct reg
 { unsigned a:3;
   unsigned b:16;
 } example;
 cout <<sizeof(example);
}
```

Созданная в примере структура занимает 19 битов. Это больше, чем два байта, но меньше, чем три. Но операция `sizeof` дает результат в три байта.

Все сказанное наводит на мысль, что задач для битовой упаковки, таких, чтобы эта упаковка действительно имела смысл, наверное, не очень много. Но "немного" не значит "нет".

Рассчитаем затраты памяти для следующей задачи. Пусть требуется составить расписание занятий крупного вуза. Алгоритм расчетов нас сейчас не интересует, обсудим только проблему хранения данных. Для этой задачи есть естественная проблема: как организовать хранение данных на занятие? Предположим, занятие описывается следующими характеристиками:

- ❒ название предмета;
- ❒ ФИО преподавателя;
- ❒ номер группы;
- ❒ потребность в специальном оборудовании.

Вряд ли это единственные характеристики, но остановимся на них. Самая простая идея хранения данных — это запись их в массив структур, в которых каждое поле будет явным образом содержать значение характеристики. Конечно, это очень затратный способ. Можно использовать небольшое усовершенствование. Ясно, что множество значений каждой характеристики может быть и велико, но вполне ограничено. Например, количество преподавателей даже крупного вуза ограничено несколькими сотнями. Поэтому составим для каждой характеристики список всех возможных значений. Тогда поля структур могут содержать не значения, а их номера в соответствующих списках. Предположим

для упрощения, что в таком массиве будет 1000 записей (тысяча занятий), тогда общий объем памяти, требуемый для всего массива:

1000 занятий × 4 компонента × 2 байта = 8000 байтов.

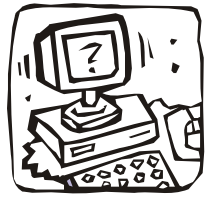
А теперь посмотрим, что получится, если паковать номера в битовых полях. Пусть для упрощения возможных:

- предметов — 50: требует 5 битов;
- преподавателей — 100: требует 6 битов;
- групп — 50: требует 5 битов;
- вариантов спецоборудования — 10: требует 3 бита.

Итого, общая потребность на одну запись составляет 19 битов или с учетом невозможности дробить целый тип получается 3 байта, а, следовательно, на весь массив нужно 3000 байтов. Простая арифметика показывает, что даже на такой небольшой структуре удалась экономия в 5000 байтов. Для баз, содержащих миллионы записей (а такие базы не редкость), экономия может оказаться огромной.

Заключение

Структуры — это именно та языковая конструкция, благодаря которой возможно относительно легко разрабатывать базы данных, а если учесть, что базы — это наиболее популярная коммерческая задача, то становится ясно, что важность структур трудно переоценить. Главное свойство структур — это объединение под одним именем данных, описывающих один объект. Еще одно важнейшее свойство касается передачи структур в другие функции. Массив также может быть передан в качестве параметра, но массив нельзя непосредственно вернуть из функции. Структура же в отличие от массива может быть возвращена оператором `return`, что также добавляет структурам функциональности в использовании.



Глава 9

Указатели

Объявление и создание указателей

Указатели — пожалуй, важнейшая конструкция данных, предлагаемая языком C. Общая идея указателя проста для понимания. Объявляя указатель, мы объявляем величину, являющуюся адресом области памяти. Сам по себе указатель не хранит никаких содержательных данных, но они могут находиться в области, на которую он указывает. Сразу заметим, что область памяти — это совершенно бесструктурный объект, а, следовательно, выделение памяти под указатель — это наиболее общая операция, позволяющая объявить данные. В силу своей общности понятие указателя связано практически со всеми структурами данных языка C.

Важнейшим преимуществом указателей перед остальными структурами данных является управляемость памяти. В момент создания области памяти можно определить ее размер в полном соответствии с текущей потребностью, а как только потребность в данных отпадет, область памяти можно вернуть в общий массив свободной памяти.

Указателем называется переменная, содержащая в себе адрес области памяти, в которой находится какая-либо структура данных. Различие с обычными переменными следующее: результатом объявления `float a` будет создание переменной с именем `a` действительного типа, а результатом объявления `float *a` будет создание переменной с именем `a`, содержащей адрес области памяти, достаточной для хранения действительного значения.

Проявляется существенное отличие в способе обращения к переменным. Пример из листинга 9.1 демонстрирует это различие.

Листинг 9.1

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int *a,b;
  *a=100; b=200;
  cout <<a<<" "<<*a<<" "<<b;
}
```

В этом примере содержимое переменной a участвует в потоке вывода дважды. Первый раз без звездочки, а второй раз со звездочкой. Указатель без звездочки — это адрес, указатель со звездочкой — это значение. Поэтому результаты вывода значений a и *a существенно различаются. Величина b не является указателем, поэтому добавлять к ее записи звездочку не имеет смысла.

Из сказанного ясно, что переменная, объявленная как указатель, фактически представляет собой две величины: адрес и значение, в то время как переменная, объявленная, как просто переменная, содержит только значение. Отсюда следует вывод, что для хранения указателей требуется больше памяти, и преимущество указателей перед обычной переменной неочевидно.

Это действительно так, но заметный проигрыш есть только в случае обычных переменных. Если же используя указатель, мы будем объявлять сложные структуры данных, например такие, как массивы, то затраты на хранение указателей перестанут быть существенными. Но зато станут очень заметными значительные преимущества. Во-первых, память, связанную с указателем, можно вернуть в свободную область памяти специальной командой, пример которой мы рассмотрим позже. Этого нельзя делать с обычными величинами. Во-вторых, что может быть даже более существенно, объектом, объявленным как указатель, проще опе-

рировать в случае, если есть необходимость обработки объекта как единого целого. Такая задача возникает, например, при передаче массива как параметра (об этом *см. главу 13*).

Имена сложных объектов часто являются указателями. Пример этого — программа из листинга 9.2.

Листинг 9.2

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int *a;
  a=new int[100];
  for (int i=0;i<=99;i++)
    a[i]=i;
  for (i=0;i<=99;i++)
    cout <<a[i]<<" ";
  cout<<"\n"<<"\n";
  for (i=0;i<=99;i++)
   { cout <<*a<<" ";
     a++;  // переход к следующему элементу
   }
}
```

Здесь массив создан через указатель на массив, но заполняется значениями, как обычный массив. После заполнения его значения выводятся на экран дважды: первый раз мы к элементам массива обращаемся по значению индекса, а второй раз используя понятие указателя. Результат один и тот же. Данный пример демонстрирует значительную степень идентичности просто массива и массива, созданного через указатель на массив. Но, объявляя массив непосредственно, мы обязаны жестко определить объем требуемой памяти. Объявление через указатель дает значительную свободу. А именно команда

```
a=new int[100];
```

позволяет объявить массив в нужный момент, а самое главное — нужного размера. В варианте с указателем вполне возможно следующее объявление:

```
int t=100;
int *a;
a=new int[t];
```

Здесь значение размера массива есть переменная величина. То же самое в случае прямого объявления уже не будет законным. А именно следующий вариант объявления ошибочен:

```
int t=100;
int a[t];
```

Следующий пример показывает, что значит работать с объектом как с единым целым и какие удобства мы можем получить от использования указателей. В примере из листинга 9.3 объявлены два массива. Один объявлен прямо и заполнен последовательными числами. Второй объявлен через указатель. Такая техника объявления второго массива дает возможность передать значения первого второму одним присвоением

```
b=a;
```

После выполнения данного оператора имена `a` и `b` фактически будут указывать на одну и ту же область памяти.

Листинг 9.3

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int a[10];
  for (int i=0;i<=9;i++) a[i]=i;
  int *b;
  b=new int[10];
  b=a;
  for (i=0;i<=9;i++)
   cout <<b[i]<<" ";
}
```

Однако необходимо помнить, что такое присвоение не передает значения массива a массиву b, выполняется только передача адреса массива a. Это дает существенные преимущества в скорости присвоения, если массив a велик. Но тогда необходимо учитывать, что любое изменение данных массива b означает точно такое же изменение данных массива a. Фактически, a и b — два имени одного и того же массива.

У внимательного читателя должен возникнуть вопрос. Команда

```
b=new int[10];
```

была реально выполнена, и ее результатом стало выделение памяти. Если же в результате присвоения b=a был передан адрес массива a, то что будет с областью выделенной в момент объявления b? Ответ будет таков: область, изначально выделенная под массив b, окажется потерянной. Причем просто потерянной. Она не будет возвращена в область свободной памяти. Фактически, наша программа содержит грубую ошибку, но она будет откомпилирована и выполнена, ошибка пройдет незамеченной. Существенные проблемы, связанные с потерей памяти, начнутся при работе с большими объемами данных. Программа, записанная в листинге 9.4, свободна от указанной ошибки и, кроме того, демонстрирует, что a и b — действительно, не более чем два имени одной области.

Листинг 9.4

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int a[10];
  for (int i=0;i<=9;i++) a[i]=i;
  int *b;
  // b=new int[10];
  b=a;
  for (i=0;i<=9;i++)
```

```
    cout <<b[i]<<" ";
  cout <<'\n';
  for (i=0;i<=9;i++) b[i]*=b[i];
  for (i=0;i<=9;i++)
    cout <<a[i]<<" ";
}
```

В этой программе заполняется массив a, его адрес передается массиву b. Далее значения b выводятся, чем демонстрируется его идентичность массиву a. После чего значения b изменяются, и выводится массив a. Можно увидеть, что его значения равны измененным значениям массива b. Существенная разница в отсутствии команды

```
b=new int[10];
```

Она закомментирована. В результате указатель b создается, а память не выделяется. Именно этот механизм используется для передачи массива как параметра, о чем мы уже упоминали и о чем было обещано поговорить позднее.

Доступ к элементам структуры

Для доступа к значению указателя используется звездочка. Но только в том случае, если структура данных однородна, как например массив. В случае разнородной структуры данных — в языке C такие данные так и называются "структурами", и мы уже изучали их ранее — доступ к компонентам осуществляется несколько иначе. Пример из листинга 9.5 показывает, как именно это выполняется.

Листинг 9.5

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  struct example{int a;char b;};
```

```
  example *my;
  // my=new example;
  my->a=5;
  my->b='h';
}
```

Доступ к элементам объединения осуществляется точно так же. Обратите внимание, что операция new в этом примере не используется. Необходимая память под одну структуру будет выделена в момент объявления в динамической памяти. Поэтому команда, занимающаяся созданием указателя, закомментирована, но комментарии вполне можно убрать, это тоже будет вполне корректная, работающая программа (листинг 9.6).

Листинг 9.6

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  struct example{int *a;char b;};
  example *my;
  int *u;
  my->a=new int[10];
  for (int i=0;i<=9;i++)
    my->a[i]=i;
  for (i=0;i<=9;i++)
    cout<<my->a[i]<<" ";
}
```

А сейчас немного более сложный пример — листинг 9.7.

Листинг 9.7

```
#include <conio.h>
#include <iostream.h>
void main()
```

```
{ clrscr();
  struct example{int a;int b;};
  struct example1{int c; example *my;};
  example1 *MY;
  MY->my->a=1;
}
```

В этой программе структура содержит указатель на структуру, поэтому операция доступа -> применяется дважды для присваивания значения компоненту a. Если бы в структуре example1 объявление вложенной структуры было бы выполнено без указателей, то последние три оператора имели бы следующий вид:

```
struct example1{int c; example my;};
example1 *MY;
MY->my.a=1;
```

Язык C вполне допускает создание указателя на указатель. Таким способом, например, можно создать двумерный массив. Пример из листинга 9.8 показывает, как это осуществить. В программе распечатывается таблица умножения, причем элементы таблицы хранятся в двумерном массиве. Явного объявления массива нет, но есть объявление указателя, указывающего на указатель, который в свою очередь указывает на величину целого типа. Первым действием после объявления сложного указателя выделяется область памяти под массив указателей на указатели величин целого типа, фактически создается массив массивов.

Далее в теле внешнего цикла операцией new выделяется память уже под массив целых. Затем в теле внутреннего цикла очередная ячейка памяти заполняется значением произведения, это произведение выводится в соответствующей точке экрана, после чего операция a++ переводит указатель к следующей ячейке памяти, а операция *a++, выполняемая в теле внешнего цикла, осуществляет переход к следующему массиву.

Листинг 9.8

```
#include <iostream.h>
#include <conio.h>
void main()
```

```
{ clrscr();
  int **a;
  *a=new int[10];
  for (int i=1;i<=9;i++)
   { *a=new int[10];
     for (int j=1;j<=9;j++)
      { **a=i*j;
        gotoxy(i*3,j*3);
        cout <<**a;
        a++;
      }
     *a++;
   }
}
```

Еще одна возможность создания двумерного массива посредством указателя представлена в программе из листинга 9.9. В предыдущем примере мы абсолютно корректно создали двумерный массив, но вся работа с ним в программе выполнялась через указатели. Если есть желание получать доступ к массиву традиционным образом, используя операцию индексации, то и это возможно. Программа из листинга 9.9 показывает технику такой работы.

Листинг 9.9

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int **b;
  for (int i=1;i<=9;i++)
   for (int j=1;j<=9;j++) b[i][j]=i*j;
  for (i=1;i<=9;i++)
   for (j=1;j<=9;j++)
    { gotoxy(i*3,j*3);
      cout <<b[i][j];
    }
}
```

Эта программа выдает на экран таблицу умножения (наш любимый пример). Двумерный массив объявлен как указатель на указатель, а обращение к элементам выполняется через операцию индексации.

Указатель и операция создания объекта, адрес которого будет содержать указатель, тесно связаны еще с двумя операциями, существующими в C, и, наверное, логично будет упомянуть о них в этой главе. Это `sizeof` — операция определения размера объекта данных и `&` — операция получения адреса. В примере программы из листинга 9.10 показаны две возможности использования операции `sizeof`. В поток вывода направлены два значения. Первое — это размер в байтах памяти, занимаемой величиной `a`, и второе — это размер памяти, выделяемой на тип `int`. А еще немного ниже поток вывода получает размер в байтах структуры, содержащей два двухбайтных числа и один массив в 10 байтов. И, наконец, в последнем примере определяется объем памяти, занятый массивом целых чисел.

Листинг 9.10

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int a;
  cout << sizeof(a)<<"  "<<sizeof(int)<<"\n";
  struct example{int t;int a; char f[10];};
  example b;
  cout <<sizeof(b)<<"\n";
  int r[100];
  cout <<sizeof(r);
}
```

Возможность расчета размера занимаемой памяти полезна в самых различных ситуациях. В *главе 12*, рассказывающей о файлах данных, эта операция используется для определения порции байтов,

записываемой в файл. Эта операция окажется незаменимой в случае необходимости скопировать структуру данных в некоторую другую область памяти.

Обратите внимание, что в качестве аргумента функция берет либо имя типа, либо выражение. Это очень важный момент. Рассмотрим, к примеру, следующий фрагмент:

```
int *a;
a=new int[100];
cout <<sizeof(a);
```

Если вы полагаете, что результатом будет размер созданного массива, то это серьезная ошибка. В этом фрагменте в качестве аргумента находится указатель. Вот его размер в байтах и будет результатом вывода, а не размер области памяти, на которую он указывает.

> ***Примечание***
>
> Когда операция `sizeof` применяется к имени типа структуры или объединения, то результатом будет *фактический размер* в байтах. Этот размер включает пустые участки, находящиеся либо внутри структуры, либо завершающие ее. Такие участки часто используются для выравнивания компонентов структуры на границах слов физической памяти. В результате чего этот результат может не соответствовать размеру, вычисленному путем сложения размеров памяти, которая необходима для хранения компонентов по отдельности.

Программа из листинга 9.11 иллюстрирует операцию определения адреса. Первыми операторами программы объявляется целая переменная `a`, и ее адрес выводится на экран. Это не очень содержательный пример, но запустите программу и посмотрите, как выглядит адрес переменной величины.

Продолжение программы уже более содержательно. Во-первых, создается структура и объявляется переменная `r`, являющаяся структурой объявленного типа. Чтобы было нагляднее, два поля структуры заполняются соответствующими величинами. Далее объявляется указатель `p` на структуру, но память под него не выделяется. Этот указатель предназначен для работы с той же структурой, которую именует и переменная `r`. Для того чтобы

привязать указатель к области памяти r, в программе вычисляется адрес r и этот адрес присваивается указателю p. А для того чтобы убедиться в успешности выполненной операции, выводятся значения заполненных полей, но уже с использованием указателя.

Листинг 9.11

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int a;
  cout << &a<<"\n";
  struct example{int b;char e[100];};
  example r;
  r.b=6;
  r.e[2]='f';
  example *p;
  p=&r;
  cout <<p->b<<"  "<<p->e[2];
}
```

Еще одна интересная возможность использования ссылок (так называются выражения вида &*имя_объекта*) показана в *главе 13*, посвященной увязке нескольких функций C в одну цельную программу. Указатели могут указывать и на функцию, т. к. функция в языке C — в некотором смысле такой же объект, как и структура данных, из чего следует, что функцию возможно вызывать, используя переменную, являющуюся указателем, но об этом *см. в главе 13*.

Выделение и освобождение памяти

Вспомните о классах памяти. Указатели, как и обычные переменные, могут быть автоматическими и статическими. В приведенных примерах мы не всегда использовали операцию выделения памяти, но указатели все же работали. Что же дает операция new?

Посредством new мы выделяем память в так называемой основной памяти, в то время как просто объявление выделяет память в стеке. Память, выделенную операцией new, необходимо специальным образом возвращать в общую свободную память. Для этого в C используется специальная команда delete для простых переменных и delete[] для массивов.

Арифметические операции над указателями

Язык C дает достаточно большую свободу в части выполнения над указателями арифметических операций. Присваивание — далеко не единственная возможность арифметического преобразования указателей, что хорошо показывает программа из листинга 9.12.

Листинг 9.12

```
#include <iostream.h>
#include <conio.h>
void main(void)
{ clrscr();
  int *a,*b;
  a=new int[10];
  b=a;
  for (int i=1;i<=10;i++)
   { *a=i*i;
     a++;
   }
  a=b+5;
  for (i=1;i<=5;i++)
   { cout <<*a<<" ";
     a++;
   }
}
```

В данном примере создается динамический массив, заполняемый квадратами целых чисел. Указатель b необходим для запоминания адреса начала динамического массива. В процессе выполнения первого из циклов указатель a уходит от начала области. Оператором a=b мы можем вернуть его в начало, но в программе выполняется a=b+5, в результате чего распечатка вычисленных значений начнется не с первого адреса, а с пятого. Возможность применения к указателям арифметических операций предоставляет программисту огромные возможности в части управления динамическим массивом.

Рассмотрим в качестве примера следующую задачу. Дан массив, заполненный возрастающими числами, точные значения которых неизвестны. Необходимо выяснить, есть ли в данном массиве заданное число. Просто перебор массива нас не устраивает в силу его большого размера. Конечно, для тестового примера большой массив вводить не будем, но имеем в виду, что программа должна работать для массива любого размера.

Идея решения такова. Если двигаться по списку в направлении искомого числа с некоторым шагом, то можно либо попасть точно на это число, либо его перескочить. В случае промаха необходимо сделать две вещи: во-первых, уменьшить шаг движения указателя, например, в два раза и, во-вторых, изменить направление движения на противоположное. Подробно излагать алгоритм не будем, программа достаточно прозрачна (листинг 9.13).

Листинг 9.13

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
void main(void)
{ clrscr();
  int *a,*b;
  a=new int[1000];
  b=a;
  randomize();
```

```
  long s=0;
  // Заполнение массива возрастающими числами
  for (int i=0;i<1000;i++)
   { s+=random(10)+1;
     *a=s;
     a++;
   }
  a=b;
  int res=0,n;
  cin>>n;
  int step=500,l=1;
  do
  { a+=l*step;
    if ((*a>n && l==1 ) || (*a<=n && l==-1))
     { step=div(step,2).quot;
       l=-l;
     }
    if (*a==n) res=1;
  }
  while (step>0);
  cout <<res;
}
```

Если бы язык не разрешал обычные арифметические операции с указателями, то для решения подобной задачи либо пришлось бы отказаться от динамических массивов, либо существенно усложнить логику построения программы. В связи с тем, что имя массива есть указатель на массив, точнее на его начало, возможны весьма экзотические формы обращения к массиву. Рассмотрим следующий пример (листинг 9.14).

Листинг 9.14

```
#include <conio.h>
#include <iostream.h>
void main(void)
```

```
{ clrscr();
  int t[10],*a;
  a=t;
  for (int i=0;i<=9;i++)
   { *a=i;
     a++;
   }
  for (i=0;i<=9;i++)
   cout <<t[i]<<" ";
}
```

В этой программе область памяти, занятая массивом, заполняется с помощью указателя, имеющего смысл указателя на просто целое. Тот же эффект будет получен, если оператор `a=t` заменить на `a=&t[0]`. Действительно, элемент `t[0]` находится в самом начале области памяти, содержащей массив, и, следовательно, передача его адреса эквивалентна обращению к имени массива.

Используя адресную природу имени массива, ту же самую задачу заполнения массива значениями можно решить еще проще. Рассмотрим программу из листинга 9.15.

Листинг 9.15

```
#include <conio.h>
#include <iostream.h>
void main(void)
{ clrscr();
  int t[10];
  for (int i=0;i<=9;i++)
    *(t+i)=i;
  for (i=0;i<=9;i++)
   cout <<t[i]<<" ";
}
```

Здесь ссылка на массив `*(t+i)` выглядит достаточно экзотично, но ничего удивительного в ней нет. То, что к указателю можно

прибавить целое и получить при этом адрес, мы уже знаем, а операция "звездочка" обеспечит доступ к ячейке памяти по данному адресу. Два рассмотренных ранее примера призваны показать гибкость связи между понятием указателя и понятием массива, но у вас не должно создаваться впечатления, что с именем массива можно вообще поступать как угодно. Нельзя забывать, что имя массива — это константа, поэтому запись вида:

```
int a[10];
a++;
```

будет незаконна, несмотря на то, что такая же по смыслу, но иная по механизму запись

```
int *a;
a=new int[10];
a++
```

является вполне допустимой. Это как раз и показывает, что, несмотря на столь сильную связь, указатель и массив — все же различные понятия.

Операции умножения и деления для указателей не определены, но вполне возможна операция вычитания. Рассмотрим следующую задачу. Пусть дан массив. Предположим, что почти все числа, записанные в него, положительны и больше нуля, но есть некоторое количество нулей. Необходимо выяснить, сколько чисел находится до первого нуля. Решение представлено в листинге 9.16.

Листинг 9.16

```
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
void main(void)
{ clrscr();
  randomize();
  int *a,*b;
  b=a;
  for (int i=0;i<1000;i++)
```

```
   {*a=random(100);
    a++;
   }
  for (a=b;*a!=0;a++);
  cout <<a-b;
}
```

Данная задача легко решается посредством обычного массива, но использование указателей делает расчетный цикл очень изящным и избавляет программу от дополнительной переменной. Впрочем, это, конечно, вопрос вкуса, мы же просто показали имеющуюся возможность.

Замечание о шаге указателя

В отношении арифметических операций, например операции ++, разумно задать вопрос о шаге. На сколько ячеек памяти осуществляется переход, например, в операции *указатель*+5? Ранее мы игнорировали такую деталь, но деталь эта важная. Действительно, если предположить, что указатель смещается на адрес, то результаты смещения в таком фрагменте

```
char *a;
a++;
```

и таком

```
float *a;
a++
```

будут различными.

Поэтому, видимо, единственным разумным решением для C будет смещение на размер величины, на которую ссылается указатель. И язык C действительно так и поступает, причем по такому правилу величина смещения определяется не только для простых типов, но и для любых прочих. Пример из листинга 9.17 это демонстрирует.

Листинг 9.17

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  struct name
  { int a;
    char c[10];
  };
  name *my,*my1;;
  my=new name[10];
  my1=my;
  for (int i=0;i<10;i++)
   { my->a=i;
     my++;
   }
  my1+=5;
  cout<<my1->a;
}
```

В программе указатель ссылается на структуру, занимающую 12 байтов, следовательно, оператор `my1+=5` выполняет смещение на 60 байтов и переходит к шестой записи массива. Следующий пример интересен не только в плане арифметики указателей. Этим примером мы покажем еще одну возможность создания указателя на массив. В программе в очередной раз создается таблица умножения (листинг 9.18).

