

*Основы основ — проще простого!*



Примеры программ работают со всеми компиляторами C++, включая Visual C++

# C++

**руководство для начинающих**  
Второе издание

БЕСПЛАТНЫЙ  
КОД  
в INTERNET

**БОНУС!**  
[WWW.OSBORNE.COM](http://WWW.OSBORNE.COM)

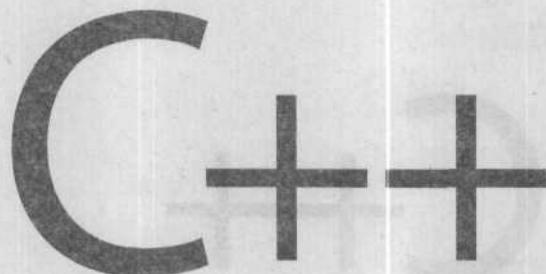
**Герберт Шилдт**

Автор бестселлеров по программированию — во всем мире продано более 3 млн. экземпляров его книг!

# C++

**руководство для начинающих**

Второе издание



# A Beginner's Guide

Second Edition

*HERBERT SCHILDT*

**McGraw-Hill/Osborne**

New York Chicago San Francisco  
Lisbon London Madrid Mexico City Milan  
New Delhi San Juan Seoul Singapore Sydney Toronto

Современное  
язык  
программирования

# C++

**руководство для начинающих**

**Второе издание**

**ГЕРБЕРТ ШИЛДТ**



Москва • Санкт-Петербург • Киев  
2005

ББК 32.973.26-018.2.75

Ш57

УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией С.Н. Тригуб

Перевод с английского и редакция Н.М. Ручко

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:

info@williamspublishing.com, <http://www.williamspublishing.com>

115419, Москва, а/я 783; 03150, Киев, а/я 152

Шилдт, Герберт.

Ш57 С++: руководство для начинающих, 2-е издание. : Пер. с англ. — М. : Издательский дом "Вильямс", 2005. — 672 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0840-X (рус.)

В этой книге описаны основные средства языка C++, которые необходимо освоить начинающему программисту. После рассмотрения элементарных понятий (переменных, операторов, инструкций управления, функций, классов и объектов) читатель легко перейдет к изучению таких более сложных тем, как перегрузка операторов, механизм обработки исключительных ситуаций (исключений), наследование, полиморфизм, виртуальные функции, средства ввода-вывода и шаблоны. Автор справочника — общепризнанный авторитет в области программирования на языках C и C++, Java и C# — включил в свою книгу множество тестов для самоконтроля, которые позволяют быстро проверить степень освоения материала, а также разделы "вопросов и ответов", способствующие более глубокому изучения основ программирования даже на начальном этапе.

ББК 32.973.26-018.2.75

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 Osborne Media.

Authorized translation from the English language edition published by Osborne Publishing. Copyright © 2004 by The McGraw-Hill Companies.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International. Copyright © 2005

ISBN 5-8459-0840-X (рус.)

ISBN 0-07-223215-3 (англ.)

© Издательский дом "Вильямс", 2005

© The McGraw-Hill Companies, 2004

# Оглавление

Об авторе	15
Введение	17
Модуль 1. Основы C++	21
Модуль 2. Типы данных и операторы	69
Модуль 3. Инструкции управления	109
Модуль 4. Массивы, строки и указатели	157
Модуль 5. Введение в функции	211
Модуль 6. О функциях подробнее	261
Модуль 7. Еще о типах данных и операторах	303
Модуль 8. Классы и объекты	349
Модуль 9. О классах подробнее	397
Модуль 10. Наследование	465
Модуль 11. C++-система ввода-вывода	521
Модуль 12. Исключения, шаблоны и кое-что еще	571
Приложение А. Препроцессор	639
Приложение Б. Использование устаревшего C++-компилятора	653
Предметный указатель	656

# Содержание

<b>Об авторе</b>	<b>15</b>
<b>Введение</b>	<b>17</b>
Как организована эта книга	17
Практические навыки	18
Тест для самоконтроля	18
Вопросы для текущего контроля	18
Спросим у опытного программиста	18
Учебные проекты	18
Никакого предыдущего опыта в области программирования не требуется	18
Требуемое программное обеспечение	19
Программный код – из Web-пространства	19
Для дальнейшего изучения программирования	19
<b>Модуль 1. Основы C++</b>	<b>21</b>
1.1. Из истории создания C++	22
Язык С: начало эры современного программирования	23
Предпосылки возникновения языка C++	24
Рождение C++	25
Эволюция C++	26
1.2. Связь C++ с языками Java и C#	27
1.3. Объектно-ориентированное программирование	29
Инкапсуляция	30
Полиморфизм	31
Наследование	32
1.4. Создание, компиляция и выполнение C++-программ	32
Ввод текста программы	34
Компиляирование программы	34
Выполнение программы	35
Постстрочный “разбор полетов” первого примера программы	36
Обработка синтаксических ошибок	38
1.5. Использование переменных	40
1.6. Использование операторов	41
1.7. Считывание данных с клавиатуры	44
Вариации на тему вывода данных	46
Познакомимся еще с одним типом данных	47
1.8. Использование инструкций if и for	52

Инструкция if	52
Цикл for	54
1.9. Использование блоков кода	56
Точки с запятой и расположение инструкций	58
Практика отступов	59
1.10. Понятие о функциях	62
Библиотеки C++	64
1.12. Ключевые слова C++	65
1.13. Идентификаторы	66
<b>Модуль 2. Типы данных и операторы</b>	<b>69</b>
Почему типы данных столь важны	70
2.1. Типы данных C++	70
Целочисленный тип	72
Символы	75
Типы данных с плавающей точкой	77
Тип данных bool	78
Тип void	80
2.2. Литералы	83
Шестнадцатеричные и восьмеричные литералы	84
Строковые литералы	84
Управляющие символьные последовательности	85
2.3. Создание инициализированных переменных	87
Инициализация переменной	87
Динамическая инициализация переменных	88
Операторы	89
2.4. Арифметические операторы	89
Инкремент и декремент	90
2.5. Операторы отношений и логические операторы	92
2.6. Оператор присваивания	98
2.7. Составные операторы присваивания	99
2.8. Преобразование типов в операторах присваивания	100
Выражения	101
2.9. Преобразование типов в выражениях	101
Преобразования, связанные с типом bool	102
2.10. Приведение типов	102
2.11. Использование пробелов и круглых скобок	104

<b>Модуль 3. Инструкции управления</b>	<b>109</b>
3.1. Инструкция if	110
Условное выражение	112
Вложенные if-инструкции	114
“Лестничная” конструкция if-else-if	115
3.2. Инструкция switch	117
Вложенные инструкции switch	121
3.3. Цикл for	125
Вариации на тему цикла for	127
Отсутствие элементов в определении цикла	129
Бесконечный цикл for	130
Циклы без тела	131
Объявление управляющей переменной цикла в заголовке инструкции for	132
3.4. Цикл while	134
3.6. Использование инструкции break для выхода из цикла	143
3.7. Использование инструкции continue	145
3.8. Вложенные циклы	151
3.9. Инструкция goto	152
<b>Модуль 4. Массивы, строки и указатели</b>	<b>157</b>
4.1. Одномерные массивы	158
На границах массивов без пограничников	162
4.2. Двумерные массивы	163
4.3. Многомерные массивы	165
4.4. Строки	169
Основы представления строк	169
Считывание строк с клавиатуры	170
4.5. Некоторые библиотечные функции обработки строк	173
Функция strcpy()	173
Функция strcat()	173
Функция strcmp()	173
Функция strlen()	174
Пример использования строковых функций	174
Использование признака завершения строки	175
4.6. Инициализация массивов	177
Инициализация “безразмерных” массивов	180
4.7. Массивы строк	182
4.8. Указатели	183

Что представляют собой указатели	184
4.9. Операторы, используемые с указателями	185
О важности базового типа указателя	186
Присваивание значений с помощью указателей	189
4.10. Использование указателей в выражениях	190
Арифметические операции над указателями	190
Сравнение указателей	192
4.11. Указатели и массивы	193
Индексирование указателя	196
Строковые константы	198
Массивы указателей	203
Соглашение о нулевых указателях	205
4.12. Многоуровневая непрямая адресация	206
<b>Модуль 5. Введение в функции</b>	<b>211</b>
Основы использования функций	212
5.1. Общий формат C++-функций	212
5.2. Создание функции	213
5.3. Использование аргументов	215
5.4. Использование инструкции return	216
Возврат значений	220
5.5. Использование функций в выражениях	222
Правила действия областей видимости функций	224
5.6. Локальная область видимости	225
5.7. Глобальная область видимости	231
5.8. Передача указателей и массивов в качестве аргументов функций	235
Передача функции указателя	235
Передача функции массива	237
Передача функциям строк	240
5.9. Возвращение функциями указателей	242
Функция main()	243
5.10. Передача аргументов командной строки функции main()	244
Передача числовых аргументов командной строки	246
5.11. Прототипы функций	248
Стандартные заголовки содержат прототипы функций	250
5.12. Рекурсия	250
<b>Модуль 6. О функциях подробнее</b>	<b>261</b>
6.1. Два способа передачи аргументов	262
6.2. Как в C++ реализована передача аргументов	262

## **10 Содержание**

6.3. Использование указателя для обеспечения вызова по ссылке	264
6.4. Ссыльные параметры	266
6.5. Возврат ссылок	271
6.6. Независимые ссылки	275
Ограничения при использовании ссылок	276
6.7. Перегрузка функций	277
Автоматическое преобразование типов и перегрузка	282
6.8. Аргументы, передаваемые функции по умолчанию	292
Сравнение возможности передачи аргументов по умолчанию с перегрузкой функций	294
Об использовании аргументов, передаваемых по умолчанию	296
6.9. Перегрузка функций и неоднозначность	298
<b>Модуль 7</b>	<b>303</b>
<b>Еще о типах данных и операторах</b>	<b>303</b>
7.1. Спецификаторы типа const и volatile	304
Спецификатор типа const	304
Спецификатор типа volatile	307
Спецификаторы классов памяти	308
Спецификатор класса памяти auto	308
7.2. Спецификатор класса памяти extern	308
7.3. Статические переменные	310
7.4. Регистровые переменные	315
7.5. Перечисления	318
7.6. Ключевое слово typedef	322
7.7. Поразрядные операторы	322
Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ	323
7.8. Операторы сдвига	330
7.9. Оператор "знак вопроса"	339
7.10. Оператор "запятая"	340
7.11. Составные операторы присваивания	342
7.12. Использование ключевого слова sizeof	343
Сводная таблица приоритетов C++-операторов	345
<b>Модуль 8. Классы и объекты</b>	<b>349</b>
8.1. Общий формат объявления класса	350
8.2. Определение класса и создание объектов	351
8.3. Добавление в класс функций-членов	357
8.4. Конструкторы и деструкторы	367

8.5. Параметризованные конструкторы	370
Добавление конструктора в класс Vehicle	373
Альтернативный вариант инициализации объекта	374
8.6. Встраиваемые функции	376
Создание встраиваемых функций в объявлении класса	378
8.7. Массивы объектов	388
8.8. Инициализация массивов объектов	389
8.9. Указатели на объекты	391
<b>Модуль 9. О классах подробнее</b>	<b>397</b>
9.1. Перегрузка конструкторов	398
9.2. Присваивание объектов	399
9.3. Передача объектов функциям	402
Конструкторы, деструкторы и передача объектов	403
Передача объектов по ссылке	405
Потенциальные проблемы при передаче параметров	407
9.4. Возвращение объектов функциями	408
9.5. Создание и использование конструктора копии	411
9.6. Функции-“друзья”	415
9.7. Структуры и объединения	421
Структуры	421
Объединения	424
9.8. Ключевое слово this	428
9.9. Перегрузка операторов	430
9.10. Перегрузка операторов с использованием функций-членов	431
О значении порядка operandов	435
Использование функций-членов для перегрузки	
унарных операторов	435
9.11. Перегрузка операторов с использованием функций-не членов класса	441
Использование функций-“друзей” для перегрузки	
унарных операторов	447
Советы по реализации перегрузки операторов	448
<b>Модуль 10. Наследование</b>	<b>465</b>
10.1. Понятие о наследовании	466
Доступ к членам класса и наследование	470
10.2. Управление доступом к членам базового класса	474
10.3. Использование защищенных членов	477
10.4. Вызов конструкторов базового класса	483

## **12 Содержание**

---

10.5. Создание многоуровневой иерархии	493
10.6. Наследование нескольких базовых классов	497
10.7. Когда выполняются функции конструкторов и деструкторов	498
10.8. Указатели на производные типы	501
Ссылки на производные типы	502
10.9. Виртуальные функции и полиморфизм	502
Понятие о виртуальной функции	502
Наследование виртуальных функций	506
Зачем нужны виртуальные функции	507
Применение виртуальных функций	509
10.10. Чисто виртуальные функции и абстрактные классы	514
<b>Модуль 11. C++-система ввода-вывода</b>	<b>521</b>
Сравнение старой и новой C++-систем ввода-вывода	522
11.1. Потоки C++	523
Встроенные C++-потоки	524
11.2. Классы потоков	524
11.3. Перегрузка операторов ввода-вывода	526
Создание перегруженных операторов вывода	527
Использование функций-“друзей” для перегрузки	
операторов вывода	529
Перегрузка операторов ввода	530
Форматированный ввод-вывод данных	533
11.4. Форматирование данных с использованием	
функций-членов класса ios	533
11.5. Использование манипуляторов ввода-вывода	540
11.6. Создание собственных манипуляторных функций	543
Файловый ввод-вывод данных	545
11.7. Как открыть и закрыть файл	546
11.8. Чтение и запись текстовых файлов	549
11.9. Неформатированный ввод-вывод данных в двоичном режиме	551
Считывание и запись в файл блоков данных	554
11.10. Использование других функций ввода-вывода	556
Версии функции get()	557
Функция getline()	558
Функция обнаружения конца файла	559
Функции peek() и putback()	559
Функция flush()	559

11.11. Произвольный доступ	565
11.12. Получение информации об операциях ввода-вывода	568
<b>Модуль 12. Исключения, шаблоны и кое-что еще</b>	<b>571</b>
12.1. Обработка исключительных ситуаций	572
Основы обработки исключительных ситуаций	572
Использование нескольких catch-инструкций	578
Перехват всех исключений	581
Определение исключений, генерируемых функциями	583
Повторное генерирование исключений	585
Шаблоны	587
12.2. Обобщенные функции	587
Функция с двумя обобщенными типами	590
Явно заданная перегрузка обобщенной функции	591
12.3. Обобщенные классы	594
Явно задаваемые специализации классов	597
12.4. Динамическое распределение памяти	603
Инициализация динамически выделенной памяти	607
Выделение памяти для массивов	608
Выделение памяти для объектов	609
12.5. Пространства имен	614
Понятие пространства имен	615
Инструкция using	619
Неименованные пространства имен	622
Пространство имен std	622
12.6. Статические члены класса	623
Статические члены данных класса	623
Статические функции-члены класса	626
12.7. Динамическая идентификация типов (RTTI)	628
12.8. Операторы приведения типов	633
Оператор dynamic_cast	633
Оператор const_cast	635
Оператор static_cast	635
Оператор reinterpret_cast	635
<b>Приложение А. Препроцессор</b>	<b>639</b>
Директива #define	640
Макроопределения, действующие как функции	642
Директива #error	644

## 14 Содержание

Директива #include	644
Директивы условной компиляции	645
Директивы #if, #else, #elif и #endif	645
Директивы #ifdef и #ifndef	648
Директива #undef	649
Использование оператора defined	649
Директива #line	650
Директива #pragma	650
Операторы препроцессора “#” и “##”	651
Зарезервированные макроимена	652
<b>Приложение Б. Использование устаревшего C++-компилитора</b>	<b>653</b>
Два простых изменения	655
<b>Предметный указатель</b>	<b>656</b>

## Об авторе

Герберт Шилдт (Herbert Schildt) — признанный авторитет в области программирования на языках C, C++ Java и C#, профессиональный Windows-программист, член комитетов ANSI/ISO, принимавших стандарт для языков C и C++. Продано свыше 3 миллионов экземпляров его книг. Они переведены на все самые распространенные языки мира. Шилдт — автор таких бестселлеров, как *Полный справочник по C*, *Полный справочник по C++*, *Полный справочник по C#*, *Полный справочник по Java 2*, и многих других книг, включая: *Руководство для начинающих по C*, *Руководство для начинающих по C#* и *Руководство для начинающих по Java 2*. Шилдт — обладатель степени магистра в области вычислительной техники (университет шт. Иллинойс). Его контактный телефон (в консультационном отделе): (217) 586-4683.

