Начни писать программы на C++, пользуясь этой книгой

C++ для "чайников"™

4-е издание

Верный помощник при изучении языка C++

Для сомневающихся

Стефан Р. Дэвис

IDG BOOKS WORLDWIDE

ДИАЛЕКТИКА

C++ для "чайников"

4-е издание

C++ FOR DUMMIES®

4TH EDITION

by Stephen R. Davis

IDG Books Worldwide, Inc.

An International Data Group Company

Foster City, CA • Chicago, IL • Indianapolis, IN • New York, NY

Стефан Р. Дэвис

ДИАЛЕКТИКА

Москва • Санкт-Петербург • Киев
2003

ББК 32.973.26-018.2$_{я}$75
Д94
УДК 681.3.07

Компьютерное издательство ""Диалектика"

Перевод с английского *Д.М. Мищишина* и канд. техн. наук *ИВ. Красикова*

Под редакцией канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в издательство "Диалектика"
по адресу: info@dialektika.com, http://www.dialektika.com

Дэвис, Стефан, Р.

Д94 C++ для "чайников", 4-е издание. : Пер. с англ. : — М. : Издательский дом "Вильямс", 2003. — 336 с. : ил. : Парал. тит. англ.

ISBN 5-8459-0160-X (рус.)

Книга, которая у вас в руках, — это введение в язык программирования C++. Она начинается с азов: от читателя не требуется каких-либо знаний в области программирования. В отличие от других книг по программированию па C++, в этой книге вопрос "почему" считается не менее важным, чем вопрос "как". И поэтому перед изложением конкретных особенностей языка C++ читателю разъясняется, как они действуют в целом. Ведь каждая структурная особенность языка — это отдельный штрих единой картины. Прочитав книгу, вы сможете написать на C++ вразумительную программу и, что не менее важно, будете понимать, почему и как она работает.

Книга рассчитана на пользователей с различным уровнем подготовки.

ББК 32.973.26-018.2$_{я}$75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства IDG Books Worldwide, Inc.

Copyright © 2001 by Dialektika Computer Publishing.

Original English language edition copyright © 2000 by IDG Books Worldwide, Inc.

All rights reserved including the right of reproduction in whole or in part in any form.

This edition published by arrangement with the original publisher, IDG Books Worldwide, Inc., Foster City, California, USA.

Windows is a trademark of Microsoft Corporation... For Dummies, Dummies Man, and the IDG Books Wordwide logos are trademarks under exclusive license to IDG Books Worldwide, Inc., from International Data Group, Inc. Used by permission.

ISBN 5-8459-0160-X (рус.)
ISBN 0-7645-0746-X (англ.)

© Компьютерное издательство "Диалектика", 2001
© IDG Books Worldwide, Inc., 2000

Оглавление

Часть I. Первое знакомство с C++ 21
Глава 1. Написание вашей первой программы 23
Глава 2. Премудрости объявления переменных 33
Глава 3. Выполнение математических операций 41
Глава 4. Выполнение логических операций 46
Глава 5. Операторы управления программой 55
Часть П. Становимся программистами 67
Глава 6. Создание функций 69
Глава 7. Хранение последовательностей в массивах 79
Глава 8. Первое знакомство с указателями в C++ 90
Глава 9. Второе знакомство с указателями 100
Глава 10. Прочие функциональные особенности 111
Глава 11. Отладка программ на C++ 119
Часть III. "Классическое" программирование 139
Глава 12. Знакомство с объектно-ориентированным программированием 141
Глава 13. Классы в C++ 144
Глава 14. Работа с классами 149
Глава 15. Создание указателей на объекты 160
Глава 16. Защищенные члены класса: не беспокоить! 173
Глава 17. Создание и удаление объектов: конструктор и деструктор 179
Глава 18. Аргументация конструирования 187
Глава 19. Копирующий конструктор 201
Глава 20. Статические члены 211
Часть IV. Наследование 225
Глава 21. Наследование классов 227
Глава 22. Знакомство с виртуальными функциями-членами: настоящие ли они 233
Глава 23. Разложение классов 245
Часть V. Полезные особенности 265
Глава 24. Перегрузка операторов 267
Глава 25. Перегрузка оператора присвоения 280
Глава 26. Использование потоков ввода-вывода 286
Глава 27. Обработка ошибок и исключения 299
Глава 28. Множественное наследование 306
Часть VI. Великолепная десятка 315
Глава 29. Десять способов избежать ошибок 317
Приложение А. Словарь терминов 322

Содержание

Об авторе 12
Благодарности 13
Введение 14
Об этой книге 14
О чем эта книга 14
Что такое C++ 14
Соглашения, используемые в книге 15
Что можно не читать 15
Нелепые предположения 15
Как организована эта книга 16
В конце каждой части... 16
Часть 1. Первое знакомство с C++ 17
Часть 2. Становимся программистами 17
Часть 3. "Классическое" программирование 17
Часть 4. Наследование 17
Часть 5. Полезные особенности 17
Часть 6. Великолепная десятка 17
Использованные в этой книге пиктограммы 18
Что дальше 18
Часть I. Первое знакомство с C++ 19
Глава 1. Написание вашей первой программы 21
Постигая концепции C++ 21
Что такое программа 22
Как пишут программы 22
Создание первой программы 23
Введение кода 23
Создание выполнимого файла 25
Выполнение программы 26
GNU — это не Windows 26
Помощь в GNU C++ 27
Разбор программ 27
Определение структуры программ C++ 27
Использование в исходном коде комментариев 27
Использование инструкций в программах 28
Объявления 28
Генерирование вывода 29
Вычисление выражений 29
Сохранение результатов выражения 30
Обзор программы Convert продолжается... 30
Глава 2. Премудрости объявления переменных 31
Объявление переменных 31
Объявление разных типов переменных 32
Ограничения, налагаемые на целые числа в C++ 32
Решение проблемы усечения дробной части 33
Ограничения, налагаемые на числа с плавающей точкой 34
Объявления типов переменных 35
Типы констант 36
Специальные символы 36
Выражения смешанного типа 37
Глава 3. Выполнение математических операций 39

Бинарная арифметика 39
Анализ выражений 40
Определение порядка операций 41
Выполнение унарных операций 42
Использование операторов присвоения 42
Глава 4. Выполнение логических операций 44
Зачем нужны логические операторы 44
Использование простых логических операторов 44
Логические операции и действительные переменные 45
Бинарные числа в C++ 47
Десятичная система счисления 47
Другие системы счисления 48
Двоичная система счисления 48
Выполнение битовых логических операций 49
Битовые операции с одним битом 49
Использование битовых операторов 50
Простой пример 51
Зачем придуманы эти глупые операторы 52
Глава 5. Операторы управления программой 53
Управление ходом программы с помощью команд ветвления 53
Выполнение циклов 55
Цикл while 55
Использование операторов инкремента и декремента 56
Использование цикла for 57
Избегайте бесконечных циклов 59
Специальные операторы циклов 60
Вложенные команды управления 62
Инструкция выбора 63
Часть II. Становимся программистами 65
Глава 6. Создание функций 67
Написание и использование функций 67
Подробный анализ функций 69
Простые функции 70
Функции с аргументами 70
Перегрузка функций 73
Определение прототипов функций 74
Хранение переменных в памяти 75
Глава 7. Хранение последовательностей в массивах 77
Преимущества массивов 77
Работа с массивами 78
Инициализация массива 80
Выход за границы массива 81
Использовать ли массивы 81
Определение и использование массивов 82
Использование символьных массивов 82
Управление строками 84
Написание функции, соединяющей две строки 84
Функции C++ для работы со строками 86
Обработка символов типа wchar_t 87
Устранение устаревших функций вывода 87
Глава 8. Первое знакомство с указателями в C++ 88
Что такое адрес 88
Использование указателей 89

Сравнение указателей и почтовых адресов 90
Использование разных типов указателей 91
Передача указателей функциям 93
Передача аргументов по значению 93
Передача значений указателей 93
Передача аргументов по ссылке 94
Использование кучи 94
Область видимости 94
Проблемы области видимости 96
Использование блока памяти 96

Глава 9. Второе знакомство с указателями 98

Операции с указателями 98
Повторное знакомство с массивами в свете указателей 99
Использование операций над указателями для адресации внутри массива 100
Использование указателей для работы со строками 101
Операции с указателями других типов 104
Отличия между указателями и массивами 104
Объявление и использование массивов указателей 105
Использование массивов строк 106
Доступ к аргументам main() 107

Глава 10. Прочие функциональные особенности 109

Зачем разбивать программу на модули 109
Пример большой программы 110
Разделение программы FunctionDemo 111
Отделение модуля sumSequence() 111
Создание модуля MainModule.cpp 113
Создание файла проекта 113
Создание файла проекта в GNU C++ 114
Создание файла проекта в Visual C++ 114
Использование директивы #include 115
Использование стандартных библиотек C++ 116

Глава П. Отладка программ на C++ 117

Определение типа ошибки 117
Использование отладочной печати 117
Выявление "жучка" № 1 118
Выявление "жучка" № 2 121
Использование отладчика 124
Что такое отладчик 124
Выбор отладчика 124
Запуск тестовой программы 125
Пошаговое выполнение программы 126
Пошаговый режим с входом в функции 128
Использование точек останова 129
Просмотр и редактирование переменных 129

Первая программа BUDGET 132

Часть III. "Классическое" программирование 137

Глава 12. Знакомство с объектно-ориентированным программированием 139

Микроволновые печи и уровни абстракции 139
Приготовление блюд с помощью функций 140
Приготовление "объектно-ориентированных" блюд 140
Классифицирование микроволновых печей 140
Зачем нужна классификация 141

Глава 13. Классы в C++ **142**
Введение в классы 142
Формат класса 142
Обращение к членам класса 143
Пример программы 144
Глава 14. Работа с классами **147**
Активизация объектов 147
Моделирование реальных объектов 148
Зачем нужны функции-члены 148
Добавление функции-члена 149
Создание функции-члена 149
Именование членов класса 150
Вызов функций-членов 150
Обращение к функциям-членам 151
Доступ к членам из функции-члена 151
Разрешение области видимости 153
Определение функции-члена 154
Определение функций-членов вне класса 155
Перегрузка функций-членов 156
Глава 15. Создание указателей на объекты **158**
Определение массивов указателей 158
Объявление массивов объектов 159
Объявление указателей на объекты 160
Разыменование указателей на объекты 160
Использование стрелок 161
Передача объектов функциям 161
Вызов функции с передачей объекта по значению 161
Вызов функции с передачей указателя 162
Зачем передавать указатель 162
Передача объекта по ссылке 164
Возврат к куче 165
Использование связанных списков 165
Массив 165
Связанный список 166
Другие операции над связанным списком 167
Свойства связанных списков 168
Программа LinkedListData 168
Глава 16. Защищенные члены класса: не беспокоить! **171**
Защищенные члены 171
Зачем нужны защищенные члены 171
Как устроены защищенные члены 172
Чем хороши защищенные члены 173
Защита внутреннего устройства класса 173
Классы с ограниченным интерфейсом 174
Обращение к защищенным членам 174
"Друг всегда уступить готов место в шлюпке и круг..." 174
Глава 17. Создание и удаление объектов: конструктор и деструктор **177**
Создание объектов 177
Использование конструкторов 178
Зачем нужны конструкторы 178
Работа с конструкторами 179
Что такое деструктор 182
Зачем нужен деструктор 182

Работа с деструкторами 183
Глава 18. Аргументация конструирования 185
Как снабдить конструктор аргументами 185
Зачем конструкторам нужны аргументы 186
Как использовать конструктор с аргументами 186
Перегрузка конструктора 188
Определение конструкторов по умолчанию 190
Конструирование членов класса 191
Управление последовательностью конструирования 195
Локальные объекты создаются последовательно 196
Статические объекты создаются один раз 196
Все глобальные объекты создаются до вызова main() 196
Порядок создания глобальных объектов не определен 197
Члены создаются в порядке их объявления 198
Деструкторы удаляют объекты в порядке, обратном порядку их создания 198
Глава 19. Копирующий конструктор 199
Копирование объекта 199
Зачем это нужно 199
Использование конструктора копирования 200
Автоматический конструктор копирования 201
"Мелкие" и "глубокие" копии 203
Временные объекты 206
Глава 20. Статические члены 209
Определение статических членов 209
Зачем нужны статические данные 209
Использование статических членов 210
Обращение к статическим данным-членам 210
Применение статических данных-членов 212
Объявление статических функций-членов 212
Бюджет с классами — BUDGET2.CPP 215
Часть IV. Наследование 223
Глава 21. Наследование классов 225
Зачем нужно наследование 225
Как наследуется класс 227
Конструирование подкласса 229
Отношение СОДЕРЖИТ 229
Глава 22. Знакомство с виртуальными функциями-членами: настоящие ли они 231
Зачем нужен полиморфизм 234
Как работает полиморфизм 235
Полиморфное приготовление закуски 237
Когда функция не является виртуальной 239
Виртуальные особенности 241
Глава 23. Разложение классов 243
Разложение 243
Реализация абстрактных классов 246
Концепция абстрактных классов 248
Создание полноценного класса из абстрактного 249
Передача абстрактных классов 251
Нужны ли чисто виртуальные функции 251
Рационализация бюджета: BUDGET3.CPP 253

Часть V. Полезные особенности 263
Глава 24. Перегрузка операторов 265
Перегрузка операторов: давайте жить в гармонии 265
Операторная функция 266
А подробнее? 269
operator+() 269
operator*-+() 271
Операторы как функции-члены 272
Еще одна перегрузка 273
Перегрузка операторов с помощью неявного преобразования типов 274
Приведение объектов пользовательских типов 275
Оператор явного преобразования 276
Правила для неявных преобразований 277
Глава 25. Перегрузка оператора присвоения 279
Опасная работа, коварная работа, кошмарная работа... 279
Знакомство с перегрузкой оператора присвоения 280
Глубокая проблема создания мелких копий 281
Почленный подход к С 282
Возврат результата присвоения 283
Зашита членов 283
Глава 26. Использование потоков ввода-вывода 285
Нырнем в поток... 285
Знакомство с подклассами fstream 287
Подклассы strstream 290
Манипулирование манипуляторами 291
Написание собственных операторов вставки 294
Создание "умных" операторов 296
Глава 27. Обработка ошибок и исключения 299
Зачем нужен новый механизм обработки ошибок 300
Механизм исключительных ситуаций 301
Так что же мы будем бросать? 303
Глава 28. Множественное наследование 306
Механизм множественного наследования 306
Устранение неоднозначностей множественного наследования 307
Виртуальное наследование 308
Конструирование объектов 312
Отрицательные стороны множественного наследования 313
Часть VI. Великолепная десятка 315
Глава 29. Десять способов избежать ошибок 317
Включение всех предупреждений и сообщений об ошибках 317
Добейтесь чистой компиляции 318
Используйте последовательный стиль программирования 318
Ограничивайте видимость 318
Комментируйте свою программу 320
Хотя бы один раз выполните программу пошагово 320
Избегайте перегрузки операторов 320
Работа с кучей 320
Используйте исключительные ситуации для обработки ошибок 321
Избегайте множественного наследования 321
Приложение А. Словарь терминов 322
Предметный указатель 327

Моим друзьям и семье, которые помогли мне
стать "чайником" в еще большей степени,
чем я есть на самом деле

Об авторе

Стефан Р. Дэвис (Stephen R. Davis) — автор множества книг, включая такие бестселлеры, как C++ *для "чайников", More C++ for Dummies* и *Windows 95 Programming for Dummies.* Стефан работает в компании Valtech, специализирующейся в области обучения информатике (Даллас, Техас).

Благодарности

Я считаю странным то, что на обложке любой книги, особенно такой, как эта, написано только одно имя. В действительности свой труд в создание книги *...для "чайников"* вкладывает громадное число людей. Для начала я хотел бы поблагодарить своего главного редактора Мэри Кордер (Mary Corder) и агента Клодетт Мур (Claudette Moore), направлявших меня при формировании материала этой книги. Во время работы над книгой я значительно повысил свой уровень как редактор и корректор, и в этом мне помогли редакторы первого и второго изданий Сьюзен Пинк (Susan Pink) и третьего издания Келли Юинг (Kelly Ewing) и Коллин Вильямс (Colleen Williams). Кроме того, я благодарен техническим редакторам Грегу Гантли (Greg Guntle), Гаррет Пиз (Garrett Pease) и Джеффу Банкстону (Jeff Bankston) (первое, второе и третье издания соответственно) без их участия эта книга была бы намного хуже. И если бы не помощь координатора первого и второго изданий Сьюзанны Томас (Suzanne Thomas), эта книга вообще не была бы напечатана. Однако, несмотря ни на что, на обложке представлено только одно имя, а значит, ответственность за все неточности в тексте должен нести именно его обладатель.

Хочу также поблагодарить свою жену Дженни и сына Кинси за их терпение и преданность.

И наконец, новости о последних событиях из мира животных в моем доме. Для тех, кто не читал ни одной из моих книг, объясняю, что такая сводка встречается в них регулярно.

Мои две собаки, Скутер и Труди, чувствуют себя нормально, хотя Труди почти ослеп. Наши два кролика, Бивас и Батхед, отправились на большую зеленую небесную лужайку после почти полуторалетнего проживания на газоне перед нашим домом. Мы завели двух кошек, Боба и Марли, когда писалась книга *More C++ for Dummies,* Марли умерла от кошачьей лейкемии, а Боб продолжает жить и радовать нас.

Если вы хотите пообщаться со мной по поводу программирования на C++, полуслепых собак или бродячих кроликов, пишите мне по адресу: srdavis@ACM.org.

Введение

Об этой книге

Добро пожаловать в четвертое издание книги C++ *для "чайников"*. В ней вы найдете всю необходимую для изучения C++ информацию, описанную доступным языком и не отягощенную излишними подробностями.

О чем эта книга

Книга, которая у вас в руках, — это введение в язык программирования C++.

Она начинается с азов: от читателя не требуется каких-либо знаний в области программирования (и в этом основное отличие от предыдущего издания, которое предполагает знание языка C).

В отличие от других книг по программированию на C++, в этой книге вопрос "почему" считается не менее важным, чем вопрос "как". И потому перед изложением конкретных особенностей языка C++ я старался объяснить читателю, как они действуют в целом. Ведь каждая структурная особенность языка — это отдельный штрих единой картины.

Если вы не понимаете, зачем нужны те или иные особенности языка, постарайтесь понять, как они работают. Прочитав книгу, вы сможете написать на C++ вразумительную программу и, что не менее важно, будете понимать, почему и как она работает.