Листинг 9.18

```
#include <conio.h>
#include <iostream.h>
void main()
{ clrscr();
  typedef int mass[10];
```

```
  mass *a,*b;
  a=new mass[10];
  b=a;
  for (int i=1;i<10;i++)
  {for (int k=1;k<10;k++)
     *a[k]=k*i;
   a++;
  }
  a=b;
  for (i=1;i<10;i++)
  {for (int k=1;k<10;k++)
    { gotoxy(i*3,k*3);
      cout <<*a[k];
    }
   a++;
  }
}
```

Разберем этот пример более подробно.

- Первым действием в программе создается синоним типа, имеющего смысл целочисленного массива из десяти элементов.
- Затем объявляется указатель данного типа. Указатель `*a` после своего объявления показывает на динамическую область размером в десять целых чисел.
- Операция `new` обеспечивает выделение в основной памяти десяти областей, каждая из которых рассчитана на 10 целых чисел.
- Оператор `b=a` запоминает начало первой из областей.
- В теле внутреннего цикла заполняется очередная область (каждая из областей рассчитана на 10 чисел, мы же заполняем только девять; нулевой элемент игнорируется из соображений удобства).
- Оператор `a++` выполняет скачок в 20 байтов к следующей области на 10 чисел.

- Оператор `a=b` позволяет вспомнить указателю `a` начало первой из десяти областей. Этот оператор, конечно, не обязателен, мы могли бы далее воспользоваться указателем `b`.
- Циклы вывода разбирать подробно не будем, они работают полностью идентично расчетным.

Указатель в никуда

Язык С позволяет присвоить указателю адрес несуществующего объекта, так сказать, создать ссылку в никуда. Для такого адреса используется специальная константа `NULL`. Обратите внимание, что константа записана прописными буквами. (Вы помните, что для С прописные буквы и строчные — это разные буквы?) Это константа целого типа и ее целочисленное значение равно нулю. То есть результатом следующей операции:

```
cout << NULL;
```

будет ноль. С этой константой мы еще встретимся в следующей главе.

Заключение

Примеров и теоретических рассуждений данной главы, наверное, достаточно, чтобы понять: указатель — в языке С исключительно мощная и гибкая структура данных, действительно позволяющая с данными делать практически все, что угодно. Но плата за гибкость и мощь тоже достаточно высока. Ошибки, совершаемые в программах, имеющих указатели, наиболее сложны для начинающих, а зачастую и для искушенных программистов. К тому же большая часть ошибок не видна компилятору. Логика использования указателей далеко не так очевидна, как, например, логика работы с массивами. Особенные трудности в понимании создают такие конструкции, как указатель на указатель. А в языке С цепочка ссылок может быть сколько угодно длинной. Например, нет ничего незаконного в следующем объявлении:

```
char ****a;
```

Такой "многоэтажный" указатель можно, например, использовать для создания многомерного массива, но, конечно, чтобы не запутаться во множестве ссылок, необходим опыт и тренированное внимание. Несмотря на значительное количество материала, мы рассказали здесь про указатели не все. Объяснения будут продолжены еще в трех главах. Главы, посвященные строкам *(см. главу 10)* и связным спискам *(см. главу 11)* — это целиком главы об указателях, и в *главе 13* о разработке сложной программы вы встретитесь с важным видом использования указателей, это ссылка на функцию.

Глава 10

Строки

Нультерминальные строки С

В современных компиляторах есть классы для самых разных типов строк, но любой компилятор языка C/C++ понимает так называемые *нультерминальные строки*. Фактически, строки языка С — это не новый тип данных, а символьный массив, для которого оговорено только одно специальное условие — такая строка должна завершаться символом `\0`, именуемым *терминальным нулем*, поэтому они и называются нультерминальными. Если это небольшое условие не соблюдать, то некоторые свойства строки у такого массива останутся, но лучше все же терминальный ноль не игнорировать, т. к. это даст в ваше распоряжение достаточно мощный набор функций, работающих со строками, как единым целым.

Как уже было сказано, типа "нультерминальная строка" в языке С нет. Но такую строку можно организовать как указатель на динамический символьный массив. В листинге 10.1 приведен пример программы, в которой строка создается, вводится с клавиатуры и выводится на экран монитора.

Листинг 10.1

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
```

```
  char *s;        // Объявление строки неопределенной длины
  s=new char[10]; // Создание строки в 10 символов
  cin >> s;
  cout << s;
}
```

Уже в этом примере видна важнейшая особенность механизма обработки строк. Если бы был создан указатель на целое число, то, например, его ввод осуществлялся бы так:

```
cin >> *s;
```

Согласно правилам обращения с указателем звездочка необходима для доступа к значению, а имя указателя без звездочки содержит не значение, а адрес. Мы же в командах ввода и вывода прописали имя указателя без звездочки. Это оттого, что строка есть массив, а для массива не существует единственной ячейки, где хранится значение, поэтому использование звездочки не даст всю строку.

Оператор

```
s=new char[10];
```

фактически выделяет место для строки, в основной памяти, поэтому его использование не обязательно. Если эту команду пропустить, то строка будет создана, но в стековой памяти, что для данного примера совершенно не принципиально. В примерах, рассматриваемых далее, строки чаще всего будут создаваться в основной памяти посредством операции `new`. Эта операция избавляет нас от необходимости тратить память стека, выделенную для функции, но не дает выполнить инициализацию строки. Если строки размещать динамически, то их инициализация возможна в момент объявления. Вот два примера:

```
char *a="пример инициализации строки";
char b[]="еще один пример инициализации строки";
```

В первом примере строка объявлена как указатель, во втором — как массив неопределенной длины. В какой-то степени оба определения эквивалентны, это следует из того, что имя массива есть указатель.

Набор функций, используемых для обработки нультерминальных строк, наверное, сильно зависит от реализации. Поэтому мы рассмотрим некоторые стандартные функции, используемые в реализации Turbo C. А затем посмотрим технику работы с такими строками при условии, что ни одна из стандартных функций не известна.

Первый пример представлен в листинге 10.2.

Листинг 10.2

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
void main()
{ clrscr();
  char *s;
  s=new char[10];
  cin >> s;
  for (int i=0;i<=strlen(s);i++)
  cout << s[i]<<"  ";
}
```

В первом примере вводится строка и затем выводится посимвольно, для чего используется функция, определяющая длину строки, а сама строка обрабатывается, как массив.

В следующем примере попробуем сложить две строки (листинг 10.3). Это достаточно распространенная операция, и нам необходимо научиться ее выполнять.

Листинг 10.3

```
#include <string.h>
#include <stdio.h>
void main(void)
{  char *dest;
```

```
   dest=new char[25];
   char *blank = " ", *c = " Язык C ", *turbo = "Turbo";
   strcpy(dest, turbo);
   strcat(dest, blank);
   strcat(dest, c);
   printf("%s\n", dest);
}
```

Эту же программу можно переписать в таком варианте (листинг 10.4).

Листинг 10.4

```
#include <string.h>
#include <stdio.h>
void main(void)
{  char dest[25];
   char *blank = " ", *c = " Язык C ", *turbo = "Turbo";
   strcpy(dest, turbo);
   strcat(dest, blank);
   strcat(dest, c);
   printf("%s\n", dest);
}
```

Способ объявления строки во втором варианте существенно иной, а результат работы программы будет таким же. Это следствие того факта, что в языке С имя массива является в то же самое время указателем на область памяти, занимаемую массивом.

А теперь попробуйте заменить команду

```
strcpy(dest, turbo);
```

на команду

```
strcat(dest, turbo);
```

С первого взгляда эта замена вполне логична, т. к. нам требуется именно конкатенация, и функции `strcat` должно быть достаточно, но это не так, и программа перестает работать корректно.

Мы более не будем подробно разбирать действие функций обработки строк. Их описания есть в справочной системе компилятора, и достаточно просто внимательного прочтения, чтобы разобраться в правилах их использования. Значительно более интересно попробовать выполнять обработку строк, пользуясь только знанием структуры нультерминальной строки. Знать необходимо немного:

- адрес начала строки хранится в указателе;
- с именем указателя на строку можно обращаться, как с именем одномерного символьного массива;
- строка завершается терминальным символом `\0`.

Теперь попробуем переписать первый пример (см. листинг 10.2) без использования функции вычисления длины (листинг 10.5).

Листинг 10.5

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
void main(void)
{  clrscr();
   char *s;
   s=new char[20];
   cin >> s;
   int num=0;
   while (s[num]!='\0') num++;
   cout << num;
}
```

Здесь мы воспользовались тем, что строка — это массив, и достаточно просто двигаться по массиву до тех пор, пока не будет обнаружен терминальный ноль.

Следующая программа выполняет операцию *конкатенации*. А именно добавляет две строки, вводимые с клавиатуры к третьей пустой (листинг 10.6).

Листинг 10.6

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
void main(void)
{  clrscr();
   char *a,*b,*c;
   a=new char[20];
   cin >> a;
   b=new char[20];
   cin >> b;
   c=new char[40];
   char *w;
   w=c;
   while (*a!='\0')
    {*c=*a;c++;a++;}
   while (*b!='\0')
    {*c=*b;c++;b++;}
   *c='\0';
   c=w;
   cout << c;
}
```

Идея программы заключается в проходе копируемых строк и переносе символов в строку назначения. Эту работу выполняют два цикла по условию. По завершению работы упомянутых циклов два последних штриха: во-первых, необходимо к концу строки назначения добавить терминальный ноль и, во-вторых, в указатель, связанный со строкой назначения, вернуть адрес ее начала. Для того чтобы это было возможно выполнить, перед циклами копирования адрес начала строки назначения был запомнен в дополнительном указателе.

Еще одна стандартная операция над строками — это копирование части строки определенной длины, начиная с конкретной позиции (листинг 10.7).

Листинг 10.7

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
void main(void)
{  clrscr();
   char *a,*b,*w;
   a=new char[100];
   b=new char[100];
   cin >> a;
   int n,count;
   cin >>n>>count;
   for (int i=1;i<n;i++) a++;
   w=b;
   for (i=1;i<=count;i++)
    { *b=*a;
      a++;b++;
    }
  *b='\0';
  b=w;
  cout << b;
}
```

Указатель a показывает на начало строки. Нам необходимо передвинуть его на n позиций. Эту операцию выполняет первый цикл for. После его завершения указатель a находится на позиции, с которой требуется начинать копирование, а указатель b на начале строки b. Во втором цикле указатель a проходит последовательно count элементов, каковые записываются в строку b. По завершению цикла к строке b добавляется терминальный ноль и указателю b присваивается его исходное значение, сохраненное в специальном указателе w.

Наверное, приведенных примеров достаточно для понимания механизма работы с нультерминальными строками. Но все же, при-

ведем еще один. Попробуем найти позицию первого вхождения строки в строку (листинг 10.8).

Листинг 10.8

```
#include <string.h>
#include <iostream.h>
#include <conio.h>
void main(void)
{  clrscr();
   char *a,*b;
   a=new char[100];
   b=new char[100];
   cin >> a;
   cin >> b;
   int n=0,count=0;
   int flag=0;
   while (*a!='\0' && flag==0)
    { for(char *w=b;*w!='\0' && *a==*w && *a!='\0';w++,a++);
      if (*w=='\0') {count=n+1;flag=1;}
      n++;
      a++;
    }
  cout <<count;
}
```

В ходе работы внешнего цикла указатель `a` перемещается по своей строке, и одновременно с ним пытается идти вдоль строки `b` указатель `w`. На каждом шаге двух указателей их содержимое сравнивается. Если вдруг окажется, что они не равны, то указатель `a` продолжает движение, а указатель `w` сбрасывается в начало строки и начинает свой путь заново.

Программа прекращает свою работу в двух случаях:

- указатель `a` дошел до конца своей строки. Если при этом указатель `w` не достиг конца строки `b`, то, следовательно, строка `b` в строку `a` не входит и будет возвращен ноль;

- если указатель w достиг конца строки b, то независимо от состояния указателя a возвращается номер позиции, с которой удалось найти вхождение.

Последний пример выглядит немного более сложным, но принципы работы те же. Указатель, связанный со строкой, можно двигать по строке в обе стороны, и в каждый момент времени указатель обеспечивает доступ к одному элементу строки. Доступ можно получить с помощью операции звездочка, но возможно обратиться к элементу строки, как к элементу массива. В момент определения значения строки, например, при вводе ее с клавиатуры, указатель устанавливается на начало строки. Завершается строка терминальным символом \0, который можно установить вручную. При операциях ввода терминальный ноль устанавливается автоматически.

О вложении присваивания

Присваивание и сравнение — не одно и то же. В ходе присваивания вычисляется выражение, и значение идентификатора, стоящего слева от знака присваивания, изменяется. В ходе вычисления сравнения величины, участвующие в сравнении, не изменяются, а результатом сравнения будет логическое выражение. Из сказанного следует, что использовать присваивание в заголовках циклов некорректно. Однако пример, приведенный в листинге 10.9, опровергает это утверждение.

Листинг 10.9

```
#include <conio.h>
#include <iostream.h>
void main(void)
{ clrscr();
  char *a="123456789";
  char *b;
  char *c=b;
  while ((*b++=*a++)!='\0');
```

```
  // while (*b++=*a++);
  cout <<b<<"\n";
  cout <<c;
}
```

В этой программе строка a копируется в строку b до тех пор, пока не будет встречен терминальный ноль. Это достаточно удивительный пример вложения в условие присваивания и операций увеличения значения, но возможна еще более короткая запись. Снимите комментарий со второго оператора `while`, а первый наоборот закомментируйте, и вы увидите, что ничего не изменилось, программа по-прежнему работает корректно. Это хороший пример, показывающий гибкость использования операций и указателей в языке C.

Пример упорядочения строк

Построение строк через указатели создает интересные возможности для их упорядочения. Любой алгоритм, от самого простого, например пузырьковой сортировки, до самых эффективных, предполагает, что в процессе работы некоторые пары строк будут меняться местами. Для упорядочения массивов чисел эта необходимость не вызывает возражения. Ситуация со строками несколько иная. Проблему может создать длина строки. Если строка содержит два, три или несколько десятков символов, то, конечно, их перекачка в другую строку не обременит компьютер, но если строки длинные, например, по несколько килобайт, что вполне реально, что даже самый быстрый алгоритм может оказаться неэффективным. Указатели дают замечательный выход из затруднения. А именно для того чтобы поменять содержимое двух строк, достаточно поменять адреса двух указателей на строки, само же содержимое строк может оставаться на месте. Программа, записанная в листинге 10.10, именно это и делает.

Листинг 10.10

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

```
void main()
{ clrscr();
  char *a[5]=
  { "111111111111",
    "2222222222",
    "3333333",
    "4444",
    "555"
  };
 for (int i=0;i<5;i++)
  for (int k=0;k<5-i;k++)
   if (strlen(a[k])>strlen(a[k+1]))
    { char *g;
      // Ниже выполняется обмен не содержимым строк,
      // а адресов
      g=a[k];
      a[k]=a[k+1];
      a[k+1]=g;
    }
 for (i=0;i<5;i++)
  cout<<a[i]<<"\n";
}
```

Пример сортировки показывает еще один важный технологический момент — инициализацию массива строк. Заметьте, что строки, инициализирующие элементы массива, имеют разную длину. Проблему определения длины берет на себя компилятор. Программу можно еще немного оптимизировать, если длины строк вычислить только один раз и запомнить в дополнительном массиве, но т. к. нас интересует только техника работы со строками, мы этого делать не будем.

Полезные функции

Посвятим еще немного времени функциям обработки строк, предоставляемым компилятором компании Borland. Схема изложе-

ния раздела будет такова. Для каждой функции, т. к. это принято в справочной системе компилятора, будем кратко указывать:

- действие;
- синтаксис;
- короткий пример.

Функция *strstr()*

Действие: ищет вхождение подстроки в строку. В случае удачи возвращает указатель на элемент, с которого начинается вхождение.

Синтаксис:

```
<указатель_на_строку> strstr(<указатель_на_строку>,
                             <указатель_на_подстроку>)
```

Пример представлен в листинге 10.11.

Листинг 10.11

```
#include <conio.h>
#include <string.h>
#include <iostream.h>
void main()
{ clrscr();
  char *a="1234567890";
  char *b="456";
  cout <<strstr(a,b);
}
```

Результат вывода: `4567890`.

Функции *strlow()* и *strupr()*

Действие: функция `strlow()` переводит все символы строки в нижний регистр, а функция `strupr()` — в верхний регистр.

Синтаксис:

<*указатель_на_строку*> strlow(<*указатель_на_строку*>)

Пример представлен в листинге 10.12.

Листинг 10.12

```
#include <conio.h>
#include <string.h>
#include <iostream.h>
void main()
{ clrscr();
  char *a="ABCDE";
  cout <<strlwr(a);
  cout <<strupr(a);
}
```

Функция *strset()*

Действие: заменяет все символы строки на указанный символ.

Синтаксис:

<*указатель_на_строку*> strset(<*указатель_на_строку*>, <*символ*>)

Пример приведен в листинге 10.13.

Листинг 10.13

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
void main()
{  clrscr();
   char *a = "123456789";
   char s = 'c';
   printf("Строка перед обработкой: %s\n", a);
   strset(a, s);
   printf("Строка после обработки:  %s\n", a);
}
```

Функция *strcmp()*

Действие: сравнивает две строки (в смысле лексического порядка).

Синтаксис:

<*целое*> strcmp(<*указатель_на_строку*>, <*указатель_на_строку*>)

Пример приведен в листинге 10.14.

Листинг 10.14

```
#include <string.h>
#include <conio.h>
#include <iostream.h>
void main()
 { clrscr();
   char *str1 = "abcd", *str2 = "abklb";
   cout<<strcmp(str2, str1);
}
```

Заключение

Данная глава полностью посвящена строковым переменным, и в ней ни слова не говорится о строковых константах. Это от того, что константам в нашей книге посвящена отдельная достаточно емкая глава, и мы просто не повторяемся.

Обратите внимание, что в основном строки в данной главе объявлялись по следующей схеме:

char **Имя_строки*

Имя_строки = new char[*длина_строки*]

Но был пример, в котором мы строку объявили так, как обычно объявляем массив. Было ли это ошибкой? Нет. Все объясняется уже известным вам фактом — имя массива играет роль указателя на область памяти, в которой массив хранится. Поэтому нультерминальную строку можно объявлять и без использования указателя. Принципиально важно только одно — это завершение строки терминальным нулем.

Глава 11

Связные списки и деревья

Построение связного списка

Рассмотрим следующую ситуацию. Предположим, есть необходимость разместить в памяти большой массив данных, например массив структур. Механизм решения такой задачи не сложен — объявляем указатель на структуру и затем операцией `new` выделяем необходимый объем памяти. Все очень просто, но только на первый взгляд. Вполне может оказаться, что при наличии достаточно большого объема свободной памяти, единой области такого размера не найдется. Свободная память может оказаться разбитой на кусочки различного размера. Операция `new` в такой ситуации будет беспомощна, т. к. она способна выделить именно цельную область. А о том, как быть, если такой цельной области в наличии не окажется, мы и поговорим в данной главе.

Связным списком называется структура данных, одно из полей которой указывает на следующую структуру. Связный список графически можно представить так, как показано на рис. 11.1.

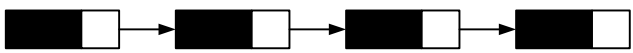


Рис. 11.1. Связный список

Темным помечены поля содержательных данных, а белое — это поле, содержащее указатель на следующую запись. Такая конструкция данных обладает одним недостатком и одним компенсирующим его достоинством. Недостаток — чтобы добраться

до конкретной записи, необходимо пройти все предыдущие. При достаточно длинном списке, это может потребовать существенного времени.

Достоинство — каждый элемент связного списка гарантированно создается в свободной области памяти достаточного размера. Чтобы в этом убедиться, рассмотрим более подробно процесс создания связного списка.

Объявление указателя на связный список выполняется так:

```
struct Имя_записи {Список_содержательных_полей;
                   Указатель_на_Имя_записи}
```

Имя переменной есть указатель на *Имя_записи*.

Такая технология объявления как раз и гарантирует, что записи списка будут создаваться на свободных местах. Действительно, имя переменной есть указатель, для его реального создания необходимо выполнение операции new, которая как раз и осуществляет поиск гарантированно свободного места. А для того чтобы создать новую запись, необходимо опять применить операцию new к компоненту, который обозначен как *Указатель_на_Имя_записи*, что гарантирует корректность создания очередного элемента списка. Но что не гарантирует new — это местоположение элементов списка. Поэтому рис. 11.1, иллюстрирующий список, не нужно понимать буквально. Реально записи могут располагаться самым различным образом. Далее мы рассмотрим несколько программ, описывающих несложные технологические приемы работы со связными списками.

Задача 1. Обход связного списка

Первый пример самый простой. Здесь создается связный список, состоящий из одного числового поля и указателя на следующую запись. Числовые поля заполняются последовательными числами. Создается дополнительный указатель, используемый для запоминания начала связного списка. Программа представлена в листинге 11.1.