## **Введение**

Язык C++ предназначен для разработки высокопроизводительного программного обеспечения и чрезвычайно популярен среди программистов. При этом он обеспечивает концептуальный фундамент (синтаксис и стиль), на который опираются другие языки программирования. Не случайно ведь потомками C++ стали такие почитаемые языки, как C# и Java. Более того, C++ можно назвать универсальным языком программирования, поскольку практически все профессиональные программисты на том или ином уровне знакомы с C++. Изучив C++, вы получите фундаментальные знания, которые позволят вам освоить любые аспекты современного программирования.

Цель этой книги — помочь читателю овладеть базовыми элементами C++-программирования. Сначала, например, вы узнаете, как скомпилировать и выполнить C++-программу, а затем шаг за шагом будете осваивать более сложные темы (ключевые слова, языковые конструкции, операторы и пр.). Текст книги подкрепляется многочисленными примерами программ, тестами для самоконтроля и учебными проектами, поэтому, проработав весь материал этой книги, вы получите глубокое понимание основ C++-программирования.

Я хочу подчеркнуть, что эта книга — лишь стартовая площадка. C++ — это большой (по объему средств) и не самый простой язык программирования. Необходимым условием успешного программирования на C++ является знание не только ключевых слов, операторов и синтаксиса, определяющего возможности языка, но и библиотек классов и функций, которые существенно помогают в разработке программ. И хотя некоторые элементы библиотек рассматриваются в этой книге, все же большинство из них не нашло здесь своего отражения. Чтобы стать первоклассным программистом на C++, необходимо в совершенстве изучить и C++-библиотеки. Знания, полученные при изучении этой книги, позволят вам освоить не только библиотеки, но и все остальные аспекты C++.

## **Как организована эта книга**

Эта книга представляет собой самоучитель, материал которого равномерно распределен по разделам, причем успешное освоение каждого следующего предполагает знание всех предыдущих. Книга содержит 12 модулей, посвященных соответствующим аспектам C++. Уникальность этой книги состоит в том, что она включает несколько специальных элементов, которые позволяют закрепить уже пройденный материал.

## **Практические навыки**

Все модули содержат разделы (отмеченные пиктограммой “Важно!”), которые важно не пропустить при изучении материала книги. Их значимость для последовательного освоения подчеркивается тем, что они пронумерованы и перечислены в начале каждого модуля.

## **Тест для самоконтроля**

Каждый модуль завершается тестом для самоконтроля. Ответы можно найти на Web-странице этой книги по адресу: <http://www.osborne.com>.

## **Вопросы для текущего контроля**

В конце каждого значимого раздела содержится тест, проверяющий степень вашего понимания ключевых моментов, изложенных выше. Ответы на эти вопросы приводятся внизу соответствующей страницы.

## **Спросим у опытного программиста**

В различных местах книги вы встретите специальные разделы “Спросим у опытного программиста”, которые содержат дополнительную информацию или интересные комментарии по данной теме. Эти разделы представлены в формате “вопрос-ответ”.

## **Учебные проекты**

Каждый модуль включает один или несколько проектов, которые показывают, как на практике можно применить изложенный здесь материал. Эти учебные проекты представляют собой реальные примеры, которые можно использовать в качестве стартовых вариантов для ваших программ.

## **Никакого предыдущего опыта в области программирования не требуется**

Для освоения представленного здесь материала никакого предыдущего опыта в области программирования не требуется. Поэтому, если вы никогда раньше не программируали, можете смело браться за эту книгу. Конечно же, в наши дни многие читатели уже имеют хотя бы минимальный опыт в программировании. Например, вам, возможно, уже приходилось писать программы на Java или C#. Как вы скоро узнаете, C++ является “родителем” обоих этих языков. Поэтому, если вы уже знакомы с Java или C#, то вам не составит труда освоить и C++.

## Требуемое программное обеспечение

Чтобы скомпилировать и выполнить программы, приведенные в этой книге, вам понадобится любой из таких современных компиляторов, как Visual C++ (Microsoft) и C++ Builder (Borland).

## Программный код — из Web-пространства

Помните, что исходный код всех примеров программ и учебных проектов, приведенных в этой книге, можно бесплатно загрузить с Web-сайта по адресу: <http://www.osborne.com>. Загрузка кода избавит вас от необходимости вводить текст программ вручную.

## Для дальнейшего изучения программирования

Книга *C#: руководство для начинающих* — это ваш “ключ” к серии книг по программированию, написанных Гербертом Шилдтом. Ниже перечислены те из них, которые могут представлять для вас интерес.

Те, кто желает подробнее изучить язык C++, могут обратиться к следующим книгам.

- Полный справочник по C++,
- C++: базовый курс,
- Освой самостоятельно C++,
- *STL Programming from the Ground Up*,
- Справочник программиста по C/C++.

Тем, кого интересует программирование на языке Java, мы рекомендуем такие издания.

- *Java 2: A Beginner's Guide*,
- Полный справочник по Java 2,
- *Java 2: Programmer's Reference*,
- Искусство программирования на Java.

Если вы желаете научиться программировать на C#, обратитесь к следующим книгам.

- *C#: A Beginner's Guide,*
- *Полный справочник по C#.*

Тем, кто интересуется Windows-программированием, мы можем предложить такие книги Шилдта.

- *Windows 98 Programming from the Ground Up,*
- *Windows 2000 Programming from the Ground Up,*
- *MFC Programming from the Ground Up,*
- *The Windows Annotated Archives.*

Если вы хотите поближе познакомиться с языком С, который является фундаментом всех современных языков программирования, обратитесь к следующим книгам.

- *Полный справочник по C,*
- *Освой самостоятельно C.*

Если вам нужны четкие ответы, обращайтесь к Герберту Шилдту, общепризнанному авторитету в области программирования.

## От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

Информация для писем из:

России: 115419, Москва, а/я 783

Украины: 03150, Киев, а/я 152

# Модуль 1

## Основы C++

- 1.1.** Из истории создания C++
- 1.2.** Связь C++ с языками Java и C#
- 1.3.** Объектно-ориентированное программирование
- 1.4.** Создание, компиляция и выполнение C++-программ
- 1.5.** Использование переменных
- 1.6.** Использование операторов
- 1.7.** Считывание данных с клавиатуры
- 1.8.** Использование инструкций управления *if* и *for*
- 1.9.** Использование блоков кода
- 1.10.** Понятие о функциях
- 1.12.** Ключевые слова C++
- 1.13.** Идентификаторы

Если и существует один компьютерный язык, который определяет суть современного программирования, то, безусловно, это C++. Язык C++ предназначен для разработки высокопроизводительного программного обеспечения. Его синтаксис стал стандартом для других профессиональных языков программирования, а его принципы разработки отражают идеи развития вычислительной техники в целом. C++ послужил фундаментом для разработки языков будущего. Например, как Java, так и C# – прямые потомки языка C++. Сегодня быть профессиональным программистом высокого класса означает быть компетентным в C++. C++ – это ключ к современному программированию.

Цель этого модуля – представить C++ в историческом аспекте, проследить его истоки, проанализировать его взаимоотношения с непосредственным предшественником (C), рассмотреть его возможности (области применения) и принципы программирования, которые он поддерживает. Самым трудным в изучении языка программирования, безусловно, является то, что ни один его элемент не существует изолированно от других. Компоненты языка работают вместе, можно сказать, в дружном “коллективе”. Такая тесная взаимосвязь усложняет рассмотрение одного аспекта C++ без изучения других. Зачастую обсуждение одного средства предусматривает предварительное знакомство с другим. Для преодоления подобных трудностей в этом модуле приводится краткое описание таких элементов C++, как общий формат C++-программы, основные инструкции управления и операторы. При этом мы не будем пока углубляться в детали, а сосредоточимся на общих концепциях создания C++-программы.

**ВАЖНО!**

## **1.1. Из истории создания C++**

История создания C++ начинается с языка С. И немудрено: C++ построен на фундаменте С. C++ и в самом деле представляет собой супермножество языка С. C++ можно назвать расширенной и улучшенной версией языка С, предназначеннной для поддержки объектно-ориентированного программирования (его принципы описаны ниже в этом модуле). C++ также включает ряд других усовершенствований языка С, например расширенный набор библиотечных функций. При этом “вкус и запах” C++ унаследовал непосредственно из языка С. Чтобы до конца понять и оценить достоинства C++, необходимо понять все “как” и “почему” в отношении языка С.

## Язык С: начало эры современного программирования

Изобретение языка С отметило начало новой эры современного программирования. Его влияние нельзя было переоценить, поскольку он коренным образом изменил подход к программированию и его восприятие. Его синтаксис и принципы разработки оказали влияние на все последующие компьютерные языки, поэтому язык С можно назвать одной из основных революционных сил в развитии вычислительной техники.

Язык С изобрел Дэнис Ритч (Dennis Ritchie) для компьютера PDP-11 (разработка компании DEC – Digital Equipment Corporation), который работал под управлением операционной системы (ОС) UNIX. Язык С – это результат процесса разработки, который сначала был связан с другим языком – BCPL, созданным Мартином Ричардсом (Martin Richards). Язык BCPL индуцировал появление языка, получившего название В (его автор – Кен Томпсон (Ken Thompson)), который в свою очередь в начале 70-х годов привел к разработке языка С.

Язык С стал считаться первым современным “языком программиста”, поскольку до его изобретения компьютерные языки в основном разрабатывались либо как учебные упражнения, либо как результат деятельности бюрократических структур. С языком С все обстояло иначе. Он был задуман и разработан реальными, практикующими программистами и отражал их подход к программированию. Его средства были многократно обдуманы, отточены и протестированы людьми, которые действительно работали с этим языком. В результате этого процесса появился язык, который понравился многим программистам-практикам и быстро распространился по всему миру. Его невиданный успех обусловило то, что он был разработан программистами для программистов.

Язык С возник в результате революции *структурированного программирования* 60-х годов. До появления структурированного программирования обозначились проблемы в написании больших программ, поскольку программисты пользовались логикой, которая приводила к созданию так называемого “спагетти-кода”, состоящего из множества переходов, последовательность которых было трудно проследить. В структурированных языках программирования эта проблема решалась путем использования хорошо определенных инструкций управления, подпрограмм, локальных переменных и других средств. Появление структурированных языков позволило писать уже довольно большие программы.

Несмотря на то что в то время уже существовали другие структурированные языки (например Pascal), С стал первым языком, в котором успешно сочетались мощь, изящество и выразительность. Его лаконизм, простота синтаксиса и философия были настолько привлекательными, что в мире программирования непо-

стижимо быстро образовалась целая армия его приверженцев. С точки зрения сегодняшнего дня этот феномен даже трудно понять, но, тем не менее, язык C стал своего рода долгожданным “свежим ветром”, который вдохнул в программирование новую жизнь. В результате C был общепризнан как самый популярный структурированный язык 1980-х годов.

## Предпосылки возникновения языка C++

Приведенная выше характеристика языка C может вызвать справедливое недоумение: зачем же тогда, мол, был изобретен язык C++? Если C – такой успешный и полезный язык, то почему возникла необходимость в чем-то еще? Оказывается, все дело в *сложности*. На протяжении всей истории программирования усложнение программ заставляло программистов искать пути, которые бы позволили справиться со сложностью. Язык C++ можно считать одним из способов ее преодоления. Попробуем лучше раскрыть взаимосвязь между постоянно возрастающей сложностью программ и путями развития языков программирования.

Отношение к программированию резко изменилось с момента изобретения компьютера. Например, программирование для первых вычислительных машин состояло в переключении тумблеров на их передней панели таким образом, чтобы их положение соответствовало двоичным кодам машинных команд. Пока длины программ не превышали нескольких сотен команд, такой метод еще имел право на существование. Но по мере их дальнейшего роста был изобретен язык ассемблер, чтобы программисты могли использовать символическое представление машинных команд. Поскольку программы продолжали расти в размерах, желание справиться с более высоким уровнем сложности вызвало появление языков высокого уровня, разработка которых дала программистам больше инструментов (новых и разных).

Первым широко распространенным языком программирования был, конечно же, FORTRAN. Несмотря на то что это был очень значительный шаг на пути прогресса в области программирования, FORTRAN все же трудно назвать языком, который способствовал написанию ясных и простых для понимания программ. Шестидесятые годы двадцатого столетия считаются периодом появления структурированного программирования. Именно такой метод программирования и был реализован в языке C. С помощью структурированных языков программирования можно было писать программы средней сложности, причем без особых героических усилий со стороны программиста. Но если программный проект достигал определенного размера, то даже с использованием упомянутых структурированных методов его сложность резко возрастала и оказывалась непреодолимой для возможностей программиста. К концу 70-х к “критической” точке подошло довольно много проектов.

Решить эту проблему “взялись” новые технологии программирования. Одна из них получила название *объектно-ориентированного программирования* (ООП). Вооружившись методами ООП, программист мог справляться с программами гораздо большего размера, чем прежде. Но язык С не поддерживал методов ООП. Стремление получить объектно-ориентированную версию языка С в конце концов и привело к созданию C++.

Несмотря на то что язык С был одним из самых любимых и распространенных профессиональных языков программирования, настало время, когда его возможности по написанию сложных программ достигли своего предела. Желание преодолеть этот барьер и помочь программисту легко справляться с еще более объемными и сложными программами — вот что стало основной причиной создания C++.

## Рождение C++

Язык C++ был создан Бьерном Страуструпом (Bjarne Stroustrup) в 1979 году в компании Bell Laboratories (г. Муррей-Хилл, шт. Нью-Джерси). Сначала новый язык получил имя “С с классами” (C with Classes), но в 1983 году он стал называться C++.

Страуструп построил C++ на фундаменте языка С, включающем все его средства, атрибуты и основные достоинства. Для него также остается в силе принцип С, согласно которому программист, а не язык, несет ответственность за результаты работы своей программы. Именно этот момент позволяет понять, что изобретение C++ не было попыткой создать новый язык программирования. Это было скорее усовершенствование уже существующего (и при этом весьма успешного) языка.

Большинство новшеств, которыми Страуструп обогатил язык С, было предназначено для поддержки объектно-ориентированного программирования. По сути, C++ стал объектно-ориентированной версией языка С. Взяв язык С за основу, Страуструп подготовил плавный переход к ООП. Теперь, вместо того, чтобы изучать совершенно новый язык, С-программисту достаточно было освоить только ряд новых средств, и он мог пожинать плоды использования объектно-ориентированной технологии программирования.

Создавая C++, Страуструп понимал, насколько важно, сохранив изначальную суть языка С, т.е. его эффективность, гибкость и принципы разработки, внести в него поддержку объектно-ориентированного программирования. К счастью, эта цель была достигнута. C++ по-прежнему предоставляет программисту свободу действий и власть над компьютером (которые были присущи языку С), значительно расширяя при этом его (программиста) возможности за счет использования объектов.

Несмотря на то что C++ изначально был нацелен на поддержку очень больших программ, этим, конечно же, его использование не ограничивалось. И в самом деле, объектно-ориентированные средства C++ можно эффективно применять практически к любой задаче программирования. Неудивительно, что C++ используется для создания компиляторов, редакторов, компьютерных игр и программ сетевого обслуживания. Поскольку C++ обладает эффективностью языка C, то программное обеспечение многих высокоеффективных систем построено с использованием C++. Кроме того, C++ — это язык, который чаще всего выбирается для Windows-программирования.

## ЭВОЛЮЦИЯ C++

С момента изобретения C++ претерпел три крупные переработки, причем каждый раз язык как дополнялся новыми средствами, так и в чем-то изменялся. Первой ревизии он был подвергнут в 1985 году, второй — в 1990, а третья произошла в процессе стандартизации, который активизировался в начале 1990-х. Специально для этого был сформирован объединенный ANSI/ISO-комитет (я был его членом), который 25 января 1994 года принял первый проект предложенного на рассмотрение стандарта. В этот проект были включены все средства, впервые определенные Страуструпом, и добавлены новые. Но в целом он отражал состояние C++ на тот момент времени.