Эта книга не обучает программированию для Windows. Научиться этому можно в два этапа. Сначала необходимо усвоить C++, а затем приобрести книгу *Windows 95 Programming for Dummies.*

Что такое C++

C++ представляет собой объектно-ориентированный низкоуровневый язык программирования, отвечающий стандартам ANSI и Международной организации стандартов (International Standards Organization — ISO). *Объектная ориентированность* C++ означает, что он поддерживает стиль программирования, упрощающий кодирование крупномасштабных программ и обеспечивающий их расширяемость. Будучи низкоуровневым языком, C++ может генерировать весьма эффективные высокоскоростные программы. Сертификация ANSI и ISO обеспечила переносимость C++: написанные на нем программы совместимы с большинством современных сред программирования.

Уже в самом названии содержится намек на то, что C++ является следующим поколением языка программирования C — результатом добавления новых веяний академической компьютерной мысли к старому доброму C. На C++ можно делать все то же, что и на C, и даже таким же образом. Но это нечто большее, чем просто C, наряженный в новые одежды. Дополнительные возможности C++ весьма значительны и требуют не только некоторых размышлений, по и привычки, однако результат того заслуживает.

Для опытного программиста на C язык C++ может показаться одновременно и захватывающим и расстраивающим. Представьте себе немца, читающего по-датски. Это очень похожие ситуации. Программист, использующий C, сможет понять смысл

программ на C++, но из-за значительных отличий между языками его трактовка не всегда будет адекватной. Эта книга поможет вам перейти от C к C++ настолько мягко, насколько это возможно. Однако напомним, что для читателей C++ *для "чайников"* опыт программирования на C вовсе не обязателен.

Соглашения, используемые в книге

Описываемые сообщения или любая другая информация, отображаемая на экране, будет выглядеть так:

```
Hi mom!
```

Программный код будет представлен таким же образом:

```
// программа
void main()
{
 ...
}
```

Если вы решите набирать программу вручную, следите за тем, чтобы ее текст полностью соответствовал напечатанному в книге, за исключением количества пробелов, которое может быть произвольным.

Всяческие компьютерные сообщения, такие как команды и имена функций, будут выглядеть вот так. После имен функций всегда следуют открывающая и закрывающая скобки, например myFavoriteFunction() .

Иногда для выполнения некоторых действий в книге рекомендуется использовать специальные команды клавиатуры. Например, когда в тексте содержится инструкция: нажать <Ctrl+C>, вам следует, удерживая нажатой клавишу <Ctrl>, нажать клавишу <C>. Вводить знак "плюс" при этом не нужно.

Время от времени будут использоваться команды меню, например File⇨Open.

В этой строке для открытия меню File и выбора нужной команды из него предлагается использовать клавиатуру или мышь.

Что можно не читать

C++ является слишком большим куском, чтобы проглотить его сразу. Вы столкнетесь и с легкими, и с достаточно сложными моментами. Чтобы уберечь вас от перенасыщения информацией, не актуальной для вас в текущий момент, некоторые технические подробности будут отмечены специальными пиктограммами (см. раздел "Использованные в этой книге пиктограммы").

Некоторые сведения общего характера будут помещаться в выделенных врезках. Почувствовав, что информация воспринимается с трудом, смело пропускайте этот раздел во время первого чтения (но помните, что по возможности его нужно будет прочитать, поскольку в конечном счете незнание каких-то моментов неизбежно скажется на ваших программах).

Нелепые предположения

Чтобы освоить материал книги C++ *для "чайников"*, совершенно не обязательно иметь какой-то опыт в программировании. Конечно, если он есть, это только плюс, но его отсутствие не должно вас тревожить.

В предыдущем издании этой книги предполагалось, что вы уже немного знакомы с языком С. Идея была в том, что изучающий С++ должен был основываться на уже имеющихся знаниях языка С. Однако такой методологический подход оказался ошибочным. Прежде всего, многие принципы С++ в корне отличаются от основополагающих принципов С, несмотря на обманчивую схожесть их синтаксиса. К тому же среди изучающих С++ большинство составляют все-таки не программисты на С, а новички в программировании.

Четвертое издание *С++ для "чайников"* начинается с основных понятий, используемых в программировании. Затем книга ведет читателя от программ из простых синтаксических конструкций до концепций объектно-ориентированного программирования. Читателю, осилившему ее всю, не составит большого труда в нужный момент произвести впечатление на друзей или блеснуть своей осведомленностью на вечеринках.

Как организована эта книга

Каждая новая структурная возможность языка будет охарактеризована следующим образом:

- ✓ что представляет собой эта возможность;
- ✓ зачем она включена в язык;
- ✓ как она работает.

Разделы книги щедро снабжены небольшими фрагментами программного кода. Каждый из них иллюстрирует представленные особенности или основные моменты некоторых моих разработок. Эти фрагменты не всегда закончены и в основном не представляют собой ничего существенного.

Примечание. Необходимость соблюдать формат книги требовала переноса очень длинных строк кода. В конце таких строк появляется стрелка, которая напоминает о том, что следует продолжать ввод, не торопясь нажимать клавишу <Enter>. Я очень старался свести эти длинные строки кода к минимуму.

В конце каждой части...

В дополнение в конце частей 2, 3 и 4 приводятся тексты серии программ BUDGET. Эти программы достаточно объемны, чтобы позволить вам получить какое-то представление о реальных программах.

К тому же очень важно понимать, как разные структурные единицы языка С++ сосуществуют в готовой программе. Поэтому, хотя забот у меня и так было предостаточно, я занялся разработкой примеров обучающих программ. Наверное, от недостатка фантазии мне пришлось потратить много времени на придумывание программных сюжетов. Хотелось бы, чтобы представленные в них особенности языка были очевидны читателю.

В конце концов в качестве примера было решено использовать программу BUDGET. Она рождается как простая, процедурно ориентированная программа. Постепенно обрастая структурными особенностями, описанными в каждой новой части, к концу книги программа BUDGET предстанет перед вами во всей красе своего объектно-ориентированного содержимого. Возможно, работа с этой программой покажется вам страшной тратой времени. Если это так, вы можете пропустить первые варианты программы (хотя, по мнению нашего редактора, замысел довольно удачный). Тем не менее я надеюсь, что, разобрав программу BUDGET, вы постигнете тайну согласованной работы возможностей С++.

Часть /. Первоезнакомство с C++

Эта часть является отправной точкой нашего путешествия в мир C++. Вы начнете его с нелегкого испытания — написания своей первой компьютерной программы. Затем перейдете к изучению синтаксиса языка.

Часть 2.Становимся программистами

В этой части новоприобрстенные знания основных команд C++ пополнятся способностью объединять фрагменты программного кода в модули и повторно использовать их в программах.

Здесь также представлена внушающая наиболее благоговейный страх тема: указатели в C++. Если вам это ни о чем не говорит, не волнуйтесь — скоро вы обо всем узнаете.

Часть 3."Классическое" программирование

В этой части дело запутывается все больше и больше: начинается обсуждение объектно-ориентированного программирования. По правде говоря, объектно-ориентированный подход к построению программ и есть главная причина возникновения и активного использования C++. Ведь отказавшись от объектно-ориентированных особенностей C++, мы просто возвратимся к его предшественнику — языку программирования C. В этом разделе обсуждаются такие понятия, как классы, конструкторы, деструкторы, и прочие не менее страшные термины. Не волнуйтесь, если пока что вы не совсем понимаете, о чем идет речь.

Часть Ч. Наследование

Возможность наследования — это как раз то главное свойство объектно-ориентированного программирования, которое обеспечило ему известность и распространенность. Обсуждение этой одной из наиболее важных концепций, понимание которой служит ключом к эффективному программированию на C++, и является темой четвертой части. Теперь дороги назад нет: закончив освоение этого материала, вы сможете назвать себя настоящим объектно-ориентированным профаммистом.

Часть 5 Полезные особенности

К моменту знакомства с этой частью вы уже будете знать все необходимое для эффективного программирования на C++. Здесь же затрагиваются некоторые оставшиеся дополнительные вопросы. Если вы чувствуете, что голова все еще кружится от избытка с трудом воспринимаемой информации, можете пока удержаться от чтения этой части.

Часть 6.Великолепнаядесятка

Разве книга для "чайников" может считаться законченной без такой полезной напутствующей части? В ее единственной главе вы узнаете наилучшие способы избежать ошибок в программах.

использованные в этой книге пиктограммы

Технические подробности, которые можно пропустить при первом чтении.

Советы, которые помогут сохранить много времени и усилий.

Запомните — это важно.

Тоже важное напоминание. Это указание о том, что здесь легко допустить одну из труднонаходимых ошибок и даже не догадаться о ней.

Что дальше

Обучиться языку программирования — задача отнюдь не тривиальная. Я попытаюсь сделать это настолько мягко, насколько возможно, но вы должны будете поднатужиться и освоить некоторые элементы серьезного программирования. Так что разомните пальцы, приготовьте для книжки почетное место рядом с клавиатурой и — приступим!

Часть I

Первое знакомство с C++

"В сети ему нет равных; благо, что есть электронная почта, потому что я не могу понять ни слова из того, что он говорит."

В этой части...

И новейшие потрясающие воображение авиационные симуляторы, и незамысловатые, но мощные вычислительные программы состоят из одних и тех же базовых блоков.

В этой части вы найдете основные сведения, необходимые для написания самых потрясающих программ.

Глава 1

Написание вашей первой программы

В этой главе...

- ✓ Постигая концепции C++
- ✓ Что такое программа
- ✓ Как пишут программы
- ✓ Создание первой программы
- ✓ Выполнение программы
- ✓ Разбор программ
- ✓ Вычисление выражений

Итак, мы на старте. Никого вокруг нет — только вы, я и книга. Сосредоточьтесь и постарайтесь овладеть некоторыми фундаментальными понятиями.

Компьютер — это поразительно быстрая, но невероятно глупая машина. Он может выполнить то и только то, что прикажешь (причем с умом!), — ни больше, ни меньше.

К нашему глубокому сожалению, компьютер не понимает привычного человеку языка — ни английского, ни русского, ни какого-либо другого. Знаю, вы хотите возразить: "Я видел компьютеры, понимающие английский". В действительности язык понимала выполняемая компьютером специально разработанная программа. (Это объяснение не совсем корректно. Но, с другой стороны, если я захочу рассказать своему маленькому сыну что-то слишком для него сложное, то постараюсь объяснить это доступными для него словами и понятиями.)

Компьютеры понимают язык, который называют машинным или языком программирования. Человеку крайне сложно разговаривать машинным языком. Поэтому в качестве посредника между компьютерами и людьми решили использовать такие языки *высокого уровни,* как C++. Они более или менее понятны людям и конвертируются в машинный язык, воспринимаемый компьютерами.

Постигая концепции C++

В начале семидесятых консорциум очень умных людей разрабатывал компьютерную систему Multix. Ее предназначением было обеспечение недорогого всеобщего доступа к графическим, текстовым и другим файлам, к электронной почте, эротике (ладно, это я уже переборщил). Конечно, это была совершенно глупая идея, и в целом проект провалился.

Небольшая группа инженеров, работающих в лабораториях Белла, решила использовать фрагменты Multix в небольшой операционной системе, которую окрестили Unix (Un-ix, Mult-ix — словом, все понятно?).

Эти инженеры не имели одной большой мощной машины, а лишь несколько маломощных машин разных производителей. Поскольку все они были разные, каждую программу требовалось перерабатывать под каждую машину. Чтобы избежать этих мучений, был разработан небольшой, но мощный язык, который назвали C.

Язык C оказался действительно мощным и очень скоро завоевал передовые позиции среди средств разработки программного обеспечения. Однако со временем в программировании появились новые технологии (например, достойное самого большого внимания объектно-ориентированное программирование), которые постепенно вытесняли язык C. Не желая остаться за бортом, инженерное сообщество усовершенствовало C, дополнив его новыми возможностями и получив в результате новый язык программирования — C++.

Язык C++ включает:

- ✓ словарь понятных для людей команд, которые конвертируются в машинный язык;
- ✓ структуру языка (или *грамматику*), которая позволяет пользователям составлять из команд работающие программы.

Примечание. Словарь известен также как семантика, а грамматика — как синтаксис языка.

Что такое программа

Программа — это текстовый файл, содержащий последовательность команд, связанных между собой по законам грамматики C++. Этот файл называют исходным текстом (возможно, потому, что он является началом всех наших страданий). Исходный файл в C++ имеет расширение .CPP, так же как файлы Microsoft Word оканчиваются на .DOC или командные файлы MS DOS имеют окончание .BAT. Расширение .CPP всего лишь соглашение, но в мире персональных компьютеров оно, по сути, стало законом.

Задача программирования — это написание такой последовательности команд, после преобразования которой в машинный язык можно получить программу, выполняющую наши желания. Такие машинно-исполнимые программы имеют расширение .EXE[1]. Процесс превращения программы C++ в исполнимую называется компиляцией.

Пока все выглядит достаточно легко, не так ли? Но это лишь цветочки. Продолжим...

Как пишут программы

Для написания программы вам нужны две вещи; редактор для создания исходного .CPP-файла и программа, которая преобразует исходный текст в понятный машине код .EXE-файла, выполняющего ваши команды. Инструмент, осуществляющий такое превращение, называется компилятором.

Современные инструменты разработки программ обычно совмещают в себе и компилятор и редактор. После ввода текста программы для создания выполнимого файла нужно только щелкнуть на кнопке.

Одна из популярнейших сред разработки — Visual C++ фирмы Microsoft. В ней можно скомпилировать и выполнить все программы, представленные в этой книге; однако не все из вас являются владельцами этого программного продукта из-за его довольно высокой стоимости даже у уличных торговцев (кстати, многие в это не поверят, но далеко не все программисты работают в Windows — есть и другие операционные системы). К счастью, существуют и общедоступные среды разработки программ C++, наиболее популярная из которых GNU C++.

---

[1] *Как правило, но, вообще говоря, это выполняется не всегда.* — Прим. ред.

Все свободно распространяющиеся программы можно найти в Internet. Некоторые из этих программ не совсем бесплатны — для их получения вы все-таки должны будете внести небольшую сумму. За использование GNU C++ вы не должны ничего платить.

GNU характеризуется шуточным определением "GNU — это не Unix" (GNU is Not Unix). Эта шутка возвращает нас к раннему периоду C++, когда признавалась только эта операционная система. На самом деле GNU представляет собой серию инструментов, разработанных представителями Фонда свободного программного обеспечения (Free Software Foundation).

GNU C++ вовсе не плод нездорового воображения группы разработчиков, а полнофункциональная среда для разработки программ на C++. Она поддерживает все структурные особенности языка C++ и может обеспечить выполнение любых про" грамм из этой книги (и изо всех других книг по C++)[2].

GNU C++ не является пакетом разработки программ для Windows. Если я разгадал ваши тайные желания, то у вас нет другого выхода, кроме как приобрести коммерческий пакет наподобие Visual C++.

Далее предполагается, что читатель посетил узел по адресу `www.deloreon.com` (или любой другой из сотен узлов, на которых можно найти GNU C++), загрузил последнюю версию GNU C++ и инсталлировал ее на своем компьютере, следуя приведенным в загруженном пакете инструкциям. После этого совместными усилиями будет написана наша первая программа. Задача, которую нужно реализовать, — преобразование введенной пользователем температуры по Цельсию в температуру по Фаренгейту.

Создание первой программы

Сейчас вы приступите к созданию своей первой программы на C++. Для этого потребуется ввести программный код в файл `CONVERT.CPP`, а потом скомпилировать его в выполнимую программу.

Введение кода

При создании любой программы на C++ первым шагом становится введение команд языка с помощью текстового редактора. Сердцем пакета GNU C++ является утилита `rhide`. Она представляет собой редактор, который связывает различные возможности GNU C++ в единый интегрированный пакет. В этой главе `rhide` будет использоваться для создания файла `Convert.cpp`.

1. Откройте окно MS DOS двойным щелчком на пиктограмме MS DOS в меню Windows Пуск⇨Программы.

   GNU C++ является утилитой, работающей из командной строки. Запускать `rhide` вы всегда будете из системного приглашения MS DOS.

2. Создайте каталог `C:\CPP_For_Dummies\Chap01` (подразумевается, что рабочим диском является C).

Вы можете использовать любое угодное душе имя каталога, но намного легче манипулировать именами каталогов MS DOS, которые не содержат пробелов. Еще луч-

[2] *Пожалуй, это слишком смелое утверждение, если речь идет о книге с применением последних стандартов C++, но для подавляющего большинства книг данное утверждение верно.* — Прим. ред.

ше использовать имена каталогов из восьми или менее символов — тогда с ними гарантированно можно будет работать в любой среде.

Находясь в каталоге ChapOl, запустите `rhide` из приглашения MS DOS.

Создайте пустой файл, выбрав пункт New в меню File. Откроется пустое окно, в котором необходимо ввести текст программы (причем он должен точно соответствовать книжному оригиналу).

Пусть количество отступов и пробелов вас не волнует: не так важно, сколько пробелов вы поставили перед началом строки или между соседними словами. Однако C++ весьма чувствителен к другому: надо следить, чтобы все команды набирались в нижнем регистре[3].

Интерфейс rhide

Этот интерфейс существенно отличается от интерфейса Windows-программ. Программы Windows "разрисовывают" выводимую на экран информацию, и это придает им более изысканный вид.

А вот интерфейс rhide — текстовый. Он использует символы из арсенала персонального компьютера, так что облик rhide не так изящен. Интерфейс rhide не поддерживает изменение размеров окна, оно стандартного размера (80x25 символов). Но все же rhide поддерживает многое из того, к чему вы привыкли: выпадающие меню, множественные окна, интерфейс с использованием мыши, "горячие" клавиши.

Для более опытных из вас достаточно напомнить, что rhide очень похож на интерфейс набора инструментальных средств ныне "усопшего" Borland.

```
//
//  Программа для преобразования
//   градусов Цельсия в градусы Фаренгейта:
//  Fahrenheit = NCelsius  *  (212 - 32)/100 + 32
//
#include <stdio.h>
#include <iostream.h>
int main(int nNumberofArgs, char* pszArgs[])
{
    // Введите температуру в градусах Цельсия
    int nNCelsius;
    cout << "Введите температуру по Цельсию: ";
    cin  >> nNCelsius;

    // для приведенной формулы преобразования
    //вычислим преобразующий множитель
    int nNFactor;
    nNFactor = 212 - 32;

    // используем вычисленный множитель для
    // преобразования градусов Цельсия в
    // градусы Фаренгейта

    int nFahrenheit;
    nFahrenheit = nNFactor * nNCelsius/100 + 32;
```

[3] *В C++ отнюдь не запрещается использование символов в верхнем регистре — просто язык* чувствителен к регистру, *а это значит, что int main и* `Int Main`, *например, означают совершенно разные вещи.* — Прим. ред.

```
    // вывод результатов
    cout << "Температура по Фаренгейту: ";
    cout << nFahrenheit;

    return 0;
}
```

После ввода этого кода выберите Save As в меню File и сохраните файл под именем Conversion.cpp.

Хотя это вам может показаться и не очень впечатляющим, но только что вы создали вашу первую программу на C++!