Листинг 11.1

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  struct example{int a;example *next;};
  example *my,*my1;
  my=new example;
  my1=my;                    // запоминаем начало
  for (int i=1;i<=10;i++)
   { my->a=i;
     my->next=new example; // создание нового элемента

     my=my->next;           // переход к новому элементу
   }
  my=my1;                    // возврат к началу списка
  for (i=1;i<=10;i++)
   { cout <<my->a<<" ";     // распечатка числового поля
     my=my->next;           // переход к следующему элементу
   }
}
```

Задача 2. Обход связного списка неизвестной длины

Предположим, что некоторый список уже был создан, но длина его неизвестна. (В предыдущей задаче длина списка была фиксированной — 10 элементов.) Необходимо этот список неизвестной длины пройти.

Во-первых, для того чтобы обход стал возможным, необходимо завершить построение списка специальным образом. А именно указатель на следующую запись последнего элемента надо установить в никуда. Для этого существует специальное ключевое слово `NULL`. Обратите внимание, что `NULL` записано большими

(прописными) буквами. Это обязательно. Если последнее значение указателя установлено в NULL, то этим обстоятельством можно воспользоваться и читать записи связного списка до тех пор, пока не будет достигнут адрес NULL.

Текст программы приведен в листинге 11.2.

Листинг 11.2

```
#include <iostream.h>
#include <conio.h>
void main(void)
{ clrscr();
  struct example{int a;example *next;};
  example *my,*my1;
  my=new example;
  my1=my;
  for (int i=1;i<=10;i++)
   { my->a=i;
     my->next=new example;
     my=my->next;
   }
  my->next=NULL; // установка последнего указателя в NULL
/* Цикл, записанный далее, кроме того, что обеспечивает
корректный проход связного списка еще и демонстрирует мощь
цикла for. Обратите внимание, что параметром цикла является
указатель, и вся информация, необходимая для того, чтобы этим
указателем управлять, записана уже в тексте заголовка.
А именно начинается движение указателя с начала списка
до тех пор, пока не будет достигнут указатель в НИКУДА,
и шаг цикла заключается в переходе на очередной
элемент списка. */
  for (my=my1;my->next!=NULL;my=my->next)
    cout <<my->a<<" ";
}
```

Задача 3. Обход связного списка в двух направлениях

Отличие решения поставленной задачи от двух предыдущих в том, что запись связного списка содержит два указателя. Один на запись вперед и один на запись назад.

Текст программы представлен в листинге 11.3.

Листинг 11.3

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  struct example{int a;example *next,*pred;};
  example *my,*my1,*my2;
  my=new example;
  my1=my;
  for (int i=1;i<=10;i++)
   { my->a=i;
     my->next=new example;
/* Указатель my2 играет очень важную роль. После создания
новой записи указатель my уходит на нее, и адрес предыдущей
записи просто забывается, так что к тому моменту, когда будет
необходимо в следующей записи запомнить адрес предыдущей,
этот предыдущий адрес уже будет забыт. А указатель my2
и содержит адрес предыдущей записи. */
     my2=my;
     my=my->next;
     my->pred=my2;
   }
// проход списка в прямом направлении
  my=my1;
  for (i=1;i<=10;i++)
  { cout <<my->a<<" ";
```

```
/* Оператор my1=my; совершенно необходим. Дело в том, что
при выходе из данного цикла указатель my указывает
не на последнюю запись, что необходимо, а на следующую
за последней. А указатель my1 содержит адрес именно
последней. */
    my1=my;
    my=my->next;
  }
 cout <<"\n";
// проход списка в обратном направлении
 for (i=1;i<=10;i++)
  { cout <<my1->a<<" ";
    my1=my1->pred;
  }
}
```

Несколько замечаний о методах обхода

В *главе 9*, рассказывающей о технике работы с указателями, есть раздел, посвященный арифметике указателей. Оттуда мы знаем, что указатели можно складывать, вычитать и сравнивать. Указатель на связный список — это, конечно, тоже указатель, и, следовательно, следующий фрагмент не содержит синтаксической ошибки:

```
struct uk{int a; uk *next;};
uk *my;
my++;
```

Синтаксической ошибки не содержит, но и смысла не имеет и легко может привести к ошибке исполнения. В начале главы мы уже говорили, что записи связного списка не обязаны находиться в памяти рядом. Фактически они могут быть раскиданы как угодно по памяти. Арифметические же операции просто смещают указатель на определенное количество байтов, и только. Это означает, что, выполнив над таким указателем арифметическую

операцию, мы переместим указатель некорректно. Конечно, может случиться так, что записи списка действительно окажутся рядом, но это будет, скорее, случайность, на которую нельзя серьезно рассчитывать.

Еще одно замечание. Если связный список велик и есть необходимость быстрого движения по нему, можно создать дополнительные поля указателей, которые будут хранить ссылки не на следующую запись, а, например, через десять или через сто элементов, в зависимости от потребности задачи. Можно создавать указатели, показывающие на конкретные элементы связного списка, которые обладают определенными свойствами.

И последнее. Процесс создания связного списка принципиально нуждается в операции выделения памяти new. Следующий фрагмент не содержит ошибки синтаксиса:

```
for (int i=1;i<=20;i++)
{ my->a=i;
  my=my->next;
}
```

но и работать не будет.

Динамические переменные — указатели создаются в момент объявления, но никак не в момент использования. Поэтому обращение

```
my=my->next;
```

это обращение в никуда.

Задача 4. Вставка элемента в связный список

Идея решения заключается в разрыве списка в определенной точке так, как это показано на рис. 11.2.

Для того чтобы это получилось, необходимо дойти по списку до нужного элемента, затем его указатель на следующую запись перенаправить на вставляемую запись (т. е. присвоить ему адрес новой записи), а указатель вставляемой записи направить на сле-

дующую запись в списке (т. е. присвоить ему адрес следующей записи).

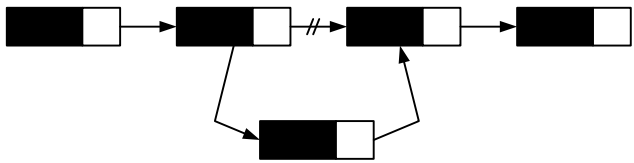


Рис. 11.2. Вставка элемента в связный список

Текст программы — в листинге 11.4.

Листинг 11.4

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  struct example{int a;example *next;};
  example *my,*my1,*my2;
  my=new example;
  my1=my;
// создается связный список
  for (int i=1;i<=10;i++)
   { my->a=i;
     my->next=new example;
     my=my->next;
   }
// создается вставляемая запись
  my2=new example;
  my2->a=5000;
  my=my1;
// проходим список до определенной записи
  for (i=1;i<=5;i++) my=my->next;
// переменяем адреса соответствующих указателей
```

```
  my2->next=my->next;
  my->next=my2;
  my=my1;
  for (i=1;i<=11;i++)
   { cout <<my->a<<" ";
     my=my->next;
   }
}
```

Задача 5. Вставка связного списка

Даны два связных списка: *первый* и *второй*, и дан *номер позиции*. Требуется вставить *второй* список в первый, начиная с заданного *номера позиции*.

Идея решения та же, что и в предыдущей задаче, и новый пример нужен лишь для демонстрации возможностей техники работы с указателями. Как видно из решения, длина связных списков роли не играет (листинг 11.5). Они могут состоять из одного элемента или из многих, ничего от этого не меняется.

Листинг 11.5

```
#include <iostream.h>
#include <conio.h>
void main()
{ struct zapis{int a; zapis *next;};
  zapis *u,*uk,*uk1,*uk1_start,*uk1_end,*uk2_start,*uk2_end;
  clrscr();
  int n;cin >>n;
  uk1_end=new zapis; uk1_start=uk1_end;
  cin >> uk1_end->a;
  for (int i=1;i<n;i++)
   { uk1_end->next=new zapis;
     uk1_end=uk1_end->next;
     cin >> uk1_end->a;
   }
```

```
  int m;cin>>m;
  uk2_end=new zapis; uk2_start=uk2_end;
  cin >> uk2_end->a;
  for (i=1;i<m;i++)
   { uk2_end->next=new zapis;
     uk2_end=uk2_end->next;
     cin >> uk2_end->a;
   }
  int l;cin>>l;
  uk=uk1_start;
  for (i=1;i<=l-1;i++) uk=uk->next;
  u=uk->next;
  uk->next=uk2_start;
  uk2_end->next=u;
  for (i=1;i<=n+m;i++)
   { cout <<uk1_start->a<<" ";
     uk1_start=uk1_start->next;
   }
}
```

Задача 6. Сортировка по возрастанию массива, организованного в виде связного списка

Для упорядочивания выберем самый простой метод — метод пузырька, т. к. для нас сейчас важна отработка не метода сортировки, а метода работы со связными списками. Суть пузырьковой сортировки заключается в понятии неправильной пары. Таковой называется пара рядом стоящих элементов, таких, что предыдущий больше последующего. В массиве рядом стоящие элементы — это элементы, чьи индексы отличаются на единицу. В связном списке рядом стоящие элементы — это непосредственно связанные.

Создадим два указателя, и пусть они "бегут" по списку синхронно, только первый указатель начнет свой путь с первого элемента,

а второй — со второго. Тогда у нас будет гарантия, что эти два указателя всегда будут показывать на элементы, стоящие рядом. Обсуждать подробно метод пузырька мы здесь не будем.

Текст программы приведен в листинге 11.6.

Листинг 11.6

```
#include <iostream.h>
#include <conio.h>
void main()
{ struct list{int a; list *next;};
  list *my,*my1,*my2;
  clrscr();
  my=new list;
  my1=my;
  int n; cin >>n;
  for (int i=1;i<=n;i++)
   { cin >>my->a;
     my->next=new list;
     my=my->next;
   }
  for (i=1;i<n;i++)
   { my=my1; my2=my->next;
     for (int k=1;k<=n-i;k++)
      { if (my->a>my2->a)
          { int c=my->a;
            my->a=my2->a;
            my2->a=c;
          }
        my=my->next;
        my2=my2->next;
      }
   }
  my=my1;
  for (i=1;i<=n;i++)
   { cout <<my->a<< "  ";
```

```
    my=my->next;
  }
}
```

Деревья

Задача 7. Построение двоичного дерева

Двоичное дерево отличается от прочих тем, что имеет две ветви. Будем называть их правой и левой. На рис. 11.3 показано дерево глубиной 2.

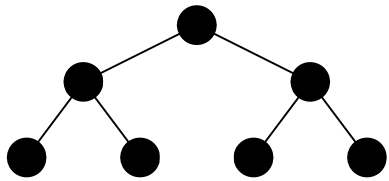


Рис. 11.3. Двоичное дерево глубиной 2

Каждый узел дерева порождает новое дерево, поэтому вполне логично построить рекурсивную программу, ядром которой будет процедура, запускаемая из каждого узла для левой и правой ветви (листинг 11.7).

Листинг 11.7

```
#include <iostream.h>
#include <conio.h>
struct tree{int a; tree *left; tree *right;};
void Create_tree(tree *,int);
void View_tree(tree *,int);
void main(void)
{ clrscr();
  tree *uk;
  uk=new tree;
  uk->a=0;
  Create_tree(uk,0);
```

```
  View_tree(uk,0);
}
void Create_tree(tree *uk,int n)
{ tree *uk1;
  if (n<3)
   { uk->left=new tree;
     uk1=uk->left;
     uk1->a=2*(n+1);
     Create_tree(uk1,n+1);
     uk->right=new tree;
     uk1=uk->right;
     uk1->a=3*(n+1);
     Create_tree(uk1,n+1);
   }
}
void View_tree(tree *uk,int n)
{ if (n<3)
   { cout<<uk->a<<" ";
     View_tree(uk->left,n+1);
     View_tree(uk->right,n+1);
   }
}
```

В программе обход дерева выполняют две функции. Отличие между ними в том, что функция `Create_tree` содержит операцию `new`, поэтому функция создает дерево. Функция `View_tree` получает на вход адрес верхнего узла и просто выполняет проход уже построенных узлов.

Задача 8. Обмен местами значения левой и правой ветви двоичного дерева

Пусть построено некоторое двоичное дерево и каждому его узлу присвоено случайное число, например, так, как показано на рис. 11.4.

Результатом работы алгоритма (листинг 11.8) должно быть дерево, изображенное на рис. 11.5.

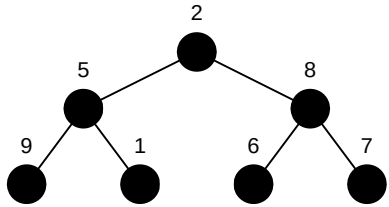


Рис. 11.4. Двоичное дерево с присвоенными числами узлам

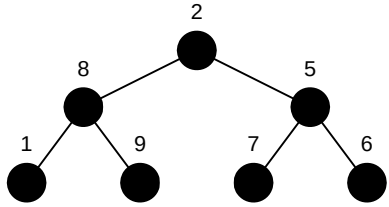


Рис. 11.5. Результат работы алгоритма

Листинг 11.8

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
struct zapis{int a; zapis *left;zapis *right;};
int max;
void Create_Tree(zapis *,int);
void View_Tree(zapis *,int,int,int);
void Change(zapis *,int);
void main()
{ zapis *tree1,*tree2;
  randomize();
  clrscr();
  cin>>max;
  tree1=new zapis;
  tree1->a=random(10);
  tree2=tree1;
  Create_Tree(tree1,1);
```

```
  View_Tree(tree2,40,1,1);
  Change(tree2,1);
  View_Tree(tree2,40,1,12);
}
void Create_Tree(zapis *tree,int n)
{ zapis *new_tree;
  if (n<=max)
   { tree->left=new zapis;
     new_tree=tree->left;
     new_tree->a=random(10);
     Create_Tree(new_tree,n+1);
     tree->right=new zapis;
     new_tree=tree->right;
     new_tree->a=random(10);
     Create_Tree(new_tree,n+1);
   }
}
void View_Tree(zapis *tree,int x,int n,int y)
{ gotoxy(x,y);cout <<tree->a;
  if (n<=max)
   { View_Tree(tree->left,x-div(20,n).quot,n+1,y+3);
     View_Tree(tree->right,x+div(20,n).quot,n+1,y+3);
   }
}
void Change(zapis *tree,int n)
{ zapis *left,*right;
  left=tree->left;
  right=tree->right;
  int a=left->a;
  left->a=right->a;
  right->a=a;
  if (n<max)
   { Change(left,n+1);
     Change(right,n+1);
   }
}
```

Методы построения и обхода дерева нам уже известны. Нового в этой программе — это, во-первых, способ красивого расположения элементов дерева на экране монитора и функция обмена значениями. Функция `Change` пользуется в своей работе тем обстоятельством, что в каждом узле известны адреса его левого и правого соседа, а поменять местами значения труда не составляет.

Заключение

Достаточно часто программистам приходится работать со сложным образом структурированными данными, а не просто набором чисел. Это, например, графы самой различной конфигурации. Связные списки дают мощный инструмент для построения и обработки именно таких структур. Мы рассмотрели только простые случаи: линейный связный список и двоичное дерево, т. к. изучение сложных алгоритмов не входит в цели данной книги.

Глава 12

Файлы данных

Неформатный ввод/вывод

Файловый тип не является базовым типом языка C. В сущности, это даже и не тип данных, а, скорее, способ организации ввода/вывода, при котором поток данных направляется в файл и выбирается из файла.

Ввод/вывод в файл, как и ввод/вывод на консоль, возможен в двух вариантах: форматный и неформатный. Мы начнем изучение файлового ввода/вывода с неформатного метода.

Сущность метода заключается в указании:

- объекта, из которого считывается или в который записывается информация;
- имени файла, хранящего информацию;
- размер порции информации (в байтах);
- количество порций, считываемых за один раз.

Таким образом работают две функции: `fread` — чтение данных из файла и `fwrite` — запись данных в файл. Далее приведен их точный синтаксис.

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t n,
              FILE*stream);
```

Здесь:

- `size_t` — тип, используемый для возврата информации о памяти, занимаемой объектом. Для дальнейших примеров не

нужен, и в записи функций чтения/записи его можно игнорировать;

- `*ptr` — адрес переменной, являющейся источником или, соответственно, местом назначения для читаемых (записываемых) байтов;
- `size` — количество считываемых (записываемых) байтов;
- `n` — количество повторений операции чтения (записи);
- `*stream` — указатель на файл.

В примере, приведенном в листинге 12.1, источником данных является символьный массив, содержащий две буквы. Этот массив записывается в файл с именем `file` 10 раз, для чего в теле цикла используется функция записи, в которой указывается имя массива (как известно, имя массива содержит его адрес), указание на то, что записывать необходимо 2 байта. Количество операций — одна, и файловый указатель имеет имя — `f`.

Заметим сразу, что количество операций записи более чем 1 указать нельзя, т. к. повторная операция будет продолжать чтение из того же массива, а он исчерпан уже первой операцией. Прежде чем выполнять операции чтения/записи, необходимо файл открыть, для чего используется функция `fopen`. Ее синтаксис таков:

```
FILE *fopen(const char *filename, const char *mode);
```

Функции требуется немногое: только имя файла и способ его открытия. Список возможных способов:

- `r` — файл открывается только для чтения;
- `w` — открывается для записи. Если файл с таким именем уже существует, то он будет переписан;
- `a` — файл открывается для добавления данных к концу файла, а если такой файл не существует, то открывается для записи;
- `r+` — открывается существующий файл для обновления (чтения и записи);
- `w+` — открывается существующий файл для обновления (чтения и записи). Если такой файл уже существует, то он будет переписан;

- `a+` — открывается для обновления, начиная с конца файла. Если такой файл уже существует, то он будет переписан.

И последняя функция, используемая в данной программе, `fclose` — это функция, закрывающая файл. Ее синтаксис, видимо, ясен из примера.

Листинг 12.1

```
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
void main()
{ FILE *f;
  clrscr();
  char a[2];
  a[0]='a';a[1]='b';
  f=fopen("file","w");
  for (int i=1;i<=10;i++)
    fwrite(a,2,1,f);
  char buf[2];
  fclose(f);
  f=fopen("file","r");
  for (i=1;i<=10;i++)
   { fread(buf,2,1,f);
     buf[2]='\0';
     cout <<buf;
   }
}
```

Отдельно стоит пояснить объявление `FILE *f`. `FILE`. Это структура данных, содержащаяся в файле stdio.h, и ее цель — описание величин, управляющих процессом ввода/вывода. Как именно она устроена, можно посмотреть в справочной системе компилятора. Мы же здесь подробно рассматривать эту структуру не будем, т. к. практически для любой задачи ввода/вывода достаточно объявления, принятого по умолчанию.

Следующий пример (листинг 12.2) поясняет различие между количеством байтов, которым оперирует функция, и количеством операций. Пример показывает взаимозаменяемость этих величин. Производится запись четыре раза по два байта. А считывается наоборот два раза по четыре байта. В результате строка `buf` получает то же значение, что и исходная строка `a`.

Листинг 12.2

```
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
void main()
{ FILE *f;
  clrscr();
  char *a;
  a=new char[20];
  a="12345678";
  f=fopen("file","w");
  fwrite(a,2,4,f);
  char *buf;
  buf=new char[20];
  fclose(f);
  f=fopen("file","r");
  fread(buf,4,2,f);
  buf[8]='\0';
  cout <<buf;
}
```

Важнейшая функция файлов данных — это хранение записей баз данных. Язык C для разработки баз данных предлагает конструкцию структуры. Поэтому важно уметь читать из файла и записывать в файл структуру как единое целое. Следующий пример демонстрирует технику работы со структурами (листинг 12.3).

Программа записывает в файл структуру `my1`, а затем считывает данные из файла в структуру `my2`, после чего значения этих

двух структур должны быть одинаковы. В функции записи указывается адрес структуры источника данных и ее размер, вычисляемый функцией `sizeof`.

Листинг 12.3

```
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
void main()
{ struct data{char a;int t;};
  data my,my1;
  my.a='a';
  my.t=45;
  FILE *f;
  clrscr();
  f=fopen("file","w");
  fwrite(&my,sizeof(my),1,f);
  fclose(f);
  f=fopen("file","r");
  fread(&my1,sizeof(my),1,f);
  cout <<my1.a<<" "<<my1.t;
}
```

Поставим перед собой новую задачу. Дан файл. Пусть это будет файл символов. Для упрощения в программе, представленной в листинге 12.4, это один многократно повторяемый символ. Необходимо изменить размер файла. Это может оказаться необходимым, например, в случае удаления некоторого количества байтов. В этом случае, для того чтобы в файле не образовалась пустота, потребуется перезаписать часть файла от местоположения удаленных байтов до конца файла. Но тогда, если мы удаляем *N* байтов, в конце файла *N* байтов окажутся продублированными, и этот хвост потребуется удалить.

Программа записывает в файл 100 чисел. Затем, для того чтобы убедиться в корректности выполненной операции, файл закрыва-

ется, сразу же открывается, но уже только на чтение, содержимое файла считывается и выводится на экран.

Затем файл снова открывается на модификацию и изменяется его размер. Размер файла изменяет функция `chsize`. Но эта функция первым аргументом требует номер файла (каждому файлу в момент своего открытия назначается номер, по которому к нему можно получить доступ). Поэтому прежде, чем вызывается функция `chsize`, в программе отрабатывается функция `fileno`, действие которой как раз и заключается в определении номера файла.

После того как номер файла определен, вызывается еще одна полезная функция — `filelength`. Ее аргумент — также номер файла, а результат — длина файла в байтах. И, наконец, после определения длины вызывается функция `chsize`, изменяющая длину. Если длина файла, указанная в качестве второго аргумента, окажется меньше существующей, то файл будет обрезан, если больше, то файл будет увеличен.

Листинг 12.4

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <io.h>
void main()
{ clrscr();
  FILE *f=fopen("proba","w");
  char g='f';
  for (int i=1;i<=100;i++)  fwrite(&g,1,1,f);
  fclose(f);
  f=fopen("proba","r");
  for (i=1;i<=100;i++)
   { fread(&g,1,1,f);
     cout << g<<" ";
   }
```

```
 fclose(f);
 cout <<"\n\n";
 f=fopen("proba","r");
 int n=fileno(f);
 int L=filelength(n);
 chsize(n,L/2);
 while (!feof(f))
   { fread(&g,1,1,f);
     cout << g<<" ";
   }
}
```

Форматный файловый ввод/вывод

Два примера, записанных далее, не делают ничего нового. Как и предыдущие примеры, эти программы занимаются записью и считыванием данных. Но делают они это немного иначе. Для того чтобы пользоваться функциями `fread` и `fwrite`, необходимо точно знать количество байтов записываемого или считываемого объекта. Это, конечно, не составляет существенной проблемы, даже более того, байтовые операции дают значительную свободу программисту, но иногда задачу по определению объема данных можно возложить на программу.

В первом примере (листинг 12.5) записывается, а затем считывается набор строк. Для этого используются функции форматного ввода/вывода. Для ввода по формату достаточно указать тип формата. В нашем примере указано, что записывается строка. Управляющий символ `\n` необходим для того, что указать в файле точку, в которой текущая строка завершается.

Для того чтобы записанные строки корректно прочитать, необходимо формат чтения описать так же, как и формат записи. Единственное, управляющий символ `\n` уже не нужен, т. к. при чтении символы читаются в любом случае до метки конца строки.