Вскоре после завершения работы над первым проектом стандарта C++ произошло событие, которое заставило значительно расширить существующий стандарт. Речь идет о создании Александром Степановым (Alexander Stepanov) стандартной библиотеки шаблонов (Standard Template Library — STL). Как вы узнаете позже, STL — это набор обобщенных функций, которые можно использовать для обработки данных. Он довольно большой по размеру. Комитет ANSI/ISO проголосовал за включение STL в спецификацию C++. Добавление STL расширило сферу рассмотрения средств C++ далеко за пределы исходного определения языка. Однако включение STL, помимо прочего, замедлило процесс стандартизации C++, причем довольно существенно.

Помимо STL, в сам язык было добавлено несколько новых средств и внесено множество мелких изменений. Поэтому версия C++ после рассмотрения комитетом по стандартизации стала намного больше и сложнее по сравнению с исходным вариантом Страуструпа. Конечный результат работы комитета датируется 14 ноября 1997 года, а реально ANSI/ISO-стандарт языка C++ увидел свет в 1998 году. Именно эта спецификация C++ обычно называется *Standard C++*. И именно она описана в данной книге. Эта версия C++ поддерживается всеми основными C++-компиляторами, включая Visual C++ (Microsoft) и C++ Builder

(Borland). Поэтому код программ, приведенных в этой книге, полностью применим ко всем современным C++-средам.

**ВАЖНО!**

## 1.2. Связь C++ с языками Java и C#

Помимо C++, существуют два других современных языка программирования: Java и C#. Язык Java разработан компанией Sun Microsystems, а C# – компанией Microsoft. Поскольку иногда возникает путаница относительно того, какое отношение эти два языка имеют к C++, попробуем внести ясность в этот вопрос.

C++ является родительским языком для Java и C#. И хотя разработчики Java и C# добавили к первоисточнику, удалили из него или модифицировали различные средства, в целом синтаксис этих трех языков практически идентичен. Более того, объектная модель, используемая C++, подобна объектным моделям языков Java и C#. Наконец, очень сходно общее впечатление и ощущение от использования всех этих языков. Это значит, что, зная C++, вы можете легко изучить Java или C#. Схожесть синтаксисов и объектных моделей – одна из причин быстрого освоения (и одобрения) этих двух языков многими опытными C++-программистами. Обратная ситуация также имеет место: если вы знаете Java или C#, изучение C++ не доставит вам хлопот.

Основное различие между C++, Java и C# заключается в типе вычислительной среды, для которой разрабатывался каждый из этих языков. C++ создавался с целью написания высокоеффективных программ, предназначенных для выполнения под управлением определенной операционной системы и в расчете на ЦП конкретного типа. Например, если вы хотите написать высокоеффективную программу для выполнения на процессоре Intel Pentium под управлением операционной системы Windows, лучше всего использовать для этого язык C++.

Языки Java и C# разработаны в ответ на уникальные потребности сильно распределенной сетевой среды Internet. (При разработке C# также ставилась цель упростить создание программных компонентов.) Internet связывает множество различных типов ЦП и операционных систем. Поэтому возможность создания межплатформенного (совместимого с несколькими операционными средами) переносимого программного кода для Internet при решении некоторых задач стало определяющим условием при выборе языка программирования.

Первым языком, отвечающим таким требованиям, был Java. Используя Java, можно написать программу, которая будет выполняться в различных вычислительных средах, т.е. в широком диапазоне операционных систем и типов ЦП. Таким образом, Java-программа может свободно “бороздить просторы” Internet. Несмотря на то что Java позволяет создавать переносимый программный код, ко-

торый работает в сильно распределенной среде, цена этой переносимости — эффективность. Java-программы выполняются медленнее, чем C++-программы. То же справедливо и для C#. Поэтому, если вы хотите создавать высокоэффективные приложения, используйте C++. Если же вам нужны переносимые программы, используйте Java или C#.

### Спросим у опытного программиста

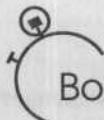
**Вопрос.** *Каким образом языки Java и C# позволяют создавать межплатформенные (совместимые с несколькими операционными средами) переносимые программы и почему с помощью C++ невозможно достичь такого же результата?*

**Ответ.** Все дело в типе объектного кода, создаваемого компиляторами. В результате работы C++-компилятора мы получаем машинный код, который непосредственно выполняется конкретным центральным процессором (ЦП). Другими словами, C++-компилятор связан с конкретными ЦП и операционной системой. Если нужно выполнить C++-программу под управлением другой операционной системой, необходимо перекомпилировать ее и получить машинный код, подходящий для данной среды. Поэтому, чтобы C++-программа могла выполняться в различных средах, нужно создать несколько выполняемых версий этой программы.

Используя языки Java и C#, можно добиться переносимости программного кода за счет специальной компиляции программ, позволяющей перевести исходный программный код на промежуточный язык, именуемый *псевдокодом*. Для Java-программ этот промежуточный язык называют *байт-кодом* (bytecode), т.е. машинно-независимым кодом, генерируемым Java-компилятором. В результате компиляции C#-программы получается не исполняемый код, а файл, который содержит специальный псевдокод, именуемый *промежуточным языком Microsoft* (*Microsoft Intermediate Language – MSIL*). В обоих случаях этот псевдокод выполняется специальной операционной системой, которая для Java называется *виртуальной машиной Java* (Java Virtual Machine – JVM), а для C# — средством *Common Language Runtime* (CLR). Следовательно, Java-программа сможет выполняться в любой среде, содержащей JVM, а C#-программа — в любой среде, в которой реализовано средство CLR.

Поскольку специальные операционные системы для выполнения Java- и C#-кода занимают промежуточное местоположение между программой и ЦП, выполнение Java- и C#-программ сопряжено с расходом определенных системных ресурсов, что совершенно излишне при выполнении C++-программ. Вот поэтому C++-программы обычно выполняются быстрее, чем эквивалентные программы, написанные на Java и C#.

И последнее. Языки C++, Java и C# предназначены для решения различных классов задач. Поэтому вопрос “Какой язык лучше?” поставлен некорректно. Уместнее сформулировать вопрос иначе: “Какой язык наиболее подходит для решения данной задачи?”. 1  
Основы C++



### Вопросы для текущего контроля

1. От какого языка программирования произошел C++?
2. Какой основной фактор обусловил создание C++?
3. Верно ли, что C++ является предком языков Java и C#?\*

ВАЖНО!

## 1.3. Объектно-ориентированное программирование

Основополагающими для разработки C++ стали принципы объектно-ориентированного программирования (ООП). Именно ООП явилось толчком для создания C++. А раз так, то, прежде чем мы напишем самую простую C++-программу, важно понять, что собой представляют принципы ООП.

Объектно-ориентированное программирование объединило лучшие идеи структурированного с рядом мощных концепций, которые способствуют более эффективной организации программ. В самом общем смысле любую программу можно организовать одним из двух способов: опираясь на **код** (действия) или на **данные** (информация, на которую направлены эти действия). При использовании лишь методов структурированного программирования программы обычно опираются на код. Такой подход можно выразить в использовании “кода, действующего на данные”.

Механизм объектно-ориентированного программирования основан на выборе второго способа. В этом случае ключевым принципом организации программ является использование “данных, управляющих доступом к коду”. В любом объектно-ориентированном языке программирования определяются данные и процедуры, которые разрешается применить к этим данным. Таким образом, тип данных в точности определяет, операции какого вида можно применить к этим данным.

1. Язык C++ произошел от C.
2. Основной фактор создания C++ – повышение сложности программ.
3. Верно: C++ действительно является родительским языком для Java и C#.

Для поддержки принципов объектно-ориентированного программирования все языки ООП, включая C++, характеризуются следующими общими свойствами: инкапсуляцией, полиморфизмом и наследованием. Рассмотрим кратко каждое из этих свойств.

## Инкапсуляция

**Инкапсуляция** — это такой механизм программирования, который связывает воедино код и данные, им обрабатываемые, чтобы обезопасить их как от внешнего вмешательства, так и от неправильного использования. В объектно-ориентированном языке код и данные могут быть связаны способом, при котором создается самодостаточный *черный ящик*. В этом “ящике” содержатся все необходимые (для обеспечения самостоятельности) данные и код. При таком связывании кода и данных создается объект, т.е. *объект* — это конструкция, которая поддерживает инкапсуляцию.

### Спросим у опытного программиста

**Вопрос.** Я слышал, что термин “метод” применяется к подпрограмме. Тогда правда ли то, что метод — это то же самое, что и функция?

**Ответ.** В общем смысле ответ: да. Термин “метод” популяризовался с появлением языка Java. То, что C++-программист называет функцией, Java-программист называет методом. C#-программисты также используют термин “метод”. Ввиду такой широкой популярности этот термин все чаще стали применять и к C++-функции.

Внутри объекта код, данные или обе эти составляющие могут быть закрытыми в “рамках” этого объекта или открытыми. *Закрытый* код (или данные) известен и доступен только другим частям того же объекта. Другими словами, к закрытому коду или данным не может получить доступ та часть программы, которая существует вне этого объекта. *Открытый* код (или данные) доступен любым другим частям программы, даже если они определены в других объектах. Обычно открытые части объекта используются для предоставления управляемого интерфейса с закрытыми элементами объекта.

Базовой единицей инкапсуляции является класс. Класс определяет новый тип данных, который задает формат *объекта*. Класс включает как данные, так и код, предназначенный для выполнения над этими данными. Следовательно, *класс связывает данные с кодом*. В C++ спецификация класса используется для по-

строения объектов. *Объекты* — это экземпляры класса. По сути, класс представляет собой набор планов, которые определяют, как строить объект.

В классе данные объявляются в виде *переменных*, а код оформляется в виде *функций* (подпрограмм). Функции и переменные, составляющие класс, называются его *членами*. Таким образом, переменная, объявленная в классе, называется *членом данных*, а функция, объявленная в классе, называется *функцией-членом*. Иногда вместо термина *член данных* используется термин *переменная экземпляра* (или *переменная реализации*).

## ПОЛИМОРФИЗМ

*Полиморфизм* (от греческого слова *polymorphism*, означающего “много форм”) — это свойство, позволяющее использовать один интерфейс для целого класса действий. В качестве простого примера полиморфизма можно привести руль автомобиля. Для руля (т.е. интерфейса) безразлично, какой тип рулевого механизма используется в автомобиле. Другими словами, руль работает одинаково, независимо от того, оснащен ли автомобиль рулевым управлением прямого действия (без усилителя), рулевым управлением с усилителем или механизмом реечной передачи. Если вы знаете, как обращаться с рулем, вы сможете вести автомобиль любого типа.

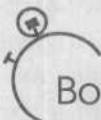
Тот же принцип можно применить к программированию. Рассмотрим, например, стек, или список, добавление и удаление элементов к которому осуществляется по принципу “последним прибыл — первым обслужен”. У вас может быть программа, в которой используются три различных типа стека. Один стек предназначен для целочисленных значений, второй — для значений с плавающей точкой и третий — для символов. Алгоритм реализации всех стеков — один и тот же, несмотря на то, что в них хранятся данные различных типов. В необъектно-ориентированном языке программисту пришлось бы создать три различных набора подпрограмм обслуживания стека, причем подпрограммы должны были бы иметь различные имена, а каждый набор — собственный интерфейс. Но благодаря полиморфизму в C++ можно создать один общий набор подпрограмм (один интерфейс), который подходит для всех трех конкретных ситуаций. Таким образом, зная, как использовать один стек, вы можете использовать все остальные.

В более общем виде концепция полиморфизма выражается фразой “один интерфейс — много методов”. Это означает, что для группы связанных действий можно использовать один обобщенный интерфейс. Полиморфизм позволяет понизить уровень сложности за счет возможности применения одного и того же интерфейса для задания *общего класса действий*. Выбор же *конкретного действия* (т.е. функции) применительно к той или иной ситуации ложится “на плечи” компилятора. Вам, как программисту, не нужно делать этот выбор вручную. Ваша задача — использовать общий интерфейс.

## Наследование

*Наследование* – это процесс, благодаря которому один объект может приобретать свойства другого. Благодаря наследованию поддерживается концепция иерархической классификации. В виде управляемой иерархической (нисходящей) классификации организуется большинство областей знаний. Например, яблоки *Красный Делишес* являются частью классификации *яблоки*, которая в свою очередь является частью класса *фрукты*, а тот – частью еще большего класса *пища*. Таким образом, класс *пища* обладает определенными качествами (съедобность, питательность и пр.), которые применимы и к подклассу *фрукты*. Помимо этих качеств, класс *фрукты* имеет специфические характеристики (сочность, сладость и пр.), которые отличают их от других пищевых продуктов. В классе *яблоки* определяются качества, специфичные для яблок (растут на деревьях, не тропические и пр.). Класс *Красный Делишес* наследует качества всех предыдущих классов и при этом определяет качества, которые являются уникальными для этого сорта яблок.

Если не использовать иерархическое представление признаков, для каждого объекта пришлось бы в явной форме определить все присущие ему характеристики. Но благодаря наследованию объекту нужно доопределить только те качества, которые делают его уникальным внутри его класса, поскольку он (объект) наследует общие атрибуты своего родителя. Следовательно, именно механизм наследования позволяет одному объекту представлять конкретный экземпляр более общего класса.



### Вопросы для текущего контроля

1. Назовите принципы ООП.
2. Что является основной единицей инкапсуляции в C++?
3. Какой термин в C++ используется для обозначения подпрограммы?\*

ВАЖНО!

## 1.4. Создание, компиляция и выполнение C++-программ

Пора приступить к программированию. Для этого рассмотрим первую простую C++-программу. Начнем с ввода текста, а затем перейдем к ее компиляции и выполнению.

1. Принципами ООП являются инкапсуляция, полиморфизм и наследование.
2. Базовой единицей инкапсуляции в C++ является класс.
3. Для обозначения подпрограммы в C++ используется термин “функция”.

## Спросим у опытного программиста

**Вопрос.** Вы утверждаете, что объектно-ориентированное программирование (ООП) представляет собой эффективный способ управления большими по объему программами. Но не означает ли это, что для относительно небольших программ использование ООП может внести неоправданные затраты системных ресурсов? И справедливо ли это для C++?

**Ответ.** Нет. Ключевым моментом для понимания применения ООП в C++ является то, что оно позволяет писать объектно-ориентированные программы, но не является обязательным требованием. В этом и заключается одно из важных отличий C++ от языков Java и C#, которые предполагают использование строгой объектной модели в каждой программе. C++ просто предоставляет программисту возможности ООП. Более того, по большей части объектно-ориентированные свойства C++ при выполнении программы совершенно незаметны, поэтому вряд ли стоит говорить о каких-то ощутимых затратах системных ресурсов.

/\*

Это простая C++-программа.

Назовите этот файл Sample.cpp.

\*/

```
#include <iostream>
using namespace std;

// C++-программа начинается с функции main().
int main()
{
    cout << "C++-программирование - это сила!";

    return 0;
}
```

Итак, вы должны выполнить следующие действия.

1. Ввести текст программы.
2. Скомпилировать ее.
3. Выполнить.

Прежде чем приступать к выполнению этих действий, необходимо определить два термина: исходный код и объектный код. *Исходный код* — это версия программы, которую может читать человек. Она сохраняется в текстовом файле. Приведенный выше листинг — это пример исходного кода. Выполняемая версия программы (она создается компилятором) называется *объектным* или *выполняемым кодом*.

## Ввод текста программы

Программы, представленные в этой книге, можно загрузить с Web-сайта компании Osborne с адресом: [www.osborne.com](http://www.osborne.com). При желании вы можете ввести текст программ вручную (это иногда даже полезно: при вводе кода вручную вы запоминаете ключевые моменты). В этом случае необходимо использовать какой-нибудь текстовый редактор (например WordPad, если вы работаете в ОС Windows), а не текстовой процессор (word processor). Дело в том, что при вводе текста программ должны быть созданы исключительно текстовые файлы, а не файлы, в которых вместе с текстом (при использовании текстового процессора) сохраняется информация о его форматировании. Помните, что информация о форматировании помешает работе C++-компилятора.

Имя файла, который будет содержать исходный код программы, формально может быть любым. Но C++-программы обычно хранятся в файлах с расширением .cpp. Поэтому называйте свои C++-программы любыми именами, но в качестве расширения используйте .cpp. Например, назовите нашу первую программу Sample.cpp (это имя будет употребляться в дальнейших инструкциях), а для других программ (если не будет специальных указаний) выбирайте имена по своему усмотрению.

## Компилирование программы

Способ компиляции программы Sample.cpp зависит от используемого компилятора и выбранных опций. Более того, многие компиляторы, например Visual C++ (Microsoft) и C++ Builder (Borland), предоставляют два различных способа компиляции программ: с помощью компилятора командной строки и интегрированной среды разработки (Integrated Development Environment — IDE). Поэтому для компилирования C++-программ невозможно дать универсальные инструкции, которые подойдут для всех компиляторов. Это значит, что вы должны следовать инструкциям, приведенным в сопроводительной документации, прилагаемой к вашему компилятору.