Создание выполнимого файла

После сохранения на диске исходного файла Conversion.cpp самое время сгенерировать выполнимый машинный код.

Для этого нужно выбрать пункт Make меню Compile или просто нажать клавишу <F9>. Интерфейс rhide откроет в нижней части экрана еще одно небольшое окно, отражающее ход процесса компиляции. Если все в порядке, после сообщения Creating Conversion.exe вы увидите слова no errors.

Ошибки инсталляции GNU C++

В процессе инсталляции могут возникнуть довольно распространенные ошибки, которые испортят ваши выдающиеся достижения в программировании. Две наиболее распространенные ошибки можно диагностировать, пытаясь скомпилировать программу.

Сообщение Bad command or file name означает, что MS DOS не может найти gcc.exe, т.е. компилятор GNU C++. Причиной этого может быть неправильная инсталляция продукта или некорректное задание пути c:\djgpp\bin к каталогу, где находится gcc.exe. Попытайтесь реинсталлировать GNU C++ и проверьте, присутствует ли в файле autoexec.bat строка SET PATH=c:\djgpp\bin;%PATH%. Перезагрузите компьютер.

Сообщение gcc.exe: Conversion.cpp: No such file or directory (ENOENT) означает, что gcc не понимает используемых длинных имен файлов (что характерно для MS DOS). Эту проблему можно решить, подредактировав файл c:\djgpp\djgpp.env и присвоив в нем свойству LFN значение Y.

И еще одно предупреждение: GNU C++ не понимает имен файлов, которые содержат пробелы. В этом случае не поможет даже флажок, разрешающий длинные имена.

Когда компилятор GNU C++ сталкивается в программе с какой-либо некорректностью, он генерирует сообщение об ошибке. Ошибки в программном коде являются таким же распространенным явлением, как снег на Аляске. Вы обязательно столкнетесь с многочисленными предупреждениями и сообщениями об ошибках, возможно, даже при работе с простенькой программой Conversion.cpp. Чтобы продемонстрировать процесс исправления ошибок, изменим в строке 13 cin>>nCelsius; на cin>>>nCelsius;.

Это нарушение кажется совсем невинным — и вы, и я вскоре бы о нем забыли. Но при компиляции rhide генерирует следующие сообщения:

```
Compiling: Conversion.cpp
In function 'int main(int, char **)':
Conversion.cpp(13) Error: parse error before '>'
There were some errors
```

Таким многословным образом компилятор сообщает, что GNU C++ не может понять значения записи ">>>" из 13-й строки программы.

Термин *parse* обозначает, что ошибка была найдена при проведении синтаксического анализа команд C++.

Отредактируйте файл, удалив лишний знак ">". Теперь, нажав <F9>, вы благополучно создадите выполнимую программу `Conversion.exe`.

Почему C++ так требователен

Как видим, C++ смог определить, что мы испортили в предыдущем примере. Однако если **GNU** C++ нашел ошибку, то почему же он сам не решит эту проблему — и дело с концом?

Ответ достаточно прост. Хотя в данном случае GNU C++ считает, что мы всего лишь допустили опечатку при вводе символов ">>", полностью положиться на его интуицию нельзя. Ведь правильной командой в действительности может оказаться совершенно другая, не имеющая никакого отношения к ошибочной команде. Если бы компилятор исправлял ошибки так, как считает нужным, то **GNU** C++ скрывал бы от разработчиков многие реальные проблемы.

Требуется много усилий и времени, чтобы найти скрытую в программе ошибку. Намного лучше позволить найти эту ошибку компилятору. Мы тратим наше драгоценное время, создавая ошибки. Но зачем же расходовать его еще и на их поиск, если **GNU** C++ может их выловить, не тратя нашего времени. Каким, как вы думаете, будет мой выбор?..

Выполнение программы

Пришло время испытания вашего нового творения. Для выполнения программы нужно запустить файл CONVERT.EXE И обеспечить его входными данными. Полученный результат можно использовать для анализа.

Чтобы запустить программу из среды GNU C++, нужно выбрать пункт Run из меню Run или нажать <Ctrl+F9>.

При этом откроется окно, в котором вам предложат ввести температуру по Цельсию. Для проверки правильности внесите какую-то заранее известную температуру, например 100°. После нажатия клавиши <Enter> программа возвращает эквивалентную температуру по Фаренгейту, т.е. 212°. Однако, так как rhide закрывает окно сразу же по завершении программы, увидеть результат вы не сможете; rhide открывает предупредительное окно с сообщением, что программа завершена с нулевым кодом ошибки. Несмотря на устрашающее выражение "код ошибки", ноль как раз обозначает отсутствие ошибок в процессе выполнения программы.

Чтобы увидеть результат выполнения программы, щелкните на пункте User Screen меню Windows или нажмите <Alt+5>. Это окно отображает предыдущее окно сеанса MS DOS. В нем можно увидеть последние 25 строк выполнения программы, в том числе вычисленное значение температуры по Фаренгейту.

Поздравляю! Используя GNU C++, вы только что ввели, скомпилировали и запустили свою первую программу.

GNU - это не Windows

Заметьте, что пакет GNU C++ не предназначен для разработки программ Windows. Написать Windows-приложение с помощью GNU C++ теоретически можно, но без использования внешних библиотек, как в Visual C++, сделать это весьма непросто.

Windows-программы имеют ярко выраженный визуально-ориентированный оконный интерфейс. А `Convension.exe` является 32-битовой программой, которая выполняется в среде Windows, но Windows-программой в настоящем смысле ее не назовешь.

Если вы не знаете, чем 32-битовая программа отличается от 16-битовых, не волнуйтесь об этом. Как уже отмечалось, эта книга не о написании программ для Windows. Интерфейс программ, разработанных нами в среде GNU C++, представляет собой набор командных строк в окне MS DOS.

Начинающим Windows-программистам огорчаться не следует: ваше время не пропадет зря. Изучение C++ совершенно необходимо как предварительное условие для написания Windows-программ.

Помощь в GNU C++

GNU C++ обеспечивает разработчиков системой помощи через пользовательский интерфейс `rhide`. Разместите свой курсор на конструкции, смысл которой неясен, и нажмите <F1>. Раскроется окно с имеющейся по этому вопросу информацией. Для отображения тематического списка помощи нужно выбрать Help⇨Index.

Справочная информация, предоставляемая GNU C++, не так исчерпывающа, как справка некоторых других инструментальных средств (например, Microsoft Visual C++). Если разместить курсор на слове "`int`" и нажать <F!>, появится окно, описывающее редактор, — но ведь это не совсем то, что мы искали. Внимание GNU C++ в основном сосредоточено на библиотеке функций и опциях компиляции. К счастью, после того как вы овладеете языком C++, помощи GNU C++ окажется достаточно для разработки большинства приложений.

Разбор программ

Хотя разбор программы, написанной другим, — вещь не самая впечатляющая, но на этапе вашего становления как программиста заниматься этим очень даже полезно. Рассмотрим далее программу `Conversion.cpp` и найдем элементы, общие для всех программ.

Определение структуры программ C++

Каждая программа, написанная с использованием материала этой книги, в своей основе будет иметь одну и ту же базовую схему:

```
//Это комментарии, которые компьютер игнорирует
#include <stdio.h>
#include<iostream.h>
int main(int nNumberofArgs, char* pzArgs[ ])
{
    //...здесь записывается код программы...
    return 0;
}
```

Если не вникать в детали, то выполнение программы начинается с кода, который помещен между открывающей и закрывающей скобками.

Использование в исходном коде комментариев

Нетрудно заметить, что первые несколько строк `Conversion.cpp` являются обычным текстом. Значит, или GNU C++ оказался более понятливым, чем я его представил, или — что вероятнее всего — этот код предназначается для человеческих глаз.

Оформленные таким образом строки называют комментариями. Чаще всего в комментариях программист объясняет конкретные действия, которые он собирается реализовать в следующем фрагменте кода. Компилятор комментарии игнорирует.

Комментарии в C++ начинаются с двойной косой черты (//) и заканчиваются переходом на новую строку. В их тексте можно использовать любые символы. Длина комментариев не ограничена, но, так как желательно, чтобы они не превосходили размеров экрана, обычно *придерживаются* нормы не более 80 символов.

Во времена печатных машинок перевод каретки означал начало новой строки. Но ввод с клавиатуры — это не печатание на машинке. В этом случае новая строка является символом, который завершает текущую командную строку.

Допустима и другая форма комментариев, при которой игнорируется все, что `/*` заключается в такие скобки `*/`; однако эта форма комментариев в C++ почти не используется.

Присутствие в программах игнорируемых компьютером команд C++ (или любого другого языка программирования) может покачаться странным. Однако все компьютерные языки предлагают те или иные способы оформления комментариев. Объяснения программиста раскрывают ход его мыслей при написании программного кода. Ведь замыслы программиста могут быть совсем неочевидными для людей, которые захотят воспользоваться программой или ее модифицировать. Да и сам автор программы, взглянув на нее через месяц, не всегда сможет вспомнить ее суть.

Использование инструкций в программах

Все программы C++ в своей основе имеют то, что называют инструкциями. В этом разделе рассмотрим такие из них, которые составляют остов программы Convert.

Инструкция — это команда, которую понимает компилятор. Все инструкции, кроме комментариев, оканчиваются точкой с запятой (для комментариев на то есть свои причины, но все же иногда это неудобно; мне кажется, что во избежание путаницы после комментариев точку с запятой следовало бы ставить тоже).

При запуске программы первой выполняется инструкция, находящаяся после открывающей фигурной скобки, а затем поочередно выполняются и все остальные инструкции.

Просмотрев программу, можно увидеть, что пробелы, символы табуляции и новой строки появляются на протяжении всей программы. Переход на новую строку осуществляется практически после каждой инструкции. Все эти символы называют *непечатаемыми*, так как на экране монитора их увидеть нельзя.

Для повышения удобочитаемости допускается добавление символов пробела в любом месте программы (но не внутри слов!).

Игнорируя пропуски, язык C++ учитывает регистр. Например, переменные `fullspeed` и `FullSpeed`, с его точки зрения, не имеют между собой ничего общего.

Объявления

Строка `int nNCelsius;` является инструкцией объявления. *Объявление*— это инструкция, которая определяет переменную. Переменная — это контейнер, в котором хранятся значения некоторого типа. Переменная может содержать числовые или символьные значения.

Термин "переменная" был заимствован из алгебры, где он является стереотипным для следующих выражений:

```
x = 10
У = 3 * x
```

Во втором выражении y — множество, которое задается формулой `3*x`. Но что такое x? Переменная x играет роль контейнера для хранения каких-то значений. В нашем случае значением x является 10, но с таким же успехом можно определить значение x равным 20, 30 или -1. Вторая формула имеет смысл при любом числовом значении x.

В алгебре можно начать работу с выражений типа x = 10. Программируя на C++, переменную x также необходимо определить перед ее использованием.

В C++ переменная имеет тип и имя. Переменная, определенная в строке 11, называется nNCelsius. Согласно объявлению она целочисленная (подобные названия типов, наверное, имеют целью развить у программистов ассоциативное мышление — тип `int` представляет собой сокращенное `integer`).

Для C++ имя переменной не имеет никакого специфического значения. Имя должно начинаться с букв английского алфавита A-z или a-z[4]. Остальные символы могут быть буквами, цифрами от 0 до 9 или подчеркивающей чертой (_). Имена переменных могут быть настолько длинными, насколько это вам удобно.

Существует негласная договоренность о том, что имена переменных должны начинаться *со* строчной буквы. Каждое слово внутри имени переменной пишется с прописной буквы, например `myVariable`.

Старайтесь давать переменным короткие, но наглядные имена. Избегайте таких имен, как x, потому что они не несут никакого смысла. Примером достаточно наглядного имени переменной может служить `lengthOfLineSegment`.

Генерирование вывода

Строки, начинающиеся с cout и cin, называют инструкциями ввода-вывода, или сокращенно I/O (input/output) (как и все инженеры, программисты любят сокращения и аббревиатуры).

Первая инструкция I/O выводит фразу "*Введите температуру по Цельсию*" в cout (произносится как "си-аут"). В C++ cout — это имя стандартного устройства вывода. В нашем случае таким устройством является монитор.

В следующей строке все происходит с точностью до наоборот. Со стандартного устройства ввода мы получаем значение и сохраняем его в целой переменной nNCelsius. Стандартным устройством ввода для C++ в данном случае служит клавиатура. Этот процесс является аналогом упоминаемой выше алгебраической формулы x = 10 в C++. Профамма будет считать значением nNCelsius любое целое число, введенное пользователем.

вычисление выражений

Почти все профаммы выполняют вычисления того или иного вида. В C++ выражением называется инструкция, которая выполняет какие-либо вычисления. Иными словами, выражение — это инструкция, которая имеет значение. Команда, генерирующая это значение, называется оператором.

Например, в программе Conversion можно назвать "вычисляющим выражением" совокупность строк с объявлением переменной `nNFactor` и определением ее значения

---

[4] *Вообще говоря, имя может также начинаться с символа подчеркивания, хотя на практике это используется довольно редко.* — Прим. ред.

как результата вычислений. Эта команда вычисляет разницу между 212 и 32. В данном примере оператором является знак "минус" ("-"), а выражением — "212-32".

Сохранение результатов выражения

Разговорный язык может быть далеко не однозначным. Яркий тому пример — слово *равный*. Оно может употребляться в значении "одинаковый" (например, *равные силы),* а может применяться в математике для построения выражений типа "у равен утроенному x".

Чтобы избежать двусмысленности, программисты на C++ называют знак "=" *оператором присвоения.* Оператор присвоения сохраняет результат выражения, находящегося справа от "=", в переменной, записанной слева. Программисты говорят, что "переменной nNFactor присвоено значение 212-32".

Обзор программы Convert продолжается...

Второе выражение, представленное в Conversion.cpp, несколько сложнее первого. В нем используются всем известные математические символы: "*" для умножения, "/" для деления, "+" для сложения. В этом случае, однако, вычисления выполняются не просто с константами, а с переменными.

Значение переменной nNFactor (кстати, уже вычисленное) умножается на значение переменной nNCelsius (которое было введено с клавиатуры). Результат делится на 100 и к нему прибавляется 32. Результат всего выражения приводится к целому типу и присваивается переменной nFahrenheit.

Последние команды выводят строку *"Температура по Фаренгейту:"* и отображают значение переменной nFahrenheit.

Глава 2

Премудрости объявления переменных

В этой главе...

- ✓ Объявление переменных
- ✓ Объявление разных типов переменных
- ✓ Объявления типов переменных
- ✓ Выражения смешанного типа

Одним из основных в C++ является понятие *переменной*. Переменную можно представить, как небольшую шкатулку, в которой хранятся вещи для дальнейшего многократного использования. Понятие переменной заимствовано из математики. Инструкция вида

```
x = 1
```

сохраняет значение 1 в переменной x. После такого присвоения математики могут использовать переменную x вместо константы 1, пока не изменят значение x на другое.

В C++ переменные используются таким же образом. После присвоения `x = 1;` и до следующего изменения содержимого переменная x становится обозначением числа 1 в программе. При этом говорят, что значение x есть 3.

К сожалению, в C++ возни с переменными несколько больше, чем в математике. Эта глава как раз и повествует о заботах, связанных с использованием переменных в C++.

Объявление переменных

Все числа, с которыми работает C++, хранятся в небольших "ларцах", которые называются переменными. В математике с переменными обращаются достаточно свободно. Допускаются формулировки наподобие

$$\begin{cases} (x+2) = y/2 \\ x+4 = y \end{cases}$$, найти x и y.

Уверен, вам не нужно объяснять, что такой способ задания переменных действительно однозначен. К сожалению, C++ не так сообразителен (как я уже упоминал, компьютеры ну просто очень глупы!).

Прежде чем использовать в программе новую переменную, вы должны ее объявить:

```
int x;
x = 10;
int y;
y = 5;
```

Таким образом, мы объявили переменные x, y и определили, что они могут содержать значения типа `int` (типы переменных обсуждаются в следующем разделе). Объявлять переменные можно в любом удобном для вас месте программы, но обязательно перед их использованием.

Объявление разных типов переменных

Вы, вероятно, думаете, что переменная в математике — это совершенно аморфное хранилище для любой информации, которая взбредет в голову. Ведь в принципе можно свободно написать следующее:

```
x = 1;
x = 2.3
x = "Это - предложение."
x = Техас
```

Но C++ не настолько гибкий язык. (С другой стороны, C++ очень легко может справиться с совершенно непосильными для нас задачами. Например, ему ничего не стоит сложить миллион чисел всего за одну секунду[5].) В C++ переменные могут хранить значения только одного типа. Причиной тому является большая разница в размерах памяти, необходимой для хранения значений переменных разных типов. Если некоторые данные программы могут состоять всего из одного числа, то довольно часто разработчикам приходится манипулировать целыми предложениями.

Добавлю, что особенности использования переменных разных типов различны. Пока вы встречались только с переменными типа ir.t:

```
int x;
x = 1;
```

В C++ тип int определяет множество целых чисел. Напомню, что целым называется число, не имеющее дробной части.

Целые числа используют для самых разных видов вычислений. Детально этому учат в младшей школе, приблизительно до шестого класса, и лишь потом начинается путаница с дробями[6]. Та же тенденция характерна и для C++, в котором более 90% всех переменных имеют тип int[7].

К сожалению, иногда использование в программах переменных типа ir.t приводит к ошибочным результатам. Когда в первой главе вы работали с программой, преобразующей температуру, существовала (пусть неявно) проблема: программа могла работать только с целыми значениями температуры. Отмечу, что в этой конкретной программе использование исключительно целых чисел вряд ли приведет к отрицательным последствиям. Но при проведении серьезных метеорологических исследований усечение дробной части температурных значений может поставить под вопрос истинность полученных результатов. Простейшим примером может служить определение значений температуры для книги рекордов Гиннеса. В этом случае требуется высокая точность вычислений, но из-за отбрасывания дробных частей достичь ее невозможно.

Ограничения, налагаемые на целые числа в C++

Целочисленные переменные в C++ представляются типом int. На переменные этого типа накладываются те же ограничения, что и на их эквиваленты в математике.

---

[5] *Заметим, что складывает числа не язык, а компьютер — язык только передает ему задание.* — Прим. ред.

[6] *Автор имеет в виду американскую школу; впрочем, современные тенденции в отечественной школе примерно те же.* — Прим. ред.

[7] *Эта величина опять-таки существенно зависит от типа разрабатываемой программы.* — Прим. ред.

Округление до целых значений

Рассмотрим проблему вычисления среднего трех чисел. Введем три целочисленные переменные — nValue1, nValue2, nValue3. Среднее значение вычисляется по формуле

```
(nValue1+ nValue2(• nValue3) / 3
```

Поскольку все три значения являются целыми, их сума тоже будет целым числом. Например, сумма чисел I, 2 и 2 равна 5. Но если 5 поделить на 3, получим $1\frac{2}{3}$, или 1,666.... В отличие от людей (обладающих разумом), компьютеры (которым он свойственен далеко не всегда) приводят полученный результат к целому значению, просто отбрасывая его дробную часть. При этом 1,666 утратит свой "дьявольский" остаток и превратится в 1.