Листинг 12.5

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
void main()
{ clrscr();
  char *y;
  y=new char[50];
  FILE *f=fopen("proba","w");
  for (int i=1;i<=5;i++)
   { scanf("%s",y);
     fprintf(f,"%s\n",y);
   }
  fclose(f);
  f=fopen("proba","r");
  for (i=1;i<=5;i++)
   { fscanf(f,"%s",y);
     printf("%s",y);
   }
}
```

Следующий пример показывает, что в файл можно записывать не только данные одной природы (листинг 12.6). Типы записываемых и читаемых данных могут быть разными. В этом нет ничего необычного, т. к. для C файл — это набор байтов, и понятие типа для файла просто не существует. Описывая формат, мы не определяем тип файла, а задаем способ его обработки во время операций чтения и записи.

Листинг 12.6

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

```
{ clrscr();
  char y='h';
  FILE *f=fopen("proba","w");
  for (int i=1;i<=5;i++)
    fprintf(f,"%c%d\n",y,i);
  fclose(f);
  int t;
  f=fopen("proba","r");
  for (i=1;i<=5;i++)
   { fscanf(f,"%c%d\n",&y,&t);
     cout<<y<<t<<"  ";
   }
}
```

Естественно, что, пользуясь функциями форматного ввода/вывода, нельзя работать со структурами, т. к. нет и быть не может форматных описателей для структуры. Таким образом, функции неформатного ввода/вывода более универсальны.

Полезные функции

Функция *fseek()*

Действие: устанавливает файловый указатель в заданную позицию.

Синтаксис:

```
fseek(FILE <имя файлового указателя>,
           <отступ от исходной точки>,
           <тип исходной точки>)
```

Тип исходной точки — это один из следующих:

- `SEEK_SET` — начало файла;
- `SEEK_CUR` — текущая позиция файлового указателя;
- `SEEK_END` — конец файла.

Пример приведен в листинге 12.7.

Листинг 12.7

```
#include <stdio.h>
#include <conio.h>
void main()
{  clrscr();
   FILE *f;
   f=fopen("proba","w+");
   fseek(f,10,SEEK_SET);
}
```

Функция *fgetpos()*

Действие: получает значение текущей позиции файлового указателя.

Синтаксис:

```
<целое> fgetpos(FILE <имя файлового указателя>,
                <fpos_t указатель>)
```

Пример представлен в листинге 12.8.

Листинг 12.8

```
#include <stdio.h>
#include <conio.h>
void main()
{  clrscr();
   FILE *f;
   fpos_t N;
   f=fopen("proba","w+");
   fgetpos(f,&N);
}
```

Функция *rewind()*

Действие: возвращает файловый указатель в начало файла.

Синтаксис:

```
void rewind(FILE <файловый указатель>)
```

Пример представлен в листинге 12.9.

Листинг 12.9

```
#include <stdio.h>
void main()
 {
    FILE *fp;
    fp = fopen("namefile","w+");
    fprintf(fp,"abcdefghijklmnopqrstuvwxyz");
    rewind(fp);
}
```

Функция *creat()*

Действие: создает файлы (низкоуровневая функция).

Синтаксис:

```
<целое> creat(<путь к файлу>, <способ открытия>)
```

Пример представлен в листинге 12.10.

Листинг 12.10

```
#include <sys\stat.h>
#include <string.h>
#include <io.h>
void main()
{
   int handle;
   char buf[11] = "0123456789";
   handle = creat("namefile", S_IREAD |S_IWRITE);
   write(handle, buf, strlen(buf));
   close(handle);
}
```

Заключение

Рассмотренные в этой главе методы создания файлов, а также чтения и записи данных не исчерпывают всех возможностей, предоставляемых библиотеками компилятора. Но того, что рассмотрено, вполне достаточно для разработки сложных программ, управляющих потоками данных. Для изучения полного набора возможностей воспользуйтесь справочной системой компилятора.

Самое главное, что необходимо вынести из этой главы, заключается, наверное, в одном предложении: "Файл данных — это последовательность байтов, для которой отсутствует понятие типа". Собственно, это главная идея всех методов ввода/вывода языка С. Файловые данные — только лишь частный случай этого общего правила. И обратите внимание, что не существует какой-то единой библиотеки, в которую были бы записаны все функции и структуры данных, обслуживающие файлы данных. Причина этого в том, что, как уже было сказано, такого типа данных, как файл, в С не существует, а есть поток данных, организацию которого можно выполнить самыми различными средствами.

Глава 13

Построение сложной программы

Общие сведения

Настоящая глава не о том, как написать логически сложную программу, а о том, как создать программу, состоящую из нескольких функций. Проблема увязки нескольких функций в единое целое — это, прежде всего, проблема передачи данных. Поэтому глава и посвящена передаче и возврату из функции данных, объявленных как базовые типы, как структуры, как массивы, посредством указателей и без них. Еще мы немного коснемся проблемы вызова и описания функции. И в конце главы разберем, как вызвать функцию не по имени, а по адресу.

До сего момента все программы, приводимые в качестве примеров, состояли только из одной функции `main()`. Ее наличие обязательно в любой программе, но не всегда разумно ограничивать программу одной функцией. Часто бывает более правильно разбить программу на отдельные части — функции, каждая из которых имеет какой-то свой отдельный смысл и реализует законченный алгоритм. Это создает хорошую структуру, программа становится более прозрачной и читаемой.

Пусть, например, требуется написать код, выполняющий три действия:

- ввод массива сложных структур данных;
- обработку (не важно, какую именно);
- вывод каким-либо специальным образом.

Эти три действия можно записать в одной функции `main()`, и в этом не будет ровным счетом никакого нарушения. Но можно функцию `main()` записать так:

```
void main()
 { input_data();
   work();
   print_data();
 }
```

Тем самым, программа не стала короче, но функция `main()` в таком варианте выглядит компактно, а с тремя функциями, вызываемыми в `main()`, далее можно работать по отдельности.

В этой главе мы не будет рассматривать смысловые вопросы. В деле разработки сложной программы есть проблемы чисто технические, но очень важные, — это передача данных между функциями. Организация передачи данных и есть тот связующий механизм, который из нагромождения функций создает единое целое. Поэтому мы ограничимся рядом примеров, в которых показаны различные случаи обмена данными между функциями.

Начнем с небольшого замечания об общей структуре программы. С-программа состоит из набора функций, расположенных относительно друг друга самым произвольным образом. При этом их взаимное расположение не имеет никакого значения. Ни одна из функций не имеет привилегированного положения перед другими. Единственная привилегия есть у функции с именем `main()`, с нее начинается выполнение программы. Это означает, что функция `main()` должна присутствовать в любой программе. Но и `main()` может в тексте находиться практически где угодно. Конечно, функции не могут пересекаться и вкладываться. Это можно выразить кратко так: никакая часть текста функции не может быть частью текста другой функции. Разрешение на произвольное расположение функций приводит к противоречию с важным правилом, имеющимся, наверное, в любом языке программирования: любой объект до момента своего первого использования должен быть определен.

Рассмотрим следующий пример:

```
void main()
{ int y=example();
}
int example()
{ return 8;
}
```

Этот пример противоречит правилу. А именно объект `example` встречается до того, как он описан. Чтобы устранить это противоречие, в языке С вводится понятие прототипа. *Прототип* — это краткое описание заголовка функции. Для компилятора такого описания вполне достаточно, чтобы не создавать конфликтной ситуации с основным правилом. Вместе с описанием прототипа пример, записанный выше, будет выглядеть так:

```
int example();
void main()
{ int y=example();
}
int example()
{ return 8;
}
```

Обратите внимание, что заголовок функции не завершается точкой с запятой в отличие от описания прототипа. Различие обусловлено тем, что описание прототипа состоит только из одного заголовка, а функция — из заголовка и тела. Оператор "точка с запятой" завершает оператор, поэтому если мы после заголовка поставим знак `;`, то тем самым сообщим компилятору, что описание завершено, тела функции не будет, а это неправда. В случае же прототипа, это действительно так. Общий синтаксис описания прототипа таков:

Тип_возвращаемого_значения имя_функции(*список типов формальных параметров*);

Допустимо в прототипе указывать не просто список типов, а и имена формальных параметров, но острой необходимости в этом нет.

Кроме того, указание имен ограничивает программиста в создании имен для формальных параметров.

Передача одномерного массива

Пример 1. Расчет максимального в массиве

Эта задача уже была использована в *главе 1*. Повтор сделан умышленно. Во-первых, вы, может быть, пропустили элементарное введение в язык, а во-вторых, сейчас мы этот пример обсудим с другой точки зрения, поэтому полного дублирования не будет.

В программе расчет максимума оформлен в виде отдельной функции, результаты своих расчетов она возвращает командой `return max`, и, кроме того, в заголовке и в описании прототипа указывается тип возвращаемого значения как целый (листинг 13.1). Входящих данных для функции `maximum()` нет. Данные о массиве и его длине передаются самым простым и грубым способом. Они просто объявлены как глобальные переменные.

Листинг 13.1

```
#include <iostream.h>
#include <conio.h>
int maximum();
int a[100],n;
void main(void)
{ clrscr();
  cin >>n;
  for (int i=0;i<n;i++) cin >> a[i];
  int max=maximum();
  cout << max;
}
int maximum()
{ int max=a[0];
```

```
  for (int i=1;i<n;i++)
    if (max<a[i]) max=a[i];
  return max;
}
```

Такой стиль написания программ, когда все переменные объявляются до всех функций и, таким образом, являются для всех функций внешними, может показаться очень привлекательным. Действительно, в этом случае все функции знают про все переменные, нет необходимости дополнительно заботиться о том, чтобы сообщить функции требуемые ей данные. Но ощущение привлекательности такого стиля очень обманчиво. Кроме достоинств, есть два больших недостатка. Во-первых, внешние (глобальные) переменные существуют всегда, вне зависимости от потребности в них, что может привести к существенному перерасходу памяти. Во-вторых, связи между переменными внутри функции становятся неочевидными. Каждая переменная — это дополнительная связь между функциями, и чем больше таких связей, тем сложнее логика взаимодействия функций. Поэтому очень важно уметь обходиться без глобальных переменных. Рассмотренный пример, конечно, в силу своей простоты не может убедить, что "чем больше связей, тем сложнее логика", но когда вы начнете создавать большие и сложные программы, у вас будет очень много возможностей убедиться, что это действительно так.

Дальнейшее содержание главы посвящено технике передачи данных. Общая идея такова. В вызове функции указываются фактические параметры, которые могут быть как константами, так и именами переменных, а в заголовке функции указываются имена формальных параметров. В момент передачи данных под каждый формальный параметр выделяется память, и параметр инициализируется по тем же правилам, по которым инициализируются обычные переменные. При передаче тип формального параметра сопоставляется с типом фактического параметра, и выполняются все требуемые преобразования типов. Есть, правда, специальные правила для передачи массивов (и именно этим мы займемся чуть

позже), и есть специальные средства для передачи параметров без указания типа (но об этом — в конце главы).

Пример 2. Расчет максимального с передачей массива, как параметра

В новой программе передается не только массив, но и величина, имеющая смысл длины массива. Конечно, наиболее существенным здесь является механизм передачи массива. Обратите внимание на описание прототипа. В его заголовке указывается, что будет передан целый массив и целое число. В операторе вызова записано просто имя массива и имя переменной. "Имя переменной" обозначает собой переменную, поэтому величина, получаемая как величина `n`, есть величина локальная для функции `maximum()`, и какое-либо воздействие на величину `n` в теле функции `maximum()` не повлечет за собой изменения значения величины `n`, объявленной в функции `main()`.

В случае с массивом все обстоит иначе. Имя массива — это указатель на начало области памяти, содержащей массив. Таким образом, в функцию `maximum()` передается не копия массива, а указатель. Следовательно, массив, обрабатываемый в функции `main()`, и массив в функции `maximum()` — это один и тот же массив.

Текст программы представлен в листинге 13.2.

Листинг 13.2

```
#include <iostream.h>
#include <conio.h>
int maximum(int [],int);
void main(void)
{ clrscr();
  int a[100],n;
  cin >>n;
  for (int i=0;i<n;i++) cin >> a[i];
  int max=maximum(a,n);
```

```
  cout << max;
}
int maximum(int a[100],int n)
{ int max=a[0];
  for (int i=1;i<n;i++)
   if (max<a[i]) max=a[i];
  return max;
}
```

Заметим, что явное указание длины массива в прототипе не является обязательным.

Пример 3. Расчет максимального с передачей массива, как указателя

В предыдущем примере фактически тоже в каком-то смысле передается указатель, но мы должны были в заголовке указать размер получаемого массива. Теперь же в этом нет никакой необходимости. Но, как и в прошлом примере, массив, используемый в функции `main()`, и массив, используемый в функции `maximum()`, — это один и тот же массив.

Текст программы — в листинге 13.3.

Листинг 13.3

```
#include <iostream.h>
#include <conio.h>
int maximum(int *,int);
void main(void)
{ clrscr();
  int a[100],n;
  cin >>n;
  for (int i=0;i<n;i++) cin >> a[i];
  int max=maximum(a,n);
  cout << max;
}
```

```
int maximum(int *a,int n)
{ int max=a[0];
  for (int i=1;i<n;i++)
    if (max<a[i]) max=a[i];
  return max;
}
```

В данном примере мы использовали уже известное нам соотношение между именем массива и типом "указатель", а именно нам уже известно, что имя массива и есть указатель на область памяти, содержащую значения массива, или, иначе, операция "квадратные скобки". Это операция, обеспечивающая доступ к элементам области памяти, на которую ссылается указатель. Еще одна возможность передать массив связана с созданием собственного типа или, как это более точно для C, с созданием имени-синонима. В листинге 13.4 приведен пример с все тем же расчетом максимального.

Листинг 13.4

```
#include <iostream.h>
#include <conio.h>
typedef int mass[100];
int maximum(mass,int);
void main()
{ mass a;
  int n;
  cin>>n;
  for (int i=0;i<n;i++) cin>>a[i];
  int max=maximum(a,n);
  cout <<max;
}
int maximum(mass b,int m)
{ int max=b[0];
  for (int i=1;i<m;i++)
    if (max<b[i]) max=b[i];
  return max;
}
```

Передача многомерных массивов

Механизм без участия указателей принципиально такой же, как и в случае одномерных массивов, отличие заключается в том, что размер массива необходимо указывать явным образом. Но это не является обязательным при работе с одномерными массивами. В листинге 13.5 представлена еще одна наша часто используемая задача построения таблицы умножения.

Листинг 13.5

```
#include <iostream.h>
#include <conio.h>
void example(int [10][10]);
void main()
{ clrscr();
  int b[10][10];
  for (int i=1;i<=9;i++)
   for (int j=1;j<=9;j++) b[i][j]=i*j;
   example(b);
}
void example(int a[10][10])
{ for (int i=1;i<=9;i++)
   for (int j=1;j<=9;j++)
    { gotoxy(i*3,j*3);
      cout <<a[i][j];
    }
}
```

Безусловно, требуется передача только второй и дальнейших размерностей. Первая размерность, в каком-то смысле, особая, и ее можно опустить. В нашем примере объявление функции `example()` можно записать так:

```
void example(int a[][10])
```

и это было бы вполне корректное объявление.

В следующей программе показано, как передать двумерный массив с использованием указателей (листинг 13.6). Здесь нет операции new для обеспечения выделения памяти в общем массиве памяти, но механизм, наверное, понятен.

Листинг 13.6

```
#include <iostream.h>
#include <conio.h>
void example(int **)
;
void main()
{ clrscr();
  int **b;
  for (int i=1;i<=9;i++)
   for (int j=1;j<=9;j++) b[i][j]=i*j;
   example(b);
}
void example(int **a)
{ for (int i=1;i<=9;i++)
   for (int j=1;j<=9;j++)
    { gotoxy(i*3,j*3);
      cout <<a[i][j];
    }
}
```

Передача параметров по ссылке

Операция определения адреса объекта & уже была рассмотрена ранее. А здесь ею воспользуемся. В программе из листинга 13.7 в главной функции объявлены и проинициализированы две целые переменные. После объявления и инициализации они обе переданы в функцию example(), но переданы разными способами. Величина a передана по ссылке, а величина b — по значению. В результате локальная переменная a главной функции и локальная переменная y функции example() указывают на одну и ту же

область памяти, поэтому увеличение на 1 значения y приводит к увеличению на 1 и значения a в главной функции. С величиной переменной b этого не происходит, т. к. b и w указывают на различные области памяти.

Листинг 13.7

```
#include <iostream.h>
#include <conio.h>
void example(int &y,int w)
{ cout <<y<<"  "<<w<<"\n";
  y++;
  w++;
}
void main()
{ clrscr();
  int a=5,b=10;
  example(a,b);
  cout <<a<<"  "<<b;
}
```

Параметр по умолчанию

Рассмотрим следующую ситуацию. Пусть в некоторую функцию требуется передать несколько параметров, но значения некоторых из них следует получать извне, а некоторых приняты по умолчанию. То есть если в вызове функции для параметра передается значение, то параметр его получает, а если не передается, то его значение равно значению, принятому по умолчанию. В примере, приведенном в листинге 13.8, функция `example()` нуждается в двух параметрах. Но для параметра `b` в прототипе описано значение по умолчанию, следовательно, если в вызове функции `example()` окажутся два фактических параметра, то значение второго из них и будет значением формального параметра `b`. Если же фактический параметр окажется только один, то значение `b` будет значением по умолчанию.

Листинг 13.8

```
#include <iostream.h>
#include <conio.h>
int example(int,int b=5);
void main()
{ clrscr();
  cout <<example(10,10)<<"  ";
  cout <<example(10);
}
int example(int a,int b)
{ return a*b;
}
```

Параметр, определенный по умолчанию, проходит проверку объявления типа в момент объявления функции, а вычисляется во время ее вызова. Из чего следует, что задавать параметр по умолчанию можно только для последних параметров. Далее показаны два примера с ошибкой и один правильный:

```
int example (int a, int b=9, int y=7); // правильно
int example (int a=7, int b, int y=9); // ошибка
int example (int a=7, int b=6; int y); // ошибка
```

О возвращаемых значениях

В рассмотренных ранее примерах мы видели, как обеспечивается возврат значений. Если функция что-либо возвращает, то тип возвращаемого значения должен быть указан в заголовке, и реальный тип возвращаемого значения должен совпадать с типом, указанным в заголовке, иначе будет выполнено преобразование типов, что может не совпасть с вашими намерениями. Если в заголовке указано, что функция возвращает значение, но ни одного оператора `return` в ходе работы выполнено не будет, то это создаст ситуацию ошибки (но компилятор ее не заметит), и будет возвращено неопределенное значение. Обратите ваше внимание

на формулировку, повторим еще раз: "но ни одного оператора `return` в ходе работы функции выполнено не будет", в этой формулировке есть важный нюанс. В тексте функции может присутствовать оператор возврата, но не факт, что он будет выполнен.

Разберем следующий пример (листинг 13.9).

Листинг 13.9

```
#include <iostream.h>
#include <conio.h>
int example(int,int *);
void main()
{ clrscr();
  int *a,n;
  a=new int[10];
  cin >>n;
  for (int i=0;i<n;i++) cin>>a[i];
  cout <<example(n,a);
}
int example(int n,int *b)
{ for (int i=0;i<n;i++)
   if (b[i]<0) return -1;
}
```

В тексте функции есть оператор возврата, но если в переданном ей массиве не будет ни одного отрицательного числа, то оператор выполнен не будет. Однако функция завершит свою работу, и, следовательно, значение будет возвращено, но это возвращаемое значение окажется совершенно неопределенным. Компилятор TurboC, на который мы ориентируемся, заметит эту ситуацию, хотя и не определит ее как ошибочную.

Запретов на тип возвращаемого значения для функций С почти нет. Можно возвращать структуру, и, естественно, можно возвращать любой базовый тип. Нельзя возвращать массив, но можно

возвращать указатель на любой тип, и это позволяет легко обойти указанное ограничение. Пример — в листинге 13.10.

Листинг 13.10

```
#include <iostream.h>
#include <conio.h>
int *example();
void main()
{ clrscr();
  int *a;
  a=example();
  for (int i=0;i<10;i++) cout<<a[i]<<" ";
}
int *example()
{ int *a;
  a=new int[10];
  for (int i=0;i<10;i++) a[i]=i*i;
  return a;
}
```

Функция `example()` создает массив, заполняет его квадратами целых чисел и затем возвращает функции `main()` адрес на этот массив. А далее функция `main()` выводит значения массива. Из данного примера должно быть ясно, что принципиальная возможность вернуть адрес исчерпывает любые потребности как в передаче данных, так и в их возврате. Но, кроме того, функции разрешается возвращать не только адрес, но и указатель на адрес, и такая цепочка указателей может быть практически сколь угодно длинной.

В ряде примеров этой главы в качестве типа возвращаемого значения указывается тип `void`. Часто в описаниях языка C можно встретить указание на то, что тип `void` — это указатель на объект неизвестного типа, т. е. фактически это указание на то, что возвращаемого значения нет.

Допустимо применение оператора `return` без указания возвращаемого значения. В этом случае выполняется передача управления в точку вызова функции без возврата значения. Пример — в листинге 13.11.

Листинг 13.11

```
#include <iostream.h>
void example();
void main(void)
{ example();
}
void example()
{  for (int i=0;i<10;i++)
    if (i<5) cout <<i*i<<" "; else return;
}
```

В этой программе вывод квадратов чисел прерывается на значении `i=5`, управление программой передается в функцию `main()`, и при этом никакого значения в точку вызова, конечно же, не возвращается.

Перегрузка функций

Если две функции выполняют различные действия, то целесообразно назвать их по-разному. Но вполне может оказаться, что различие между ними не принципиально, и поэтому разумно как раз наоборот дать им одно и то же имя. В то же самое время, конечно, нельзя допустить, чтобы в ходе исполнения программы функции вызывались произвольным образом. Следует обеспечить, несмотря на одинаковые имена, передачу потока выполнения именно в ту функцию, которая необходима по замыслу программиста. А это, в свою очередь, означает, что в процессе работы программы смысл имени функции будет переопределяться в зависимости от ситуации. Такое переопределение имени

называется *перегрузкой функций*. Механизм перегрузки строится на различии в списке формальных параметров. В момент вызова функции управление передается той, у которой список формальных параметров соответствует списку полученных фактических. В примере, приведенном в листинге 13.12, определены две функции, и обе используются последовательно в функции `main()`.

Листинг 13.12

```
#include <conio.h>
#include <iostream.h>
int example(int, int);
int example(int);
void main(void)
{ clrscr();
  cout <<example(2,3)<<" "<<example(4);
}
int example(int a,int b)
{ return a*b;
}
int example(int a)
{ return a*a;
}
```

Для определения соответствующего имени существенно важно количество фактических и формальных параметров. Поэтому в нашей функции `main()` вполне может появиться следующая запись:

```
cout << example(3,4.5);
```

Программа с такой записью будет откомпилирована и выполнена, но это совсем не означает возможности полного произвола в описаниях. Компилятор вполне в состоянии заботиться о предотвращении некоторых конфликтных ситуаций. Следующий пример компилятор не пропустит (листинг 13.13).

Листинг 13.13