Но, если вы используете такие популярные компиляторы, как Visual C++ и C++ Builder, то проще всего в обоих случаях компилировать и выполнять программы, приведенные в этой книге, с использованием компиляторов командной

строки. Например, чтобы скомпилировать программу Sample.cpp, используя Visual C++, введите следующую командную строку:

```
C:\...>cl -GX Sample.cpp
```

Опция -GX предназначена для повышения качества компиляции. Чтобы использовать компилятор командной строки Visual C++, необходимо выполнить пакетный файл VCVARS32.bat, который входит в состав Visual C++. (В среде Visual Studio .NET можно перейти в режим работы по приглашению на ввод команды, который активизируется выбором команды Microsoft Visual Studio .NET⇒Visual Studio .NET Tools⇒Visual Studio .NET Command Prompt из меню Пуск⇒Программы). Чтобы скомпилировать программу Sample.cpp, используя C++ Builder, введите такую командную строку:

```
C:\...>bcc32 Sample.cpp
```

В результате работы C++-компилятора получается выполняемый объектный код. Для Windows-среды выполняемый файл будет иметь то же имя, что и исходный, но другое расширение, а именно расширение .exe. Итак, выполняемая версия программы Sample.cpp будет храниться в файле Sample.exe.

## Выполнение программы

Скомпилированная программа готова к выполнению. Поскольку результатом работы C++-компилятора является выполняемый объектный код, то для запуска программы в качестве команды достаточно ввести ее имя в режиме работы по приглашению. Например, чтобы выполнить программу Sample.exe, используйте эту командную строку:

```
C:\...>Sample.cpp
```

Результаты выполнения этой программы таковы:

C++-программирование – это сила!

Если вы используете интегрированную среду разработки, то выполнить программу можно путем выбора из меню команды Run (Выполнить). Безусловно, более точные инструкции приведены в сопроводительной документации, прилагаемой к вашему компилятору. Но, как упоминалось выше, проще всего компилировать и выполнять приведенные в этой книге программы с помощью командной строки.

Необходимо отметить, что все эти программы представляют собой консольные приложения, а не приложения, основанные на применении окон, т.е. они выполняются в сеансе приглашения на ввод команды (Command Prompt). При этом вам, должно быть, известно, что язык C++ не просто подходит для Windows-программирования, C++ – основной язык, применяемый в разработке Windows-приложений. Однако ни одна из программ, представленных в этой книге, не использует

графический интерфейс пользователя (*graphics user interface – GUI*). Дело в том, что Windows — довольно сложная среда для написания программ, включающая множество второстепенных тем, не связанных напрямую с языком C++. Для создания Windows-программ, которые демонстрируют возможности C++, потребовалось бы написать сотни строк кода. В то же время консольные приложения гораздо короче графических и лучше подходят для обучения программированию. Освоив C++, вы сможете без проблем применить свои знания в сфере создания Windows-приложений.

## Построчный “разбор полетов” первого примера программы

Несмотря на то что программа `Sample.cpp` довольно мала по размеру, она, тем не менее, содержит ключевые средства, характерные для всех C++-программ. Поэтому мы подробно рассмотрим каждую ее строку. Итак, наша программа начинается с таких строк.

```
/*
Это простая C++-программа.
```

Назовите этот файл `Sample.cpp`.

Это — *комментарий*. Подобно большинству других языков программирования, C++ позволяет вводить в исходный код программы комментарии, содержание которых компилятор игнорирует. С помощью комментариев описываются или разъясняются действия, выполняемые в программе, и эти разъяснения предназначаются для тех, кто будет читать исходный код. В данном случае комментарий просто идентифицирует программу. Конечно, в реальных приложениях комментарии используются для разъяснения особенностей работы отдельных частей программы или конкретных действий программных средств. Другими словами, вы можете использовать комментарии для детального описания всех (или некоторых) ее строк.

В C++ поддерживается два типа комментариев. Первый, показанный в начале рассматриваемой программы, называется *многострочным*. Комментарий этого типа должен начинаться символами `/*` (косая черта и “звездочка”) и заканчиваться ими же, но переставленными в обратном порядке `(* /)`. Все, что находится между этими парами символов, компилятор игнорирует. Комментарий этого типа, как следует из его названия, может занимать несколько строк. Второй тип комментариев (*однострочный*) мы рассмотрим чуть ниже.

Приведем здесь следующую строку программы.

```
#include <iostream>
```

В языке C++ определен ряд **заголовков** (header), которые обычно содержат информацию, необходимую для программы. В нашу программу включен заголовок `<iostream>` (он используется для поддержки C++-системы ввода-вывода), который представляет собой внешний исходный файл, помещаемый компилятором в начало программы с помощью директивы `#include`. Ниже в этой книге мы ближе познакомимся с заголовками и узнаем, почему они так важны.

Рассмотрим следующую строку программы:

```
using namespace std;
```

Эта строка означает, что компилятор должен использовать пространство имен `std`. Пространства имен — относительно недавнее дополнение к языку C++. Подробнее о них мы поговорим позже, а пока ограничимся их кратким определением. *Пространство имен* (namespace) создает декларативную область, в которой могут размещаться различные элементы программы. Пространство имен позволяет хранить одно множество имен отдельно от другого. С помощью этого средства можно упростить организацию больших программ. Ключевое слово `using` информирует компилятор об использовании заявленного пространства имен (в данном случае `std`). Именно в пространстве имен `std` объявлена вся библиотека стандарта C++. Таким образом, используя пространство имен `std`, вы упрощаете доступ к стандартной библиотеке языка. (Поскольку пространства имен — относительно новое средство, старые компиляторы могут его не поддерживать. Если вы используете старый компилятор, обратитесь к приложению Б, чтобы узнать, как быть в этом случае.)

Очередная строка нашей программе представляет собой *однострочный комментарий*.

```
// C++-программа начинается с функции main().
```

Так выглядит комментарий второго типа, поддерживаемый в C++. Однострочный комментарий начинается с пары символов `//` и заканчивается в конце строки. Как правило, программисты используют многострочные комментарии для подробных и потому более простиенных разъяснений, а однострочные — для кратких (построчных) описаний инструкций или назначения переменных. Вообще-то, характер использования комментариев — личное дело программиста.

Перейдем к следующей строке.

```
int main()
```

Как сообщается в только что рассмотренном комментарии, именно с этой строки и начинается выполнение программы.

Все C++-программы состоят из одной или нескольких функций. (Как упоминалось выше, под *функцией* мы понимаем подпрограмму.) Каждая C++-функция имеет имя, и только одна из них (ее должна включать каждая C++-программа) называется `main()`. Выполнение C++-программы начинается и заканчивается

(в большинстве случаев) выполнением функции `main()`. (Точнее, C++-программа начинается с вызова функции `main()` и обычно заканчивается возвратом из функции `main()`.) Открытая фигурная скобка на следующей (после `int main()`) строке указывает на начало кода функции `main()`. Ключевое слово `int` (сокращение от слова *integer*), стоящее перед именем `main()`, означает тип данных для значения, возвращаемого функцией `main()`. Как вы скоро узнаете, C++ поддерживает несколько встроенных типов данных, и `int` — один из них.

Рассмотрим очередную строку программы:

```
cout << "C++-программирование - это сила!";
```

Это инструкция вывода данных на консоль. При ее выполнении на экране компьютера отобразится сообщение `C++-программирование - это сила!`. В этой инструкции используется оператор вывода `<<`. Он обеспечивает вывод выражения, стоящего с правой стороны, на устройство, указанное с левой. Слово `cout` представляет собой встроенный идентификатор (составленный из частей слов *console output*), который в большинстве случаев означает экран компьютера. Итак, рассматриваемая инструкция обеспечивает вывод заданного сообщения на экран. Обратите внимание на то, что эта инструкция завершается точкой с запятой. В действительности все выполняемые C++-инструкции завершаются точкой с запятой.

Сообщение `"C++-программирование - это сила!"` представляет собой *строку*. В C++ под строкой понимается последовательность символов, заключенная в двойные кавычки. Как вы увидите, строка в C++ — это один из часто используемых элементов языка.

А этой строкой завершается функция `main()`:

```
return 0;
```

При ее выполнении функция `main()` возвращает вызывающему процессу (в роли которого обычно выступает операционная система) значение 0. Для большинства операционных систем нулевое значение, которое возвращает эта функция, свидетельствует о нормальном завершении программы. Другие значения могут означать завершение программы в связи с какой-нибудь ошибкой. Слово `return` относится к числу ключевых и используется для возврата значения из функции. При нормальном завершении (т.е. без ошибок) все ваши программы должны возвращать значение 0.

Закрывающая фигурная скобка в конце программы формально завершает ее.

## Обработка синтаксических ошибок

Ведите текст только что рассмотренной программы (если вы еще не сделали это), скомпилируйте ее и выполните. Каждому программисту известно, насколько легко

при вводе текста программы в компьютер вносятся случайные ошибки (опечатки). К счастью, при попытке скомпилировать такую программу компилятор "просигналит" сообщением о наличии *синтаксических ошибок*. Большинство C++-компиляторов попытаются "увидеть" смысл в исходном коде программы, независимо от того, что вы ввели. Поэтому сообщение об ошибке не всегда отражает истинную причину проблемы. Например, если в предыдущей программе случайно опустить открывающую фигурную скобку после имени функции `main()`, компилятор укажет в качестве источника ошибки инструкцию `cout`. Поэтому при получении сообщения об ошибке просмотрите две-три строки кода, непосредственно предшествующие строке с "обнаруженной" ошибкой. Ведь иногда компилятор начинает "чуять недоброе" только через несколько строк после реального местоположения ошибки.

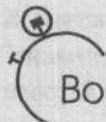
### Спросим у опытного программиста

**Вопрос.** *Помимо сообщений об ошибках мой C++-компилятор выдает в качестве результатов своей работы и предупреждения. Чем же предупреждения отличаются от ошибок и как следует реагировать на них?*

**Ответ.** Действительно, многие C++-компиляторы выдают в качестве результатов своей работы не только сообщения о неисправимых синтаксических ошибках, но и предупреждения (warning) различных типов. Если компилятор "уверен" в некорректности программного кода (например, инструкция не завершается точкой с запятой), он сигнализирует об ошибке, а если у него есть лишь "подозрение" на некорректность при видимой правильности с точки зрения синтаксиса, то он выдает предупреждение. Тогда программист сам должен оценить, насколько справедливы подозрения компилятора.

Предупреждения также можно использовать (по желанию программиста) для информирования о применении в коде неэффективных конструкций или устаревших средств. Компиляторы позволяют выбирать различные опции, которые могут информировать об интересующих вас вещах. Программы, приведенные в этой книге, написаны в соответствии со стандартом C++ и при корректном вводе не должны генерировать никаких предупреждающих сообщений.

Для примеров этой книги достаточно использовать обычную настройку компилятора. Но вам все же имеет смысл заглянуть в прилагаемую к компилятору документацию и поинтересоваться, какие возможности по управлению процессом компиляции есть в вашем распоряжении. Многие компиляторы довольно "интеллектуальны" и могут помочь в обнаружении неочевидных ошибок еще до того, как они перерастут в большие проблемы. Знание принципов, используемых компилятором при составлении отчета об ошибках, стоит затрат времени и усилий, которые потребуются от программиста на их освоение.



### Вопросы для текущего контроля

1. С чего начинается выполнение C++-программы?
2. Что такое cout?
3. Какое действие выполняет инструкция #include <iostream>?\*

**ВАЖНО!**

## 1.5. Использование переменных

Возможно, самой важной конструкцией в любом языке программирования является переменная. *Переменная* – это именованная область памяти, в которой могут храниться различные значения. При этом значение переменной во время выполнения программы можно изменить. Другими словами, содержимое переменной изменяемо, а не фиксированно.

В следующей программе создается переменная с именем length, которой присваивается значение 7, а затем на экране отображается сообщение Значение переменной length равно 7.

```
// Использование переменной.
```

```
#include <iostream>
using namespace std;

int main()
{
    int length; // ← Здесь объявляется переменная.

    length = 7; // Переменной length присваивается число 7.

    cout << "Значение переменной length равно ";
    cout << length; // Отображается значение переменной
                     // length, т.е. число 7.
```

1. Любая C++-программа начинает выполнение с функции main().
2. Слово cout представляет собой встроенный идентификатор, который имеет отношение к выводу данных на консоль.
3. Она включает в код программы заголовок <iostream>, который поддерживает функционирование системы ввода-вывода.

```
    return 0;  
}
```

Как упоминалось выше, для C++-программ можно выбирать любые имена. Тогда при вводе текста этой программы дадим ей, скажем, имя `VarDemo.cpp`.

Что же нового в этой программе? Во-первых, инструкция

```
int length; // Здесь объявляется переменная.
```

объявляет переменную с именем `length` целочисленного типа. В C++ все переменные должны быть объявлены до их использования. В объявлении переменной помимо ее имени необходимо указать, значения какого типа она может хранить. Тем самым объявляется *тип* переменной. В данном случае переменная `length` может хранить целочисленные значения, т.е. целые числа, лежащие в диапазоне  $-32\ 768 - 32\ 767$ . В C++ для объявления переменной целочисленного типа достаточно поставить перед ее именем ключевое слово `int`. Ниже вы узнаете, что C++ поддерживает широкий диапазон встроенных типов переменных. (Более того, C++ позволяет программисту определять собственные типы данных.)

Во-вторых, при выполнении следующей инструкции переменной присваивается конкретное значение:

```
length = 7; // Переменной length присваивается число 7.
```

Как отмечено в комментарии, здесь переменной `length` присваивается число 7. В C++ *оператор присваивания* представляется одиночным знаком равенства (`=`). Его действие заключается в копировании значения, расположенного справа от оператора, в переменную, указанную слева от него. После выполнения этой инструкции присваивания переменная `length` будет содержать число 7.

Обратите внимание на использование следующей инструкции для вывода значения переменной `length`:

```
cout << length; // Отображается число 7.
```

В общем случае для отображения значения переменной достаточно в инструкции `cout` поместить ее имя справа от оператора `<<`. Поскольку в данном конкретном случае переменная `length` содержит число 7, то оно и будет отображено на экране. Прежде чем переходить к следующему разделу, попробуйте присвоить переменной `length` другие значения (в исходном коде) и посмотрите на результаты выполнения этой программы после внесения изменений.

**ВАЖНО!**

## 1.6. Использование операторов

Подобно большинству других языков программирования, C++ поддерживает полный диапазон арифметических операторов, которые позволяют выполнять

действия над числовыми значениями, используемыми в программе. Приведем самые элементарные.

- + Сложение
- Вычитание
- \* Умножение
- / Деление

Действие этих операторов совпадает с действием аналогичных операторов в алгебре.

В следующей программе оператор "\*" используется для вычисления площади прямоугольника, заданного его длиной и шириной.

```
// Использование оператора.
```

```
#include <iostream>
using namespace std;

int main()
{
    int length; // Здесь объявляется переменная.
    int width; // Здесь объявляется вторая переменная.
    int area; // Здесь объявляется третья переменная.

    length = 7; // Число 7 присваивается переменной length.
    width = 5; // Число 5 присваивается переменной width.

    area = length * width; // ← Здесь вычисляется площадь
                           // прямоугольника, а результат произведения
                           // значений переменных length и width
                           // присваивается переменной area.

    cout << "Площадь прямоугольника равна ";
    cout << area; // Здесь отображается число 35.

    return 0;
}
```

В этой программе сначала объявляются три переменные `length`, `width` и `area`. Затем переменной `length` присваивается число 7, а переменной `width` – число 5. После этого вычисляется произведение значений переменных `length` и `width` (т.е. вычисляется площадь прямоугольника), а результат умножения присваивается переменной `area`. При выполнении программы отображает следующее.

Площадь прямоугольника равна 35.

В этой программе в действительности нет никакой необходимости в использовании переменной `area`. Поэтому предыдущую программу можно переписать следующим образом:

```
// Упрощенная версия программы вычисления площади
// прямоугольника.

#include <iostream>
using namespace std;

int main()
{
    int length; // Здесь объявляется переменная.
    int width; // Здесь объявляется вторая переменная.

    length = 7; // Число 7 присваивается переменной length.
    width = 5; // Число 5 присваивается переменной width.

    cout << "Площадь прямоугольника равна ";
    cout << length * width; // Здесь отображается число 35
                           // (результат выполнения
                           // операции length * width
                           // выводится на экран напрямую).

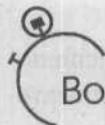
    return 0;
}
```

В этой версии площадь прямоугольника вычисляется в инструкции `cout` путем умножения значений переменных `length` и `width` с последующим выводом результата на экран.