Для многих приложений усечение дробной части числа не представляет большой проблемы. Зачастую оно может быть даже полезным (разумеется, сказанное не касается математических или экономических программ). Однако такое округление целых может весьма пагубно сказаться на работе других программ. Рассмотрим следующую, эквивалентную приведенной выше формулу:

```
nValue1/3 + nValue2/3 + nValue3/3
```

Подставляя в нее те же значения 1, 2 и 2, в результате получим 0. Это случилось потому, что каждое слагаемое оказалось числом, меньшим 1. Компьютер округлил их до 0, а сумма трех нудей, как известно, равна 0. Так что такого приведения к целочисленным значениям, вообще говоря, нужно избегать.

Ограничения диапазона

Второй проблемой переменной типа int является ограниченный диапазон возможных ее значений. Максимальным значением обычной целочисленной переменной является число 2 147 483 647, минимальным — -2 147 483 648, т.е. общий диапазон — около 4 млрд чисел[8].

Решение проблемы усечения дробной части

Рассмотренные особенности переменных типа int делают невозможным их использование в некоторых приложениях. Но, к счастью, C++ умеет работать и с десятичными числами, которые могут иметь ненулевую дробную часть (математики называют их *действительными числами).* Используя действительные числа, можно избежать большинства перечисленных проблем. Заметьте, что десятичные числа могут иметь ненулевую дробную часть, а могут и не иметь, оставаясь действительными. В C++ число 1.0 является таким же действительным числом, как и 1.5. Эквивалентным им целым числом является просто 1.

В C++ действительные числа определены как числа с плавающей точкой, или просто float. Используя выражение "с плавающей точкой", имеют в виду, что десятичную запятую (или используемую вместо нее в программах точку) в десятичных числах можно перемешать вперед и назад настолько, насколько этого требуют вычисления. Действительные переменные объявляются так же, как и переменные типа int:

```
float fValue;
```

---

[8] *Вообще говоря, диапазон представимых типом int значений определяется множеством факторов — в первую очередь компилятором, на выбор типа int которого оказывает огромное влияние тип компьютера, поэтому считать определенным раз и навсегда, что диапазон значений int простирается от -2^{32} до $+2^{32}-1$, нельзя.* — Прим. ред.

Начиная с этой строки, во всей остальной части программы переменная fValue может принимать значения типа float. Тип уже объявленной переменной изменить нельзя: fValue является действительной переменной и останется ею до конца программы. Рассмотрим, как решается присущая целочисленным переменным проблема отбрасывания дробной части. Для этого в объявлении все переменные определим как действительные (тип float):

```
1/3 + 2/3 + 2/3
```

Это эквивалентно выражению

```
0.333... + 0.666... + 0.666...,
```

которое равно

```
1.666...
```

Ограничения, налагаемые на числа с плавающей точкой

Хотя числа с плавающей точкой могут решить многие вычислительные проблемы, на их использование тоже существуют ограничения. Проблемы отчасти противоположны тем, которые характерны для целочисленных переменных. Действительные переменные не могут использоваться для перечисления, с ними сложнее работать компьютеру, и они тоже страдают от ошибок округления (хотя намного меньше, чем переменные типа int).

Перечисление

Использовать переменные с плавающей точкой для простого перечисления нельзя. C++ не умеет определять, какое целочисленное значение подразумевается под действительным числом.

Например, ясно, что 1.0 есть 1. Но что такое 0.9 или 1.1? Следует ли их рассматривать как 1? Так что C++ избегает многих проблем, требуя использовать при перечислении только целые значения.

Скорость вычислений

Исторически сложилось так, что процессор компьютера выполняет операции с целыми числами гораздо быстрее, чем с действительными. Для сложения 1000 целых чисел процессору может потребоваться столько же времени, сколько для выполнения только 200 вычислений с плавающей точкой.

Однако с увеличением производительности микропроцессоров проблема скорости вычислений становится все менее важной. Большинство современных процессоров содержат специальные вычислительные схемы, которые позволяют вычислять выражения с плавающей точкой почти так же быстро, как и целочисленные выражения.

Потеря точности

Действительные переменные не могут решить всех вычислительных проблем. Обычно их точность ограничена приблизительно шестью разрядами, но есть и расширенный варианте типа для действительных чисел, который может содержать после десятичной точки до 15 значимых разрядов.

Чтобы понять эту проблему, представим 1/3 в виде бесконечной последовательности 0.333.... Однако математическое понятие периода в программировании не имеет смысла, так как точность компьютерных вычислений ограничена и где-то наша дробь должна оборваться (что зависит от использованного для хранения числа типа переменной). Поэтому, усреднив числа 1, 2, 2, мы получим не точное, а приблизительное значение 1.666667.

В некоторых случаях ошибки округления может исправлять сам C++; например, выводя информацию на экран, вместо числа 0.999999 C++ выдаст пользователю значение 1.

Ограниченность диапазона

Тип данных `float` также ограничен, хотя диапазон чисел с плавающей точкой намного обширнее диапазона целочисленных переменных. Максимальным значением типа `int` является число чуть больше 2 млрд. Максимальное значение переменной типа `float` приблизительно равно 10^{38}, т.е. 1 с 38 нулями[9].

Представляя переменные с плавающей точкой в стандартном виде, C++ учитывает после десятичной точки только первые шесть разрядов. Остальные разряды становятся жертвами ошибочных округлений. Поэтому действительная переменная может хранить значение 123 000 000 без потери точности из-за округления, в то время как значение 123 456 789 приведет к ошибке округления.

Объявления типов переменных

Вы уже знаете, что все переменные в программе должны быть объявлены и что им должен быть назначен тип. В табл. 2.1 представлен список некоторых стандартных типов переменных языка C++ с указанием их достоинств и недостатков.

Таблица 2.1. Переменные C++

ПЕРЕМЕННАЯ	ПРИМЕР	ХАРАКТЕРИСТИКА
`int`	`1`	Простые положительные или отрицательные числа, используемые для перечисления
`float`	`1.0F`	Действительные числа
`double`	`1.0`	Расширенная версия `float`: использует больше памяти, допускает работу с большим диапазоном и обеспечивает более высокую точность вычислений
`char`	`c`	Символьный тип; значением переменных может быть символ алфавита, цифра, знак препинания или знак арифметической операции. Не годится для арифметических операций
`string`	`"this is a string"`	Строка символов, составляющая предложение
`long`	`10L`	Потенциально расширенная версия типа `int`. В GNU C++ и Microsoft Visual C++ разницы между типами `long` и `int` нет

Следующий оператор объявляет переменные `lVariable` **типа** `long` **и** `dVariable` **типа** `double` **и присваивает им начальные значения:**

```
//объявление переменной и установка ее равной 1
long lVariable;
lVariable = 1
```

[9] *Следует отдавать себе отчет, что это не означает, будто тип float может представить 10^{38} разных значений; вспомните, что говорилось несколько выше о количестве разрядов в числах этого типа.* — Прим. ред.

```
//объявление переменной типа double и ее инициализация
double dVariable;
dVariable = 1.0;
```

Объявить и инициализировать переменную можно одним оператором:

```
int nVariable = 1; //объявление переменной и ее инициализация
```

Единственное преимущество инициализации переменной в объявлении — уменьшение размеров текстов программ. Но даже и это весьма важное достижение.

Переменная типа char может содержать единственный символ, в то время как строковая переменная — строку символов. Поэтому *a* можно интерпретировать и как символ *a,* и как строку, содержащую только букву *a* (в действительности string не является типом переменной, но в большинстве случаев его можно рассматривать именно так). В главе 9, "Второе знакомство с указателями", вы найдете детальное описание этого типа данных.

Символ *a* и строка *a* — это далеко не одно и то же. Если вы захотите присвоить символьной переменной строковое значение (или наоборот), вы не сможете этого сделать даже в том случае, когда строка содержит единственный символ.

Типы констант

Константой называют произвольную постоянную величину (например, 1, 0.5 или 'c'). Подобно переменным, константы имеют свой тип. В выражении n = 1; константа 1 имеет тип int. Чтобы привести 1 к типу long, нужно написать n = 1L;. Для лучшего понимания можно провести следующую аналогию: если под 1 понимать поездку на грузовике, то 1L можно интерпретировать как путешествие на лимузине. Их маршруты могут быть совершенно одинаковыми, но согласитесь, что путешествовать вторым способом гораздо удобнее.

Константу 1 можно также привести к действительному числу 1.0. Однако заметим, что по умолчанию типом действительной константы является double. Поэтому 1.0 будет числом типа double, а не float.

Специальные символы

Для работы с любыми печатаемыми символами можно использовать переменные типа char или string. Но значениями переменных, используемых в качестве символьных констант, могут быть и непечатаемые символы. В табл. 2.2 приведено описание некоторых важных непечатаемых символов.

Таблица 2.2. Специальные символы

СИМВОЛЬНЫЕ КОНСТАНТЫ	ОБОЗНАЧЕНИЕ ДЕЙСТВИЙ
\n	Новая строка
\t	Табуляция
\0	Нулевой символ
\\	Обратная косая черта

С символом новой строки вы уже встречались раньше. Он позволяет разделить строку в любом месте на две части. Например, строка

```
"Это первая строка\nЭто вторая строка"
```

при выводе на экран будет выглядеть так:

```
Это первая строка
Это вторая строка
```

По аналогии символ табуляции \t перемещает выводимую информацию к следующей позиции табуляции. В зависимости от типа компьютера, на котором вы запустите программу, эта позиция может изменяться. Символ "обратная косая черта" используется для обозначения специальных символов, поэтому, чтобы вывести его на экран, необходимо записать два символа: \\.

Коллизии между C++ и именами файлов MS DOS

В MS DOS для разделения имен файлов в указаниях пути используется символ обратной косой черты. Так, `root\folderA\file` представляет собой путь к файлу `file` в папке `folderA`, которая является подкаталогом каталога `root`.

К сожалению, функциональное предназначение обратной косой черты в MS DOS и C++ не совпадает. Обратная косая черта в C++ используется для обозначения управляющих символов, а ее саму можно вывести с помощью символов \\. Поэтому путь MS DOS `root\folderA\file` представляется в C++ строкой `root\\folderA\\file`.

Выражения смешанного типа

C++ позволяет использовать в одном выражении переменные разных типов. Например, позволено складывать целые и действительные переменные. В следующем выражении переменная nValue является целой:

```
//в следующем выражении перед выполнением операции сложения
// значение nValue1 преобразуется к типу double
int nValue1 = 1;
double fValue = nValue1 + 1.0;
```

Выражение, в котором два операнда относятся к разным типам, называется *выражением смешанного типа*. Тип генерируемого в результате значения будет соответствовать более мощному типу операнда. В нашем случае перед началом вычислительного процесса nValue1 конвертируется в тип double. По тому же принципу выражение одного типа может быть присвоено переменной другого типа, например:

```
//в следующем задании целая часть
// fVariable сохраняется в nVariable
float fVariable = 1.0;
int nVariable;
nVariable = fVariable;
```

Если переменная в левой стороне равенства относится к типу менее мощному, чем переменная справа, то при таком присвоении можно потерять точность или диапазон значений (например, если значение переменной fVariable превышает диапазон допустимых значений переменной nVariable).

Соглашения по именованию

Вы могли заметить, что имя каждой переменной начинается с определенного символа, который, как может показаться, совсем ни к чему (эти специальные символы приведены в таблице ниже). С помощью соглашений по использованию этих символов можно мгновенно распознать, что `dVariable` — это переменная типа double.

Эти символы помогают программисту распознавать типы переменных, не обращаясь к их объявлениям в другом месте программы. Так, нетрудно определить, что в представленном ниже выражении осуществляется присвоение смешанного типа (переменная типа long присваивается целочисленной переменной):

```
nVariable = lVariable;
```

- Для C++ использование этих специальных символов не имеет никакого значения. При желании для обозначения переменной типа int вы можете использовать любую другую букву. Но использование "первобуквенного" соглашения упрощает понимание, и многие программисты постоянно используют подобные схемы в своей работе.

Символ	Тип
n	int
l	long
f	float
d	double
c	character
sz	string

Преобразование типа большего размера в меньший называется *понижающим приведением* (demotion), а обратное преобразование — *повышающим приведением* (promotion).

Использование в C++ выражений смешанного типа — идея далеко не самая блестящая; их лучше избегать, не позволяя C++ делать преобразования за вас.

Глава 3

Выполнение математических операций

В этой главе...

- ✓ Бинарная арифметика
- ✓ Анализ выражений
- ✓ Определение порядка операций
- ✓ Выполнение унарных операций
- ✓ Использование операторов присвоения

Переменные придуманы математиками не только для того, чтобы было что описывать и в чем сохранять какие-то значения. Над переменными можно выполнять самые разные действия: складывать, перемножать, вычитать и т.д. Список возможных операций достаточно обширен.

Эти основные математические операции используются и в программах C++. Ведь приложения без вычислительных возможностей себя совершенно не оправдывают. Кому нужна страховая программа, которая не может подсчитать даже суммы взносов?

Операции C++ внешне идентичны обыкновенным арифметическим операциям, выполняемым на клочке бумаги; разве что применяемые в вычислениях переменные перед использованием нужно объявлять:

```
int var1;
int var2 = 1;
var1 = 2 * var2;
```

В этом примере объявлены две переменные, var1 и var2. Переменной var2 присвоено начальное значение 1, var1 определена как результат удвоения переменной var2.

В этой главе вы найдете описание всего множества математических операторов C++.

Бинарная арифметика

Бинарными называются операторы, которые имеют два аргумента. В выражениях типа var1 *op* var2 оператор *op* бинарный. Самыми распространенными бинарными операторами являются простые математические операции, изучаемые еще за школьными партами. Бинарные операции, которые поддерживает C++, приведены в табл. 3.1.

Таблица 3.1. Математические операции в порядке приоритета

ПРИОРИТЕТ	ОПЕРАТОР	ЗНАЧЕНИЕ
1	+ (унарный)	Реально ничего не изменяет
1	- (унарный)	Возвращает противоположное по знаку, равное по модулю значение
2	++ (унарный)	Оператор инкремента, увеличивает значение аргумента на 1
2	— (унарный)	Оператор декремента, уменьшает значение аргумента на 1

ПРИОРИТЕТ	ОПЕРАТОР	ЗНАЧЕНИЕ
3	* (бинарный)	Умножение
3	/ (бинарный)	Деление
3	% (бинарный)	Остаток (деление по модулю)
4	+(бинарный)	Сложение
4	- (бинарный)	Вычитание
5	=,*=,%=,+=,-= (специальные)	Операторы присвоений

Как видите, операторы умножения, деления, деления по модулю, сложения и вычитания имеют вид обычных математических операций. Да они и работают так же, как соответствующие им арифметические операции:

```
float var = 133 / 12;
```

Значение большинства операторов вам хорошо известно еще из начальной школы, кроме разве что операции деления по модулю.

По своей сути этот оператор означает получение остатка от деления. Например, 4 входит в 15 три раза, и остаток при этом составляет 3. Выражаясь терминами C++, 15, деленное по модулю 4, равно 3.

```
int var = 15 % 4; //переменной var присваивается значение 3
```

Программисты всегда пытаются удивить непрограммистов, а потому в C++ деление по модулю определяется так:

```
IntValue % IntDivisor
```

эквивалентно

```
IntValue - (IntValue / IntDivisor)* IntDivisor
```

Вот пример:

```
15 % 4 равно     15 - (15/4) * 4
                 15 - 3*4
                 15 - 12
                 3
```

Для действительных переменных оператор деления по модулю не определен, поскольку он целиком основан на использовании округления (округления рассматривались в главе 2, "Премудрости объявления переменных").

Анализ выражений

Самым распространенным видом инструкций в C++ является выражение. *Выражением* в C++ называют любую последовательность операторов (длиной не меньше одного), которая возвращает значение. Все выражения типизированы. Тип выражения определяется типом возвращаемого значения. Например, значение выражения 1 + 2 равняется 3, следовательно, это целочисленное выражение (тут нужно вспомнить, что константы без дробной части определяются как имеющие тип `int`). Синтаксическая конструкция, включающая математический оператор, является выражением, так как в результате выполнения любой операции получается число.

Выражения бывают как сложными, так и крайне простыми. Необычной особенностью C++ является то, что он понимает под выражением любой законченный оператор. Поэтому корректным оператором является, например, `1;`.

Он тоже представляет собой выражение, потому что его значение 1, а тип int. В операторе

```
z = x * y + w;
```

можно выделить пять выражений:

```
x * y + w
x * y
x
y
w
```

Необычный аспект C++ состоит в том, что выражение само по себе является завершенной инструкцией, т.е. упомянутое выражение `1;` — завершенная инструкция C++.

Определение порядка операций

Все операторы выполняют определенные функции. Чтобы установить порядок выполнения различных операторов, им назначены приоритеты. Рассмотрим выражение

```
int var = 2 * 3 + 1;
```

Если сложение выполнить перед умножением, то значением выражения будет 2 * 4 = 8. Если сперва выполнить умножение, то получим значение 6 + 1 = 7.

Приоритеты операций определяют порядок выполнения вычислений. Из табл. 3.1 видно, что приоритет операции умножения выше, чем сложения, т.е. результат все же равен 7 {приоритеты используются и в арифметике, и C++ следует именно им).

А что происходит, когда в одном выражении используется два оператора с одинаковым приоритетом?

```
int var = 8 / 4 / 2;
```

Как в этом случае следует поступить: сначала 8 поделить на 2 или 2 на 4? Если в одном выражении присутствуют операции с одинаковыми приоритетами, они выполняются слева направо (то же правило применяется и в арифметике). Поэтому в предыдущем примере сперва делим 8 на 4, получая 2, а затем делим его на 2, получая ответ — 1.

В выражении

```
x / 100 + 32
```

x делится на 100 и к результату добавляется 32. Но что, если программисту нужно поделить x на сумму 100 и 32? В таком случае ему придется использовать скобки:

```
x / (100 + 32)
```

При вычислении такого выражения x будет делиться на 132.

Заметим, что начальное выражение

```
x / 100 + 32
```

идентично следующему:

```
(x / 100) + 32
```

Почему это действительно так? Потому что C++ сначала выполняет операции с высшим приоритетом. А приоритет операций умножения и деления выше, чем сложения и вычитания. Поэтому скобки, указывающие на высокий приоритет данной операции, можно опустить.

На основе сказанного можно сделать вывод: в случае необходимости приоритет оператора можно повысить, используя скобки.

Выполнение унарных операций

С арифметическими бинарными операторами вы неоднократно встречались с самого первого класса. О существовании же унарных операций вы могли и не подозревать, хотя наверняка одну из них использовали довольно часто (имея дело с отрицательными числами).