```
#include <conio.h>
#include <iostream.h>
int example(float, int);
int example(int,float);
void main(void)
{ clrscr();
  cout <<example(2,3);
}
int example(float a,int b)
{ return a*b;
}
int example(int a,float b)
{ return a+b;
}
```

Причина недовольства компилятора в явном противоречии между двумя функциями `example()` и вызовом. Ясно, что в момент вызова придется выбирать между двумя одинаково неподходящими функциями `example()`.

Целесообразно воспользоваться перегрузкой, если две функции выполняют логически одно и то же, но при этом отличаются, например, типом используемых данных. Тогда можно отработать логику на одной из них, а вторую получить простым копированием текста с последующей заменой структур данных. Пример из листинга 13.14 суммирует два массива, один из которых целочисленный, а второй действительного типа.

Листинг 13.14

```
#include <iostream.h>
#include <conio.h>
float summa(float [],int);
int summa(int [],int);
void main()
```

```
{ int a[100];
  float b[100];
  int n,m;
  clrscr();
  cin>>n;
  for (int i=0;i<n;i++) cin>>a[i];
  cin>>m;
  for (i=0;i<m;i++) cin>>b[i];
  cout<<summa(a,n)<<"\n";
  cout<<summa(b,m);
}
float summa(float mass[100],int n)
{ float sum=0;
  for (int i=0;i<n;i++) sum+=mass[i];
  return sum;
}
int summa(int mass[100],int n)
{ int sum=0;
  for (int i=0;i<n;i++) sum+=mass[i];
  return sum;
}
```

Обратите внимание, что два текста функций `summa()` действительно отличаются только типом данных, их логика совершенно одинакова.

Указатель на функцию

Ранее было сказано, что возможность обратиться к объекту по адресу практически исчерпывает все потребности обращения к структурам данных. Но указатели позволяют обращаться не только к данным, но и к функциям. В языке С понятие объекта не ограничивается структурами данных, а расширяется на программные единицы. Мы можем определить адрес функции, передать его некоторой переменной и затем выполнить функцию,

используя переменную-указатель и не используя имя функции. В листинге 13.15 показан пример.

Листинг 13.15

```
#include <iostream.h>
#include <conio.h>
// Объявлен указатель p1 на функцию вида void имя(void)
void (*p1)();
// Объявлен указатель p1 на функцию вида int имя(int)
int (*p2)(int);
void example1();
int  example2(int);
void main(void)
{ clrscr();
  p1=&example1;    // указателю передан адрес функции
  (*p1)();         // выполнен вызов функции
  cout<<"\n";
  p2=&example2;    // указателю передан адрес функции
  int a=(*p2)(10); // выполнен вызов функции
  cout<<"Sum="<<a;
}
void example1()
{  for (int i=0;i<10;i++)
    cout <<i*i<<" ";
}
int example2(int a)
{ int sum=0;
  for (int i=0;i<10;i++)
    sum+=a*i;
  return sum;
}
```

Синтаксис объявлений и вызовов ясен из этих двух объявлений и вызовов функций, а для того чтобы четче понять мощь новых

возможностей, рассмотрим более содержательную задачу. Предположим, существует некоторая структура данных (в нашем примере — массив), заполненная какими-то содержательными значениями. И пусть с этими данными необходимо осуществлять операции, последовательность выполнения которых зависит от результата очередного шага обработки. Такая ситуация возможна, например, при обработке базы данных. В базах данных цикл обработки — это последовательность операций, выбираемых на каждом шаге пользователем из меню базы. Программа, записанная в листинге 13.16, показывает технику использования указателей на функцию в такой ситуации.

Листинг 13.16

```
#include <conio.h>
#include <iostream.h>
int (*uk[3])(int *);
int example1(int *);
int example2(int *);
int example3(int *);
void main(void)
{ clrscr();
  uk[0]=&example1;
  uk[1]=&example2;
  uk[2]=&example3;
  int *b;
  b=new int[10];
  for (int i=0;i<=9;i++) b[i]=i;
  int c;
  do
  { clrscr();
    cout <<"Введите код операции";
    cin>>c;
    int k=(*uk[c])(b);
    cout <<"Результат обработки "<<k;
  }
```

```
  while (c!=4);
}
int example1(int *a)
{ return 1;
}
int example2(int *a)
{ return 1;
}
int example3(int *a)
{ return 1;
}
```

Мы не описываем подробно, что делает та или иная функция, для демонстрации общей технологии это не имеет значения. Существенно важно то, что три функции в цикле вызываются по номеру одним оператором присваивания. Если бы в C не было возможности вызова функции по указателю, то тот же фрагмент пришлось бы записать, например, так:

```
cin >> c;
switch (c)
{ case 1: k=example1(b);
  case 2: k=example2(b);
  case 3: k=example3(b);
}
```

Конечно, для нашего примера разница небольшая, но в случае большого количества функций обработки, выигрыш в компактности текста, конечно, может быть значительнее.

Комментарии

Комментарий — это часть программы, не воспринимаемая компилятором как исполняемая. Это, в принципе, точное определение комментария. Осмысленным он быть не обязан, но, конечно, программисты используют комментарии в точном соответствии со смыслом этого термина, т. е. для внесения в программу пояс-

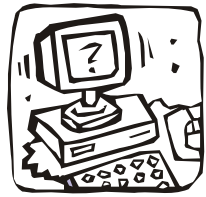
няющих записей. Создавая комментарии, вы можете не скупиться. Сколько бы текста вы в них ни записали, это никак не скажется на объеме кода, комментарии не компилируются, и, следовательно, текст кода не утяжеляют.

Язык C предлагает два типа комментариев:

- `//` — комментирует все знаки, находящиеся правее; комментируется только одна строка;
- `/*` — комментирует весь текст, находящийся правее, до знака `*/`, завершающего комментарий. Количество комментируемых строк не ограничивается.

Заключение

Функция языка C — это мощное и удобное средство построения программы. Функция — это тот минимальный кирпичик, из которого можно построить любое, сколь угодно сложное здание. Но для того чтобы здание было крепким, необходимо самым тщательным образом продумать о том, что именно будет делать каждая из функций и как наиболее эффективно организовать между функциями обмен данными. Языковых возможностей более чем достаточно для организации сколь угодно сложной схемы передачи данных.



Глава 14

Графика языка С

Общая информация о графических режимах

Начнем с простого примера, представленного в листинге 14.1.

Листинг 14.1

```
#include <graphics.h>
void main()
{ int dr=DETECT,md;
  initgraph(&dr,&md,"");
  line(10,10,200,200);
}
```

Это очень простая программа, выполняющая только две операции: активацию графического режима и прорисовку одной линии от точки с координатами `x=10`, `y=10` до точки с координатами `x=200`, `y=200`. Начинается текст программы с директивы `include`, подключающей файл graphics.h, который содержит прототипы графических функций. Первая строка функции `main()` объявляет две целые переменные

```
int dr=DETECT,md;
```

необходимые для активации графического режима. Первая переменная — это номер, идентифицирующий тип графического драйвера. Возможные варианты перечислены в табл. 14.1.

Таблица 14.1. *Типы графического драйвера*

Константа	Значение
DETECT	0
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

Далее мы всегда будем пользоваться константой DETECT, указание которой дает команду компилятору самостоятельно определить подходящий драйвер.

Вторая переменная (в нашем примере md) необходима для определения графического режима. В справочной системе компилятора дана подробная таблица всех возможных графических режимов. Мы здесь приводить ее не будем. То значение, которое дается компилятором по умолчанию, для наших целей вполне достаточно.

Активизируется графический режим следующей функцией:

```
initgraph(&dr,&md,"");
```

Два первых параметра этой функции мы только что обсудили, третий параметр — это путь к файлу драйвера. В нашем примере путь не указан. Это означает, что программа должна искать в текущей папке файл egavga.bgi. Если этого файла в текущей папке нет, то программа не будет работать, даже если в ней самой нет ошибок. Возможны и другие файлы драйверов, но мы всегда будем активировать графику именно так, как это сделано в нашем примере.

И, наконец, завершается пример функцией, рисующей одну диагональную линию. В точке завершения программы графический режим закрывается. В *приложении 2* приведено много программ, работающих с графикой. Поэтому мы сейчас не будем тратить время на детальные примеры. Остаток же главы посвятим небольшому перечню графических функций, для того чтобы создать хорошее представление о возможностях компилятора компании Borland. Еще раз повторимся. Материал, предлагаемый далее, описывает не столько сам язык С, сколько библиотеки функций. Впрочем, это обычная ситуация. Большая часть так называемых возможностей языка — это на самом деле возможности библиотек, и если их выбросить, то рассказ собственно о языке займет не так уж много места.

Некоторые функции

Функция *arc()*

Действие: прорисовка дуги.

Синтаксис:

```
arc(int x, int y, int angle1, int angle2, int radius)
```

Здесь:

- `x`, `y` — координаты центра дуги;
- `angle1`, `angle2` — начальный и конечный угол прорисовки;
- `radius` — радиус дуги.

Функция *bar()*

Действие: прорисовка прямоугольника.

Синтаксис:

```
bar(int x1, int y1, int x2, int y2)
```

Здесь:

- `x1`, `y1` — координаты левой верхней точки диагонали;
- `x2`, `y2` — координаты правой нижней точки диагонали.

Функция *bar3d()*

Действие: прорисовка параллелепипеда.

Синтаксис:

```
bar(int x1, int y1, int x2, int y2, int a, int b)
```

Здесь:

- `x1`, `y1` — координаты левой верхней точки внутренней диагонали;
- `x2`, `y2` — координаты правой нижней точки внутренней диагонали;
- `a` — глубина;
- `b` — определяет, где находится вершина.

Функция *circle()*

Действие: прорисовка окружности.

Синтаксис:

```
circle(int x, int y, int radius)
```

Здесь:

- `x`, `y` — координаты центра;
- `radius` — радиус окружности.

Функция *cleardevice()*

Действие: очистка экрана.

Синтаксис:

```
cleardevice()
```

Функция *closegraph()*

Действие: закрывает графический режим.

Синтаксис:

```
closegraph()
```

Функция *drawpoly()*

Действие: прорисовка ломаной линии.

Синтаксис:

```
drawpoly(int n, int *poly)
```

Здесь:

- `n` — количество вершин многоугольника;
- `*poly` — указатель на массив, содержащий значения координат. Координаты в массив записываются парами.

Синтаксис функции не так очевиден, поэтому проиллюстрируем ее действие законченной программой (листинг 14.2).

Листинг 14.2

```
#include <graphics.h>
void main()
{ int dr=DETECT,md;
  initgraph(&dr,&md,"");
  int coord[8]=
    {10,10,110,10,110,110,10,110};
  drawpoly(4,coord);
}
```

Функция *ellipse()*

Действие: прорисовка эллипса.

Синтаксис:

```
ellipse(int x, int y, int angle1, int angle2,
        int xradius, int yradius)
```

Здесь:

- `x`, `y` — координаты центра;
- `angle1`, `angle2` — начальный и конечный углы дуги в градусах;
- `xradius`, `yradius` — радиусы по осям координат.

Функция *floodfill()*

Действие: заливка замкнутого контура.

Синтаксис:

```
floodfill(int x, int y, int color)
```

Здесь:

- `x`, `y` — координаты точки, с которой начинается закраска;
- `color` — цвет контура, которым выполняется закраска.

Функция *getbkcolor()*

Действие: определяет текущий цвет фона.

Синтаксис:

```
int getbkcolor()
```

Функция *getcolor()*

Действие: определяет текущий цвет рисуемого контура.

Синтаксис:

```
int getcolor()
```

Функции *getmaxx()* и *getmaxy()*

Действие: функция `getmaxx()` вычисляет максимально возможное значение координаты *x*, а функция `getmaxy()` — максимально возможное значение координаты *y*.

Синтаксис:

```
int getmaxx()
int getmaxy()
```

Функция *getpixel()*

Действие: вычисляет цвет текущей точки.

Синтаксис:

```
unsigned getpixel(int x, int y)
```

Здесь `x`, `y` — координаты текущей точки.

Функции *getx()* и *gety()*

Действие: функция `getx()` вычисляет текущую координату *x*, а функция `gety()` — текущую координату *y*.

Синтаксис:

```
int getx()
int gety()
```

Функция *linerel()*

Действие: рисует линию от текущей точки на указанный отрезок.

Синтаксис:

```
linerel(int dx, int dy)
```

Здесь `dx`, `dy` — приращения по осям.

Функция *lineto()*

Действие: рисует линию от текущей точки до указанной.

Синтаксис:

```
lineto(int x, int y)
```

Здесь `x`, `y` — координаты точки, до которой следует выполнять рисование линии.

Функция *moverel()*

Действие: смещает текущую позицию на вектор.

Синтаксис:

```
moverel(int dx, int dy)
```

Здесь `dx`, `dy` — приращения по осям.

Функция *moveto()*

Действие: смещает текущую позицию в указанную точку.

Синтаксис:

```
moveto(int x, int y)
```

Здесь `x`, `y` — координаты точки.

Функция *outtext()*

Действие: выводит текст на экран в текущей позиции.

Синтаксис:

```
outtext(char *string)
```

Здесь `string` — выводимая строка.

Функция *outtextxy()*

Действие: выводит текст на экран в позиции (*x*, *y*).

Синтаксис:

```
outtext(int x, int y, char *string)
```

Здесь:

- `x`, `y` — координаты точки, в которой выводится строка текста;
- `string` — выводимая строка.

Функция *putpixel()*

Действие: установка точки на экране.

Синтаксис:

```
putpixel(int x, int y, int color)
```

Здесь:

- `x`, `y` — координаты точки;
- `color` — цвет точки.

Функция *rectangle()*

Действие: рисует прямоугольник по вершинам диагонали.

Синтаксис:

```
rectangle(int x1, int y1, int x2, int y2)
```

Здесь:

- `x1`, `y1` — координаты левой верхней точки;
- `x2`, `y2` — координаты правой нижней точки.

Функция *sector()*

Действие: рисует закрашенный сектор.

Синтаксис:

```
sector(int x, int y, int angle1, int angle2,
       int xradius, int yradius)
```

Здесь:

- `x`, `y` — координаты центра;
- `angle1`, `angle2` — начальный и конечный угол прорисовки в градусах;
- `xradius`, `yradius` — радиусы по осям.

Функция *setbkcolor()*

Действие: устанавливает цвет фона.

Синтаксис:

```
setbkcolor(int color)
```

Здесь `color` — цвет.

Функция *setcolor()*

Действие: устанавливает цвет линий.

Синтаксис:

```
setcolor( int color)
```

Здесь `color` — цвет.

Функция *setfillstyle()*

Действие: устанавливает цвет и шаблон закраски.

Синтаксис:

```
setfillstyle(int pattern, int color)
```

Здесь:

- `pattern` — шаблон закраски;
- `color` — цвет закраски.

Глава 15

Директивы

Директива *include*

Работая с языком программирования, вы имеете дело не с абстрактным описанием этого языка, а с конкретной реализацией, иначе говоря, со средой, обеспечивающей успешность ваших усилий в программировании. Главным элементом среды, конечно, будет компилятор, его действия мы обсуждаем на протяжении всей книги. Данная глава единственная, в которой мы будем говорить не о компиляторе, а о препроцессоре — менее важном, но существенном механизме среды программирования. Задача препроцессора заключается в преобразовании текста программы до того, как этот текст увидит компилятор.

Для препроцессора существует собственный набор команд. Он невелик, и одну команду из этого набора мы уже использовали, это команда `include`. Далее команды препроцессора будем называть директивами. Директива `include` включает некоторый текст, хранящийся в отдельном файле, в компилируемый текст. Используемые нами повсеместно директивы `include` занимаются включением заголовочных файлов, т. е. файлов, описывающих прототипы функций, но возможно включение и текста программы.

В листинге 15.1 приведен пример программы, уже нами многократно использованной для разных целей. Это программа поиска наибольшего значения в массиве. Данная программа разбита на два файла, один из которых головной, с него начинается компиляция, второй файл, содержащий функцию, подключается препроцессором директивой `include`.

Данная функция хранится в файле с именем maximum.cpp в текущей папке.

Листинг 15.1

```
int maximum(int *a,int n)
{ int max=*a;
  for (int i=0;i<n;i++)
   {if (max<*a) max=*a;
    a++;
   }
  return max;
}
```

Главный файл — с программой, место его расположения роли не играет.

```
#include <conio.h>
#include <iostream.h>
#include "maximum.cpp"
void main()
{ clrscr();
  int *a,*b;
  a=new int[100];
  b=a;
  int n; cin>>n;
  for (int i=0;i<n;i++)
   { cin >>*a;
     a++;
   }
  int max=maximum(b,n);
  cout <<max;
}
```

Обратите внимание, что дополнительный файл не является самодостаточной программой, поэтому его нельзя компилировать и выполнять. Кроме того, не забывайте, что директива включает

именно файл, поэтому если оба текста загружены в среду, то внесение изменений во включаемый текст не отразится в основном тексте до тех пор, пока вы не сохраните сделанные изменения в файле.

Директива include в нашем тексте работает в двух формах. В первой форме для заголовочных файлов имена файлов заключены в скобки <>, во второй форме для включаемого файла имя файла заключено в кавычки "". От того, как мы опишем имя файла, зависит, где препроцессор будет его искать. Для угловых скобок файл ищется так, как это прописано в установках среды. В случае кавычек файл ищется по указанному пути (можно указать детально путь), начиная с текущей папки.

Директива *define*

Директива define предназначена для определения так называемых макросов, иначе говоря, имен для последовательностей символов. Пример:

```
#define PI 3.14
```

Данный макрос определяет именованную константу — число π. Это простейший пример. Принципиально макрос может использоваться для любых замен. Макрос может быть использован даже вместо оператора. Программа, приведенная в листинге 15.2, выполняет ввод массива значений, суммирование элементов массива и вывод суммы. Все три действия описаны в макросах.

Листинг 15.2

```
#include <conio.h>
#include <iostream.h>
#define input cin>>a[i]
#define calc int sum=0;for (i=0;i<5;i++) sum+=a[i]
#define output cout <<sum
void main()
{ clrscr();
```

```
  int a[100];
  for (int i=0;i<5;i++)
   { input;
   }
  calc;
  output;
}
```

Пример показывает, что принципиально возможно практически всю программу оформить в виде макросов, если бы это было разумно. Иногда это действительно полезно. Наш пример, благодаря макросам, выглядит очень просто и понятно, но плата за использование макросов существенная. Если вы в своей деятельности пользуетесь программами-отладчиками, то макросы создадут для вас очень заметные проблемы. Например, макросы абсолютно не разбираются в проблемах типизации C и ничего не знают об областях видимости. Кроме того, для макросов не существует механизма перегрузки, как для функций. А самое существенное, пожалуй, — это то, что в точке определения макроса компилятор еще не работает, а, следовательно, ошибка в макросе может быть обнаружена только там, где макрос расширяется до определяемого им выражения. Это обстоятельство приводит к трудно обнаруживаемым ошибкам. Но от большинства проблем, создаваемых макросами, можно избавиться, если все необходимые макросы записывать в начале файла программы.

Более интересная форма макроса и более функциональная — это макрос с параметром. Пример приведен в листинге 15.3.

Листинг 15.3

```
#include <conio.h>
#include <iostream.h>
#define ab(x,y) x*y
void main(void)
{ clrscr();
  int sum=0;
```

```
  for (int i=0;i<5;i++)
   sum+=ab(i,2);
  cout <<sum;
}
```

Здесь определен макрос, перемножающий два числа, в круглых скобках указаны два параметра. Конечно, не вполне понятно, зачем такой макрос может быть нужен. Но мы ведь сейчас просто показываем технику их использования на простых примерах. Впрочем, повторимся, макросы вряд ли полезны для сокращения текста программы, их полезность заключается в возможности сложные выражения заменять на имена, которые четко и ясно указывают на смысл замененного выражения. С их использованием можно сложный код сделать легко понятным для человека, с языком С плохо знакомым, но хорошо владеющим другим языком программирования или просто английским. Мы можем целые фрагменты программы заменять на понятные английские слова или короткие фразы.

Строкообразующий оператор

В примере, приведенном далее, создан макрос, выводящий на экран значение параметра. В операции вывода перед именем параметра стоит оператор #.