Хочу обратить ваше внимание на то, что с помощью одной и той же инструкции можно объявить не одну, а сразу несколько переменных. Для этого достаточно разделить их имена запятыми. Например, переменные `length`, `width` и `area` можно было бы объявить таким образом.

```
int length, width, area; // Все переменные объявлены в
                         // одной инструкции.
```

В профессиональных C++-программах объявление нескольких переменных в одной инструкции – обычная практика.



## Вопросы для текущего контроля

1. Необходимо ли переменную объявлять до ее использования?
2. Покажите, как переменной `min` присвоить значение 0.
3. Можно ли в одной инструкции объявить сразу несколько переменных?\*

**ВАЖНО!**

### 1.7. Считывание данных с клавиатуры

В предыдущих примерах действия выполнялись над данными, которые явным образом были заданы в программе. Например, в программе вычисления площади выполнялось умножение значений сторон прямоугольника, причем эти множители (7 и 5) являлись частью самой программы. Безусловно, процесс вычисления площади прямоугольника не зависит от размера его сторон, поэтому наша программа была бы гораздо полезнее, если бы при ее выполнении пользователю предлагалось ввести размеры прямоугольника с клавиатуры.

Чтобы пользователь мог ввести данные в программу с клавиатуры, можно применить оператор “`>>`”. Это C++-оператор ввода. Для считывания данных с клавиатуры используется такой формат этого оператора.

```
cin >> var;
```

Здесь `cin` – еще один встроенный идентификатор. Он составлен из частей слов `console input` и автоматически поддерживается средствами C++. По умолчанию идентификатор `cin` связывается с клавиатурой, хотя его можно перенаправить и на другие устройства. Элемент `var` означает переменную (указанную с правой стороны от оператора “`>>`”), которая принимает вводимые данные.

Рассмотрим новую версию программы вычисления площади, которая позволяет пользователю вводить размеры сторон прямоугольника.

```
/*
```

```
    Интерактивная программа, которая вычисляет
    площадь прямоугольника.
```

```
*/
```

```
#include <iostream>
```

1. Да, в C++ переменные необходимо объявлять до их использования.
2. `min = 0;`
3. Да, в одной инструкции можно объявить сразу несколько переменных.

```
using namespace std;

int main()
{
    int length; // Здесь объявляется переменная.
    int width; // Здесь объявляется вторая переменная.

    cout << "Введите длину прямоугольника: ";
    cin >> length; // Ввод значения длины прямоугольника
                    // (т.е. значения переменной length)
                    // с клавиатуры.

    cout << "Введите ширину прямоугольника: ";
    cin >> width; // Ввод значения ширины прямоугольника
                   // (т.е. значения переменной width)
                   // с клавиатуры.

    cout << "Площадь прямоугольника равна ";
    cout << length * width; // Отображение значения площади.

    return 0;
}
```

Вот один из возможных результатов выполнения этой программы.

```
Введите длину прямоугольника: 8
Введите ширину прямоугольника: 3
Площадь прямоугольника равна 24
```

Обратите особое внимание на эти строки программы.

```
cout << "Введите длину прямоугольника: ";
cin >> length; // Ввод значения длины прямоугольника
```

Инструкция `cout` приглашает пользователя ввести данные. Инструкция `cin` считывает ответ пользователя, запоминая его в переменной `length`. Таким образом, значение, введенное пользователем (которое в данном случае представляет собой целое число), помещается в переменную, расположенную с правой стороны от оператора `>>` (в данном случае переменную `length`). После выполнения инструкции `cin` переменная `length` будет содержать значение длины прямоугольника. (Если же пользователь введет нечисловое значение, переменная `length` получит нулевое значение.) Инструкции, обеспечивающие ввод и считывание значения ширины прямоугольника, работают аналогично.

## Вариации на тему вывода данных

До сих пор мы использовали самые простые типы инструкций `cout`. Однако в C++ предусмотрены и другие варианты вывода данных. Рассмотрим два из них. Во-первых, с помощью одной инструкции `cout` можно выводить сразу несколько порций информации. Например, в программе вычисления площади для отображения результата использовались следующие две строки кода.

```
cout << "Площадь прямоугольника равна ";
cout << length * width;
```

Эти две инструкции можно заменить одной.

```
cout << "Площадь прямоугольника равна " << length * width;
```

Здесь в одной инструкции `cout` используется два оператора “`<<`”, которые позволяют сначала вывести строку “Площадь прямоугольника равна”, затем значение площади прямоугольника. В общем случае в одной инструкции `cout` можно соединять любое количество операторов вывода, предварив каждый элемент вывода “своим” оператором “`<<`”.

Во-вторых, до сих пор у нас не было необходимости в переносе выводимых данных на следующую строку, т.е. в выполнении последовательности команд “возврат каретки/перевод строки”. Но такая необходимость может появиться довольно скоро. В C++ упомянутая выше последовательность команд генерируется с помощью символа *новой строки*. Чтобы поместить этот символ в строку, используйте код `\n` (обратная косая черта и строчная буква “`n`”). Теперь нам остается на практике убедиться, как работает последовательность `\n`.

```
/*
```

В этой программе демонстрируется использование кода `\n`, который генерирует переход на новую строку.

```
*/
#include <iostream>
using namespace std;

int main()
{
    cout << "один\n";
    cout << "два\n";
    cout << "три";
    cout << "четыре";

    return 0;
}
```

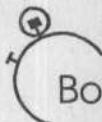
При выполнении эта программа генерирует такие результаты.

один

два

тричетыре

Символ новой строки можно размещать в любом месте строки, а не только в конце. Вам стоит на практике опробовать различные варианты применения кода `\n`, чтобы убедиться в том, что вам до конца понятно его назначение.



### Вопросы для текущего контроля

1. Какой оператор используется в C++ для ввода данных?
2. С каким устройством по умолчанию связан идентификатор `cin`?
3. Что означает код `\n?*`

## Познакомимся еще с одним типом данных

В предыдущих программах использовались переменные типа `int`. Однако переменные типа `int` могут содержать только целые числа. И поэтому их нельзя использовать для представления чисел с дробной частью. Например, переменная типа `int` может хранить число 18, но не значение 18,3. К счастью, в C++, кроме `int`, определены и другие типы данных. Для работы с числами, имеющими дробную часть, в C++ предусмотрено два типа данных с плавающей точкой: `float` и `double`. Они служат для представления значений с одинарной и двойной точностью соответственно. Чаще в C++-программах используется тип `double`.

Для объявления переменной типа `double` достаточно написать инструкцию, подобную следующей.

```
double result;
```

Здесь `result` представляет собой имя переменной, которая имеет тип `double`. Поэтому переменная `result` может содержать такие значения, как 88,56, 0,034 или -107,03.

1. Для ввода данных в C++ используется оператор “`>>`”.
2. По умолчанию идентификатор `cin` связан с клавиатурой.
3. Код `\n` означает символ новой строки.

Чтобы лучше понять разницу между типами данных `int` и `double`, рассмотрим следующую программу.

```
/*
Эта программа иллюстрирует различия между типами
данных int и double.

*/
#include <iostream>
using namespace std;

int main() {
    int ivar;      // Здесь объявляется переменная типа int.
    double dvar;  // Здесь объявляется переменная типа double.

    ivar = 100;   // Переменная ivar получает значение 100.

    dvar = 100.0; // Переменная dvar получает значение 100.0.

    cout << "Исходное значение переменной ivar: " << ivar
        << "\n";
    cout << "Исходное значение переменной dvar: " << dvar
        << "\n";

    cout << "\n"; // Выводим пустую строку.

    // А теперь делим оба значения на 3.
    ivar = ivar / 3;
    dvar = dvar / 3.0;

    cout << "Значение ivar после деления: " << ivar << "\n";
    cout << "Значение dvar после деления: " << dvar << "\n";

    return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

Исходное значение переменной ivar: 100

Исходное значение переменной dvar: 100

Значение ivar после деления: 33

Значение dvar после деления: 33.3333

Как видите, при делении значения переменной `ivar` на 3 выполняется целочисленное деление, в результате которого (33) теряется дробная часть. Но при делении на 3 значения переменной `dvar` дробная часть сохраняется.

В этой программе есть еще новая деталь. Обратите внимание на эту строку.

```
cout << "\n"; // Выводим пустую строку.
```

При выполнении этой инструкции происходит переход на новую строку. Используйте эту инструкцию в случае, когда нужно разделить выводимые данные пустой строкой.

## Спросим у опытного программиста

**Вопрос.** *Почему в C++ существуют различные типы данных для представления целых чисел и значений с плавающей точкой? Почему бы все числовые значения не представлять с помощью одного типа данных?*

**Ответ.** В C++ предусмотрены различные типы данных, чтобы программисты могли создавать эффективные программы. Например, вычисления с использованием целочисленной арифметики выполняются гораздо быстрее, чем вычисления, производимые над значениями с плавающей точкой. Таким образом, если вам не нужны значения с дробной частью, то вам и не стоит зря расходовать системные ресурсы, связанные с обработкой таких типов данных, как `float` или `double`. Кроме того, для хранения значений различных типов требуются разные по размеру области памяти. Поддерживая различные типы данных, C++ позволяет программисту выбрать наилучший вариант использования системных ресурсов. Наконец, для некоторых алгоритмов требуется использование данных конкретного типа. И потом, широкий диапазон встроенных C++-типов данных предоставляет программисту проявить максимальную гибкость при реализации решений разнообразных задач программирования.

## Проект 1.1. Преобразование футов в метры

FtоМ.cpp

Несмотря на то что в предыдущих примерах программ были продемонстрированы важные средства языка C++, они полезны больше с точки зрения теории. Поэтому важно закрепить даже те немногие сведения, которые вы уже перечерпнули о C++, при создании практических программ. В этом проекте мы напишем программу перевода футов в метры. Программа

должна предложить пользователю ввести значение в футах, а затем отобразить значение, преобразованное в метры.

Один метр равен приблизительно 3,28 футам. Следовательно, в программе мы должны использовать представление данных с плавающей точкой. Для выполнения преобразования в программе необходимо объявить две переменные типа `double`: одну для хранения значения в футах, а другую для хранения значения, преобразованного в метры.

### Последовательность действий

1. Создайте новый C++-файл с именем `FtoM.cpp`. (Конечно, вы можете выбрать для этого файла любое другое имя.)
2. Начните программу следующими строками, которые разъясняют назначение программы, включите заголовок `iostream` и укажите пространство имен `std`.

```
/*
Проект 1.1.
```

Эта программа преобразует футы в метры.

Назовите программу `FtoM.cpp`.

```
*/
```

```
#include <iostream>
using namespace std;
```

3. Начните определение функции `main()` с объявления переменных `f` и `m`.
  4. Добавьте код, предназначенный для ввода значения в футах.
- ```
int main() {
    double f; // содержит длину в футах
    double m; // содержит результат преобразования в метрах
```
5. Добавьте код, которые выполняет преобразование в метры и отображает результат.
- ```
m = f / 3.28; // преобразование в метры
cout << f << " футов равно " << m << " метрам.;"
```

6. Завершите программу следующим образом.

```
    return 0;
}
```

7. Ваша законченная программа должна иметь такой вид.

```
/*
```

Проект 1.1.

Эта программа преобразует футы в метры.

Назовите программу FtoM.cpp.

```
*/
```

```
#include <iostream>
using namespace std;

int main() {
    double f; // содержит длину в футах
    double m; // содержит результат преобразования в ме-
    трах

    cout << "Введите длину в футах: ";
    cin >> f; // считывание значения, выраженного в футах

    m = f / 3.28; // преобразование в метры
    cout << f << " футов равно " << m << " метрам. ";

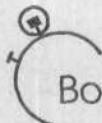
    return 0;
}
```

8. Скомпилируйте и выполните программу. Вот как выглядит один из возможных результатов ее выполнения.

Введите длину в футах: 5

5 футов равно 1.52439 метрам.

9. Попробуйте ввести другие значения. А теперь попытайтесь изменить программу, чтобы она преобразовывала метры в футы.



## Вопросы для текущего контроля

1. Какое C++-ключевое слово служит для объявления данных целочисленного типа?
2. Что означает слово `double`?
3. Как вывести пустую строку (или обеспечить переход на новую строку)?\*

**ВАЖНО!**

## 1.8. Использование инструкций управления `if` и `for`

Внутри функции выполнение инструкций происходит последовательно, сверху вниз. Однако, используя различные инструкции управления, поддерживаемые в C++, такой порядок можно изменить. Позже мы рассмотрим эти инструкции подробно, а пока кратко представим их, чтобы можно было воспользоваться ими для написания следующих примеров программ.

### Инструкция `if`

Можно избирательно выполнить часть программы, используя инструкцию управления условием `if`. Инструкция `if` в C++ действует подобно инструкции "IF", определенной в любом другом языке программирования (например C, Java и C#). Ее простейший формат таков:

```
if(условие) инструкция;
```

Здесь элемент `условие` – это выражение, которое при вычислении может оказаться равным значению ИСТИНА или ЛОЖЬ. В C++ ИСТИНА представляется ненулевым значением, а ЛОЖЬ – нулем. Если `условие`, или условное выражение, истинно, элемент `инструкция` выполнится, в противном случае – нет. Например, при выполнении следующего фрагмента кода на экране отобразится фраза 10 меньше 11, потому что число 10 действительно меньше 11.

```
if(10 < 11) cout << "10 меньше 11";
```

- 
1. Для объявления данных целочисленного типа служит ключевое слово `int`.
  2. Слово `double` является ключевым и используется для объявления данных с плавающей точкой двойной точности.
  3. Для вывода пустой строки используется код `\n`.

Теперь рассмотрим такую строку кода:

```
if(10 > 11) cout << "Это сообщение никогда не отобразится";
```

В этом случае число 10 не больше 11, поэтому инструкция `cout` не выполнится. Конечно же, операнды в инструкции `if` не должны всегда быть константами. Они могут быть переменными.

В C++ определен полный набор операторов отношений, которые используются в условном выражении. Перечислим их.

<code>==</code>	Равно
<code>!=</code>	Не равно
<code>&gt;</code>	Больше
<code>&lt;</code>	Меньше
<code>&gt;=</code>	Больше или равно
<code>&lt;=</code>	Меньше или равно

Обратите внимание на то, что для проверки на равенство используется двойной знак “равно”.

Теперь рассмотрим программу, в которой демонстрируется использование инструкции `if`.

```
// Демонстрация использования инструкции if.
```

```
#include <iostream>
using namespace std;

int main() {
    int a, b, c;

    a = 2;
    b = 3;

    if(a < b) cout << "a меньше b\n"; // ←———— Инструкция if

    // Эта инструкция ничего не отобразит.
    if(a == b) cout << "Этого вы не увидите.\n";

    cout << "\n";

    c = a - b; // Переменная с содержит -1.

    cout << "Переменная с содержит -1.\n";
```

```
if(c >= 0) cout << "Значение с неотрицательно.\n";
if(c < 0) cout << "Значение с отрицательно.\n";

cout << "\n";

c = b - a; // Теперь переменная с содержит 1.
cout << "Переменная с содержит 1.\n";
if(c >= 0) cout << "Значение с неотрицательно.\n";
if(c < 0) cout << "Значение с отрицательно.\n";

return 0;
}
```

При выполнении эта программа генерирует такие результаты.

a меньше b

Переменная с содержит -1.

Значение с отрицательно.

Переменная с содержит 1.

Значение с неотрицательно.

## Цикл for

Мы можем организовать повторяющееся выполнение одной и той же последовательности инструкций с помощью специальной конструкции, именуемой **циклом**. В C++ предусмотрены различные виды циклов, и одним из них является цикл **for**. Для тех, кто уже не новичок в программировании, отмечу, что в C++ цикл **for** работает так же, как в языках C# или Java. Рассмотрим простейший формат использования цикла **for**.