Унарными называются те операторы, которые имеют только один аргумент, например *-a*.

Унарными математическими операторами являются +, -, ++ и --. Рассмотрим некоторые из них:

```
int var1 = 10;
int var2 = -var1;
```

Здесь в последнем выражении используется унарный оператор "-".

Оператор "минус" изменяет знак своего аргумента (в примере это var1) на противоположный. Положительные числа становятся отрицательными и наоборот. Оператор "плюс" знака аргумента не изменяет и фактически вообще ни на что не влияет.

Операторы ++ и -- вы можете увидеть впервые. Они увеличивают или уменьшают на 1 значение аргумента и поэтому называются операторами инкремента и декремента (от англ. *increment* (увеличивать) и *decrement* (уменьшать). — *Прим. перев.)*. К действительным переменным их применение недопустимо. После выполнения приведенного ниже фрагмента значение переменной будет равно 11.

```
int var = 10; // Инициализация переменной
    var++;    // Ее увеличение; значение переменной равно 11
```

Операторы инкремента и декремента могут находиться либо перед аргументом (префиксная форма), либо после него (постфиксная форма). В зависимости от способа записи, выполнение операторов инкремента и декремента имеет свои особенности. Рассмотрим оператор инкремента (принципы работы оператора декремента те же).

Предположим, что переменная п имеет значение 5. Оба способа применения к п оператора инкремента (++п и п++) приведут к результату 6. Разница между ними состоит в том, что значение п в выражении ++п равно б, в то время как в выражении с постфиксной формой записи оно равно 5. Это можно проиллюстрировать следующим примером:

```
            //объявляем три целые переменные
int n1, n2, n3;

n1 = 5;
n2 = ++n1; //обе переменные - n1 и n2 - получают значение 6

n1 = 5;
n3 = n1++; // n1 принимает значение 6, а n3 - 5
```

Другими словами, переменной г.2 присваивается уже увеличенное префиксным оператором инкремента значение n1, тогда как переменной n3 передается еще не увеличенное постфиксным оператором значение n1.

Использование операторов присвоения

Операторы присвоения являются бинарными, изменяющими значения своих левых аргументов. Обыкновенный оператор присвоения "=" абсолютно необходим во всех языках программирования. Этот оператор сохраняет значение правого аргумента

в левом. Однако причуды авторов языка привели к появлению и других операторов присвоения.

Создатели C++ заметили, что присвоение часто имеет вид

```
variable = variable # constant
```

Здесь # представляет собой какой-то бинарный оператор. Следовательно, чтобы увеличить целую переменную на два, программист может написать:

```
nVariable = nVariable + 2;
```

Из этой записи следует, что к значению переменной nVariable добавляется двойка и результат снова сохраняется в nVariable.

Использование в левой и правой части выражения одной и той же переменной весьма распространенное явление в программировании.

Поскольку одна и та же переменная находится по обе стороны знака равенства, было решено просто добавить оператор, используемый при вычислении, к знаку присвоения. В таких специфических операторах присвоения допускается использование всех бинарных операторов. Поэтому указанное выше выражение можно сократить до

```
nVariable += 2;
```

Смысл этой записи таков: "значение переменной nVariable увеличено на 2".

Почему так важен оператор инкремента

Разработчики C++ заметили, что программисты прибавляют 1 чаще, чем любую другую константу. Учитывая это, в язык была добавлена соответствующая конструкция.

Кроме того, большинство процессоров способны выполнять команды инкремента быстрее, чем команды сложения. Учитывая мощность микропроцессоров, которые использовались во время создания C++, подобное нововведение было действительно важным.

Модифицированные операторы присвоения используются не так часто, как обычные, но как правило повышают удобочитаемость программ.

Глава 4

Выполнение логических операций

В этой главе...

- ✓ Зачем нужны логические операторы
- ✓ Использование простых логических операторов
- ✓ Бинарные числа в C++
- ✓ Выполнение битовых логических операций

Наиболее распространенной синтаксической конструкцией C++ является выражение. Большинство используемых выражений содержит арифметические операторы сложения (+), вычитания (-) и умножения (*). В данной главе описаны все эти типы выражений.

Существует целый класс так называемых *логических* операторов. В отличие от арифметических, этот тип операторов многими не воспринимается как операторы.

Неправда, что люди не сталкиваются с логическими операторами. Значения операторов И и ИЛИ они вычисляют постоянно. Я не буду есть овсянки без молока И сахара. И закажу себе ром ИЛИ шотландский виски. Как видите, люди очень часто используют логические операторы, не осознавая этого.

Логические операторы бывают двух типов. Операторы И и ИЛИ называются простыми логическими операторами. Операторы второго типа, или битовые операторы, уникальны, так как используются только в программировании. Этот тип операторов позволяет работать с любым битом в машинном представлении числа.

Зачем нужны логические операторы

У вас может возникнуть вопрос: "Если до сегодняшнего дня меня совершенно не волновали логические операторы, почему это должно случиться теперь?". Да потому, что программы должны "уметь" принимать решения. Программы, написанные без принятия решений, по сложности подобны приведенной в первой главе (вспомните, что все выполняемые ею действия совершенно безальтернативны). Для принятия решений в программах просто необходимо использовать логические операторы.

Использование простых логических операторов

Программы на C++ должны обладать способностью принимать решения. Программа Convert не выполняла ничего, кроме простого преобразования значений температуры, и не принимала никаких решений, основанных на входных значениях. Для принятия таких решений в программах C++ используют логические операторы.

Простые логические операторы приведены в табл. 4.1. Они могут возвращать два значения: `true` (истина) и `false` (ложь).

Таблица 4.1. Простые операторы из повседневной логики

ОПЕРАТОР	ЗНАЧЕНИЕ
==	Равенство; истинно, когда значение левого аргумента совпадает со значением правого
!=	Неравенство; противоположно равенству
> , <	Больше, меньше; истинно, когда значение левого выражения больше (или меньше) значения правого
>= , <=	Больше или равно, меньше или равно; истинно, если истиной является > или == (соответственно < или ==)
&&	И; истинно, если аргументы и слева и справа являются истиной
||	ИЛИ; истинно, если или левый, или правый аргумент являются истиной
!	НЕ; истинно, если его аргумент принимает ложное значение

Первые шесть операторов табл. 4.1 являются операторами сравнения. Оператор равенства используется для проверки равенства двух значений. Например, следующее выражение истинно, если значением n является 0, и ложно во всех других случаях:

```
n == 0;
```

Не перепутайте оператор равенства == с оператором присвоения =. Эта ошибка очень распространена, к тому же компилятор C++, вообще говоря, не считает ее ошибкой, что делает ее вдвойне опасной!

```
n = 0; // Программист хотел написать, что n == 0
```

Широко распространены в повседневной жизни операторы "больше" (>) и "меньше" (<). Приведем пример логического выражения, возвращающего значение true.

```
int n1 = 1;
int n2 = 2;
n1 < n2
```

Операторы "больше" и "меньше" внешне очень похожи, и поэтому их легко перепутать. Чтобы этого не случилось, помните, что оператор-стрелочка принимает значение true в том случае, когда из двух сравниваемых значений он указывает на меньшее.

С помощью операторов > и < можно найти случаи, когда n1 больше или меньше n2, однако при этом игнорируется возможность равенства их значений. Операторы "больше или равно" (>=), "меньше или равно" (<=), в отличие от только что рассмотренных, учитывают и эту возможность.

Так же широко используются и операторы && (И) и || (ИЛИ). Эти операторы обычно сочетаются с другими логическими операторами:

```
// истинно, если n2 больше n1 и меньше n3
(n1 < n2) && (n2 < n3);
```

В качестве ремарки: оператор "больше или равно" можно определить как[10]

```
n1 <= n2 эквивалентно (n1 < n2) || (n1 == n2)
```

Логические операции и действительные переменные

Переменные с плавающей точкой, как уже отмечалось, не могут использоваться для перечисления. Вы можете сказать: первый, второй, третий, четвертый и т.д., так

[10] *В качестве еще одной ремарки: операторы сравнения вообще достаточно взаимозаменяемы. Так, например, (a==b) эквивалентно (! (a>b) &&! (a<b)).* — Прим. ред.

как соотношения между 1, 2, 3 абсолютно точно известны. Но нет никакого смысла говорить о номере 4.535887 в последовательности (такой способ нумерации возможен лишь как обозначение чего-то между четвертым и пятым, но не действительного значения номера, так как в любом сколь угодно малом отрезке их несчетное[11] множество).

Тип `float`, представляющий в C++ действительные числа, не является перечислимым. Кроме того, в отличие от действительных чисел, числа с плавающей точкой имеют ограниченное количество разрядов, поэтому при использовании операторов сравнения с числами с плавающей точкой необходимо соблюдать осторожность. Рассмотрим следующий пример:

```
float f1 = 10.0;
float f2 = f1 / 3;
f1 == (f2 * 3.0); // Равны ли эти значения?
```

Сравнивая начальное и полученное значения, мы не обязательно получим равенство. Действительные переменные, с которыми работает компьютер, не могут содержать бесконечного числа значимых разрядов. Поэтому `f2` равняется, например, 3.3333, а не $3\frac{1}{3}$. В отличие от математики, в компьютере число троек после точки ограничено. Умножив 3.3333 на 3, вы, вероятно, получите не 10.0, а 9.9999. Такой маленькой разницей может пренебречь человек, но не компьютер. Эта машина понимает под равенством исключительно точное равенство значений.

В современных процессорах выполнение таких вычислений очень усложнено. Процессор может даже устранить ошибку округления, но точно определить, когда именно процессору вздумается это сделать, язык C++ не способен.

Проблемы могут появиться и при совершенно простых вычислениях, например:

```
float f1 = 10.0;
float f2 = 120 % 11;
f1 == f2;    //истинно ли это выражение?
```

Теоретически f1 и f2 должны быть равны (об операции деления по модулю можно прочитать в главе 3, "Выполнение математических операций"). Ошибка округления возникнуть вроде бы не должна. Однако и в этом нельзя быть уверенным: вам ведь неизвестно, как именно представляются числа с плавающей точкой внутри компьютера.

Позвольте порекомендовать более безопасное сравнение:

```
float f1 = 10.0;
float f2 = f1 / 3;
float f3 = f2 * 3.0;
(f1 - f3) < 0.0001 && (f3 - f1) < 0.0001;
```

Оно ИСТИННО В ТОМ случае, если разница между f1 и f2 меньше какого-то малого значения (в нашем случае — 0.0001); при этом небольшие погрешности вычислений на правильность сравнения не повлияют.

Сокращенные вычисления в C++

Рассмотрим следующую конструкцию:

```
условие1 && условие2
```

Если условие1 ложно, то результат не будет истинным, независимо от истинности выражения условие2. В схеме

```
условие1 || условие2
```

в случае истинности выражения `условие1` неважно, какое значение принимает условие2, — результат будет истинным.

---

[11] *Более того, в данном случае это не красивое слово, а строгий математический термин.* — Прим. ред.

Для экономии времени C++ вычисляет первым условие]., и в случае, если оно ложно (для оператора &&) или истинно (для оператора I |), выражение условие2 не вычисляется и не анализируется.

Типы логических переменных

Поскольку > является оператором, то сравнение a > 10 представляет собой выражение. Очевидно, что результатом такого выражения может быть или true (истина), или false (ложь).

Вы уже могли заметить, что среди обсуждаемых в главе 2, "Премудрости объявления переменных", типов переменных не было логических (булевых) типов. Более того, ни в одном из существующих типов C++ нет значений true или false. Тогда к какому типу отнести выражение a > 10?

В C++ для хранения логических значений используется тип int. При этом 0 обозначает false, а любое другое отличное от нуля значение является истиной (true). Выражения типа a > 10 могут принимать значения 0 (false) или, например, 1 (true).

Microsoft Visual Basic для интерпретации значений false и true также использует целые числа, но в нем операторы сравнения возвращают или 0 (false), или -1 (true).

В новом стандарте ANSI C++ для работы с булевыми переменными определен тип bool, имеющий значения true и false.

'Бшифнме чисва C++

Переменные хранятся в компьютере в виде так называемых двоичных, или бинарных, чисел, т.е. представлены в виде последовательности битов, каждый из которых может содержать два значения: 0 или 1. Скорее всего, вам не придется оперировать с числами на битовом уровне, хотя существуют ситуации, когда обойтись без этого нельзя. C--+ снабжен несколькими операторами для подобных целей.

Вряд ли вам придется часто работать с переменными на битовом уровне, поэтому остальную часть главы следует рассматривать как техническое отступление от основного повествования.

Так называемые битовые логические операторы работают с аргументами на битовом уровне. Для того чтобы понять принципы их работы, давайте рассмотрим, как компьютер хранит переменные.

Десятичная система счисления

Числа, которыми мы чаще всего пользуемся, называются *десятичными*, или числами по основанию 10. В основном программисты на C++ тоже используют десятичные переменные. Например, мы говорим, что значение переменной var равно 123.

Число 123 можно представить в виде 1*100+2*10+3*1. При этом каждое из чисел 100, 10, 1 является степенью 10.

123 = 1 * 100 + 2 * 10 + 3 * 1,

что эквивалентно следующему:

123 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0

Помните, что любое число в нулевой степени равняется 1.

Другие системы счисления

Использование числа 10 в качестве основания нашей системы счисления объясняется, по всей вероятности, тем, что издревле для подсчетов человек использовал пальцы рук. Учитывая особенности нашей физиологии, удобной альтернативной системой счисления можно было бы выбрать двадцатеричную[12] (т.е. с основанием 20).

Если бы наша вычислительная система была заимствована у собак, то она бы была восьмеричной (еще один "разряд", находящийся на задней части каждой лапы, не учитывается). Эта система счисления работала бы не менее хорошо:

$123_{10} = 1 * 8^2 + 7 * 8^1 + 3 * 8^0 = 173_8$

Индексы 10 и 8 указывают систему счисления: 10 — десятичная. 8 — восьмеричная. Основанием системы счисления может быть любое положительное число.

Двоичная система счисления

У компьютеров, видимо, пальцев поменьше (может быть, поэтому они такие недалекие?). Они предпочитают пользоваться двоичной системой счисления. Число 123 переводится в двоичную систему таким образом:

$123_{10} = 0*128 + 1*64 + 1*32 + 1*16 + 1*8 + 0*4 + 1*2 + 1*1 = 01111011_2$

Существует соглашение, которое гласит, что в записи двоичных чисел используются 4, 8, 16 или 32 (и т.д.) двоичных цифр, даже если старшие цифры — нули. Внутреннее представление числа в компьютере строится именно таким образом.

Понятие разряда применяется к числам, кратным десяти, двоичный же разряд называется битом. Восемь битов составляют байт, а слово представляется или двумя, или четырьмя байтами.

Поскольку основа двоичной системы счисления очень мала, для представления чисел необходимо использовать слишком большое количество битов. Для представления таких обычных чисел, как 123_{10}, неудобно использовать выражения вида 01111011_2. Поэтому программисты предпочитают представлять числа блоками из четырех битов.

С помощью одного четырехбитового блока можно представить любое число от 0 до 15, и такая система счисления называется шестнадцатеричной (*hexadecimal*), т.е. системой по основанию 16. Часто употребляют ее сокращенное название *hex*.

В шестнадцатеричной системе обозначения цифр от 0 до 9 остаются теми же, а числа от 10 до 15 представляются с помощью первых шести букв алфавита: А вместо 10, в вместо 11 и т. д. Следовательно, 123_{10} — это $7B_{16}$.

$123 = 7 * 16^1 + B(\text{т.е. } 11) * 16^0 = 7B_{16}$

Поскольку программисты предпочитают представлять числа с помощью 4, 8, 16 или 32 битов, шестнадцатеричные числа состоят соответственно из 1, 2, 4 или 8 шестнадцатеричных разрядов (даже если ведущие разряды равны 0).

В заключение замечу, что, так как терминал не поддерживает нижний индекс, записывать шестнадцатеричные символы в виде $7B_{16}$ неудобно. Даже в том текстовом редакторе, который я использую сейчас, довольно неудобно всякий раз менять режимы шрифтов для ввода всего двух символов. Поэтому программисты договорились начинать шестнадцатеричные числа с 0x (это странное обозначение было придумано еще во время разработки языка С). Таким образом, $7B_{16}$ равно 0x7в. Следуя этому соглашению, 0x7B равно 123, тогда как 0x123 равно 291.

К шестнадцатеричным числам можно применять все те же математические операторы, что и к десятичным. Нам трудно выполнить в уме умножение чисел Схс*СхЕ потому, что таблица умножения, которую мы учили в школе, применима только к десятичной системе счисления.

---

[12] *Что и было сделано у некоторых народов, например у майя или чукчей.* — Прим. ред.

Выражения с римскими числами

Интересно, что некоторые системы чисел значительно препятствовали развитию математики, К таким относится и так называемая римская система.

Сложить два римских числа не очень сложно:

XIX + XXVI = XXV

Последовательность выполнения сложения такова:

а) `IX+VI: I` после V "уничтожает" I перед x, поэтому в результате получаем XV;

б) `X+XX=:XXX`, если добавить еще один x, получим xxxx, или XL.

Сложность вычитания римских чисел приблизительно такая же. Однако, чтобы умножить два римских числа, требуется диплом бакалавра математики (у вас волосы станут дыбом от правила, которое объясняет, как добавить к X разряд справа так, чтобы `X*IV` равнялось XL). А уж о делении римских чисел можно писать целые докторские диссертации...

Выполнение битовых логических операций

Все числа C++ могут быть представлены в двоичном виде, т.е. с использованием только 0 и I в записи числа. В табл. 4.2 указаны операции, которые работают с числами побитово; отсюда и происходит название термина "битовые операции".

Таблица 4.2. Битовые операции

ОПЕРАТОР	ФУНКЦИЯ
~	Каждый бит меняет свое значение на противоположное: 0 заменяется 1, 1 — нулем
&	Битовое И: поочередно выполняет операцию И с парами битов левого и правого аргумента
I	Битовое ИЛИ
^	Битовое исключающее ИЛИ

С помощью битовых операций можно сохранять большое количество информации в маленьком участке памяти. В мире существует множество вещей, которые имеют только два состояния (или, максимум, четыре). Вы или женаты (замужем), или нет (хотя можете быть в разводе или еще не женаты). Вы или мужчина, или женщина (по крайней мере, так сказано в моих водительских правах). В C++ каждую такую характеристику вы можете сохранить в одном бите. Таким образом, поскольку для хранения целого числа выделяется 4 байта, в тип `int` можно упаковать значения 32 разных свойств.

Кроме того, битовые операции выполняются крайне быстро. Хранение 32 характеристик в одном типе не приводит ни к каким дополнительным затратам.