```
#include <conio.h>
#include <iostream.h>
#define print(x) cout <<#x
void main(void)
{ clrscr();
  print("Пример подстановки символьной строки");
}
```

Действие строкообразующего оператора заключается в следующем:

- фактический параметр, передаваемый макросу, заключается в кавычки;

- у строки, передаваемой макросу, отбрасываются все начальные и все конечные пробелы.

Если в данном примере убрать символ #, то данная программа станет ошибочной.

Замечание о расположении макросов

Рассмотрим программу из листинга 15.4.

Листинг 15.4

```
#include <conio.h>
#include <iostream.h>
#define first 100
#define second first*2
void main(void)
{ clrscr();
  cout <<second;
}
```

Как видно из примера, макросы вполне могут использоваться другими макросами. При этом их взаимное положение не играет существенной роли. Макросы `first` и `second` вполне можно поменять местами, это не повлечет за собой изменение результата.

Макросы вполне могут играть роль аргументов для другого макроса, что видно из следующего примера (листинг 15.5).

Листинг 15.5

```
#include <conio.h>
#include <iostream.h>
#define argument1 10
#define argument2 20
#define sum(x,y)  x+y
```

```
void main(void)
{ clrscr();
  cout <<sum(argument1,argument2);
}
```

Но макросы не могут принимать участие в операторах языка С. Записи вида

```
if (t==1) #define sum(x,y) x+y
```

просто не имеют смысла в силу того, что макросы обрабатываются до этапа компиляции программы, и, следовательно, их нет в тексте ни во время компиляции, ни тем более во время исполнения программного кода.

Заключение

Главное, что необходимо понять, макросы — это средство более удобной записи, так сказать, средство косметического характера. Макросы вряд ли будут вам полезны в деле улучшения логики программы, однако некорректное их использование может осложнить жизнь дополнительными ошибками. Но если некоторое изящество в программировании не чуждо вашему стилю, то изучить макросы будет полезно. Впрочем, есть мнение, что любое использование макроса свидетельствует о плохом проектировании программы. Определяющим здесь, конечно, будет ваше личное мнение, которое вы обязательно сформируете после того, как уверенно овладеете языком. Мы главой о макросах завершаем изложение языка С, а у вас впереди еще очень существенное приложение в виде работающих программ различной степени сложности и трудоемкости.

ПРИЛОЖЕНИЯ

Приложение 1. Терминология

Приложение 2. Примеры задач

Приложение 1

Терминология

argc

Общепринятое имя первого аргумента для функции `main()`. Содержит значение целого типа, указывающее количество аргументов, передаваемых программе из командной строки.

argv

Общепринятое имя второго аргумента для функции `main()`. Является указателем на массив строк. Традиционно первая строка — это имя программы, а каждая последующая — аргумент, передаваемый программе из командной строки.

ASCII-код

Американский стандартный код для информационного обмена. Набор из 256 кодовых комбинаций, используемых для представления букв, цифр и иных символов. Стандартизированы только 128 первых комбинаций. Остальные определяются разработчиками компьютерных систем.

Esc-последовательность

Последовательность символов, на первом месте которой стоит косая черта и далее буква или последовательность цифр.

Абстрактный декларатор

Декларатор без идентификатора, состоящий из обозначения типа и, возможно, одного или нескольких модификаторов, задающих указатель, массив или функцию.

Агрегатные типы

Массивы, структуры, объединения.

Арифметические типы

Целочисленные типы, перечислимые типы и действительные.

Ассоциативность (порядок интерпретации)

В отношении операций это правила определения приоритетов.

Базовые типы данных

Целые (в том числе символьный), действительные и перечислимые типы.

Бинарное выражение

Выражение, состоящее из двух операндов, соединенных знаком бинарной операции.

Блок

Последовательность объявлений, описаний и операторов, заключенная в фигурные скобки.

Видимость

Характеристика объекта языка, определяющая область программы, в которой к этому объекту возможно обращение по имени.

Внешний уровень

Часть программы, расположенная вне всех объявлений функций.

Внутренний уровень

Фрагменты программы, расположенные внутри объявлений функций.

Время жизни

Временной интервал выполнения программы, в течение которого существует определенный объект.

Выражение

Комбинация операндов и символов операций, формирующая единственное выражение.

Директива

Команда препроцессора, предназначенная для выполнения какого-либо действия над текстом исходной программы до компиляции.

Декларатор

Идентификатор, чей смысл может быть изменен квадратными или круглыми скобками или символом "звездочка". Составной декларатор — это декларатор, содержащий более одного модификатора.

Именованная константа

Идентификатор, описанный в директиве препроцессора `define`, который представляет собой константное значение.

Именующее выражение (lvalue)

Выражение, ссылающееся на область памяти и необходимое в качестве левой части оператора присваивания или операнда унарной операции.

Идентификатор

Имя, используемое для обозначения переменных, типов, меток и функций. Идентификатор используется в программе для доступа к обозначенным им объектам, в операторах программы. Идентификатор есть последовательность, состоящая из одной или нескольких букв, цифр, символов подчеркивания. Причем данная после-

довательность не может начинаться с цифры. Некоторое количество символов, начиная с первого, считается значимым. Это означает, что идентификаторы, отличающиеся только незначимыми символами, считаются одним и тем же идентификатором. Количество значимых символов зависит от реализации. Литеры верхнего и нижнего регистра считаются различными.

Лексема

Последовательность символов, которая компилятором не раскладывается на составные части для последующего анализа. Примером лексемы являются ключевые слова и идентификаторы.

Ключевое слово

Слово со специальным определенным именно для компилятора значением.

Компонент

Элемент структуры или объединения.

Куча

Область памяти, предназначенная для размещения автоматических переменных.

Константа

Число, символ, строка символов, чье значение не может быть изменено в ходе работы программы. Константа с плавающей точкой — это числовая константа, содержащая целую и дробную части. Символьная константа — это один символ, заключенный в апострофы. Строковая константа — это последовательность символов, заключенная в кавычки.

Ключевое слово

Специальный идентификатор, известный компилятору и имеющий определенный смысл. Ключевое слово нельзя использовать

для собственных целей. Язык C содержит следующие ключевые слова:

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

Комментарий

Последовательность символов, которую компилятор понимает как один пробел, и, следовательно, комментарий не порождает выполняемого кода.

Макрос

Идентификатор, описанный в директиве препроцессора `define`, который замещает собой другую последовательность лексем.

Модель памяти

Определение принципов использования памяти для кода программ и данных. В C поддерживаются следующие модели:

- compact — модель, допускающая существование более чем одного сегмента данных и только одного сегмента кода;
- huge — модель, допускающая существование более одного сегмента кода и более одного сегмента данных и позволяющая отдельным объектам данных занимать более одного сегмента;
- large — модель, допускающая существование более одного сегмента для данных и более одного сегмента для кода, но в этой модели объекты данных не могут занимать более одного сегмента;

- medium — модель, допускающая более одного сегмента кода и только один сегмент данных;
- small — модель, допускающая только один сегмент кода и только один сегмент данных.

Объект

Область памяти, к которой может быть произведено обращение путем считывания. Модифицируемому объекту можно также присваивать значение.

Объявление

Устанавливает связь между именем и атрибутами объявляемого объекта, т. е. переменной, типа или функции.

Описание

Выполняет объявление функции и, кроме того, выделение памяти под описываемую переменную.

Операнд

Константа или переменная, над которой производятся действия в данном выражении.

Оператор

Конструкция, управляющая ходом выполнения программы. Язык С предоставляют программисту следующие операторы:

- `break` (прерывание);
- составной оператор;
- `continue` (продолжить);
- `do` (выполнить);
- оператор-выражение;
- `for` (реализация цикла);
- `goto` (переход на метку);

- `if` (проверка условия);
- пустой оператор;
- `return` (возврат);
- `switch` (переключатель);
- `while` (условный цикл).

Препроцессор

Текстовый процессор, изменяющий содержимое исходного файла на первом этапе компиляции.

Приоритет

Относительное положение символа операции в иерархии, определяющей порядок вычисления выражений.

Псевдоним

Альтернативное имя для обозначения одной и той же области памяти.

Прототип функции

Объявление функции, устанавливающее имя функции и тип возвращаемого значения. Кроме того, может определять типы аргументов и их количество. Прототип не используется для описания тела функции.

Символы операций

Символы, определяющие допустимые операции над величинами.

Сегмент

Область памяти, равная или меньшая 64 Кбайт, которая содержит программный код или данные.

Тег

Имя, присваиваемое структурному типу, типу объединения или перечислимому типу.

Тело функции

Составной оператор, содержащий объявления локальных переменных и операторы какой-либо функции.

Формальный параметр

Описание фактического аргумента, который может быть передан функции.

Фактический параметр

Значение допустимого типа, передаваемое в функцию.

Приложение 2

Примеры задач

Задача 1. Рекуррентная функция

Условие задачи.

Вычислить значение функции, заданной следующими условиями:

- $F(1) = 1$;
- $F(2N) = F(N)$;
- $F(2N + 1) = F(N) + F(N + 1)$.

При этом запрещается использовать массивы в любом виде, в том числе динамические, и рекурсию.

Краткая идея решения.

Шаг расчетного процесса определяется четырьмя числами — четным и нечетным аргументами — и двумя количествами — количеством четных и количеством нечетных. Ядро алгоритма — правило, по которому на каждом шаге процесса, исходя из четырех текущих чисел, определяется следующая четверка чисел. Процесс расчетов заканчивается, когда нечетное число становится равно 1. По окончании расчетного цикла суммируется количество четных и количество нечетных, и эта сумма и будет результатом.

Текст программы представлен в листинге П2.1.

Листинг П2.1

```
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
void main()
{ unsigned long chet,nechet,sum_chet,sum_nechet;
  clrscr();
  cin >> nechet;
  while (div(nechet,2).rem==0) nechet=div(nechet,2).quot;
  sum_chet=sum_nechet=1;
  do
   { chet=div(nechet,2).quot;
     nechet-=chet;
     if (div(chet,2).rem==1)
      { unsigned long c=chet;
        chet=nechet;
        nechet=c;
      }
     if (div(div(chet,2).quot,2).rem==0)
      sum_chet+=sum_nechet;
     else
      { unsigned long c=sum_nechet;
        sum_nechet+=sum_chet;
        sum_chet=c;
      }
   }
  while (nechet>1);
  cout << sum_nechet;
}
```

Задача 2. Перестановки

Условие задачи.

Дано произвольное множество символов. Требуется построить все перестановки из его элементов.

Краткая идея решения.

Будем называть N-перестановкой из N нумерованных элементов ряд, полученный из предыдущего ряда смещением элементов от начала к концу. При этом первый элемент переходит на второе место, второй на третье и т. д., последний на первое. Для того чтобы получить все возможные перестановки из множества в N элементов, необходимо построить все N-перестановки, для каждой N-перестановки построить все $N - 1$ перестановки и т. д.

Текст программы представлен в листинге П2.2.

Листинг П2.2

```
#include <iostream.h>
#include <conio.h>
void printing();
void recurs(int);
int m=0,n_big;
char a[20];
void main()
{ clrscr();
  cin >>n_big;
  for (int i=1;i<=n_big;i++) cin>>a[i];
  clrscr();
  printing();
  recurs(n_big);
}
void printing()
{ m++;
  cout <<m<<") ";
  for (int j=1;j<=n_big;j++) cout <<a[j]<<" ";
  cout <<"\n";
  getch();
}
void recurs(int n)
{ for (int i=1;i<=n;i++)
```

```
   { char c=a[n];
     for (int j=n;j>1;j--) a[j]=a[j-1];
     a[1]=c;
     if (i<n) printing();
     if (n>1) recurs(n-1);
   }
}
```

Задача 3. Выборки

Условие задачи.

Дано множество символов. Нужно построить все выборки из данного множества без повторений.

Краткая идея решения.

Определим числовой массив такой же длины, как и множество элементов, на котором должны быть построены выборки. Выборку будем формировать по следующему правилу: если соответствующий элемент числового множества равен 1, то элемент множества участвует в выборке, иначе — нет. Представим себе, что вспомогательное числовое множество — это двоичное число. Очередное двоичное число можно получить из имеющегося прибавлением единицы по правилам двоичной арифметики. Тогда, если за исходное число принять множество нулей, мы можем получить все возможные двоичные числа и, соответственно, все возможные выборки.

Текст программы представлен в листинге П2.3.

Листинг П2.3

```
#include <iostream.h>
#include <conio.h>
int proverka(int,int *);
void summa(int,int *);
```

```
void printing(int,char *,int *);
void main()
{ char a[100];
  int flag[100],n;
  clrscr();
  cin>>n;
  for (int i=1;i<=n;i++)
   { cin >>a[i];
     flag[i]=0;
   }
  while (proverka(n,flag))
   { summa(n,flag);
     printing(n,a,flag);
   }
}
int proverka(int n,int *flag)
{ int f=0;
  for (int i=1;i<=n;i++)
    if (!flag[i]) f=1;
  return f;
}
void summa(int n,int *flag)
{ int i=1;
  while (flag[i] && i<=n) i++;
  if (i<=n)
   { flag[i]=1;
     for (int j=1;j<=i-1;j++) flag[j]=0;
   }
}
void printing(int n,char *a,int *flag)
{ cout <<"-----------------------------"<<"\n";
  for (int i=1;i<=n;i++)
   if (flag[i]) cout <<a[i];
  cout <<"\n";
}
```

Задача 4. Шахматная доска

Условие задачи.

Нарисовать шахматную доску, используя минимум графических возможностей.

Краткая идея решения.

Одним циклом, рисующим точку со смещением на один пиксел, за шаг мы можем нарисовать отрезок. Далее запишем цикл, смещающий отрезок на один пиксел за один шаг, это даст квадрат. Далее, запишем цикл, на каждом шаге которого квадрат смещается (например, по горизонтали) на величину стороны, это даст полоску квадратов. Затем запишем цикл, смещающий полоску на величину стороны квадрата по вертикали, это даст нам одноцветную доску. Что же касается цвета, то его перемена осуществляется после прорисовки очередного квадрата, и еще одна перемена цвета выполняется после прорисовки полосы, если количество квадратов в линии нечетно, при четном количестве вторая перемена цвета не нужна.

Текст программы представлен в листинге П2.4.

Листинг П2.4

```
#include <conio.h>
#include <iostream.h>
#include <graphics.h>
#include <stdlib.h>
void main()
{ int n;
  cin>>n;
  int dr=DETECT,md,c=2;
  initgraph(&dr,&md,"");
  for (int k=1;k<=n;k++)
   { for (int i=1;i<=n;i++)
      { for (int y=1;y<=50;y++)
```

```
          for (int x=1;x<=50;x++)
            putpixel(10+x+50*(i-1),10+y+50*(k-1),c);
        if (c==1) c=2;else c=1;
      }
     if (!div(n,2).rem)
       if (c==1) c=2;else c=1;
   }
  getch();
}
```

Задача 5. Рекурсивная снежинка

Условие задачи.

Нарисовать картинку, такую, как показано на рис. П2.1.

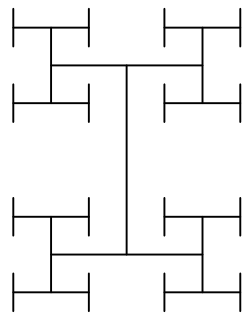


Рис. П2.1. Изображение снежинки

Краткая идея решения.

Каждый отрезок порождает еще два, поэтому в рекурсивную процедуру передаются параметры, позволяющие принять решение о том, какой отрезок должен быть построен. А именно процедуре сообщаются:

- ❒ координаты точки, от которой следует рисовать очередной отрезок;

- тип предыдущего отрезка (вертикальный или горизонтальный);
- длина предыдущего отрезка;
- номер вызова процедуры.

Получив необходимые данные, процедура прорисовывает отрезок, определяет новые значения параметров и выполняет два новых вызова.

Текст программы представлен в листинге П2.5.

Листинг П2.5

```
#include <graphics.h>
void rec_uzor(int,int,int,int,int);
void main()
{ int dr=DETECT,md;
  initgraph(&dr,&md,"");
  line(300,50,300,250);
  rec_uzor(300,50,1,200,1);
  rec_uzor(300,250,1,200,1);
}
void rec_uzor(int x,int y,int k,int l,int n)
{ int x1,x2,y1,y2;
  if (n<10)
   { l=l/2;
     if (k==1)
      { x1=x-l;x2=x+l;
        line(x1,y,x2,y);
        rec_uzor(x1,y,2,l,n+1);
        rec_uzor(x2,y,2,l,n+1);
      }
     else
      { y1=y-l;y2=y+l;
        line(x,y1,x,y2);
```

```
        rec_uzor(x,y1,1,l,n+1);
        rec_uzor(x,y2,1,l,n+1);
      }
   }
}
```

Задача 6. Ханойская башня

Условие задачи.

Даны три подставки. На первой из них лежит некоторое количество дисков, таких, что для любой пары дисков тот, что сверху, имеет радиус меньший того, что ниже. Необходимо переложить все диски на третью подставку, не нарушая следующих правил:

- за один раз можно брать только один диск;
- диск можно класть только на диск большего радиуса или на пустую подставку.

Краткая идея решения.

Обозначим подставки например так: a_1, a_2, a_3. Пусть исходная пирамида высотой N находится на подставке a_1, и ее требуется перенести на подставку a_3, используя подставку a_2 как вспомогательную. Запишем такую задачу следующим образом (a_1, a_2, a_3, N). Теперь предположим, что была решена следующая задача (a_1, a_3, a_2, $N-1$), т. е. на подставку a_2 была перенесена вся пирамида кроме самого нижнего диска. Тогда исходная задача будет решена переносом одного диска с a_1 на a_3, и далее решением задачи является задача (a_2, a_3, a_1, $N-1$). Таким образом, решение задачи о переносе ханойской башни сводится к комбинации задач, в которых подставки меняются ролями, и простых операций переноса одного диска. Роли для подставок возможны следующие:

- исходная подставка;
- подставка-цель;
- вспомогательная подставка.

Текст программы представлен в листинге П2.6.

Листинг П2.6

```
#include <graphics.h>
#include <iostream.h>
#include <conio.h>
struct mas{
  int krug[10],h;};
void hanoy(int,int,int,int,mas *);
void ris(mas *);
void perest(int,int,mas *);
int n;
void main()
{ int dr=DETECT,md;
  initgraph(&dr,&md,"");
  mas *a;
  a=new mas[3];
  cin>>n;
  for (int i=1;i<=n;i++)
    a[0].krug[i]=a[1].krug[i]=a[2].krug[i]=0;
  for (i=1;i<=n;i++) a[0].krug[i]=(n-i+1)*10;
  a[0].h=n;a[1].h=a[2].h=0;
  ris(a);
  getch();
  hanoy(0,1,2,n,a);
}
void hanoy(int a1,int a2,int a3,int m,mas *a)
{ if (m==1) perest(a1,a3,a);
  else
   { hanoy(a1,a3,a2,m-1,a);
     perest(a1,a3,a);
     hanoy(a2,a1,a3,m-1,a);
   }
}
void perest(int a1,int a3,mas *a)
{ a[a3].h++;
```

```
  a[a3].krug[a[a3].h]=a[a1].krug[a[a1].h];
  a[a1].krug[a[a1].h]=0;
  a[a1].h--;
  ris(a);
  getch();
}
void ris(mas *a)
{ cleardevice();
  for (int i=1;i<=n;i++)
   { if (a[0].krug[i]>0) circle(100,200,a[0].krug[i]);
     if (a[1].krug[i]>0) circle(300,200,a[1].krug[i]);
     if (a[2].krug[i]>0) circle(500,200,a[2].krug[i]);
   }
}
```

Задача 7. График соревнований

Условие задачи.

Требуется составить график соревнований по виду спорта, предполагающему парные состязания. Это может быть бокс или фехтование. Вид спорта роли не играет, существенно важно выполнить следующие условия:

- в каждом бою участвуют два спортсмена;
- проигравший спортсмен выбывает из соревнований;
- в любом поединке встречаются спортсмены, проведшие или равное количество боев, или у одного из них количество боев на один меньше.

Краткая идея решения.

Процесс построения графика — это процесс определения пар бойцов из оставшихся. Пары формируются соседями. Иногда возможна ситуация, в которой крайний правый боец остается без пары (пары формируются слева направо). Тогда на следующем круге формирование пар начинается не с первого бойца, а со второго.

Это гарантированно обеспечивает пару тому, кто на предыдущем круге остался без пары. Если без пары остался крайний левый, тогда на следующем круге формирование пар начинается с первого, т. е. с крайнего левого.

Текст программы представлен в листинге П2.7.

Листинг П2.7

```
#include <iostream.h>
#include <graphics.h>
#include <stdlib.h>
void ris(int *,int,int);
void main()
{ int n;cin >>n;
  int *x;x=new int[n];
  for (int i=1;i<=n;i++) x[i]=20*i-10;
  int dr=DETECT,md;initgraph(&dr,&md,"");
  ris(x,n,1);
  int k=1,tp=0;
  while (n>1)
   { k++;
     int m=div(n,2).quot;
     if (div(n,2).rem==1)
      { for (i=1+tp;i<=m+tp;i++)
          x[i]=div(x[2*i-1-tp]+x[2*i-tp],2).quot;
        m++;
        if (tp==0)
          { tp=1;
            x[m]=x[n];
          }
        else tp=0;
      }
     else
       for(i=1;i<=m;i++)
```

```
        x[i]=div(x[2*i-1]+x[2*i],2).quot;
     n=m;
     ris(x,n,k);
   }
}
void ris(int *a,int m,int y)
{ for (int i=1;i<=m;i++)
   circle(a[i],y*40,5);
}
```

Задача 8. Поиск прямоугольника наибольшей площади

Условие задачи.

Дан двумерный массив, заполненный случайным образом нулями и единицами. Найти прямоугольник наибольшей площади, заполненный единицами.

Краткая идея решения.

В программе исходный массив имеет "площадь" 1400 элементов. Это максимально возможная площадь, которая может быть заполнена единицами. Поэтому поиск начинается от этого значения. На каждом шаге поиска в случае неудачи искомая площадь уменьшается на 1, и поиск продолжается. Поиск заключается в отыскании точки исходного массива, которая может стать левой верхней вершиной прямоугольника. Для этого организуется обход всех элементов массива и в каждом из них осуществляется попытка построения прямоугольника искомой площади всех допустимых форм. Допустимость формы определяется так: пусть искомая площадь равна *S*. Предположим, что горизонтальная сторона равна *LX*. Если *S* делится на *LX* нацело, то форма со стороной LX допустима.

Текст программы представлен в листинге П2.8.

Листинг П2.8

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
int a[71][21];
int square(int,int,int,int);
void main()
{ textcolor(15);
  clrscr();
  randomize();
  for (int y=1;y<=20;y++)
   for (int x=1;x<=70;x++)
    { int c=random(40);
      if (c==39) a[x][y]=0; else a[x][y]=1;
    }
  for (y=1;y<=20;y++)
   for (x=1;x<=70;x++)
    { gotoxy(x,y);cout <<a[x][y];
    }
  getch();
  int s=1400,q=0;
  do
  { gotoxy(40,22);cout <<"      ";
    gotoxy(40,22);cout <<s;
    int ly=1;
    do
    { int lx=1;
      do
      { if (lx*ly==s)
        { int y=1;
          do
            { int x=1;
              do
```

```
               { q=square(x,y,lx,ly);
                 x++;
                 if (x+lx>70) x=70;
               }
              while ((q!=1) &&(x!=70));
              y++;
              if (y+ly>20) y=20;
            }
          while (q!=1 && y!=20);
        }
        lx++;
      }
      while (q!=1 && lx!=70);
      ly++;
    }
    while (q!=1 && ly!=20);
    s--;
  }
  while (s!=1 && q!=1);
  getch();
}
int square(int x,int y,int lx,int ly)
{ int s=0;
  for (int i=x;i<=x+lx;i++)
   for (int j=y;j<=y+ly;j++)
     if (a[i][j]==0) s=1;
  if (s==0)
   for (int i=x;i<=x+lx;i++)
    for (int j=y;j<=y+ly;j++)
     { gotoxy(i,j);
       cout <<'+';
     }
  if (s==0) return 1; else return 0;
}
```

Задача 9. Выборка из миллиарда

Условие задачи.

Из числового интервала от единицы до миллиарда выбираются случайным образом без повторений миллион чисел и записываются в файл. Необходимо за приемлемое время выяснить, какое наименьшее число отсутствует в файле. Использовать массивы или иные структуры данных, их заменяющие, запрещается.

Краткая идея решения.

Разобьем искомый миллиард на 10 интервалов. Затем прочитаем файл и подсчитаем для каждого из интервалов, сколько в него попадает чисел из этого миллиарда. Далее найдем первый из незаполненных интервалов. Неполнота определяется просто. Если миллиардный интервал разделить на 10, то в каждом из их должно оказаться 100 миллионов чисел, следовательно, неполный интервал — это тот, в котором чисел меньше, чем 100 миллионов. Если продолжать эту процедуру деления, то рано или поздно мы придем к неполному интервалу длиной в одно число. Это число и будет искомым.

Текст программы представлен в листинге П2.9.