```
for(инициализация; условие; инкремент) инструкция;
```

Элемент **инициализация** обычно представляет собой инструкцию присваивания, которая устанавливает *управляющую переменную цикла* равной некоторому начальному значению. Эта переменная действует в качестве счетчика, который управляет работой цикла. Элемент **условие** представляет собой условное выражение, в котором тестируется значение управляющей переменной цикла. По результату этого тестирования определяется, выполнится цикл **for** еще раз или нет. Элемент **инкремент** – это выражение, которое определяет, как изменяется значение управляющей переменной цикла после каждой итерации (т.е. каждого повторения элемента **инструкция**). Обратите

внимание на то, что все эти элементы цикла `for` должны отделяться точкой с запятой. Цикл `for` будет выполняться до тех пор, пока вычисление элемента условие дает истинный результат. Как только это условное выражение станет ложным, цикл завершится, а выполнение программы продолжится с инструкции, следующей за циклом `for`.

Использование цикла `for` демонстрируется в следующей программе. Она выводит на экран числа от 1 до 100.

```
// Программа, иллюстрирующая цикл for.

#include <iostream>
using namespace std;

int main()
{
    int count;

    for(count=1; count <= 100; count=count+1) // цикл for
        cout << count << " ";

    return 0;
}
```

В этом цикле переменная `count` инициализируется значением 1. При каждом повторении тела цикла проверяется условие цикла.

`count <= 100;`

Если результат проверки истинен, на экран будет выведено значение переменной `count`, после чего управляющая переменная цикла увеличивается на единицу. Когда значение переменной `count` превысит число 100, проверяемое в цикле условие станет ложным, и цикл остановится.

В профессиональных C++-программах вы не встретите инструкции

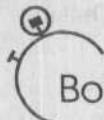
`count=count+1,`

или подобной ей, поскольку в C++ существует специальный оператор инкремента, который выполняет операцию увеличения значения на единицу более эффективно. Оператор инкремента обозначается в виде двух знаков “плюс” (`++`). Например, инструкцию `for` из приведенной выше программы с использованием оператора “`++`” можно переписать так.

```
for(count=1; count <= 100; count++) // цикл for
    cout << count << " ";
```

В остальной части книги для увеличения значения переменной на единицу мы будем использовать оператор инкремента.

В C++ также реализован оператор декремента, который обозначается двумя знаками “минус” (–). Он применяется для уменьшения значения переменной на единицу.



### Вопросы для текущего контроля

1. Каково назначение инструкции `if`?
2. Для чего предназначена инструкция `for`?
3. Какие операторы отношений реализованы в C++?\*

**ВАЖНО!**

## 1.9. Использование блоков кода

Одним из ключевых элементов C++ является *блок кода*. Блок – это логически связанная группа программных инструкций (т.е. одна или несколько инструкций), которые обрабатываются как единое целое. В C++ программный блок создается путем размещения последовательности инструкций между фигурными (открывающей и закрывающей) скобками. После создания блок кода становится логической единицей, которую разрешено использовать везде, где можно применять одну инструкцию. Например, блок кода может составлять тело инструкций `if` или `for`. Рассмотрим такую инструкцию `if`.

```
if(w < h) {
    v = w * h;
    w = 0;
}
```

Здесь, если значение переменной `w` меньше значения переменной `h`, будут выполнены обе инструкции, заключенные в фигурные скобки. Эти две инструкции (вместе с фигурными скобками) представляют блок кода. Они составляют логически неделимую группу: ни одна из этих инструкций не может выполниться без другой. Здесь важно понимать, что если вам нужно логически

1. Инструкция `if` – это инструкция условного выполнения части кода программы.
2. Инструкция `for` – одна из инструкций цикла в C++, которая предназначена для организации повторяющегося выполнения некоторой последовательности инструкций.
3. В C++ реализованы такие операторы отношений: `==`, `!=`, `<`, `>`, `<=` и `>=`.

связать несколько инструкций, это легко достигается путем создания блока. Кроме того, с использованием блоков кода многие алгоритмы реализуются более четко и эффективно.

Рассмотрим программу, в которой блок кода позволяет предотвратить деление на нуль.

// Демонстрация использования блока кода.

```
#include <iostream>
using namespace std;

int main() {
    double result, n, d;

    cout << "Введите делимое: ";
    cin >> n;

    cout << "Введите делитель: ";
    cin >> d;

    // Здесь инструкция if управляет целым блоком.
    if(d != 0) {
        cout << "Значение d не равно 0, деление осуществимо."
            << "\n";
        result = n / d;
        cout << n << " / " << d << " равно " << result;
    }

    return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

Введите делимое: 10

Введите делитель: 2

Значение d не равно 0, деление осуществимо.

10 / 2 равно 5

В этом случае инструкция if управляет целым блоком кода, а не просто одной инструкцией. Если управляющее условие инструкции if истинно (как в нашем примере), будут выполнены все три инструкции, составляющие блок. Попробуй-

те ввести нулевое значение для делителя и узнайте, как выполнится наша программа в этом случае. Убедитесь, что при вводе нуля *if*-блок будет опущен.

Как будет показано ниже, блоки кода имеют дополнительные свойства и способы применения. Но основная цель их существования — создавать логически связанные единицы кода.

### Спросим у опытного программиста

**Вопрос.** *Не страдает ли эффективность выполнения программы от применения блоков кода? Другими словами, не требуется ли дополнительное время компилятору для обработки кода, заключенного в фигурные скобки?*

**Ответ.** Нет. Обработка блоков кода не требует дополнительных затрат системных ресурсов. Более того, использование блоков кода (благодаря возможности упростить кодирование некоторых алгоритмов) позволяет увеличить скорость и эффективность выполнения программ.

## Точки с запятой и расположение инструкций

В C++ точка с запятой означает конец инструкции. Другими словами, каждая отдельная инструкция должна завершаться точкой с запятой. Как вы знаете, блок — это набор логически связанных инструкций, которые заключены между открывающей и закрывающей фигурными скобками. Блок *не* завершается точкой с запятой. Поскольку блок состоит из инструкций, каждая из которых завершается точкой с запятой, то в дополнительной точке с запятой нет никакого смысла. Признаком же конца блока служит закрывающая фигурная скобка.

Язык C++ не воспринимает конец строки в качестве признака конца инструкции. Таким признаком конца служит только точка с запятой. Поэтому для компилятора не имеет значения, в каком месте строки располагается инструкция. Например, с точки зрения C++-компилятора следующий фрагмент кода

```
x = y;
y = y+1;
cout << x << " " << y;
```

аналогичен такой строке:

```
x = y; y = y+1; cout << x << " " << y;
```

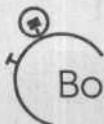
Более того, отдельные элементы любой инструкции можно располагать на отдельных строках. Например, следующий вариант записи инструкции абсолютно приемлем.

```
cout << "Это длинная строка. Сумма равна : "
    << a + b + c + d + e + f;
```

Подобное разбиение на строки часто используется, чтобы сделать программу более читабельной.

## Практика отступов

Рассматривая предыдущие примеры, вы, вероятно, заметили, что некоторые инструкции сдвинуты относительно левого края. C++ – язык свободной формы, т.е. его синтаксис не связан позиционными или форматными ограничениями. Это означает, что для C++-компилятора не важно, как будут расположены инструкции по отношению друг к другу. Но у программистов с годами выработался стиль применения отступов, который значительно повышает читабельность программ. В этой книге мы придерживаемся этого стиля и вам советуем поступать так же. Согласно этому стилю после каждой открывающей скобки делается очередной отступ вправо, а после каждой закрывающей скобки начало отступа возвращается к прежнему уровню. Существуют также некоторые определенные инструкции, для которых предусматриваются дополнительные отступы (о них речь впереди).



### Вопросы для текущего контроля

- Как обозначается блок кода?
- Что является признаком завершения инструкции в C++?
- Все C++-инструкции должны начинаться и завершаться на одной строке.  
Верно ли это?\*

## Проект 1.2. Построение таблицы преобразования футов в метры

`FtoMTable.cpp`

В этом проекте демонстрируется применение цикла `for`, инструкции `if` и блоков кода на примере создания про-

Проект  
1.2

Построение таблицы преобразования футов в метры

- Блок кода начинается с открывающей фигурной скобки (`{`), а оканчивается закрывающей фигурной скобкой (`}`).
- Признаком завершения инструкции в C++ является точка с запятой.
- Не верно. Отдельные элементы любой инструкции можно располагать на отдельных строках.

граммы вывода таблицы результатов преобразования футов в метры. Эта таблица начинается с одного фута и заканчивается 100 футами. После каждого 10 футов выводится пустая строка. Это реализуется с помощью переменной counter, в которой хранится количество выведенных строк. Обратите особое внимание на использование этой переменной.

### Последовательность действий

1. Создайте новый файл с именем FtoMTable.cpp.
2. Введите в этот файл следующую программу.

```
/*
Проект 1.2.
```

При выполнении этой программы выводится таблица результатов преобразования футов в метры.

Назовите эту программу FtoMTable.cpp.

```
*/
#include <iostream>
using namespace std;

int main() {
    double f; // содержит длину, выраженную в футах
    double m; // содержит результат преобразования в метры
    int counter; // счетчик строк

    counter = 0; // Начальная установка счетчика строк.

    for(f = 1.0; f <= 100.0; f++) {
        m = f / 3.28; // преобразуем в метры
        cout << f << " футов равно " << m << " метра.\n";

        counter++; // После каждой итерации цикла
                    // инкрементируем счетчик строк.

        // После каждой 10-й строки выводим пустую строку.
        if(counter == 10) { // Если счетчик строк досчитал
                            // до 10, выводим пустую строку.
            cout << "\n"; // выводим пустую строку
        }
    }
}
```

```
    counter = 0; // обнуляем счетчик строк
}
}

return 0;
}
```

- Обратите внимание на то, как используется переменная `counter` для вывода пустой строки после каждого десяти строк. Сначала (до входа в цикл `for`) она устанавливается равной нулю. После каждого преобразования переменная `counter` инкрементируется. Когда ее значение становится равным 10, выводится пустая строка, после чего счетчик строк снова обнуляется и процесс повторяется.
- Скомпилируйте и запустите программу. Ниже приведена часть таблицы, которую вы должны получить при выполнении программы.

1 футов равно 0.304878 метра.  
2 футов равно 0.609756 метра.  
3 футов равно 0.914634 метра.  
4 футов равно 1.21951 метра.  
5 футов равно 1.52439 метра.  
6 футов равно 1.82927 метра.  
7 футов равно 2.13415 метра.  
8 футов равно 2.43902 метра.  
9 футов равно 2.7439 метра.  
10 футов равно 3.04878 метра.

11 футов равно 3.35366 метра.  
12 футов равно 3.65854 метра.  
13 футов равно 3.96341 метра.  
14 футов равно 4.26829 метра.  
15 футов равно 4.57317 метра.  
16 футов равно 4.87805 метра.  
17 футов равно 5.18293 метра.  
18 футов равно 5.4878 метра.  
19 футов равно 5.79268 метра.  
20 футов равно 6.09756 метра.

21 футов равно 6.40244 метра.  
22 футов равно 6.70732 метра.  
23 футов равно 7.0122 метра.

24 футов равно 7.31707 метра.  
25 футов равно 7.62195 метра.  
26 футов равно 7.92683 метра.  
27 футов равно 8.23171 метра.  
28 футов равно 8.53659 метра.  
29 футов равно 8.84146 метра.  
30 футов равно 9.14634 метра.

31 футов равно 9.45122 метра.  
32 футов равно 9.7561 метра.  
33 футов равно 10.061 метра.  
34 футов равно 10.3659 метра.  
35 футов равно 10.6707 метра.  
36 футов равно 10.9756 метра.  
37 футов равно 11.2805 метра.  
38 футов равно 11.5854 метра.  
39 футов равно 11.8902 метра.  
40 футов равно 12.1951 метра.

5. Самостоятельно измените программу так, чтобы она выводила пустую строку через каждые 25 строк результатов.

**ВАЖНО!**

## 1.10. Понятие о функциях

Любая C++-программа составляется из “строительных блоков”, именуемых **функциями**. И хотя более детально мы будем рассматривать функции в модуле 5, сделаем здесь краткий обзор понятий и терминов, связанных с этой темой. Функция – это подпрограмма, которая содержит одну или несколько C++-инструкций и выполняет одну или несколько задач.

Каждая функция имеет имя, которое используется для ее вызова. Чтобы вызвать функцию, достаточно в исходном коде программы указать ее имя с парой круглых скобок. Своим функциям программист может давать любые имена, за исключением имени `main()`, зарезервированного для функции, с которой начинается выполнение программы. Например, мы назвали функцию именем `MyFunc`. Тогда для вызова функции `MyFunc` достаточно записать следующее.

```
MyFunc();
```

При вызове функции ей передается управление программой, в результате чего начинает выполняться код, составляющий тело функции. По завершении функции управление передается инициатору ее вызова.

Функции можно передать одно или несколько значений. Значение, передаваемое функции, называется *аргументом*. Таким образом, функции в C++ могут принимать один или несколько аргументов. Аргументы указываются при вызове функции между открывающей и закрывающей круглыми скобками. Например, если функция MyFunc () принимает один целочисленный аргумент, то, используя следующую запись, можно вызвать функцию MyFunc () со значением 2.

```
MyFunc(2);
```

Если функция принимает несколько аргументов, они разделяются запятыми. В этой книге под термином *список аргументов* понимаются аргументы, разделенные запятыми. Помните, что не все функции принимают аргументы. Если аргументы не нужны, то круглые скобки после имени функции остаются пустыми.

Функция может возвращать в вызывающий код значение. Некоторые функции не возвращают никакого значения. Значение, возвращаемое функцией, можно присвоить переменной в вызывающем коде, поместив обращение к функции с правой стороны от оператора присваивания. Например, если бы функция MyFunc () возвращала значение, мы могли бы вызвать ее таким образом.

```
x = MyFunc(2);
```

Эта инструкция работает так. Сначала вызывается функция MyFunc (). По ее завершении возвращаемое ею значение присваивается переменной x. Вызов функции можно также использовать в выражении. Рассмотрим пример.

```
x = MyFunc(2) + 10;
```

В этом случае значение, возвращаемое функцией, суммируется с числом 10, а результат сложения присваивается переменной x. И вообще, если в какой-либо инструкции встречается имя функции, эта функция автоматически вызывается, чтобы можно было получить (а затем и использовать) возвращаемое ею значение.

Итак, вспомним: *аргумент* — это значение, передаваемое функции. *Возвращаемое функцией значение* — это данные, передаваемые назад в вызывающий код.