Битовые операции с одним битом

Битовые операторы &, | и ~ выполняют логические операции над отдельными битами числа. Если рассматривать 0 как `false` и 1 как `true` (так принято, хотя можно ввести и другие обозначения), то для оператора ~ справедливо следующее:

```
~1(true)  равно 0(false)
~0 (true)  равно 1 (false)
```

Оператор & определяется так:

```
1(true)  & 1 (true)  равно 1(true)
1 (true)  & 0 (false) равно 0(false)
0(false) 5 0 (false) равно 0(false)
0 (false) & 1 (true)  равно 0(false)
```

Для оператора |:

```
1(true)   | 1 (true)   равно 1(true)
1 (true)  | 0 (false)  равно 1(true)
0 (false) | 0 (false)  равно 0(false)
0 (false) ! 1 (true)   равно 1(true)
```

Для последнего логического оператора, называемого "исключающим или" (XOR), прямой аналог в повседневной жизни найти труднее. Он возвращает значение true, если истинным является какой-то один (но не оба!) из его аргументов. Таблица истинности этого оператора представлена ниже (табл. 4.3).

Таблица 4.3. Таблица истинности оператора ^

^	1	0
1	0	1
0	1	0

Теперь, зная, как работают эти операторы с отдельными битами, рассмотрим их применение к двоичным числам.

Использование битовых операторов

Битовые операторы выполняются подобно любым другим арифметическим операторам. Самым легким для понимания является оператор ~. Выполнить операцию ~ над числом означает выполнить ее над каждым битом числа.

```
~0110₂  (0x6)
 1001₂  (Cx9)
```

Таким образом получаем, что ~0x6 равно 0x9.

В следующем примере продемонстрировано выполнение оператора &:

```
0110₂
&
OOHг
0010₂
```

Вычисляем, начиная со старших битов: 0 & 0 равно 0. В следующем бите 1 & 0 равно 0. В третьем бите 1 & 1 дает 1, а в последнем бите 0 & 1 дает 0.

Те же вычисления могут быть выполнены в шестнадцатеричной системе. Для этого нужно преобразовать числа в двоичное представление, выполнить операцию и преобразовать результат обратно.

```
0x06        0110₂
&           &
0x03        0011₂
0x02        0010₂
```

Расписав числа таким образом, мы получили, что Oxб & 0x3 равно 0x2.

(Попробуйте подсчитать значение выражения 0x6 | 0x3. Если вам это удастся, вы почувствуете себя на седьмом небе. Иначе очутитесь на первой из семи ступенек в преисподнюю. У меня на это ушло чуть меньше восьми минут.)

Простой пример

Следующая программа иллюстрирует работу побитовых операторов. В ней инициируются две переменные, к которым применяются операции &, |, ^. Результаты вычислений выводятся на экран.

```
// BitTest — инициируются две переменные и
//           выводятся результаты выполнения
//           операторов ~,&,| и ^
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // установка вывода в шестнадцатеричном виде
    cout.setf(ios::hex, ios::hex);
    // инициализация двух аргументов
    int nArg1 = 0x1234;
    int nArg2 = 0x00ff;

    // Выполнение логических операций
    // Сначала применяем унарный оператор NOT
    cout << "Arg1          = 0x" << nArg1 << '•\n";
    cout << "Arg2          = 0x" << nArg2 << "'\n";
    cout << "~nArg1        = 0x" << ~nArg1 << "\n";
    cout «  "~nArg2        = 0x" << ~nArg2 « "\n";

    // Теперь - бинарные операторы
    cout << "nArg1 & nArg2 = 0x"
         << (nArg1 s nArg2)
         << "\n";
    cout «  "nArg1 | nArg2 = 0x"
         «  (nArg1 | nArg2)
         << "\n";
    cout << "nArg1 ^ nArg2 = 0x"
         << (nArg1 ^ nArg2)
         << "

    return(0);
}
```

Первая инструкция нашей программы (та, которая следует за ключевым словом main) — `cout.setf(ios::hex) ;` — меняет используемый по умолчанию десятичный формат вывода на шестнадцатеричный (поверьте мне, это сработает).

В остальном программа очевидна. Присвоив значения аргументам nArg1, nArg2, она выводит все варианты побитовых вычислений над этими переменными.

Результат выполнения программы будет выглядеть следующим образом:

```
Arg1          = 0x1234
Arg2          = Oxff
~nArg1        = 0xffffedcb
~nArg2        = OxffffffOO
nArg1 S nArg2 = 0x34
nArg1 | nArg2 = 0x12ff
nArg1 ^ nArg2 = 0x12cb
```

К шестнадцатеричным числам всегда приписывается префикс Ox.

Зачем придуманы эти глупые операторы

Предназначение большинства операторов очевидно. В необходимости операторов - или + сомневаться не приходится. Не нужно рассказывать, для чего используется оператор | |. Но для начинающих программистов может быть далеко не очевидно, зачем использовать битовые операторы.

Оператор & часто используется для маскирования информации. Например, предположим, что нам нужно выделить последний значимый шестнадцатеричный разряд из четырехразрядного числа:

```
0x1234          0001  0010  OOH  0100
&               &
OxOOOF          0000  0000  0000  1111
0x0004          0000  0000  0000  0100
```

С помощью этого оператора можно также выделять и устанавливать значения отдельных битов.

Представьте себе, что в написанной вами базе данных для сохранения некоторой информации о личности используется единственный бит. Первый значимый бит равен 1. если это особа мужского пола, второй бит равен 1, если это программист, третий равен 1 в случае внешней привлекательности, а четвертый — если человек имеет собаку. Взгляните на табл. 4.4.

Таблица4.4.Значениябитов

Бит	ЗНАЧЕНИЕ
0	1 → мужчина
1	1 → программист
2	1 → привлекательный
3	1 → владелец собаки

Не особо привлекательный, имеющий собаку программист мужского рода будет закодирован числом 1101_2. Если вы хотите просмотреть все записи в поисках хорошенькой девушки без собаки, причем неважно, разбирается ли она в C++, надо использовать следующее сравнение (числа приведены в двоичной записи!):

```
(databaseValue & 1011) == 0010
                          ^    - не мужчина
                            ^  - привлекательна
                             ^ - без собаки
                  ^ ^^- важно
                    *   - не интересует
```

В этом случае число 1011 является маской, потому что оно маскирует, исключая из рассмотрения, биты с не интересующими вас характеристиками.

Глава 5

Операторы управления программой

В этой главе...

- ✓ Управление ходом программы с помощью команд ветвления
- ✓ Выполнение циклов
- ✓ Вложенные команды управления
- ✓ Инструкция выбора

Простые программки, которые появлялись в первых четырех главах, обрабатывали фиксированное количество вводов и выводов результатов вычислений, после чего завершали работу. Эти приложения были лишены какого бы то ни было контроля над работой программы, в частности не выполняли никаких проверок Но компьютерные программы могут принимать решения. Вспомните: когда пользователь нажимает клавиши, компьютер реагирует на это выполнением соответствующих команд.

Например, если пользователь нажимает <Ctrl+C>, компьютер копирует содержимое выделенного блока в буфер обмена. Если пользователь перемещает мышь, на экране двигается ее курсор. Этот список можно продолжать до бесконечности. Программы же, которые не принимают никаких решений, неизбежно скучны и однообразны.

Команды, управляющие ходом программы, указывают на то, какие действия она должна выполнить в зависимости от результата вычисления какого-либо логического выражения (о которых шла речь в главе 4, "Выполнение логических операций"). Существует три типа управляющих инструкций: операторы ветвления (или условного перехода), цикла и выбора.

Управление ходом программы с помощью команд ветвления

Проще всего управлять ходом программы с помощью инструкции ветвления, которая позволяет программе, в зависимости от результата логического выражения, решить, по какому из двух возможных путей выполнения инструкций следует двигаться дальше.

В C++ оператор условного перехода реализуется с помощью инструкции if:

```
if (m > n)
{
//1-я последовательность операторов
//инструкции, которые должны быть выполнены,
//если m больше n
}
else
{
```

```
//2-я последовательность операторов
//инструкции, которые нужно выполнить
//в противном случае
}
```

Прежде всего вычисляется логическое выражение m > n. Если его значение — true, программа выполняет первую последовательность операторов. Если же выражение ложно, управление передается второй последовательности. Оператор else не обязателен: если он опущен, C++ считает, что он существует, но является пустым.

Если в текущей ветви оператора if имеется только одна инструкция, скобки использовать необязательно. Однако очень легко сделать ошибку, которую без скобок, определяющих структуру операторов, компилятор C++ обнаружить не сможет. Поэтому намного безопаснее включать скобки всегда. Если друзья будут уговаривать вас не использовать скобки, не поддавайтесь!

Работу оператора if можно рассмотреть на следующем примере:

```
// BranchDemo — введите два числа. Если первый
// аргумент больше, выполняем операторы первой
// ветви, если меньше — второй

#include <stdio.h>
#include <iostream.h>

int main(int argc, char* pszArgs[])
{
    // вводим первый аргумент...
    int arg1;
    cout << "Введите arg1: ";
    cin  >> arg1;

    // ...второй
    int arg2;
    cout << "Введите arg2: ";
    cin  >> arg2;

    // теперь решаем, что делать:
    if (arg1 > arg2)
    {
        cout << "аргумент 1 больше, чем аргумент 2\n";
    }
    else
    {
        cout << " аргумент I не больше, чем аргумент 2\n";
    }

    return 0;
}
```

Программа считывает два целых числа, вводимых с клавиатуры, и сравнивает их. Если выражение "arg1 больше arg2" истинно, то выполняется инструкция cout « "аргумент 1 больше, чем аргумент 2\n";. Если же нет, то управление переходит к последовательности операторов, соответствующей условию else: cout << "аргумент 1 не больше, чем аргумент 2\г.";.

Выполнение циклов

Оператор перехода позволяет управлять работой программы, когда существуют альтернативные пути ее выполнения. Это усовершенствование языка хотя и весьма значительное, но все же не достаточное для написания полнофункциональных программ.

Рассмотрим проблему обновления экрана компьютера. При перерисовывании содержимого типичного дисплея компьютеру необходимо выводить на экран тысячи пикселей. Если программа не умеет повторно выполнять один и тот же фрагмент кода, вы будете вынуждены тысячи раз записывать одно и то же множество инструкций.

Для решения этой проблемы необходим способ, который позволит многократно выполнять одни и те же последовательности инструкций. Операторы цикла предоставляют возможность решить эту задачу.

Цикл while

Самый простой цикл можно организовать с помощью оператора while. Он выглядит таким образом:

```
while (условие)
{
//Этот код выполняется повторно,
//пока условие остается истинно
}
```

Сначала проверяется условие. Условием могут быть выражения вида var > 10, var1 == var2 или любые другие. Если условие истинно, выполняются инструкции в скобках. Дойдя до закрывающей скобки, компилятор передает управление в начало цикла, и все повторяется вновь. Таким образом, смысл оператора while в том, что программный код в скобках повторно выполняется до тех пор, пока не нарушится условие (этот процесс напоминает мне утренние прогулки с собакой вокруг дома, пока она не... ну а потом мы возвращаемся).

Если условие сначала было справедливо, тогда что может заставить его стать ложным? Рассмотрим следующий пример программы:

```
// WhileDemo — введите счетчик цикла.
// Программа выводит количество выполненных
// циклов while
#include <stdio.h>
#include: < lostream.h>

int main(int arg, char* pszArgs[])
{
    // ввод счетчика цикла
    int loopCount;
    cout << "Введите loopCount: ";
    cin >> loopCount;

    // теперь в цикле выводим значения
    while (loopCount > 0)
    {
        loopCount = loopCount – 1;
        cout << "Осталось выполнить "
             << loopCount << " циклов\n";
```

```
    }
    return 0;
}
```

Программа WhileDemo получает от пользователя значение счетчика цикла, которое сохраняется в переменной loopCount. Затем программа выполняет цикл while. Сначала проверяется значение переменной loopCount. Если оно больше нуля, программа входит в тело цикла (телом цикла называется код между скобками), где loopCount уменьшается на 1, и результат выводится на экран. Затем программа возвращается к началу цикла и проверяет, осталась ли переменная loopCount положительной.

Ниже представлены результаты выполнения программы, выведенные на экран. Нетрудно догадаться, что введенный мною счетчик цикла равен 5. Программа пять раз выполнила цикл, каждый раз выводя результат на экран:

```
Осталось выполнить 4 циклов
Осталось выполнить 3 циклов
Осталось выполнить 2 циклов
Осталось выполнить 1 циклов
Осталось выполнить 0 циклов
```

Если пользователь введет отрицательное значение для счетчика цикла, условие окажется ложным и тело цикла не будет выполнено ни разу. Если пользователь введет очень большое число, на выполнение программы уйдет очень много времени.

Реже используется другая версия цикла while, известная как do. . .while. Она работает аналогично, но условие завершения проверяется в конце, после выполнения тела цикла.

```
do
{
    //Тело цикла
} while (условие);
```

Поскольку условие проверяется только в конце, тело оператора *do. . .while* выполняется всегда хотя бы один раз.

Условие завершения цикла проверяется только в начале оператора while или в конце оператора do. . .while. Даже если в какой-то момент оно перестанет быть справедливым, программа продолжит выполнение цикла до следующей проверки условия.

Использование операторов инкремента и декремента

Очень часто для какого-либо подсчета в циклах программисты используют операторы инкремента или декремента. Заметим, что в следующем фрагменте программы WhileDemo для уменьшения значения счетчика используются операторы присвоения и вычитания:

```
// в цикле вызолим значения
    while (loopCount > 0)
    {
        loopCount = loopCount — 1;
        cout « "Осталось выполнить "
             << loopCount << " циклов\n";
    }
```

Используя оператор декремента, этот цикл можно записать более компактно:

```
// в цикле выводим значения
    while (loopCount > 0)
    {
        loopCount--;
        cout << "Осталось выполнить "
             « loopCount << " циклов\п";
    }
```

Смысл этого варианта цикла полностью совпадает со смыслом оригинала. Единственная разница — в способе записи.

Поскольку оператор декремента одновременно уменьшает аргумент и возвращает его значение, он может включаться в условие цикла while. В частности, допустима следующая версия цикла:

```
// в цикле выводим значения
    while (loopCount-- > 0)
    (
        cout << "Осталось выполнить "
             << loopCount << " циклов\п";
    }
```

Хотите — верьте, хотите — нет, но большинство программистов на C++ используют именно этот вариант записи. И не потому, что им нравится быть остроумными; хотя почему бы и нет? Использование в логических сравнениях операторов инкремента и декремента делает программный код легко читаемым и более компактным. И вряд ли вы, исходя из своего опыта, сможете предложить достойную альтернативу.

И в выражении loopCount--, и в --loopCount значение loopCount уменьшается; однако первое возвращает значение переменной loopCount перед его уменьшением на 1, а второе — после.

Сколько раз будет выполняться декрементированный вариант `WhileDemo`, если пользователь введет число 1? Если использовать префиксный вариант, то значение --loopCount равно 0 и тело цикла никогда не выполнится. В постфиксном варианте loopCount-- возвратит 1 и программа передаст управление в начало цикла.

У вас может сложиться ошибочное мнение, что программа, в которой используются операторы декремента, выполняется быстрее, так как содержит меньше инструкций. Однако это не совсем так. Время выполнения программы не зависит от того, какую из представленных выше операций декремента вы используете, так как современные оптимизирующие компиляторы используют минимально необходимое количество инструкций машинного языка, независимо от используемых для декремента операторов.

Использование цикла for

Другой разновидностью циклов является цикл for. Его часто предпочитают более простому циклу while. Цикл for имеет следующий вид:

```
for (инициализация; условие; увеличение)
{
    //...тело цикла
}
```

Выполнение цикла for начинается с инициализации. В ней обычно находится оператор присвоения, используемый для установки начального значения перемен-

ной цикла. Условие инициализации выполняется только один раз, при первом входе в цикл for.

Затем проверяется условие. Подобно циклу while, цикл for выполняется до тех пор, пока условие не станет ложным.

После того как выполнится код тела цикла, управление получит следующий параметр цикла for (увеличение) и значение счетчика изменится. Затем опять будет выполнена проверка условия, и процесс повторится. В этом параметре обычно записывают инкрементное или декрементное выражение, которое определяет характер изменения переменной цикла на каждой итерации, но в принципе ограничений на используемые здесь операторы нет.

Цикл for можно заменить эквивалентным ему циклом while:

```
инициализация;
while (условие)
{
    {
        // ... тело цикла
    }
    увеличение;
}
```

Все три параметра цикла for являются необязательными. C++ игнорирует отсутствие части инициализации или увеличения цикла, а если опущено условие, C++ будет выполнять цикл for вечно (или пока какой-либо другой оператор не передаст управление за пределы цикла).

Для лучшего понимания цикла for рассмотрим пример. Приведенная ниже программа ForDemo выполняет то же, что и whileDemo, но вместо while использует цикл for.

```
// ForDemo. Вводится счетчик цикла.
// На экран выводится количество выполненных
// циклов for
#include <stdio.h>
#include <iostream.h>

int main(int arg, char* pszArgs[])
{

    // ввод счетчика цикла
    int loopCount;
    cout « "Введите loopCount: ";
    cin >> loopCount;

    // работаем, пока не нарушится условие
    for (int i = loopCount; i > 0; i--)
    {
        cout << "Осталось выполнить " << i-1 << " циклов\n";
    }
    return 0;
}
```

Программа ForDemo выполняет те же действия, что и ранее рассмотренная WhileDemo. Однако вместо изменения переменной loopCount в этом варианте программы введена специальная переменная цикла.

Выполнение цикла начинается с объявления этой переменной i и инициализации ее значением переменной loopCount. Затем проверяется, является ли переменная i положительной. Если да, то программа выводит уменьшенное на 1 значение i и возвращается к началу цикла.

Цикл for также удобен и тогда, когда значение переменной цикла необходимо увеличивать, а не уменьшать. Это осуществляется небольшим изменением цикла for.

```
// ForDemo. Вводится счетчик цикла.
//На экран выводится количество выполненных
// циклов for
#include <stdio.h>
#include <iostream.h>

int main(int arg, char* pszArgs[])
{
     // ввод счетчика цикла
    int loopCount;
    cout << "Введите loopCount: ";
    cin  » loopCount;

    // вычисляем, пока не нарушится условие
        for (int i = 1; i <= loopCount; i++)
    {
        cout << "Мы завершили " << i « " цикл\п";
    }
    return 0;
}
```

Согласно последнему стандарту языка индексная переменная, объявленная в части инициализации цикла for, известна только в пределах этого цикла. Программисты на C++ в этом случае говорят, что область видимости переменной ограничена циклом for. Например, в инструкции return рассмотренного выше примера, т.е. за пределами цикла, переменная i недоступна и не может использоваться. Однако этого новейшего правила придерживаются далеко не все компиляторы, и вам нужно протестировать свой компилятор, чтобы узнать, как он действует в этом случае.