Листинг П2.9

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
void main()
{ FILE *f;
  int s[11];
  clrscr();
  f=fopen("dat","r");
  int lng=10000,x=0,a;
  for (int n=1;n<=4;n++)
   { fseek(f,0,SEEK_SET);
```

```
    for (int i=1;i<=10;i++) s[i]=0;
    while (!feof(f))
     { fread(&a,2,1,f);
       if (a>=x && a<x+lng)
        { i=div(a-x,div(lng,10).quot).quot+1;
          s[i]++;
        }
     }
    for (i=1;i<=10;i++) cout <<s[i]<<" ";
    cout<<"\n";
    getch();
    i=1;
    while (s[i]>=div(lng,10).quot) i++;
    cout <<"i="<<i;
    x+=(i-1)*div(lng,10).quot;
    lng=div(lng,10).quot;
  }
  cout <<x;
}
```

Задача 10. Раскладывание колечек по штырькам

Условие задачи.

В каждую клетку шахматной доски воткнут штырек. В нашем распоряжении есть некоторое количество колечек, которые можно нанизывать на штырьки, причем на один штырек можно нанизать несколько колечек. Требуется подсчитать, сколькими способами можно распределить все эти колечки по штырькам. Два распределения считаются разными, если на двух соответствующих штырьках находится разное количество колечек.

Краткая идея решения.

Конструкцию доски можно представить двумерным массивом. Затем двумерный массив можно развернуть в одномерный, задача

от этого никак не изменится. Далее задача сведется к двум стандартным комбинаторным задачам: первая — нужно получить все возможные выборки из "штырьков". Каждая такая выборка — это набор штырьков, на которые допустимо раскладывать "колечки". Колечки можно представить единицами. То есть если, к примеру, элемент массива равен 5, это будет означать, что на этом "штырьке" лежит 5 колечек. После построения выборки встает вторая задача — построение всех возможных заполнений. Известно, что сумма единиц должна быть равна заданному числу. В исходном положении каждый элемент массива равен 1. Если при этом оказывается, что сумма элементов больше заданного числа, то такая выборка считается недопустимой. Если же сумма меньше либо равна, то выборка допустима. Принцип прибавления единицы (добавления колечка) выглядит так. Пусть некоторое количество элементов выборки содержит некоторую сумму, обозначим ее через S_1. Тогда на оставшиеся элементы (назовем их дополнительной выборкой) приходится сумма $S - S_1$, и необходимо решить задачу заполнения дополнительной выборки суммой $S - S_1$. Ясно, что такой процесс имеет рекурсивную природу. Для полного построения рекурсии необходимо определить простейшую ситуацию. Такой ситуацией будет дополнительная выборка из одного элемента с остатком суммы S_N.

Текст программы представлен в листинге П2.10.

Листинг П2.10

```
#include <iostream.h>
#include <conio.h>
int Final();
void Print();
void Plus_1(int,int);
int Digit();
int num=0,a[100],b[100],n,l;
void main()
{ clrscr();
  cin>>n>>l;
```

```
  n*=n;
  for (int i=1;i<=n;i++) b[i]=0;
  while (Final()==1)
   { int sum=Digit();
     if (sum<=l)
      { for (int i=1;i<=n;i++) a[i]=b[i];
        if (sum==l) Print();
        Plus_1(sum,1);
      }
   }
}
int Final()
{ for (int i=1;i<=n;i++)
   if (b[i]==0) return 1;
  return 0;
}
void Print()
{ num++;
  cout <<num<<") ";
  for (int i=1;i<=n;i++)
   cout <<a[i]<<" ";
  cout <<"      ";
  for (i=1;i<=n;i++)
   cout <<b[i]<<" ";
  cout<<"\n";
}
void Plus_1(int sum,int x)
{ if (x<=n && sum<l)
    for (int i=x;i<=n;i++)
     if (b[i]==1)
      { a[i]++; sum++;
        if (sum==l) Print();
        if (sum<l) Plus_1(sum,i);
        a[i]--; sum--;
      }
```

```
}
int Digit()
{ for (int i=1;(b[i]==1 && i<=n);i++);
  b[i]=1;
  for (int k=1;k<i;k++)
    b[k]=0;
  int m=0;
  for (i=1;i<=n;i++)
    m+=b[i];
  return m;
}
```

Задача 11. Разбиение кучи камней на две равного веса

Условие задачи.

Дано множество камней разного веса. Разбросать их на две кучи максимально одинакового веса.

Краткая идея решения.

Переборное решение задачи сводится к построению выборки, которая становится первой кучей. Вторая же куча — это оставшиеся камни.

Текст программы представлен в листинге П2.11.

Листинг П2.11

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
typedef unsigned mass[20];
int proverka(mass,int);
void add_1(mass,int);
void massiv(mass, mass, mass, mass, int);
```

```
void main()
{ mass a,b,c1,c2;
  unsigned n,s1,s2,min;
  clrscr();
  cin >>n;
  for (int i=1;i<=n;i++)
   { cin >>a[i];
     b[i]=0;
   }
  int q=1;
  while (proverka(b,n))
   { add_1(b,n);
     s1=s2=0;
     for (i=1;i<=n;i++)
       if (b[i]==0) s1+=a[i]; else s2+=a[i];
     if (q)
      { min=abs(s1-s2);
        q=0;
        massiv(a,b,c1,c2,n);
      }
     else
      if (abs(s1-s2)<min)
       { min=abs(s1-s2);
         massiv(a,b,c1,c2,n);
       }
   }
  cout <<"min= "<<min<<"\n";
  for (i=1;i<=n;i++)
   if (c1[i]>0) cout<<c1[i]<<"  ";
  cout<<"\n";
  for (i=1;i<=n;i++)
   if (c2[i]>0) cout<<c2[i]<<"  ";
}
int proverka(mass b,int n)
{ for (int i=1;i<=n;i++)
```

```
   if (b[i]==0) return 1;
  return 0;
}
void add_1(mass b,int n)
{ int i=1;
  while (b[i]==1 && i<=n) i++;
  b[i]=1;
  for (int k=1;k<=i-1;k++) b[k]=0;
}
void massiv(mass a, mass b, mass c1, mass c2, int n)
{ for (int k=1;k<=n;k++)
   if (b[k])
    {c1[k]=a[k];
     c2[k]=0;
    }
   else
    {c1[k]=0;
     c2[k]=a[k];
    }
}
```

Задача 12. Движение в поле сил тяготения

Условие задачи.

Несколько тел известной массы с известными векторами начальных скоростей находятся в пустом пространстве (если другие тела и существуют, то их влиянием можно пренебречь). Необходимо построить траектории их движения.

Краткая идея решения.

Разобьем временной интервал эксперимента на малые интервалы, в пределах которых взаимодействие тел будем игнорировать, а движение считать равноускоренным. Пересчет ускорений при

этом будет выполняться на границах интервалов. При такой схеме расчетов неизбежна погрешность расчетов, но уменьшением размера интервала эту погрешность можно сделать сколь угодно малой.

Текст программы представлен в листинге П2.12.

Листинг П2.12

```
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
#include <math.h>
void main()
{ struct {float x,y,vx,vy,m,ax,ay;} body[10];
  int n,i,j;
  cin >> n;
  for (i=1;i<=n;i++)
   cin >> body[i].x >> body[i].y >> body[i].vx >> body[i].vy
       >> body[i].m;
  int dr=DETECT,md;initgraph(&dr,&md,"");
  do
   { for(i=1;i<=n;i++)
      { body[i].ax=0;
        body[i].ay=0;
        for (int j=1;j<=n;j++)
        if (i!=j)
         { float l=pow(pow(body[i].x-body[j].x,2)+
                   pow(body[i].y-body[j].y,2),0.5);
           body[i].ax=body[i].ax+body[j].m/(l*l*l)*
                     (body[j].x-body[i].x);
           body[i].ay=body[i].ay+body[j].m/(l*l*l)*
                     (body[j].y-body[i].y);
         }
        body[i].vx=body[i].vx+0.001*body[i].ax;
        body[i].vy=body[i].vy+0.001*body[i].ay;
```

```
      }
     for (i=1;i<=n;i++)
      { body[i].x=body[i].x+body[i].vx*0.001;
        body[i].y=body[i].y+body[i].vy*0.001;
        putpixel(body[i].x,body[i].y,15);
      }
   }
  while (!kbhit());
}
```

Задача 13. Одинокий путник с плохой памятью

Условие задачи.

На квадратном поле установлены препятствия произвольной формы, и помечены два пункта — *A* и *B*. Перед путником поставлена задача: найти путь из *A* в *B*. Известно, что путник не располагает картой местности, у него очень плохое зрение и очень плохая память, настолько плохая, что он практически не может запомнить приметы пройденного маршрута. Из средств ориентации путник располагает естественным чувством ориентации в пространстве своего тела, т. е. он может сказать, где у него левая рука, а где правая. Также у него есть прибор, напоминающий компас. Этот прибор всегда, в любой точке поля показывает направление на пункт *B*.

Краткая идея решения.

Путь путешественника делится на две периодически повторяющиеся части: путь по пустому пространству и путь вдоль препятствия. Путь по пустому пространству возможен по компасу. Путь вдоль препятствия заключается в обходе его до тех пор, пока количество поворотов направо не сравняется с количеством поворотов налево. Равенство этих двух количеств означает, что путешественник зашел за угол, за которым, возможно, открывается прямой путь.

Текст программы представлен в листинге П2.13.

Листинг П2.13

```
#include <graphics.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
const int n=35,ax=10,ay=10,bx=230,by=430;
int x,y;
void move();
void soround();
int stenka();
int situation(int);
void move1(int);
void move2(int);
int turn90(int);
void main()
{int dr=DETECT,md;initgraph(&dr,&md,"");
 randomize();
 for (int i=1;i<=n;i++)
  if (i!=15)
   { setcolor(i);
     setfillstyle(1,i);
     int x=50+random(400);
     int y=50+random(300);
     int dx=2+random(100);
     int dy=2+random(100);
     rectangle(x,y,x+dx,y+dy);
     floodfill(x+dx/2,y+dy/2,i);
   }
 setcolor(15);
 setfillstyle(1,15);
 circle(bx,by,2);
 floodfill(bx,by,15);
```

```
 x=ax;y=ay;
 do
  { move();
    soround();
  }
 while (abs(x-bx)>3 || abs(y-by)>3);
}
void move()
{ float x1=0,y1=0;
  int q=0;
  float l=pow(pow(bx-x+0.0,2)+pow(by-y+0.0,2),0.5);
  do
   { float dx=(bx-x)/l,dy=(by-y)/l;
     x1+=dx;y1+=dy;
     if (abs(x1)>=1)
      if (getpixel(x+x1,y)==0)
       { x+=x1;
         putpixel(x,y,15); delay(5000); putpixel(x,y,0);
         x1=0;
       }
      else q=1;
     if (abs(y1)>=1)
      if (getpixel(x,y+y1)==0)
       { y+=y1;
         putpixel(x,y,15); delay(5000); putpixel(x,y,0);
         y1=0;
       }
     else q=1;
   }
  while (q==0);
}
void soround()
{ int n=stenka(),p=1;
  do
   { int k=situation(n);
```

```
     switch (k)
      { case 1:{ move1(n);
                 p--;
                 n=stenka();
               } break;
        case 2:move2(n);break;
        case 3:{n=turn90(n);
                p++;
               }
      }
   }

  while (p!=0);
  delay(25);putpixel(x,y,0);
}
int stenka()
{if (getpixel(x,y+1)!=0) return 1;
  else if (getpixel(x+1,y)!=0) return 2;
   else if (getpixel(x,y-1)!=0) return 3;
    else return 4;
}
int situation(int n)
{ switch(n)
  { case 1:if (getpixel(x,y+1)==0) return 1;
           else if (getpixel(x+1,y)==0) return 2;
             else return 3;
    case 2:if (getpixel(x+1,y)==0) return 1;
           else if (getpixel(x,y-1)==0) return 2;
             else return 3;
    case 3:if (getpixel(x,y-1)==0) return 1;
           else if (getpixel(x-1,y)==0) return 2;
             else return 3;
    case 4:if (getpixel(x-1,y)==0) return 1;
           else if (getpixel(x,y+1)==0) return 2;
             else return 3;
```

```
  }
return 0;
}
void move1(int n)
{ putpixel(x,y,0);
  switch (n)
  { case 1:y++;break;
    case 2:x++;break;
    case 3:y--;break;
    case 4:x--;
  }
  putpixel(x,y,15);delay(5000);
}
void move2(int n)
{ putpixel(x,y,0);
  switch (n)
  { case 1:x++;break;
    case 2:y--;break;
    case 3:x--;break;
    case 4:y++;
  }
  putpixel(x,y,15);delay(5000);
}
int turn90(int n)
{ if (n<4) return n+1; else return 1;
}
```

Задача 14. Метод минимакса

Условие задачи.

Дано дерево двоичной игры определенной глубины. Для конечных ситуаций определены целочисленные оценки. Определить методом минимакса оптимальный вариант игры.

Краткая идея решения.

С каждым узлом дерева связны два узла более низкого порядка, имеющие числовые значения. В соответствии с методом минимакса, если в текущем узле решение принимает главный игрок, то данному узлу присваивается максимальная оценка, иначе — минимальная. Так как перед началом процесса числовые значения имеют только самые глубокие, то подъем оценки наверх начинается с предпоследнего уровня дерева.

Текст программы представлен в листинге П2.14.

Листинг П2.14

```
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
struct node{int ocenka; node *left; node *right;};
void Create_Tree(node *,int);
int Minimax(node *, int, int);
void main()
{ randomize();
  clrscr();
  node *uk;
  Create_Tree(uk,1);
  Minimax(uk,1,40);
}
void Create_Tree(node *uk,int n)
{ if (n<4)
   { uk->left=new node;
     Create_Tree(uk->left,n+1);
     uk->right=new node;
     Create_Tree(uk->right,n+1);
   }
  else uk->ocenka=random(20);
}
int Minimax(node *uk, int n,int x)
```

```
{ int left,right,k;
  if (n<4)
   {left=Minimax(uk->left,n+1,x-div(20,n+1).quot+n);
    right=Minimax(uk->right,n+1,x+div(20,n).quot-n);
    if (div(n,2).rem==0)
      if (left<right) k=left; else k=right;
    else if (left<right) k=right; else k=left;
   }
  else k=uk->ocenka;
  gotoxy(x,n*4);cout<<k;
  return k;
}
```

Задача 15. Экономный обход графа

Условие задачи.

Дан ориентированный граф. Необходимо совершить его полный обход с использованием минимальных дополнительных данных.

Краткая идея решения.

Данная программа есть вариант алгоритма Дойча, построенный для графов с обязательным условием: для каждого узла количество входящих ветвей не меньше количества исходящих. Это условие необходимо, т. к. входящие ветви используются для запоминания исходящих, для пути, по которому был продолжен обход.

Текст программы представлен в листинге П2.15.

Листинг П2.15

```
#include <iostream.h>
#include <conio.h>
struct {
 int count,num;
 int uk[256];
} node[100];
```

```
int tek_index,pred_index;
void print()
{ if (node[tek_index].num>0)
   { cout <<node[tek_index].num<<" ";
     node[tek_index].num=0;
     getch();
   }
}
void main()
{ clrscr();
  int n,m;
  cout<<" Введите количество узлов";cin>>n;
  for (int i=1;i<=n;i++)
   {cout<<"Узел номер"<<i<<"  ";
    cout<<" Введите значение узла-";cin>>node[i].num;
    for (int j=1;j<=255;j++) node[i].uk[j]=0;
    node[i].count=0;
    cout<<" Введите количество ссылок - ";cin>>m;
    for (j=1;j<=m;j++)
     { cout<<"Ссылка номер - "<<j<<" =";
       cin>>node[i].uk[j];
     }
   }
  tek_index=pred_index=1;
  int q=0;
  do
   { int m=1;
     while (node[tek_index].uk[m]!=0 && m<255) m++;
     if (m==255) m=0; else m--;
     if (node[tek_index].count<m)
      { print();
        m-=node[tek_index].count;
        if (tek_index>1) node[tek_index].count++;
        int c=tek_index;
        tek_index=node[tek_index].uk[m];
```

```
        if (c>1) node[c].uk[m]=pred_index;
        else node[c].uk[m]=0;
        pred_index=c;
       }
      else
       { print();
        if (node[tek_index].count>0)
          { m=node[tek_index].count;
            node[tek_index].count--;
            int c=node[tek_index].uk[m];
            node[tek_index].uk[m]=0;
            tek_index=c;
          }
        else tek_index=pred_index;
       }
      if (tek_index==1)
       { q=1;
        for (int j=1;j<=255;j++)
          if (node[1].uk[j]!=0) q=0;
       }
    }
  while (q==0);
}
```

Задача 16. Закраска односвязного контура

Условие задачи.

На плоскости дан произвольный односвязный контур. Требуется его закрасить.

Краткая идея решения.

Предположим, что одна точка внутренней области контура закрашена. Определим ее соседей, также находящихся внутри контура. Эти точки станут новыми кандидатами на закраску. Запус-

тим процесс поиска соседей для каждой новой найденной точки. Когда кандидаты на закраску закончатся, контур будет полностью закрашен.

Текст программы представлен в листинге П2.16.

Листинг П2.16

```
#include <conio.h>
#include <iostream.h>
#include <graphics.h>
#include <dos.h>
struct coord{int x,y;};
coord front[10000];
int max;
void paint(int x, int y)
{ if (getpixel(x,y)==0)
   { putpixel(x,y,15);
     max++;
     front[max].x=x;
     front[max].y=y;
   }
}
void main()
{ int n,i,x,y;
  coord point[100];
  /* cin>>n;
    for (i=1;i<=n;i++)
      cin>>point[i].x>>point[i].y;
    cin>>x>>y;
  */
  n=9;
  x=100;y=110;
  point[1].x=100;point[1].y=100;
  point[2].x=240;point[2].y=120;
  point[3].x=200;point[3].y=200;
```

```
  point[4].x=250;point[4].y=150;
  point[5].x=270;point[5].y=280;
  point[6].x=135;point[6].y=180;
  point[7].x=90;point[7].y=280;
  point[8].x=30;point[8].y=170;
  point[9].x=60;point[9].y=200;
  int dr=DETECT,md;initgraph(&dr,&md,"");
  for (i=1;i<n;i++)
    line(point[i].x,point[i].y,point[i+1].x,point[i+1].y);
  line(point[n].x,point[n].y,point[1].x,point[1].y);
  getch();
  max=1;
  front[1].x=x;front[1].y=y;
  do
   { delay(10);
     x=front[max].x;
     y=front[max].y;
     max--;
     paint(x,y-1);
     paint(x+1,y);
     paint(x,y+1);
     paint(x-1,y);
   }
  while (max>1);
  getch();
}
```

Задача 17. Живая группа ГО

Условие задачи.

В игре ГО, правила которой, конечно, сейчас изучать не будем, есть понятие живой группы. В задаче требуется, имея конкретную позицию и координаты одного камня, установить, является ли этот камень представителем живой группы.

Краткая идея.

Данная задача, несмотря на различие в формулировках, решается так же, как и предыдущая. А именно: один камень группы известен и, начиная с него, запускается процесс поиска соседей. Если процесс завершен и ни у одного из найденных соседей не обнаружилось свободного соседнего поля, то группа не живая, иначе — живая.

Текст программы представлен в листинге П2.17.

Листинг П2.17

```
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
int doska[20][20],len;
struct {int x,y;} mas[300];
int control(int x, int y)
{ switch (doska[x][y])
  { case 0: return 1;
    case 1: { int q=1;
              for (int i=1;i<=len;i++)
                if (mas[i].x==x && mas[i].y==y) q=0;
              if (q)
              { len++;
                mas[len].x=x;
                mas[len].y=y;
                floodfill(20*x+10,20*y+10,1);
                getch();
              }
              return 0;
            }
    case 2:
    case 3: return 0;
  }
}
```

```
void main()
{int x,y;
/*
 int n; cin>>n;
 for (int i=1;i<=19;i++)
   for (int j=1;j<=19;j++) doska[i][j]=0;
 for (i=1;i<=n;i++)
  { cin>>x>>y;
    doska[x][y]=1;
  }
 cin>>n;
 for (i=1;i<=n;i++)
  { cin>>x>>y;
    doska[x][y]=2;
  }
 cin>>x>>y;
 */
 doska[12][14]=1;doska[12][13]=1;doska[13][13]=1;
 doska[14][13]=1;doska[15][13]=1;doska[15][14]=1;
 doska[15][15]=1;doska[16][16]=1;doska[16][17]=1;
 doska[16][18]=1;doska[16][19]=1;doska[15][18]=1;
 doska[14][18]=1;doska[13][18]=1;doska[13][17]=1;
 doska[13][16]=1;doska[14][16]=1;doska[12][16]=1;
 doska[12][15]=1;

 doska[14][12]=2;doska[13][15]=2;doska[14][15]=2;
 doska[13][14]=2;doska[14][14]=2;doska[11][14]=2;
 doska[11][13]=2;doska[11][12]=2;doska[12][12]=2;
 doska[13][12]=2;doska[15][12]=2;doska[16][12]=2;
 doska[16][13]=2;doska[16][14]=2;doska[16][15]=2;
 doska[15][16]=2;doska[11][15]=2;doska[11][16]=2;
 doska[12][17]=2;doska[14][17]=2;doska[12][18]=2;
 doska[12][19]=2;doska[13][19]=2;doska[14][19]=2;
 doska[15][19]=2;doska[15][17]=2;doska[17][17]=2;
```

```
 doska[17][16]=2;doska[17][18]=2;doska[17][19]=2;
 x=12;y=14;
 int dr=DETECT,md;initgraph(&dr,&md,"");
 for (int i=1;i<=20;i++)
  { line(i*20,20,i*20,400);
    line(20,i*20,400,i*20);
  }
 for (i=1;i<=19;i++)
  for (int j=1;j<=19;j++)
   if (doska[i][j]>0)
    { setcolor(doska[i][j]);
      circle(20*i+10,20*j+10,5);
    }
 setfillstyle(1,1);
 floodfill(20*x+10,20*y+10,1);
 mas[1].x=x;mas[1].y=y;
 len=1;int res=0;
 while (len>0 && !res)
  { x=mas[len].x;y=mas[len].y;
    doska[x][y]=3;
    len--;
    if (x-1>=1) res=control(x-1,y);
    if (res==0)
      if (y-1>=1) res=control(x,y-1);
    if (res==0)
      if (x+1<=19) res=control(x+1,y);
    if (res==0)
      if (y+1<=19) res=control(x,y+1);
  }
 if (res) outtextxy(400,400," Группа живая");
  else outtextxy(400,400," Группа не живая");
 getch();
}
```

Задача 18. Поиск пути с наибольшим весом

Условие задачи.

Дана таблица положительных целых чисел, сформированная случайным образом. Необходимо найти путь из левого верхнего угла таблицы в правый нижний с наибольшим весом. Весом пути будем называть сумму чисел всех элементов таблицы, через которые проходит путь. Путь можно строить только двумя типами смещений: вправо на один шаг или вниз на один шаг.

Краткая идея.

Пройдем таблицу от правого нижнего угла до левого верхнего и переопределим значения элементов по следующему правилу: новое значение элемента равно сумме старого элемента и наибольшего из двух соседей (верхнего и левого). После выполненного преобразования путь по таблице строится так:

- начало пути — левый верхний элемент;
- элемент, на который необходимо перейти на следующем шаге — это наибольший из двух (нижнего и правого).

Текст программы представлен в листинге П2.18.