Рассмотрим короткую программу, которая демонстрирует использование функции. Здесь для отображения абсолютного значения числа используется стандартная библиотечная (т.е. встроенная) функция abs (). Эта функция принимает один аргумент, преобразует его в абсолютное значение и возвращает результат.

```
// Использование функции abs().
```

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
```

```

{
    int result;

    result = abs(-10); // Вызывается функция abs(), а
                       // возвращаемое ею значение
                       // присваивается переменной result.

    cout << result;

    return 0;
}

```

Здесь функции `abs()` в качестве аргумента передается число `-10`. Функция `abs()`, приняв при вызове аргумент, возвращает его абсолютное значение. Полученное значение присваивается переменной `result`. Поэтому на экране отображается число `10`.

Обратите также внимание на то, что рассматриваемая программа включает заголовок `<cstdlib>`. Этот заголовок необходим для обеспечения возможности вызова функции `abs()`. Каждый раз, когда вы используете библиотечную функцию, в программу необходимо включать соответствующий заголовок.

В общем случае в своих программах вы будете использовать функции двух типов. К первому типу отнесем функции, написанные программистом (т.е. вами), и в качестве примера такой функции можно назвать функцию `main()`. Как вы узнаете ниже, реальные C++-программы содержат множество пользовательских функций.

Функции второго типа предоставляются компилятором. К этой категории функций принадлежит функция `abs()`. Обычно программы состоят как из функций, написанных программистами, так и из функций, предоставленных компилятором.

При обозначении функций в тексте этой книги используется соглашение (обычно соблюданное в литературе, посвященной языку программирования C++), согласно которому имя функции завершается парой круглых скобок. Например, если функция имеет имя `getval`, то ее упоминание в тексте обозначится как `getval()`. Соблюдение этого соглашения позволит легко отличать имена переменных от имен функций.

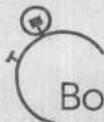
## Библиотеки C++

Как упоминалось выше, функция `abs()` не является частью языка C++, но ее “знает” каждый C++-компилятор. Эта функция, как и множество других, входит в состав *стандартной библиотеки*. В примерах этой книги мы подробно рассмотрим использование многих библиотечных функций C++.

В C++ определен довольно большой набор функций, которые содержатся в стандартной библиотеке. Эти функции предназначены для выполнения часто встречающихся задач, включая операции ввода-вывода, математические вычисления и обработку строк. При использовании программистом библиотечной функции компилятор автоматически связывает объектный код этой функции с объектным кодом программы.

Поскольку стандартная библиотека C++ довольно велика, в ней можно найти много полезных функций, которыми действительно часто пользуются программисты. Библиотечные функции можно применять подобно строительным блокам, из которых возводится здание. Чтобы не “изобретать велосипед”, ознакомьтесь с документацией на библиотеку используемого вами компилятора. Если вы сами напишете функцию, которая будет “переходить” с вами из программы в программу, ее также можно поместить в библиотеку.

Помимо библиотеки функций, каждый C++-компилятор также содержит *библиотеку классов*, которая является объектно-ориентированной библиотекой. Но, прежде чем мы сможем использовать библиотеку классов, нам нужно познакомиться с классами и объектами.



### Вопросы для текущего контроля

1. Что такое функция?
2. Функция вызывается с помощью ее имени. Верно ли это?
3. Что понимается под стандартной библиотекой функций C++?\*

**ВАЖНО!**

## 1.11. Ключевые слова C++

В стандарте C++ определено 63 ключевых слова. Они показаны в табл. 1.1. Эти ключевые слова (в сочетании с синтаксисом операторов и разделителей) образуют определение языка C++. В ранних версиях C++ определено ключевое слово `overload`, но теперь оно устарело. Следует иметь в виду, что в C++ различается строчное и прописное написание букв. Ключевые слова не являются исключением, т.е. все они должны быть написаны строчными буквами.

1. Функция – это подпрограмма, которая содержит одну или несколько C++-инструкций.
2. Верно. Чтобы вызвать функцию, достаточно в исходном коде программы указать ее имя.
3. Стандартная библиотека функций C++ – это коллекция функций, поддерживаемая всеми C++-компиляторами.

Таблица 1.1. Ключевые слова C++

asm	auto	bool	break
case	catch	char	class
const	const_class	continue	default
delete	do	double	dinamic_cast
else	enum	explicit	export
extern	false	float	for
friend	goto	if	inline
int	long	mutable	namespace
new	operator	private	protected
public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this,	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	wchar_t	while	

ВАЖНО

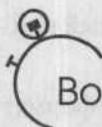
## 1.12 Идентификаторы

В C++ идентификатор представляет собой имя, которое присваивается функции, переменной или иному элементу, определенному пользователем. Идентификаторы могут состоять из одного или нескольких символов. Имена переменных должны начинаться с буквы или символа подчеркивания. Последующим символом может быть буква, цифра и символ подчеркивания. Символ подчеркивания можно использовать для улучшения читабельности имени переменной, например `line_count`. В C++ прописные и строчные буквы воспринимаются как различные символы, т.е. `myvar` и `MyVar` – это разные имена. В C++ нельзя использовать в качестве идентификаторов ключевые слова, а также имена стандартных функций (например, `abs`). Запрещено также использовать в качестве пользовательских имен встроенные идентификаторы (например, `cout`).

Вот несколько примеров допустимых идентификаторов.

Test	x	y2	MaxIncr
up	_top	my_var	simpleInterest23

Помните, что идентификатор не должен начинаться с цифры. Так, `980K` – недопустимый идентификатор. Конечно, вы вольны называть переменные и другие программные элементы по своему усмотрению, но обычно идентификатор отражает назначение или смысловую характеристику элемента, которому он принадлежит.



## Вопросы для текущего контроля

1. Какой из следующих вариантов представляет собой ключевое слово: `for`, `For` или `FOR`?
2. Символы какого типа может содержать C++-идентификатор?
3. Слова `index21` и `Index21` представляют собой один и тот же идентификатор?\*



## Тест для самоконтроля по модулю 1

1. Выше отмечалось, что C++ занимает центральное место в области современного программирования. Объясните это утверждение.
2. C++-компилятор генерирует объектный код, который непосредственно используется компилятором. Верно ли это?
3. Каковы три основных принципа объектно-ориентированного программирования?
4. С чего начинается выполнение C++-программы?
5. Что такое заголовок?
6. Что такое `<iostream>`? Для чего служит следующий код?  
`#include <iostream>`
7. Что такое пространство имен?
8. Что такое переменная?
9. Какое (какие) из следующих имен переменных недопустимо (недопустимы)?
  - `count`
  - `_count`
  - `count27`
  - `67count`
  - `if`

1. Ключевым словом здесь является вариант `for`. В C++ все ключевые слова пишутся с использованием строчных букв.
2. C++-идентификатор может содержать буквы, цифры и символ подчеркивания.
3. Нет, в C++ прописные и строчные буквы воспринимаются как различные символы.

10. Как создать односторочный комментарий? Как создать многострочный комментарий?
11. Представьте общий формат инструкции `if`. Представьте общий формат инструкции `for`.
12. Как создать блок кода?
13. Гравитация Луны составляет около 17% от гравитации Земли. Напишите программу, которая бы генерировала таблицу земных фунтов и эквивалентных значений, выраженных в лунном весе. Таблица должна содержать значения от 1 до 100 фунтов и включать пустые строки после каждой 25 строк результатов.
14. Год Юпитера (т.е. время, за которое Юпитер делает один полный оборот вокруг Солнца) составляет приблизительно 12 земных лет. Напишите программу, которая бы выполняла преобразования значений, выраженных в годах Юпитера, в значения, выраженные в годах Земли. Конечно же, здесь допустимо использование нецелых значений.
15. Что происходит с управлением программой при вызове функции?
16. Напишите программу, которая усредняет абсолютные значения пяти значений, введенных пользователем. Программа должна отображать результат.



Ответы на эти вопросы можно найти на Web-странице данной книги по адресу: <http://www.osborne.com>.

# Модуль 2

## Типы данных и операторы

- 2.1.** Типы данных C++
- 2.2.** Литералы
- 2.3.** Создание инициализированных переменных
- 2.4.** Арифметические операторы
- 2.5.** Операторы отношений и логические операторы
- 2.6.** Оператор присваивания
- 2.7.** Составные операторы присваивания
- 2.8.** Преобразование типов в операторах присваивания
- 2.9.** Преобразование типов в выражениях
- 2.10.** Приведение типов
- 2.11.** Использование пробелов и круглых скобок

Центральное место в описании языка программирования занимают типы данных и операторы. Эти элементы определяют ограничения языка и виды задач, к которым его можно применить. Нетрудно предположить, что язык C++ поддерживает богатый ассортимент как типов данных, так и операторов, что позволяет его использовать для решения широкого круга задач программирования.

Типы данных и операторы — это большая тема, которую мы начнем рассматривать с основных типов данных и наиболее часто используемых операторов. В этом модуле мы также обсудим подробнее такие элементы языка, как переменные и выражения.

## Почему типы данных столь важны

Тип переменных определяет операции, которые разрешены для выполнения над ними, и диапазон значений, которые могут быть сохранены с их помощью. В C++ определено несколько типов данных, и все они обладают уникальными характеристиками. Поэтому все переменные должны быть определены до их использования, а объявление переменной должно включать спецификатор типа. Без этой информации компилятор не сможет генерировать корректный код. В C++ не существует понятия “переменной без типа”.

Типы данных важны для C++-программирования еще и потому, что несколько базовых типов тесно связаны со “строительными блоками”, которыми оперирует компьютер: байтами и словами. Другими словами, C++ позволяет программисту действовать те же самые типы данных, которые использует сам центральный процессор. Именно поэтому с помощью C++ можно писать очень эффективные программы системного уровня.

**ВАЖНО!**

### 2.1. Типы данных C++

Язык C++ поддерживает встроенные типы данных, которые позволяют использовать в программах целые числа, символы, значения с плавающей точкой и булевы (логические) значения. Именно использование конкретных типов данных позволяет сохранять и обрабатывать в программах различные виды информации. Как будет показано ниже в этой книге, C++ предоставляет программисту возможность самому создавать типы данных, а не только использовать встроенные. Вы научитесь создавать такие типы данных, как классы, структуры и перечисления, но при этом помните, что все (даже очень сложные) новые типы данных состоят из встроенных.

В C++ определено семь основных типов данных. Их названия и ключевые слова, которые используются для объявления переменных этих типов, приведены в следующей таблице.

Тип	Название
char	Символьный
wchar_t	Символьный двубайтовый
int	Целочисленный
float	С плавающей точкой
double	С плавающей точкой двойной точности
bool	Логический (или булев)
void	Без значения

В C++ перед такими типами данных, как `char`, `int` и `double`, разрешается использовать *модификаторы*. Модификатор служит для изменения значения базового типа, чтобы он более точно соответствовал конкретной ситуации. Перечислим возможные модификаторы типов.

`signed`  
`unsigned`  
`long`  
`short`

Модификаторы `signed`, `unsigned`, `long` и `short` можно применять к целочисленным базовым типам. Кроме того, модификаторы `signed` и `unsigned` можно использовать с типом `char`, а модификатор `long` – с типом `double`. Все допустимые комбинации базовых типов и модификаторов приведены в табл. 2.1. В этой таблице также указаны гарантированные минимальные диапазоны представления для каждого типа, соответствующие C++-стандарту ANSI/ISO.

Минимальные диапазоны, представленные в табл. 2.1, важно понимать именно как *минимальные диапазоны* и никак иначе. Дело в том, что C++-компилятор может расширить один или несколько из этих минимумов (что и делается в большинстве случаев). Это означает, что диапазоны представления C++-типов данных зависят от конкретной реализации. Например, для компьютеров, которые используют арифметику дополнительных кодов (т.е. почти все современные компьютеры), целочисленный тип будет иметь диапазон представления чисел от -32 768 до 32 767. Однако во всех случаях диапазон представления типа `short int` является подмножеством диапазона типа `int`, диапазон представления которого в свою очередь является подмножеством диапазона типа `long int`. Аналогичными отношениями связаны и типы `float`, `double` и `long double`. В этом контексте термин *подмножество* означает более узкий или такой же диапазон. Таким образом, типы `int` и `long int` могут иметь одинаковые диапазоны, но диапазон типа `int` не может быть шире диапазона типа `long int`.

Таблица 2.1. Все допустимые комбинации числовых типов и их гарантированные минимальные диапазоны представления, соответствующие C++-стандарту ANSI/ISO

<i>Тип</i>	<i>Минимальный диапазон</i>
char	-128–127
unsigned char	0–255
signed char	-128–127
int	-32 768–32 767
unsigned int	0–65 535
signed int	Аналогичен типу int
short int	-32 768–32 767
unsigned short int	0–65 535
signed short int	Аналогичен типу short int
long int	-2 147 483 648–2 147 483 647
signed long int	Аналогичен типу long int
unsigned long int	0–4 294 967 295
float	1E-37–1E+37, с шестью значащими цифрами
double	1E-37–1E+37, с десятью значащими цифрами
long double	1E-37–1E+37, с десятью значащими цифрами

Поскольку стандарт C++ указывает только минимальный диапазон, который обязан соответствовать тому или иному типу данных, реальные диапазоны, поддерживаемые вашим компилятором, следует уточнить в соответствующей документации. Например, в табл. 2.2 указаны типичные размеры значений в битах и диапазоны представления для каждого C++-типа данных в 32-разрядной среде (например, в Windows XP).

## Целочисленный тип

Как вы узнали в модуле 1, переменные типа `int` предназначены для хранения целочисленных значений, которые не содержат дробной части. Переменные этого типа часто используются для управления циклами и условными инструкциями. Операции с `int`-значениями (поскольку они не имеют дробной части) выполняются намного быстрее, чем со значениями с плавающей точкой (вещественного типа).

А теперь подробно рассмотрим каждый тип в отдельности.

Поскольку целочисленный тип столь важен для программирования, в C++ определено несколько его разновидностей. Как показано в табл. 2.1, существуют короткие (`short int`), обычные (`int`) и длинные (`long int`) целочисленные значения. Кроме того, существуют их версии со знаком и без. Переменные целочисленного типа со знаком могут содержать как положительные, так и отрица-

тельные значения. По умолчанию предполагается использование целочисленного типа со знаком. Таким образом, указание модификатора `signed` избыточно (но допустимо), поскольку объявление по умолчанию и так предполагает значение со знаком. Переменные целочисленного типа без знака могут содержать только положительные значения. Для объявления целочисленной переменной без знака достаточно использовать модификатор `unsigned`.

Таблица 2.2. Типичные размеры значений в битах и диапазоны представления C++-типов данных в 32-разрядной среде

Тип	Размер в битах	Диапазон
<code>char</code>	8	-128–127
<code>unsigned char</code>	8	0–255
<code>signed char</code>	8	-128–127
<code>int</code>	32	-2 147 483 648–2 147 483 647
<code>unsigned int</code>	32	0–4 294 967 295
<code>signed int</code>	32	Аналогичен типу <code>int</code>
<code>short int</code>	16	-32 768–32 767
<code>unsigned short int</code>	16	0–65 535
<code>signed short int</code>	16	-32 768–32 767
<code>long int</code>	32	Аналогичен типу <code>int</code>
<code>signed long int</code>	32	Аналогичен типу <code>signed int</code>
<code>unsigned long int</code>	32	Аналогичен типу <code>unsigned int</code>
<code>float</code>	32	1,8E-38–3,4E+38
<code>double</code>	64	2,2E-308–1,8E+308
<code>long double</code>	64	2,2E-308–1,8E+308
<code>bool</code>	—	ИСТИНА или ЛОЖЬ
<code>w_char_t</code>	16	0–65 535

Различие между целочисленными значениями со знаком и без него заключается в интерпретации старшего разряда. Если задано целочисленное значение со знаком, C++-компилятор генерирует код с учетом того, что старший разряд значения используется в качестве *флага знака*. Если флаг знака равен 0, число считается положительным, а если он равен 1, — отрицательным. Отрицательные числа почти всегда представляются в *дополнительном коде*. Для получения дополнительного кода все разряды числа берутся в обратном коде, а затем полученный результат увеличивается на единицу. Наконец, флаг знака устанавливается равным 1.

Целочисленные значения со знаком используются во многих алгоритмах, но максимальное число, которое можно представить со знаком, составляет только

половину от максимального числа, которое можно представить без знака. Рассмотрим, например, максимально возможное 16-разрядное целое число (32 767):  
 0 1111111 11111111

Если бы старший разряд этого значения со знаком был установлен равным 1, то оно бы интерпретировалось как -1 (в дополнительном коде). Но если объявить его как `unsigned int`-значение, то после установки его старшего разряда в 1 мы получили бы число 65 535.

Чтобы понять различие в C++-интерпретации целочисленных значений со знаком и без него, выполним следующую короткую программу.

```
#include <iostream>

/* Эта программа демонстрирует различие между
   signed- и unsigned-значениями целочисленного типа.
 */

using namespace std;

int main()
{
    short int i; // короткое int-значение со знаком
    short unsigned int j; // короткое int-значение без знака

    j = 60000; // Число 60000 попадает в диапазон
                // представления типа short unsigned int, но
                // не попадает в диапазон представления типа
                // short signed int.
    i = j;      // Поэтому после присваивания числа 60000
                // переменной i оно будет интерпретироваться
                // как отрицательное.
    cout << i << " " << j;

    return 0;
}
```

При выполнении программа выведет два числа:

-5536 60000

Дело в том, что битовая комбинация, которая представляет число 60000 как короткое (`short`) целочисленное значение без знака, интерпретируется в качестве короткого `int`-значения со знаком как число -5536 (при 16-разрядном представлении).

В C++ предусмотрен сокращенный способ объявления `unsigned`, `short`- и `long`-значений целочисленного типа. Это значит, что при объявлении `int`-значений достаточно использовать слова `unsigned`, `short` и `long`, не указывая тип `int`, т.е. тип `int` подразумевается. Например, следующие две инструкции объявляют целочисленные переменные без знака.

```
unsigned x;
unsigned int y;
```

## СИМВОЛЫ

Переменные типа `char` предназначены для хранения ASCII-символов (например A, z или G) либо иных 8-разрядных величин. Чтобы задать символ, необходимо заключить его в одинарные кавычки. Например, после выполнения следующих двух инструкций

```
char ch;
ch = 'X';
```

переменной `ch` будет присвоена буква X.

Содержимое `char`-значения можно вывести на экран с помощью `cout`-инструкции. Вот пример.