Вы можете задать вопрос: если цикл for эквивалентен циклу while, зачем забивать себе им голову? Цикл for намного легче для понимания, поскольку главные части любого цикла for (инициализация, условие, увеличение) имеют фиксированное местоположение и соответствуют стандартному формату записи.

Избегайте бесконечных циклов

Бесконечным называют такой цикл, который выполняется вечно. Бесконечный цикл создается каждый раз, когда нельзя нарушить условие выполнения цикла, обычно вследствие какой-то ошибки в коде.

Рассмотрим следующую разновидность рассмотренного ранее цикла:

```
while (loopCount > 0)
f
    cout << "Осталось выполнить" <<loopCount << " циклов\п"
}
```

Программист забыл уменьшить переменную loopCount, как это делалось в ранее рассмотренных циклах. В результате получен счетчик цикла, значение которого никогда не изменяется. При этом условие выполнения цикла будет либо всегда истинно, либо всегда ложно. Выполнение такой программы никогда не закончится естественным путем, т.е., как говорят, программа зацикливается.

Но... ничто не вечно под Луной! В конечном счете правительство сменится, компьютер поломается, Microsoft обанкротится... Или цикл прервется, и вам не о чем будет тревожиться.

Существует гораздо больше способов создания бесконечных циклов, чем показано здесь; но обычно они слишком сложны для того, чтобы приводить их в такой простой книжке.

Специальные операторы циклов

В C++ определены две специальные управляющие программой команды — break и continue. Может случиться, что условие работы цикла нарушится не в начале или в конце, а где-то посередине цикла. Рассмотрим программу, которая суммирует введенные пользователем значения. Цикл прерывается, когда пользователь вводит отрицательное число.

Проблема в том, что программа не сможет выйти из цикла, пока пользователь не введет число, но должна сделать это перед тем, как значение будет добавлено к сумме.

В таких случаях используют определенную в C++ команду break. Она немедленно передает управление в конец текущего цикла. После инструкции break программа будет выполнять инструкцию, следующую сразу же после закрывающей скобки цикла.

Схематически работу команды break можно проиллюстрировать так:

```
while(условие)
{
    if (какое-то другое условие)
    {
        break; //выход из цикла
    }
         //когда программа встретит break,
         //управление будет передано этой строке
```

Вооружась **новой командой** break, **я реализовал в программе** BreakDemo **решение проблемы последовательного накопления суммы чисел.**

```
// BreakDemo – вводим множество чисел.
//             Суммируем эти числа,
//             пока пользователь не введет
//             отрицательное число
#include <stdio.h>
#include <iostream.h>

int main(int arg, char* pszArgs[])
{
    // введите счетчик цикла
    int accumulator = 0;
    cout << "Эта программа суммирует числа, "
         << "введенные пользователем\n";
    cout << "Выполнение цикла "
         << "заканчивается после "
         << "введения отрицательного числа\n";

    // бесконечный цикл
    for(;;)
    {
        // ввод следующего числа
        int value = 0;
        cout << "Введите следующее число: ";
        cin  >> value;
```

```
        // если оно отрицательно...
        if (value < 0)
        {
            // ...тогда ВЫХОДИМ из цикла
            break;
        }

        // ...иначе добавляем число к общей сумме
        accumulator = accumulator + value;
    }

    // после выхода из цикла
    // выводим результат суммирования
    cout << "\nОбщая сумма равна "
         << accumulator
         << "\n";

    return 0;
}
```

После объяснения правил пользователю (что нужно ввести, чтобы выйти из цикла, и что делает эта программа) управление получает нечто подобное бесконечному циклу for. Сначала на экран выводится запрос на введение числа с клавиатуры. Проверить его на соответствие критерию выхода из цикла можно только после того, как оно считано программой. Если введенное число отрицательно, управление переходит к оператору break и программа выходит из цикла. Если же число оказывается положительным, программа пропускает команду break и к накопленной сумме добавляет новое значение. После выхода из цикла программа выводит значение общей суммы всех введенных чисел и завершает работу.

При повторяющемся выполнении операций над переменной в цикле проследите, чтобы инициализация переменной осуществлялась еще до входа в цикл. В нашем случае программа обнулила переменную accumulator перед входом в цикл, в котором к ней добавляются новые числа.

Несколько реже используется команда continue. Столкнувшись с ней, программа немедленно возвращается к началу цикла. Остальные инструкции цикла игнорируются. В следующем фрагменте программы отрицательные числа, которые может ввести пользователь, игнорируются:

```
while (1)
{
    //Ввсд значения
    cout << "Введите значение";
    cin  >> inputVal;

    //если число отрицательное
    if (inputVal < 0)
    {
        //выводим сообщение об ошибке
        cout << "Не допускается ввод "
             << "отрицательных чисел\n";
        //возвращаемся к началу цикла
        continue;
    }
    //введено приемлемое значение
}
```

Вложенные команды управления

Возвратимся к проблеме перерисовывания экрана монитора. Конечно, некоторые типы структур циклов можно использовать для прорисовки каждого пикселя линии слева направо (или для евреев и арабов сканирование экрана осуществляется справа налево?). Но как реализовать повторяющееся перерисовывание каждой сканируемой линии сверху вниз? (Для австралийских экранов, наверное, снизу вверх?) Для этого нужно включить цикл, сканирующий пиксели слева направо, внутрь цикла, сканирующего линии сверху вниз.

В случае если команды одного цикла находятся внутри другого, цикл называют *вложенным*. В качестве примера вы можете модифицировать программу BreakDemo в программу, которая накапливает сумму чисел любого количества последовательностей. В этой программе, названной NestedDemo, во внутреннем цикле суммируются числа, введенные с клавиатуры, пока пользователь не введет отрицательное значение. Внешний цикл суммирует полученные суммы значений последовательностей до тех пор, пока накопленная сумма примет отличное от нуля значение.

```
// NestedDemo — вводится последовательность чисел.
//              Числа суммируются, пока пользователь
//              не введет отрицательное число.
//              Этот процесс будет
//              пока общая сумма не станет равной 0.
         <stdio.h>
#include <iostream.h>

int main(ir.t arg, char* pszArgs[])
{
    cout << "Эта программа суммирует "
         << "последовательности чисел.\п Ввод каждой "
         << "последовательности завершается "
         << "вводом отрицательного числа \п"
         << "Чтобы завершить ввод последовательностей, "
         << "нужно              отрицательных числа подряд\n";
    // внешний цикл работает с последовательностями чисел
    int
    do
    {
        // начинаем ввод очередной последовательности чисел
        accumulator = 0;
        cout << "\nВведите очередную последовательность\n";

        // бесконечный цикл
        for(;;)
        {
            // введение очередного числа
            int value = 0;
            cout << "Введите очередное число: ";
            cin  >>

            // если оно отрицательное...
            if (value < 0}
            {
                // ... выходим из цикла
                break;
            }
            // ...иначе добавляем число к
```

```
            // общей сумме
            accumulator = accumulator + value;
        }

        // Вывод результата вычислений...
        cout << "\nОбщая сумма равна "
            << accumulator
            << "\n";

        // ...если накопленная общая сумма чисел
        // последовательности не равна нулю,
        // начинаем работать со следующей
        // последовательностью
    } while (accumulator != 0);
    cout << "Программа завершена\n";
    return 0;
}
```

Инструкция выбора

Последняя управляющая инструкция эффективна, если существует необходимость выбора при ограниченном количестве возможных вариантов. Она похожа на усложненную инструкцию if, которая вместо проверки одного условия анализирует множество разных возможностей:

```
switch (выражение)
{
    case c1:
        //переходим сюда, если выражение == c1
        break;
    case c2:
        //переходим сюда, если выражение == c2
        break;
    default:
        //если ни одно условие не выполнено,
        // переходим сюда
}
```

Значением выражения должно быть целое число (int, long или char); c1, c2, c3 должны быть константами. Инструкция switch выполняется следующим образом: сначала вычисляется значение выражения, а затем оно сравнивается с константами, указанными после служебного слова case. Если константа соответствует значению выражения, то программа передает управление этой ветви. Если ни один вариант не подходит, выполняется условие default.

Рассмотрим для примера следующий фрагмент программы:

```
cout << "Введите 1, 2 или 3:";
cin >> choice;

switch (choice)
{
    case 1:
        //обработка случая "1"
        break;
    case 2:
        //обработка случая "2"
        break;
```

```
    case 3:
        //' обработка случая "3"
        break;
    default:
        cout << "Вы ввели не 1, не 2 и не 3\n"
}
```

Еще раз напомню, что инструкция switch эквивалентна усложненной инструкции if (с if-инструкциями); однако, если рассматривается более случаев, структура switch оказывается

Для выхода из инструкции switch необходимо использовать команды break, иначе управление будет переходить от одного случая к следующему.

Часть II

Становимся программистами

... и говорю: "Официант! Официант! У меня в супе — жучок!'. А ОН мне отвечает: "Извините, сэр, НО НАШ шеф-повар занялся программированием..."

В этой части...

Выполнять операции сложения или умножения (и даже логические операции) — это одно; а писать настоящие программы — это нечто совсем иное. В этой части рассматривается все необходимое для того, чтобы стать настоящим программистом.

Глава 6

Создание функций

В этой главе...

- ✓ Написание и использование функций
- ✓ Подробный анализ функций
- ✓ Перегрузка функций
- ✓ Определение прототипов функций
- ✓ Хранение переменных в памяти

Очень часто при написании программ возникает необходимость разделить большую программу на меньшие части, отлаживать которые намного легче. Программы из предыдущих глав слишком малы, чтобы можно было по-настоящему оценить пользу такого разделения. Но реальные программы из больших проектов состоят из тысяч (и даже миллионов!) строк. Поэтому большие программы просто невозможно написать, не разбивая их на отдельные модули.

C++ позволяет разделить код программ на части, называемые функциями. Сами функции могут быть записаны и отлажены отдельно от остального кода программы.

Хорошая функция может быть описана одним предложением с минимальным количеством слов "и" и "или". Например, функция `sumSequence()` суммирует последовательность целочисленных значений, введенных пользователем. Это определение весьма компактно и легко воспринимается.

Возможность разбивать программу на части с последующей отладкой каждой функции в отдельности существенно снижает сложность создания больших программ. Этот подход является, по сути, простейшей формой инкапсуляции (смотрите главу 12, "Знакомство с объектно-ориентированным программированием", где вопросы инкапсуляции рассматриваются подробнее.)

Написание и использование функций

Функции лучше всего изучать на примерах. Эта часть начинается с программы `FunctionDemo`, которая показывает, как упростить рассмотренную в главе 5 программу `NestDemo`, определив дополнительную функцию. На примере программы `FunctionDemo` я постараюсь объяснить, как определять и использовать функции. Эта программа будет служить образцом для их дальнейшего изучения.

`NestDemo` содержит два цикла. Во внутреннем цикле суммируется последовательность введенных пользователем чисел. Он включен во внешний цикл, который повторяет процесс, пока пользователь не изъявит желания его прекратить. Разделение этих двух циклов делает программу более наглядной.

В программе `FunctionDemo` показано, как упростить программу `NestDemo` с помощью создания функции sumSequence {] .

Согласно синтаксису C++ справа от имени функции должны присутствовать две круглые скобки. В них обычно указываются параметры функций.

```
// FunctionDemo — демонстрация использования
//                функций. Внутренний цикл
//                программы оформлен как
//                отдельная функция

#include <stdio.h>
#include <iostream.h>

// sumSequence — суммирует последовательность
//               чисел, введенных с клавиатуры, пока
//               пользователь не введет отрицательное
//               число. Возвращает сумму введенных чисел
int sumSequence(void)
{
    // бесконечный цикл
    int accumulator = 0;
    for (;;)
    {
        // ввод следующего числа
        int value = 0;
        cout << "Введите следующее число: ";
        cin  >> value;

        // если оно отрицательное...
        if (value < 0)
        {
            // ...тогда выходим из цикла
            break;
        }

        // ...иначе добавляем число к
        // переменной accumulator
        accumulator = accumulator + value;
    }

    // возвращаем значение суммы
    return accumulator;
}

int main(int arg, char* pszArgs[])
f
    cout << "Эта программа суммирует последовательности\n"
         << "чисел. Каждая последовательность\n"
         << "заканчивается отрицательным числом.\n"
         << "Ввод серий завершается вводом \n"
         << "двух отрицательных чисел подряд \n";

    // суммируем последовательности чисел...
    int accumulatedValue;
    do
    {
        // суммируем последовательности чисел,
        // введенных с клавиатуры
        cout << "\nВведите следующую последовательность\n";
```

```
        accumulatedValue = sumSequence();

        // вывод общей суммы на экран
        cout << "\nОбщая сумма равна "
             << accumulatedValue
             << "\n";

    // ...пока возвращаемая сумма не равна 0
    } while (accumulatedValue != 0);
    cout << "Программа завершена\n";
    return 0;
}
```

Вызов функции sumSequence()

Главная часть программы сконцентрирована в фигурных скобках, следующих после объявления функции main(). Эта часть кода очень напоминает программу NestDemo.

Различие состоит в том, что внутри функции main(> содержится выражение accumulatedValue = sumSequence (). В правой части ЭТОГО выражения вызывается функция sumSequence (). Возвращаемое функцией значение сохраняется в переменной accumulatedValue, а затем выводится на экран. Главная программа выполняет цикл до тех пор, пока значение суммы, возвращаемой внутренней функцией, остается отличным от 0. Нулевое значение говорит о том, что пользователь закончил вычисление сумм последовательностей.

Вызвать функцию — это значит начать выполнение кода, который содержится в ней. После завершения этого процесса программа передаст управление инструкции, следующей непосредственно за вызовом функции.

Определение функции sumSequence()

Определение этой функции начинается с инструкции int sumSequence (void). Заключенный в фигурные скобки блок кода называется *телом функции*. Как видите, тело функции sumSequence () идентично внутреннему циклу программы NestDemo.

Работа программы осуществляется следующим образом: функция main выполняет цикл, внутри которого (там, где в программе NestDemo находился внутренний цикл) происходит вызов функции sumSequence (). Она вычисляет сумму, а результат возвращает в функцию main программы, после чего продолжается выполнение внешнего цикла.

Подробный анализ функций

Функции являются первоосновой программ C++. Поэтому каждый программист должен отчетливо понимать все нюансы их определения, написания и отладки.

Функцией называют логически обособленный блок кода C++, имеющий следующий вид:

```
<тип возвращаемого значения>                    функцию)
{

    return<выражение>;
}
```

Аргументами функции называются значения, которые можно передать ей при вызове. В *возвращаемом значении* указывается результат, который функция возвращает по окончании работы. Например, в вызове функции возведения в квадрат square (10) 10 является аргументом, а возвращаемое значение равно 100.

И аргументы, и возвращаемое значение функции необязательны. Если что-то отсутствует, вместо него используется ключевое слово void. Значит, если вместо списка аргументов используется void, то при вызове функция не получает никаких аргументов (как и в рассмотренной программе FunctionDemo). Если же возвращаемый тип функции — void, то вызывающая программа не получает от функции никакого значения.

В программе FunctionDemo используется функция с именем sumSequence (), которая не имеет аргументов и возвращает значение целочисленного типа.

Тип аргументов функции по умолчанию — void, поэтому функцию `int fn(void)` можно записать как `int fn()`.

Использование функции позволяет работать с каждой из двух частей программы FunctionDemo в отдельности. При написании функции sumSequence () я концентрирую внимание на вычислении суммы чисел и не думаю об остальном коде, вызывающем эту функцию.

При написании функции main () я работаю с возвращаемой функцией sumSequence () — суммой последовательности (на этом уровне абстракции я знаю только, что выполняет функция, а не как именно она это делает).

Простые функции

Функция sumSequence () возвращает целое значение. Функции могут возвращать значение любого стандартного типа, например double или char (типы переменных рассматриваются в главе 5, "Операторы управления программой").

Если функция ничего не возвращает, то возвращаемый тип помечается как void.

Функции различаются по типу возвращаемого значения. Так, *целочисленной функцией* называют ту, которая возвращает целое значение. Функция, которая ничего не возвращает, известна как *void-функция*. Далее приведен пример функции, выполняющей некоторые действия, но не возвращающей никаких значений.

```
void echoSquare(>
{
    cout << "Введите значение:";
    cin >> value;
    cout << "\n"«value*value « "\n";
    return;
)
```

Сначала управление передается первой инструкции после открывающей скобки, затем поочередно выполняются все команды до инструкции `return` (которая в данном случае не требует аргумента). Эта инструкция передает управление вызывающей функции.

Инструкция `return` в void-функциях является необязательной. Если она отсутствует, то выполнение функции прекращается при достижении закрывающей фигурной скобки.

Функции с аргументами

Функции без аргументов используются редко, так как связь с такими функциями односторонняя, т.е. осуществляется только посредством возвращаемых значений. Аргументы функций позволяют установить двустороннюю связь — через передаваемые параметры и возвращаемые значения.

Функции с одним аргументом

Аргументами функции называют значения, которые передаются функции во время вызова. В следующем примере определяется и используется функция square{>, которая возвращает квадрат переданного в качестве аргумента числа типа double:

```
//             — демонстрирует использование
// функции, обрабатывающей аргументы

#include <stdio.h>
#include <iostream.h>

// square — возвращает квадрат аргумента
//          doubleVar — введенное значение
//          returns — квадрат doubleVar
double square(double doubleVar)
{
    return doubleVar * doubleVar;
}

// sumSequence — суммирует последовательность
// чисел, зведенных с клавиатуры и возведенных в
// квадрат, пока пользователь не введет
//отрицательное число.
// Возвращает сумму квадратов введенных чисел

int sumSequence(void)
{
    // бесконечный цикл
    int accumulator= 0;
    for(;;)
    {
        // введение следующего числа
        double dValue = 0;
        cout << "Введите следующее число: ";
        cin  >> dValue;

        // если оно отрицательное...

        if (dValue < 0)
        {
            // . . . то выходим из цикла
            break;
        }

        // ...иначе вычисляем квадрат числа
        int value = (int)square(dValue);

        // теперь добавляем квадрат к
        // accumulator
        accumulator= accumulator + value;
    }

    // возвращаем накопленное число
    return accumulator;
}

int main(int arg, char* pszArgs[])
{
```

```
    cout « "Эта программа суммирует\n"
         << "множественные серии чисел. \n"
         << "Введение каждой последовательности\n"
         << "заканчивается введением\n"
         << "отрицательного числа.\n"
         « "Серии последовательностей вводятся\n"
         << "до тех пор, пока не встретятся\n"
         << "два отрицательных

    // Продолжаем суммировать числа...
    int
    do
    {
         // суммируем последовательность чисел,
         // введенных с клавиатуры
         cout << "\nВведите следующую последовательность\n";
         accumulatedValue = sumSequence();

         // теперь выводим результат суммирования
         cout <<          сумма равна "
              << accumulatedValue
              « "\n";

    // ...пока возвращаемая сумма не равна 0
    } while (accumulatedValue != 0);
    cout << "Программа завершена\n";
    return 0;
}
```

По сути, перед вами все та же программа FunctionDemo, но теперь она суммирует квадраты введенных чисел. В функции square() играющее роль параметра число возводится в квадрат. незначительные изменения и в функции sumSequence (): если раньше мы суммировали введенные числа, то теперь суммируем значения, возвращаемые функцией square ().