Листинг П2.18

```
#include <iostream.h>
#include <conio.h>
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
void main()
{ long int a[11][11];
  int i,j;
  clrscr();
```

```
  randomize();
  for (i=1;i<=10;i++)
   for (j=1;j<=10;j++)
    { a[i][j]=random(9);
      gotoxy(i*7,j*4);cout <<a[i][j];
    }
  a[2][8]=9360;gotoxy(2*7,8*4);cout <<a[2][8];
  getch();
  for(i=10;i>=1;i--)
   for(j=10;j>=1;j--)
    { if (i<10) a[i][j]+=a[i+1][j];
      if (j<10) a[i][j]+=a[i][j+1];
      gotoxy(i*7,j*4);cout <<a[i][j];
    }
  getch();
  i=j=1;
  char string[10];
  textcolor(2);
  do
   { ltoa(a[i][j],string,10);
     gotoxy(i*7,j*4);cprintf(string);
     if (i==10) j++;
      else if (j==10) i++;
       else if (a[i+1][j]>a[i][j+1]) i++; else j++;
     delay(30000);
   }
  while (i<10 || j<10);
  ltoa(a[i][j],string,10);
  gotoxy(i*7,j*4);cprintf(string);
  getch();
}
```

Задача 19. Построение графика функции

Условие задачи.

Поставим перед собой следующую задачу: непрерывная функция задана на отрезке $[a, b]$. Требуется построить ее график и определить экстремумы.

Краткая идея.

В чем здесь проблема? Если быть точным, то следует говорить не о проблеме, а о проблемах, которых несколько.

- Отрезок построения может быть слишком мал или наоборот слишком велик. В первом случае, график окажется плохо различимым, во втором же случае часть графика выпадет за пределы экрана.
- Значения функции могут попасть в очень маленький отрезок или наоборот в слишком большой, и проблема, описанная выше, повторяется, то уже в отношении координаты y.

Указанные проблемы решаются по отдельности вполне удовлетворительно с помощью понятия масштаба. Мы, определяя масштаб по оси Ox, нужным образом можем привести отрезок $[a, b]$ к требуемому виду. Точно так же можно поступить и с осью Oy. Но может оказаться, что их масштабы будут различными.

К сожалению, проблема различия масштабов неустранима для произвольной функции. Но если вы согласны, что это проблема не так важна, как угроза потерять часть графика, то, взяв идею масштабирования за основу, мы построим хороший качественный график.

Ось Ox. Положим для определенности, что график функции необходимо вместить в экранный отрезок $[0, 600]$. Исходный отрезок мы уже обозначили как $[a, b]$. Совмещение отрезков состоит из двух операций. Во-первых, необходимо выполнить параллельный перенос для совмещения начал. Для этого достаточно вычесть из каждой точки отрезка $[a, b]$ величину a. После такого

переноса будет получен отрезок [0, $b - a$]. Далее необходимо выполнить преобразование подобия, но здесь возникает небольшая неопределенность. Если длина [a, b] меньше 600 пикселов, то отрезок необходимо увеличить, в противном случае — уменьшить. Конечно, это проблема небольшая с учетом того, что мы располагаем конструкцией выбора. Но можно поступить еще проще. Приведем полученный отрезок к единичному делением на его собственную длину, а затем умножим на 600. И все! Таким образом, для произвольной точки x, принадлежащей отрезку [a, b], формула преобразования будет выглядеть так:

$$x' = (x - a)/(b - a) \times 600.$$

Ось Oy. Здравый смысл подсказывает, что формула преобразования для вертикальной оси не должна отличаться от формулы преобразования горизонтальной. Проблема только в том, что взять в качестве отрезка для значений функции?

У множества значений функции на отрезке [a, b] есть минимум и максимум. Обозначим их, соответственно, *min* и *max*. Тогда очевидно, что все значения функции располагаются внутри отрезка [*min*, *max*]. И теперь можно записать формулу преобразования для значений ординат.

$$y' = (y - min)/(max - min) \times 400.$$

Множитель 400 имеет тот же смысл, что и множитель 600 в предыдущей формуле. То есть мы полагаем, что наш график по оси *Oy* вместится в отрезок длиной в 400 пикселов.

Оси координат в соответствии с полученными формулами преобразований будут выглядеть так, как показано на рис. П2.2. Заметьте, что точка начала координат не является нулевой точкой, и, более того, ей соответствуют два значения. По оси *Ox* значение *a* и по оси *Oy* значение *min*.

Масштабирование — ключевая проблема рисования графика. Если ее решение понятно, то остались сущие мелочи. А именно осталось учесть, что начало координат находится в левом верхнем углу, и необходимо позаботиться о вычислении минимального и максимального значений функции на отрезке.

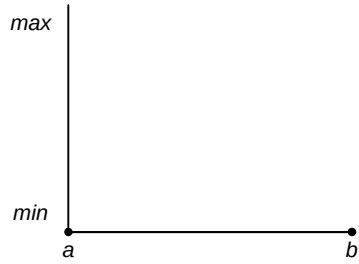


Рис. П2.2. Координатные оси

Построение графика функции. График строится на отрезке [0, 600], а массив содержит 2000 точек (см. текст программы). Это необходимо для обеспечения непрерывности графика. Если длина линии графика окажется больше, чем 2000 пикселов, то на графике появятся разрывы. Избавиться от них можно, увеличив размер массива.

Построение экстремумов. Фактически функция задана множеством значений, записанных в массив y. Это позволяет избежать сложных расчетов, предусмотренных общеизвестной схемой поиска экстремумов. А именно мы воспользуемся определением экстремума: точка графика является точкой минимума (максимума), если ее значение является наименьшим (наибольшим) в малой области, содержащей данную точку. Это определение, конечно, нельзя назвать математически строгим, но для наших целей оно подойдет. Если в качестве малой области взять двух ближайших соседей, то определение экстремума можно записать так: если точка ниже (выше) соседа слева и соседа справа, то это точка минимума (максимума).

Тогда функция построения экстремумов должна представлять собой цикл, в ходе работы которого пробегаются все точки, начиная со второй, до предпоследней, и для каждой из них проверяется описанное выше условие.

Для построения экстремумов существенно важна точность расчетов. Математический анализ утверждает, что точка, сколь угодно близкая к экстремуму по координате *x*, все равно от него отлича-

ется по значению, но в силу того, что точность наших расчетов ограничена в окрестности экстремума, могут появиться горизонтальные отрезки. Для борьбы с этим явлением можно либо повысить точность (что технически имеет пределы), либо уменьшить длину массива значений. В программе проблема решена последним способом. В результате уменьшения длины массива, расстояние между точками увеличивается и потребность в точности уменьшается, но тогда появляется риск потери точек экстремума, что, впрочем, для нашего учебного примера не существенно.

Построение программы. Работа программы состоит из трех частей: определение значений функции, затем расчеты, связанные с процедурой масштабирования, и, наконец, функция рисования графика. Оформим эти три части в виде отдельных функций.

Для хранения значений функции объявим числовой массив. Для этого есть две возможности. Во-первых, можно массив объявить глобально, и он, естественно, будет доступен везде. Во-вторых, можно воспользоваться указателем на массив, который будет передаваться из функции в функцию. Может быть, второй подход немного лучше. Отсутствие глобальных переменных делает отдельные функции программы более гибкими и более переносимыми.

Текст программы представлен в листинге П2.19.

Листинг П2.19

```
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
#include <math.h>
const N=2001;           // Длина массива значений
void vvod(float *);     // Прототип функции ввода
void calc(float *);     // Прототип функции расчетов
void ris(float *);      // Прототип функции рисования
void extremum(float *); // Прототип функции поиска экстремумов
void main(void)
```

```
{ float y[N];
  clrscr();
  vvod(y);
  calc(y);
  ris(y);
  extremum(y);
  getch();
}
void vvod(float *y)
{ float a,b,d;
  cin >>a>>b;        // Ввод границ отрезка
  d=(b-a)/(N-1);     // Расчет приращения для координаты x
  float x=a;
  for (int i=0;i<=(N-1);i++,x+=d) y[i]=sin(x*x); // Расчет
                                        // значений функции
}
void calc(float *y)
{ float max,min;
  max=min=y[0]; // Поиск минимального и максимального значений
  for (int i=1;i<=(N-1);i++)
   { if (max<y[i]) max=y[i];
     if (min>y[i]) min=y[i];
   }
  // Масштабирование массива значений
  for (i=0;i<=(N-1);i++) y[i]=(y[i]-min)/(max-min)*400;
}
void ris(float *y)
{ int dr=DETECT,md;initgraph(&dr,&md,"");
  // Рисование графика.
  // Масштабирование отрезка [a, b] выполняется
  // в функции putpixel
  // Операция i+0.0 необходима для преобразования целого i
  // в действительное
  for (int i=0;i<=(N-1);i++)
    putpixel((i+0.0)/(N-1)*600,400-y[i],15);
```

```
}
void extremum(float *y)
{ for (int i=1;i<=(N-2);i++)
   // Определение экстремума
   if ((y[i]>y[i-1] && y[i]>y[i+1]) || (y[i]<y[i-1] &&
        y[i]<y[i+1]))
     circle((i+0.0)/(N-1)*600,400-y[i],2);
}
```

Задача 20. Разработка базы данных

Разработка базы данных — это одна из наиболее популярных и часто встречающихся задач. Поэтому в данном разделе мы попробуем написать несложную программу, но такую, в которой использовались бы основные понятия из области баз данных. Однако сразу необходимо заметить, что хорошие, удобные базы содержат в себе огромное количество функций, обеспечивающих интерфейс с пользователем. Без хорошо разработанного интерфейса такая программа имеет мало смысла, но если мы будем разрабатывать все как нужно, на языке высокого уровня без использования спецсредств (специально для этого разработанных классов), то данная глава разрастется до размеров отдельной книги. Поэтому придется самым серьезным образом ограничить себя и описать только логику обработки данных.

Но прежде логики все же структура данных. Пусть цель нашей базы — хранение следующей информации о людях: фамилии, должности, размер заработной платы, года рождения. Запись такого вида описана структурой `shablon`. Возможности для обработки также пусть будут самые простые: ввод данных, их просмотр, изменение и удаление записей. Каждой такой возможности соответствует функция, вызываемая при выборе пункта главного меню. Главное меню создано в функции `main()`.

Для того чтобы не загромождать программу элементами интерфейса, о чем мы уже договорились, введено ограничение на количество видимых записей. Их не более 10. Ввести, конечно,

можно и больше, но увидеть на экране и, соответственно, обрабатывать, возможно, не более 10.

Ядро программы — это функция просмотра записей. Она вызывается из соответствующего пункта меню для собственно просмотра, и она же используется как для изменения записей, так и для их удаления. При такой организации программы специальные функции удаления и изменения фактически не нужны, но если программе поставить дополнительные задачи, например, опять же в организации интерфейса, то в функциях изменения и удаления данных могут появиться дополнительные команды, и тогда эти две функции перестанут быть идентичными просмотру. Поэтому структура программы такова, какова она есть.

Текст программы представлен в листинге П2.20.

Листинг П2.20

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <bios.h>
#include <io.h>
/*-------------------------------------------------------
Блок общих объявлений.
Здесь описаны прототипы используемых далее функций
и структура записи. Структуру лучше объявлять именно здесь,
т. к. в программе есть необходимость использовать записи
данной структуры локально в различных функциях.
-------------------------------------------------------*/
struct shablon
{ char family[20];
  char post[20];
  int sal,year;
};
FILE *f;
void New_Base();
```

```
void Enter_Data();
void View_Data(int);
void Delete_Data();
void Change_Data();
void main()
{ char g;
/*-----------------------------------------------------------
Главный цикл.
Его функция только организация главного меню. Здесь же можно
было бы один раз создать и открыть файл базы данных, но тогда
придется быть очень внимательным к процессу перемещения
файлового указателя. Открывать и закрывать файл в каждой
функции несложно, но это делает файловые операции внутри
каждой функции независимыми.
-----------------------------------------------------------*/
  do
   { clrscr();
     cout << "Новая база         - 1"<<"\n";
     cout << "Ввод данных        - 2"<<"\n";
     cout << "Просмотр данных    - 3"<<"\n";
     cout << "Удаление записи    - 4"<<"\n";
     cout << "Изменение записи   - 5"<<"\n";
     cout << "Завершение работы  - 6"<<"\n";
     g=getch();
     switch (g)
     { case '1': New_Base();break;
       case '2': Enter_Data();break;
       case '3': View_Data(0);break;
       case '4': Delete_Data();break;
       case '5': Change_Data();
     }
   }
  while (g!='6');
}
/*-----------------------------------------------------------
```

```
Самая простая функция.
Все, что она делает, - это создание файла данных в текущей
папке. После создания файл сразу же закрывается, и выводится
сообщение об успешности выполненной операции. В больших и
сложных программах, после таких операций еще принято делать
проверку на корректность выполненной операции и вывод
соответствующих сообщений (удалось или нет создать файл).
-----------------------------------------------------------*/
void New_Base()
{ f=fopen("base.dat","w");
  fclose(f);
  clrscr();
  cout <<" База создана";
  getch();
}
/*-----------------------------------------------------------
Функция ввода данных.
Пользователь имеет возможность вводить записи до тех пор,
пока в этом есть необходимость. Файл открыт для добавления
записей, поэтому вводимая информация автоматически
добавляется в конец файла. В этом месте необходимо учесть,
что часть полей структуры записи -
это символьные массивы, и для корректной
работы с ними неиспользованная часть массива заполняется
пробелами, после чего к строке добавляется терминальный ноль.
-----------------------------------------------------------*/
void Enter_Data()
{ shablon dat;
  clrscr();
  f=fopen("base.dat","a");
  char g;
  do
  { clrscr();
    cout << "Ввод новых данных";
    gotoxy(1,4);cout <<"Ввод фамилии ";cin >> dat.family;
```

```
    int N=strlen(dat.family);
    for (int i=N;i<=18;i++) dat.family[i]=' ';
    dat.family[19]='\0';
    gotoxy(1,5);cout <<"Должность ";cin >> dat.post;
    N=strlen(dat.post);
    for (i=N;i<=18;i++) dat.post[i]=' ';
    dat.post[19]='\0';
    gotoxy(1,6);cout <<"Заработная плата ";cin >> dat.sal;
    gotoxy(1,7);cout <<"Год рождения ";cin >> dat.year;
    fwrite(&dat,44,1,f);
    gotoxy(1,10);cout <<"Продолжить - y";
    g=getch();
  }
 while (g=='y');
 fclose(f);
}
/*-----------------------------------------------------------
Вспомогательная функция для вывода данных о записи.
На вход получает номер позиции курсора, бегающего по списку
считанных фамилий, затем выполняет перевод файлового указателя
в соответствующую позицию, читает данные текущей записи
и выводит их на экран монитора. 44 — это 44 байта — длина записи.
-----------------------------------------------------------*/
void Print_Record(int N)
{ fseek(f,44*N,SEEK_SET);
  shablon dat;
  fread(&dat,44,1,f);
  gotoxy(40,4);cout <<"                                    ";
  gotoxy(40,4);cout <<"Фамилия"<<dat.family;
  gotoxy(40,5);cout <<"                                    ";
  gotoxy(40,5);cout <<"Должность"<dat.post;
  gotoxy(40,6);cout <<"                                    ";
  gotoxy(40,6);cout <<"Заработная плата "<<dat.sal;
  gotoxy(40,7);cout <<"                                    ";
  gotoxy(40,7);cout <<"Год рождения   "<<dat.year;
```

```
}
/*-----------------------------------------------------------
Эта функция читает файл базы и создает список фамилий,
с которыми и выполняется дальнейшая работа по выбору записи
либо на просмотр, либо на изменение, либо на удаление.
-----------------------------------------------------------*/
int Print_Family()
{ shablon dat;
  int n=0;
  do
  { fread(&dat,44,1,f);
    if (!feof(f))
    { gotoxy(5,4+n);cout <<"Фамилия ";cout << dat.family;
      n++;
    }
  }
 while (!feof(f) && n<10);
 gotoxy(2,4);cout <<">>";
 return n;
}
/*-----------------------------------------------------------
Пожалуй, главная функция программы.
Ее цель - сформировать список фамилий и организовать
навигацию по нему. Навигация осуществляется просто
перемещением курсора. По нажатии клавиши <Enter> функция
выполняет одно из двух дополнительных действий: либо удаление
записи, либо ее изменение. Функция bioskey(0) необходима для
определения кода нажатой клавиши.
-----------------------------------------------------------*/
void View_Data(int L)
{ clrscr();
  shablon dat;
  f=fopen("base.dat","r+");
  int n=Print_Family();
```

```
  Print_Record(0);
  int c,y=0;
  do
  { c=bioskey(0);
    gotoxy(2,y+4);cout <<"  ";
    switch (c)
    { case 18432: if (y>0) y--;break;
      case 20480: if (y<n-1) y++;break;
      case 7181:
      { switch (L)
        {
/*-----------------------------------------------------------
Цель данного блока — удаление выбранной записи.
Для выполнения удаления все записи перезаписываются со
смещением 44 байта в сторону начала файла, в результате чего
стираемая запись оказывается затертой своим соседом справа.
После смещения записей, последняя оказывается
продублированной. Поэтому необходима функция
chsize(Nfile,Lfile-44); уничтожающая появившийся хвост
в 44 байта.
-----------------------------------------------------------*/
         case 1:
          { shablon dat1;
            int Nfile=fileno(f);
            int Lfile=filelength(Nfile);
            for (int i=y;i<Lfile/44-1;i++)
             { fseek(f,44*(i+1),SEEK_SET);
               fread(&dat1,44,1,f);
               fseek(f,44*i,SEEK_SET);
               fwrite(&dat1,44,1,f);
             }
            fseek(f,0,SEEK_SET);
            chsize(Nfile,Lfile-44);
            clrscr();
            n=Print_Family();
```

```
          y=0;
         }break;
/*-------------------------------------------------------
Вводятся новые данные, и старая запись заменяется, для чего
файловый указатель устанавливается на начало изменяемой
записи, и производится запись новых данных.
-------------------------------------------------------*/
        case 2:
         {
           gotoxy(5,20);cout <<"Ввод новых данных";
           gotoxy(5,21);cout <<"Фамилия ";cin >> dat.family;
           int N=strlen(dat.family);
           for (int i=N;i<=18;i++) dat.family[i]=' ';
           dat.family[19]='\0';
           gotoxy(5,22);cout <<"Должность ";cin >> dat.post;
           N=strlen(dat.post);
           for (i=N;i<=18;i++) dat.post[i]=' ';
           dat.post[19]='\0';
           gotoxy(5,23);
           cout <<"Заработная плата "; cin >> dat.sal;
           gotoxy(5,24);
           cout <<"Год рождения "; cin >> dat.year;
           fseek(f,44*y,SEEK_SET);
           fwrite(&dat,44,1,f);
           gotoxy(5,y+4);
           cout<<"Фамилия "<<dat.family<<"        ";
           gotoxy(5,20);
           cout <<"                                ";
           gotoxy(5,21);
           cout <<"                                ";
           gotoxy(5,22);
           cout <<"                                ";
           gotoxy(5,23);
           cout <<"                                ";
           gotoxy(5,24);
```

```
            cout <<"                                ";
          }
        }
      }
    }
  gotoxy(2,y+4);cout <<">>";
  Print_Record(y);
 }
 while (c!=283);
 fclose(f);
}
void Delete_Data()
{ View_Data(1);
}
void Change_Data()
{ View_Data(2);
}
```

Предметный указатель

А

Арифметическое условие 73

В

Ввод/вывод 51
  в файл 183
    неформатный 183
    форматный 189
  перевод курсора на
    следующую строку 58
  потоковый 54
  форматный 51
Время жизни 18
  переменной 39
Выражение 94

Д

Двоичное дерево:
  обмен местами левого и
    правого поддеревьев 179
  построение 178
Директива 13
  define 229
  include 227

И

Инициализация переменной 30
  автоматической 31
  статической 31

К

Класс памяти 18, 39
  auto 18, 40
  extern 41
  register 41
  static 18, 28, 40
Код символа 46
Команда:
  cin 11, 54
  count 12
  cout 54
  return 16
Комментарий 215
Константа 99
  NULL 151
  значение 99
  именованная 99

М

Массив 11, 107
  многомерный 107, 113
Метка 95

О

Область видимости 18
  переменной 39
Объединение 117, 121
Объявление:
  typedef 47
  типа 11

Оператор 77
231
break 90, 91
continue 93
goto 95
if 12, 86
return 96
switch 89
присваивания 17
пустой ; 96
составной 28, 36, 92
Операция:
delete 143
delete[] 143
new 143
sizeof 113, 140
арифметическая 61, 64
битовая 68
индексации 71
логическая 69
нахождения остатка % 65
получения адреса & 140
построения выражения 71
присваивания = 61
разрешения области видимости 38
сдвига 66
следования 70
Описатель формата 52

П

Память:
регистровая 41
статическая 28
Перегрузка функций 210
Передача:
многомерного массива 203
одномерного массива 198
параметров по ссылке 204
Переменная:
автоматическая 40
глобальная 15, 28, 38
инициализация 30
локальная 15, 28, 37
статическая 40
Перечисления 104
Поле структуры 118
Преобразование типов:
данных 43
неявное 44
явное 45
Прототип 16, 197

С

Связный список 167
вставка:
связного списка 175
элемента 173
обход 168, 169
в двух направлениях 171
сортировка 176
Строка:
вхождение одной в другую 160
конкатенация 157
копирование части 158
нультерминальная 153
обмен содержимого двух строк 162
Структура 117, 118

Т

Терминальный ноль 103, 153
Тип данных 25
char 16, 26
double 16
float 16
int 16

У

Указатель 21, 131
  NULL 151
  на функцию 212
Управляющий символ 58

Ф

Файл заголовков 13
Функция:
  arc() 219
  bar() 219
  bar3d() 220
  chsize() 188
  circle() 220
  cleardevice() 220
  closegraph() 220
  cprintf() 54
  creat() 193
  drawpoly() 221
  ellipse() 221
  fclose() 185
  fgetpos() 192
  filelength() 188
  fileno() 188
  floodfill() 222
  fopen() 184
  fread() 183
  fseek() 191
  fwrite() 183
  getbkcolor() 222
  getch() 14
  getchar() 56
  getcolor() 222
  getmaxx() 222
  getmaxy() 222
  getpixel() 222
  getx() 223
  gety() 223
  gotoxy() 58
  linerel() 223
  lineto() 223
  main() 13, 195
  moverel() 223
  moveto() 223
  outtext() 224
  outtextxy() 224
  printf() 51
  putchar(), 56
  putpixel() 224
  rectangle() 224
  rewind() 192
  scanf() 51
  sector() 225
  setbkcolor() 225
  setcolor() 225
  setfillstyle() 225
  strcmp() 166
  strlow() 164
  strset() 165
  strstr() 164
  strupr() 164
  textbkcolor() 57
  textcolor() 57

Ц

Цикл 10, 77
  do...while 19, 81
  for 11, 20, 78
  while 19, 80