Функции со многими аргументами

могут иметь не один аргумент. В этом случае аргументы разделяются запятыми. Например, следующая функция возвращает произведение двух аргументов:

```
int product(int arg1, int arg2)
{
    return arg1*arg2;
}
```

Преобразование типов

В программе содержится оператор, с которым вы раньше не встречались:

```
accumulator = accumulator + (int)dValue;
```

Префикс (int) перед dValue означает, что программист перед выполнением суммирования преобразует переменную dValue из ее текущего типа (в данном случае double) в int. Такая форма записи позволяет преобразовывать значения одного типа к другому.

допускает преобразование любых числовых величин в числовые величины другого типа. Компилятор C++ всегда выполняет приведение типов и без вашего вмешательства, но в таком случае для проверки правильности приведения он должен выдавать об этом соответствующее предупреждение. Поэтому во избежание ошибок рекомендуется использовать явное преобразование типов.

Функция main()

Служебное слово main() в нашей стандартной программе означает не что иное, как функцию (возможно, с необычными для вас аргументами и не требующую прототипа).

При компиляции программы C++ добавляет некоторый стандартный программный код, выполняемый до того, как начинает выполняться функция main (). Этот код настраивает программную среду, в которой выполняется ваша программа, например открывает потоки ввода и вывода по умолчанию.

После настройки среды выполнения этот код вызывает функцию main(), и лишь тогда происходит выполнение операторов программы. При завершении программа выходит из функции main (), после чего вторая часть стандартного кода C++ освобождает занятые программой системные ресурсы и передает управление операционной системе.

Перегрузка функций

C++ позволяет программистам называть несколько разных функций одним и тем же именем Эта возможность называется перефузкой функций (function overloading).

Вообще говоря, две функции в одной программе не могут иметь одинаковых имен, так как компилятор C++ просто не сможет их различить.

Однако используемое компилятором внутреннее имя функции включает число и типы аргументов (но не возвращаемое значение), и поэтому следующие функции являются разными:

```
void someFunction(void)
{
    //...выполнение некоторой функции
}
void someFunction(int n)
(
    //... выполнение другой функции
}
void someFunction(double d)
i
    //... выполнение еще одной функции
}
void someFunction(int n1, int
{
    //... выполнение еще одной функции,
    // отличной от предыдущих
)
```

Компилятор C++ знает, что функции someFunction (void) , someFunction(int), someFunction(double), int) не одинаковы. В мире компьютеров встречается довольно много таких вещей, для которых можно найти аналогии в реальном мире. Существуют они и для перегрузки функций.

Вы знаете, что тип аргумента *void* указывать не обязательно. Например, SumFunction(void) и SumFunction() вызывают одну и ту же функцию. Фактически функция может иметь сокращенное название, в нашем случае someFunction (), так же как и меня, можно называть просто Стефаном. Если бы Стефанов больше нигде не было, меня всегда могли бы называть только по имени. Но в действительности нас побольше, и если кому-то посчастливится попасть в общество нескольких Стефанов, то к ним придется обращаться по имени и фамилии (кстати, напомню, что меня зовут Стефан Дэвис). Пользуясь полным именем, никто не ошибется, даже если Стефанов будет несколько. Поэтому, пока полные имена функций в профамме уникальны, конфликты между ними невозможны.

Такие аналогии между компьютерным и реальным миром не должны вас удивлять, так как компьютерный мир создан людьми.

Типичное приложение может выглядеть следующим образом:

```
int intVariable1, intVariable2;
double doubleVariable;

//функции различаются по типу передаваемых аргументов
someFunction();                 //вызов someFunction(void)
someFunction(intVariable1);     //вызов someFunction(int)
someFunction(doubleVariable); //вызов someFunction(double)
someFunction(intVariable1,
                                //вызов someFunction(int, int)

// с константами функции работают аналогично
someFunction(1);                 // вызов someFunction(int)
someFunction(1.0);               // вызов someFunction(double)
someFunction(1,2);               // вызов someFunction(int, int)
```

В каждом случае типы аргументов соответствуют тем, которые значатся в полном имени каждой функции.

Тип возвращаемого значения в полное имя функции (называемое также ее сигнатурой) не входит. Следующие две функции имеют одинаковые имена (сигнатуры) и поэтому не могут использоваться в одной программе:

```
int someFunction(int n);
//полным именем этой функции является
double someFunction(int n); //имеет то же полное имя
```

Следующий код вполне допустим:

```
int someFunction(int n);
double d =                    ,•
//преобразуем тип полученного значения
```

В этом фрагменте возвращаемые функцией значения типа int преобразуются в double. Но следующий код некорректен:

```
int someFunction(int n);
double someFunction(int n);
double d = someFunction(10);
//В этом случае мы преобразуем тип полученного
//целочисленного значения или используем вторую функцию?
```

В этом случае C++ не поймет, какое значение он должен использовать — возвращаемое double-функцией или ее целочисленным вариантом.

Определение прототипов функций

Как уже отмечалось, любой фрагмент кода программист может оформить как функцию, присвоив ей полное имя, таким образом объявляя ее для дальнейшего использования.

Функции sumSequence () и square О, с которыми вы встречались этой главе, были определены до того, как вызывались. Но это не означает, что нужно всегда придерживаться именно такого порядка. Функция может быть определена в любой части модуля (модуль — это другое название исходного файла C++).

Однако должен использоваться какой-то механизм, уведомляющий функцию main() о функциях, которые могут вызываться ею. Рассмотрим следующий код:

```
int main(int argc, char* pArgs[])
{
    someFunc(1, 2);
}
int someFunc(double arg1, int arg2)
{
                    каких-то действий
}
```

При вызове функции () внутри {} полное ее имя неизвестно. Можно предположить, что именем функции является someFunc(int, int) и возвращаемое ею значение имеет тип void. Однако, как видите, это вовсе не так.

Согласен, компилятор C++ мог бы быть не таким ленивым и просмотреть весь модуль для определения сигнатуры функции. Но он этого не сделает, и с этим прихо" дится считаться[13]. Таков мир: любишь кататься — люби и саночки возить.

Поэтому нам нужно проинформировать main() о полном имени вызываемой в ней функции до обращения к ней. Для этого используют прототипы функций.

Прототип функции содержит ее полное имя с указанием типа возвращаемого значения. Использование прототипов рассмотрим на следующем примере:

```
int someFunc(double, int);
int main(int argc, char* pArgs[])
(
    someFunc(1,
|
int someFunc(double arg1, int
{
                    каких-то действий
}
```

Использованный прототип объясняет миру (по крайней мере той его части, которая следует после этого объявления), что полным именем функции someFunc() является (double, int). Теперь при ее вызове в компилятор поймет, что 1 нужно преобразовать к типу double. Кроме того, функция main() осведомлена, что someFunc() возвращает целое значение.

Вызов функции, возвращающей значение, является выражением. Следовательно, с ним можно работать так же, как и с любым другим значением этого же типа.

Хранение переменных в памяти

В памяти переменные хранятся в трех разных местах. Переменные, объявленные внутри функции, называются локальными. В следующем примере переменная localVariable является локальной по отношению к функции fn {}:

```
int
void fn()
```

---

[13] *Более того, как вы узнаете позже, тела функции в данном модуле может и не* . — Прим. ред.

```
    int localVariable;
    static int staticVariable;
}
```

До вызова fn () переменной localVariable не существует. После окончания работы функции она оставляет этот бренный мир и ее содержимое навсегда теряется. Добавлю, что доступ к ней имеет только функция остальные использовать ее не могут.

А вот переменная существует на протяжении работы всей программы и в любой момент доступна всем

Статическая переменная staticVariable является чем-то средним между локальной и глобальной переменными. Она создается, когда программа при выполнении достигает описания переменной (грубо говоря, когда происходит первый вызов ции). К тому же staticVariable доступна только из функции (). Но, в отличие от localVariable, переменная staticVariable продолжает существовать и после окончания работы функции. Если в функции () переменной присваивается какое-то значение, то оно сохранится до следующего вызова fn ().

С помощью массивов можно легко решить проблему работы с подобными последовательностями. В приведенном фрагменте объявляется массив valueArray, в котором можно хранить до 128 целых значений. Затем он заполняется числами, введенными с клавиатуры.

```
int value;

// объявление массива, способного содержать
// до 128 чисел типа int
int

//определение индекса,              для
//доступа к элементам массива; его значение
//не должно превышать
for (int i = 0; i < 128;

    cin >> value;
    //выходим из цикла, если
    //вводит              число
    if (value < 0) break;

)
```

Во второй строке кода (без учета комментариев) объявлен массив valueArray. Первым в объявлении указывается тип элементов массива (в нашем случае это int), за ним следует имя массива, последним элементом являются открывающая и закрывающая квадратные скобки, в которых записывается максимальное число элементов массива. В нашем случае массив valueArray может содержать до 128 целочисленных значений.

Компьютер считывает число с клавиатуры и сохраняет его в следующем элементе массива valueArray. Доступ к элементам массива обеспечивается с помощью имени массива и индекса, указанного в квадратных скобках. Первое целое массива обозначается как valueArray[0], второе — как valueArray[1] И

Запись valueArray представляет собой i-й элемент массива. Индексная переменная i должна быть перечислимой, т.е. ее типом может быть char, int или long. Если valueArray является массивом целых чисел, то элемент имеет тип int.

с массивами

В представленной ниже программе осуществляется ввод последовательности целых чисел (до первого отрицательного числа), затем эта последовательность и сумма ее элементов выводятся на экран.

```
//          — демонстрирует использование
//           Считывает
// целых чисел и отображает их по порядку



// объявления прототипов функций
int  sumArray(int
void                                  int

int          nArg, char* pszArgs[]}
{
     // Описываем счетчик цикла
    int             = 0;
    cout << "Эта программа суммирует числа,"
```

```
        << " введенные пользователем\n";
    cout << "Цикл прерывается, когда"
         << "              вводит"
         << " отрицательное число\n";

    // сохраняем числа в массиве
    int
    int numberOfValues = 0;
    for(; numberOfValues < 128;                +
    {
        // ввод очередного числа
            integerValue;
        cout << "Введите следующее         ";
             >> integerValue;

           если оно отрицательное...
        if {integerValue < 0)

                        выходим из цикла
            break;


            ...иначе сохраняем число в массиве
                                      integerValue


    // теперь выводим значения и их сумму
                               numberOfValues);
    cout << "Сумма введенных чисел равна "
         <<
         «  "\n";
    return 0;


// displayArray — отображает элементы массива
// integerArray длиной
void                                    int sizeOfArray)
{
    cout << "В массиве хранятся"
         << " следующие значения:\n";
    for (int i = 0; i < sizeOfArray; i++)
    {
        cout.width(3);
        cout « i « ":  " «                    << "\n";
    }
    cout << "\n";


// sumArray — возвращает сумму элементов
// целочисленного массива
int sumArray(int                  int sizeOfArray)
{
    int accumulator =
    for (int i = 0; i < sizeOfArray; i++)
    {
        accumulator +=
    }
           accumulator;
```

Программа начинается с объявления прототипов функций и displayArray (), которые понадобятся нам позже. Главная часть программы содержит довольно скучный цикл ввода значений. На этот раз вводимые значения сохраняются в массиве inputvalues.

Ввод осуществляется в цикле for. Введенное значение сначала сохраняется в локальной переменной Если оно оказывается отрицательным, оператор break прерывает выполнение цикла. Если же нет, значение integerValue копируется в массив.

Целочисленная переменная используется в качестве индекса массива. Она инициализирована нулем еще до начала цикла for. При каждой итерации индекс увеличивается. В условии выполнения цикла for осуществляется контроль за тем, чтобы количество введенных чисел не превышало т.е. размера массива (после введения 128 чисел программа к выводу элементов массива на экран независимо от того, ввел пользователь отрицательное число или нет).

> объявлении массива inputvalues было указано, что его максимальная длина равна 128. При записи большего числа данных, чем определено в объявлении, ваша программа может работать неправильно и даже аварийно завершать работу. Поэтому лучше застраховаться и оставить больше места для хранения данных. Неважно, насколько велик массив; всегда нужно следить за тем, чтобы операции с массивом не приводили к выходу за его пределы.

Функция main заканчивается выводом на экран содержимого массива и суммы его элементов. Функция displayArray () содержит обычный цикл for, который используется для прохождения по массиву. Каждый очередной элемент массива добавляется к переменной accumulator. Передаваемый функции параметр sizeOfArray включает количество значений, содержащихся в массиве.

Напомню еще раз, что индекс массива в C++ отсчитывается от 0, а не от 1. Кроме того, обратите внимание, что цикл for прерывается в тот момент, когда значение i становится равным sizeOfArray. Вы же не хотите добавлять все 128 элементов массива integerArray к accumulator? Ни один элемент массива, индекс которого больше или равен числу sizeOfArray, учитываться не будет.

Инициализация массива

Локальная переменная нежизнеспособна до тех пор, пока ей не присвоят значение. Другими словами, пока вы в ней что-то не сохраните, она будет содержать мусор. Локальное описание массива происходит так же: пока каждому элементу не присвоят какие-либо значения, в ячейках массива будет содержаться мусор. Локальную переменную следует инициализировать при ее объявлении, и еще в большей степени это справедливо для массивов. Слишком уж легко наткнуться на неработоспособную ячейку в неинициализированном массиве.

К счастью, массив может быть инициализирован сразу во время объявления, например:

```
float floatArray[5]   {0.0, 1.0,      3.0,
```

В этом фрагменте элементу loatArray[0] присваивается значение 0, floatArray[1] — 1, [2] — 2 и ТД.

Размер массива может определяться и количеством инициализирующих констант. Например, перечислив в скобках значения инициализаторов, можно ограничить размер массива floatArray пятью элементами. C++ умеет очень хорошо считать (по крайней мере, его можно спокойно использовать для этого). Так, следующее объявление идентично представленному выше:

```
float floatArray[]          1.0,      3.0,
```

Выход за границы массива

Математики пересчитывают содержимое массивов, начиная с элемента номер 1. Первым элементом математического массива x является x(1). Во многих языках программирования также начинают перечисление элементов массива с 1. Но в C++ массивы индексируются начиная с 0! Первый элемент массива C++ обозначается как Первый индекс массива C++ нулевой; поэтому последним элементом целочисленного массива является [127], а не integerArray [128].

К сожалению, в C++ не проверяется выход индекса за пределы массива. Этот язык будет рад предоставить вам доступ к элементу [200]. Более того, C++ позволит вам обратиться даже к [-15].

Приведем такую аналогию. Длину трассы можно измерять равными промежутками между многочисленными придорожными столбами. Назовем их единицами измерения длины дороги. Чтобы попасть к моему дому, нужно свернуть с главной трассы и по прямой ехать до конца дороги, длина которой равна точно девяти столбам. Если начинать отсчет с последнего телеграфного столба на трассе, тогда столб у моего дома будет иметь 10-й номер.

Отсчитывая столбы на трассе, можно получить доступ к любой позиции на дороге. Если отмерять расстояние от трассы до трассы, то получим, что оно равно нулю столбов. Следующей дискретной точкой отсчета является первый столб и т.д., пока вы не доберетесь до моего дома, который находится на расстоянии девяти столбов.

Конечно, вы можете отмерить расстояние в 20 столбов от трассы. Но дороги там уже не будет (помните, она кончается у моего дома?). Трудно точно сказать, что вы сможете там найти. Вы можете оказаться на следующей трассе, где-нибудь в поле или даже в чьей-то спальне (вот будет весело!). Угадать, куда в этом случае попадешь, довольно трудно, но хранить там еще рискованнее. Оставить вещь в поле — это одно, но забросить ее в комнату моего соседа совсем другое (я-то знаю это наверняка, так как каждый раз, когда газеты исчезают из моего почтового яшика, они оказываются в гостиной моего соседа).

То же происходит и в программировании. Пытаясь прочитать 20-й элемент (array массива, вы получите то или другое случайное число (а возможно, программа просто аварийно завершит работу). А записывая какое-то значение, вы получите непредсказуемый результат. Скорее всего, ничего в него не запишется, но возможны и другие странные реакции вплоть до краха программы.

> Самой распространенной ошибкой является неправильное обращение к последнему элементу по адресу integerArray 1128]. Хотя это всего лишь следующий за концом массива элемент, записывать или считывать его не менее опасно, чем любой другой некорректной адрес.

Использовать ли массивы

Разумеется, программа делает то же самое, что и не основанные на массивах программы, которые рассматривались раньше. Правда, в этой версии несколько изменен ввод множества чисел, но вы вряд ли будете потрясены этой особенностью.

И все же в возможности повторного отображения введенных значений кроется значительное преимущество использования массивов. Массивы позволяют программе многократно обрабатывать серии чисел. Главная программа была способна передать массив входных значений функции displayArray () для отображения, а затем в Su- для суммирования.

```
    for  int i   0; i<                    i++)
    {
            <<
    }
}
```

В программе объявлен фиксированный массив символов, содержащий, как вы могли заметить, мое имя (какое еще может быть лучше?). Этот массив передается в функцию displayCharArray () вместе с его длиной. Функция displayCharArray () идентична функции displayArray () из нашего предыдущего примера, но в этом варианте вместо чисел ею выводятся символы.

Программа работает довольно хорошо; но одно неудобство все-таки есть: раз вместе с самим массивом необходимо передавать его длину. Однако можно придумать правило, которое поможет решить нашу проблему. Если бы мы знали, что в конце массива находится специальный кодовый символ, то не потребовалось бы передавать размеры массива.

Будем использовать для указания конца массива нулевой символ.

Нулевой символ и обычный 0, вводимый с клавиатуры, — разные понятия. Кодом обыкновенного символа 0 является 0x30. Нуль-символ — это символ, каждый бит которого равен 0 (т.е. 0x00). Для того чтобы этот символ было легче отличить от 0, его часто записывают как \0.

Символ \y является символом, числовое значение которого равно y. Изменим предыдущую программу, используя это правило:

```
//              — выводит на экран массив
// символов в окне MS DOS


//            прототипов
void

int          nArg, char*
{
    char
                                'p', 'h',          0};

         << "\n";
           0;
}

// displayString — посимвольно выводит на экран строку
void
{
       (int i = 0; stringArray[i] != 0;
    {
        cout <<
    }
}
```

Массив charMyName объявляется как массив символов с дополнительным нулевым символом (\0) в конце. Программа displayString итеративно проходит по символьному массиву, пока не встретит нуль-символ.

Поскольку в функции displayString () больше нет необходимости передавать куда-либо длину символьного массива, использовать ее проще, чем displayCharArray ().