```
cout << "Это содержится в переменной ch: " << ch;
```

При выполнении этой инструкции на экран будет выведено следующее.

Это содержится в переменной ch: X

Тип `char` может быть модифицирован с помощью модификаторов `signed` и `unsigned`. Стого говоря, только конкретная реализация определяет по умолчанию, каким будет `char`-объявление: со знаком или без него. Но для большинства компиляторов объявление типа `char` подразумевает значение со знаком. Следовательно, в таких средах использование модификатора `signed` для `char`-объявления также избыточно. В этой книге предполагается, что `char`-значения имеют знак.

Переменные типа `char` можно использовать не только для хранения ASCII-символов, но и для хранения числовых значений. Переменные типа `char` могут содержать "небольшие" целые числа в диапазоне -128–127 и поэтому их можно использовать вместо `int`-переменных, если вас устраивает такой диапазон представления чисел. Например, в следующей программе `char`-переменная используется для управления циклом, который выводит на экран алфавит английского языка.

```
// Эта программа выводит алфавит.
```

```
#include <iostream>
```

## 76 Модуль 2. Типы данных и операторы

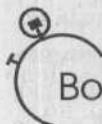
```
using namespace std;

int main()
{
    char letter; // Переменная letter типа char
                  // используется для управления циклом for.
                  ↓
    for(letter = 'A'; letter <= 'Z'; letter++)
        cout << letter;

    return 0;
}
```

Если цикл `for` вам покажется несколько странным, то учтите, что символ '`A`' представляется в компьютере как число `65`, а значения от '`A`' до '`Z`' являются последовательными и расположены в возрастающем порядке. При каждом проходе через цикл значение переменной `letter` инкрементируется. Таким образом, после первой итерации переменная `letter` будет содержать значение '`B`'.

Тип `wchar_t` предназначен для хранения символов, входящих в состав больших символьных наборов. Вероятно, вам известно, что в некоторых естественных языках (например китайском) определено очень большое количество символов, для которых 8-разрядное представление (обеспечиваемое типом `char`) весьма недостаточно. Для решения проблем такого рода в язык C++ был добавлен тип `wchar_t`, который вам пригодится, если вы планируете выходить со своими программами на международный рынок.



### Вопросы для текущего контроля

1. Назовите семь основных типов данных в C++.
  2. Чем различаются целочисленные значения со знаком и без?
  3. Можно ли использовать переменные типа `char` для представления небольших целых чисел?\*
- 
1. Семь основных типов: `char`, `wchar_t`, `int`, `float`, `double`, `bool` и `void`.
  2. Целочисленный тип со знаком позволяет хранить как положительные, так и отрицательные значения, а с помощью целочисленного типа без знака можно хранить только положительные значения.
  3. Да, переменные типа `char` можно использовать для представления небольших целых чисел.

## Спросим у опытного программиста

**Вопрос.** *Почему стандарт C++ определяет только минимальные диапазоны для встроенных типов, не устанавливая их более точно?*

**Ответ.** Не задавая точных размеров, язык C++ позволяет каждому компилятору оптимизировать типы данных для конкретной среды выполнения. В этом частично и состоит причина того, что C++ предоставляет возможности для создания высокопроизводительных программ. Стандарт ANSI/ISO просто заявляет, что встроенные типы должны отвечать определенным требованиям. Например, в нем сказано, что тип `int` "должен иметь естественный размер, предлагаемый архитектурой среды выполнения". Это значит, что в 16-разрядных средах для хранения значений типа `int` должно выделяться 16 бит, а в 32-разрядных — 32. При этом наименьший допустимый размер для целочисленных значений в любой среде должен составлять 16 бит. Поэтому, если вы будете писать программы, не "заступая" за пределы минимальных диапазонов, то они (программы) будут переносимы в другие среды. Одно замечание: каждый C++-компилятор указывает диапазон базовых типов в заголовке `<climits>`.

## Типы данных с плавающей точкой

К переменным типа `float` и `double` обращаются либо для обработки чисел с дробной частью, либо при необходимости выполнения операций над очень большими или очень малыми числами. Типы `float` и `double` различаются значением наибольшего (и наименьшего) числа, которые можно хранить с помощью переменных этих типов. Обычно тип `double` в C++ позволяет хранить число, приблизительно в десять раз превышающее значение типа `float`.

Чаще всего в профессиональных программах используется тип `double`. Дело в том, что большинство математических функций из C++-библиотеки используют `double`-значения. Например, функция `sqrt()` возвращает `double`-значение, которое равно квадратному корню из ее `double`-аргумента. Для примера рассмотрим программу, в которой функция `sqrt()` используется для вычисления длины гипотенузы по заданным длинам двух других сторон треугольника.

/\*

Здесь используется теорема Пифагора для вычисления длины гипотенузы по заданным длинам двух других сторон треугольника.

\*/

## 78 Модуль 2. Типы данных и операторы

```
#include <iostream>
#include <cmath>           // Этот заголовок необходим для
                           // вызова функции sqrt().
using namespace std;

int main() {
    double x, y, z;

    x = 5;
    y = 4;

    z = sqrt(x*x + y*y); // Функция sqrt() является частью
                           // математической C++-библиотеки.

    cout << "Гипотенуза равна " << z;

    return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

Гипотенуза равна 6.40312

Необходимо отметить, что поскольку функция `sqrt()` является частью стандартной C++-библиотеки функций, то для ее вызова в программу нужно включить заголовок `<cmath>`.

Тип `long double` позволяет работать с очень большими или очень маленькими числами. Он используется в программах научно-исследовательского характера, например, при анализе астрономических данных.

### Тип данных `bool`

Тип `bool` (относительно недавнее дополнение к C++) предназначен для хранения булевых (т.е. ИСТИНА/ЛОЖЬ) значений. В C++ определены две булевые константы: `true` и `false`, являющиеся единственными значениями, которые могут иметь переменные типа `bool`.

Важно понимать, как значения ИСТИНА/ЛОЖЬ определяются в C++. Один из фундаментальных принципов C++ состоит в том, что любое ненулевое значение интерпретируется как ИСТИНА, а нуль — как ЛОЖЬ. Этот принцип полностью согласуется с типом данных `bool`, поскольку любое ненулевое значение, используемое в булевом выражении, автоматически преобразуется в значение `true`, а нуль — в значение `false`. Обратное утверждение также справедливо: при

использовании в булевом выражении значение `true` преобразуется в число 1, а значение `false` – в число 0. Как будет показано в модуле 3, конвертируемость нулевых и ненулевых значений в их булевые эквиваленты особенно важна при использовании инструкций управления.

Использование типа `bool` демонстрируется в следующей программе.

```
// Демонстрация использования bool-значений.

#include <iostream>
using namespace std;

int main() {
    bool b;

    b = false;
    cout << "Значение переменной b равно " << b << "\n";

    b = true;
    cout << "Значение переменной b равно " << b << "\n";

    // Одно bool-значение может управлять if-инструкцией.
    if(b) cout << "Это выполнимо.\n";

    b = false;
    if(b) cout << "Это не выполнимо.\n";

    // Результатом применения оператора отношения
    // является true- или false-значение.
    cout << "10 > 9 равно " << (10 > 9) << "\n";

    return 0;
}
```

Результаты выполнения этой программы таковы.

Значение переменной b равно 0

Значение переменной b равно 1

Это выполнимо.

10 > 9 равно 1

В этой программе необходимо отметить три важных момента. Во-первых, как вы могли убедиться, при выводе на экран `bool`-значения отображается число 0

или 1. Как будет показано ниже в этой книге, вместо чисел можно вывести слова “false” и “true”.

Во-вторых, для управления `if`-инструкцией вполне достаточно самого значения `bool`-переменной, т.е. нет необходимости в использовании инструкции, подобной следующей.

```
if(b == true) ...
```

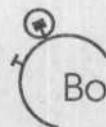
В-третьих, результатом выполнения операции сравнения, т.е. применения оператора отношения (например, “`>`”) является булево значение. Именно поэтому вычисление выражения `10 > 9` дает в результате число 1. При этом необходимость в дополнительных круглых скобках в инструкции вывода

```
cout << "10 > 9 равно " << (10 > 9) << "\n";
```

объясняется тем, что оператор “`<<`” имеет более высокий приоритет, чем оператор “`>`”.

## Тип `void`

Тип `void` используется для задания выражений, которые не возвращают значений. Это, на первый взгляд, может показаться странным, но подробнее тип `void` мы рассмотрим ниже в этой книге.



### Вопросы для текущего контроля

1. Каковы основные различия между типами `float` и `double`?
2. Какие значения может иметь переменная типа `bool`? В какое булево значение преобразуется нуль?
3. Что такое `void`\*?

- 
1. Основное различие между типами `float` и `double` заключается в величине значений, которые они позволяют хранить.
  2. Переменные типа `bool` могут иметь значения `true` либо `false`. Нуль преобразуется в значение `false`.
  3. `void` – это тип данных, характеризующий ситуацию отсутствия значения.

## Проект 2.1. "Поговорим с Марсом"

Mars.cpp

В положении, когда Марс находится к Земле ближе всего, расстояние между ними составляет приблизительно 34 000 000 миль. Предположим, что на Марсе есть некто, с кем вы хотели бы поговорить. Тогда возникает вопрос, какова будет задержка во времени между моментом отправки радиосигнала с Земли и моментом его прихода на Марс. Цель следующего проекта — создать программу, которая отвечает на этот вопрос. Для этого вспомним, что радиосигнал распространяется со скоростью света, т.е. приблизительно 186 000 миль в секунду. Таким образом, чтобы вычислить эту задержку, необходимо расстояние между планетами разделить на скорость света. Полученный результат следует отобразить в секундах и минутах.

### Последовательность действий

1. Создайте новый файл с именем Mars.cpp.
2. Для вычисления задержки необходимо использовать значения с плавающей точкой. Почему? Да просто потому, что искомый временной интервал будет иметь дробную часть. Вот переменные, которые используются в программе.

```
double distance;
double lightspeed;
double delay;
double delay_in_min;
```

3. Присвоим переменным distance и lightspeed начальные значения.

```
distance = 34000000.0; // 34 000 000 миль
lightspeed = 186000.0; // 186 000 миль в секунду
```

4. Для получения задержки во времени (в секундах) разделим значение переменной distance на значение lightspeed. Присвоим вычисленное частное переменной delay и отобразим результат.

```
delay = distance / lightspeed;
```

```
cout << "Временная задержка при разговоре с Марсом: "
<<
delay << " секунд.\n";
```

5. Для получения задержки в минутах разделим значение переменной delay на 60 и отобразим результат.

```
delay_in_min = delay / 60.0;
```

6. Приведем программу Mars .cpp целиком.

```
/*
Проект 2.1.

"Поговорим с Марсом"
*/

#include <iostream>
using namespace std;

int main() {
    double distance;
    double lightspeed;
    double delay;
    double delay_in_min;

    distance = 34000000.0; // 34 000 000 миль
    lightspeed = 186000.0; // 186 000 миль в секунду

    delay = distance / lightspeed;

    cout << "Временная задержка при разговоре с Марсом: "
    <<
        delay << " секунд.\n";

    delay_in_min = delay / 60.0;

    cout << "Это составляет " << delay_in_min << " минут.";

    return 0;
}
```

7. Скомпилируйте и выполните программу. Вы должны получить такой результат.

Временная задержка при разговоре с Марсом: 182.796  
секунд.

Это составляет 3.04659 минут.

8. Самостоятельно отобразите временную задержку, которая будет иметь место при ожидании ответа с Марса.

ВАЖНО!

## 2.2. Литералы

*Литералы* — это фиксированные (предназначенные для восприятия человеком) значения, которые не могут быть изменены программой. Они называются также *константами*. Мы уже использовали литералы во всех предыдущих примерах программ. А теперь настало время изучить их более подробно.

Литералы в C++ могут иметь любой базовый тип данных. Способ представления каждого литерала зависит от его типа. *Символьные литералы* заключаются в одинарные кавычки. Например, 'a' и '%' являются символьными литералами.

*Целочисленные литералы* задаются как числа без дробной части. Например, 10 и -100 — целочисленные литералы. *Вещественные литералы* должны содержать десятичную точку, за которой следует дробная часть числа, например 11.123. Для вещественных констант можно также использовать экспоненциальное представление чисел.

Все литературные значения имеют некоторый тип данных. Но, как вы уже знаете, есть несколько разных целочисленных типов, например `int`, `short int` и `unsigned long int`. Существует также три различных вещественных типа: `float`, `double` и `long double`. Интересно, как же тогда компилятор определяет тип литерала? Например, число 123.23 имеет тип `float` или `double`? Ответ на этот вопрос состоит из двух частей. Во-первых, C++-компилятор автоматически делает определенные предположения насчет литералов. Во-вторых, при желании программист может явно указать тип литерала.

По умолчанию компилятор связывает целочисленный литерал с совместимым и одновременно наименьшим по занимаемой памяти типом данных, начиная с типа `int`. Следовательно, для 16-разрядных сред число 10 будет связано с типом `int`, а 103 000 — с типом `long int`.

Единственным исключением из правила “наименьшего типа” являются вещественные (с плавающей точкой) константы, которым по умолчанию присваивается тип `double`.

Во многих случаях такие стандарты работы компилятора вполне приемлемы. Однако у программиста есть возможность точно определить нужный тип. Чтобы задать точный тип числовой константы, используйте соответствующий суффикс. Для вещественных типов действуют следующие суффиксы: если вещественное число завершить буквой F, оно будет обрабатываться с использованием типа `float`, а если буквой L, подразумевается тип `long double`. Для целочисленных типов суффикс U означает использование модификатора типа `unsigned`, а суффикс L — `long`. (Для задания модификатора `unsigned long` необходимо указать оба суффикса U и L.) Ниже приведены некоторые примеры.

Тип данных	Примеры констант
int	1, 123, 21000, -234
long int	35000L, -34L
unsigned int	10000U, 987U, 40000U
unsigned long	12323UL, 900000UL
float	123.23F, 4.34e-3F
double	23.23, 123123.33, -0.9876324
long double	1001.2L

## Шестнадцатеричные и восьмеричные литералы

Иногда удобно вместо десятичной системы счисления использовать восьмеричную или шестнадцатеричную. В *восьмеричной* системе основанием служит число 8, а для выражения всех чисел используются цифры от 0 до 7. В восьмеричной системе число 10 имеет то же значение, что число 8 в десятичной. Система счисления по основанию 16 называется *шестнадцатеричной* и использует цифры от 0 до 9 плюс буквы от A до F, означающие шестнадцатеричные “цифры” 10, 11, 12, 13, 14 и 15. Например, шестнадцатеричное число 10 равно числу 16 в десятичной системе. Поскольку эти две системы счисления (шестнадцатеричная и восьмеричная) используются в программах довольно часто, в языке C++ разрешено при желании задавать целочисленные литералы не в десятичной, а в шестнадцатеричной или восьмеричной системе. Шестнадцатеричный литерал должен начинаться с префикса 0x (нуль и буква x) или 0X, а восьмеричный – с нуля. Приведем два примера.

```
int hex = 0xFF; // 255 в десятичной системе
int oct = 011; // 9 в десятичной системе
```

## Строковые литералы

Язык C++ поддерживает еще один встроенный тип литерала, именуемый *строковым*. *Строка* – это набор символов, заключенных в двойные кавычки, например "это тест". Вы уже видели примеры строк в некоторых cout-инструкциях, с помощью которых мы выводили текст на экран. При этом обратите внимание вот на что. Хотя C++ позволяет определять строковые литералы, он не имеет встроенного строкового типа данных. Строки в C++, как будет показано ниже в этой книге, поддерживаются в виде символьных массивов. (Кроме того, стандарт C++ поддерживает строковый тип с помощью библиотечного класса *string*.)

## Спросим у опытного программиста

**Вопрос.** Вы показали, как определить символьный литерал (для этого символ необходимо заключить в одинарные кавычки). Надо понимать, что речь идет о литералах типа `char`. А как задать литерал (т.е. константу) типа `w_char`?

**Ответ.** Для определения двубайтовой константы, т.е. константы типа `w_char`, необходимо использовать префикс `L`. Вот пример.

```
w_char wc;
wc = L'A';
```

Здесь переменной `wc` присваивается двубайтовая константа, эквивалентная символу `A`. Вряд ли вам придется часто использовать "широкие" символы, но в этом может возникнуть необходимость при выходе ваших заказчиков на международный рынок.

## Управляющие символьные последовательности

С выводом большинства печатаемых символов прекрасно справляются символьные константы, заключенные в одинарные кавычки, но есть такие "экземпляры" (например, символ возврата каретки), которые невозможно ввести в исходный текст программы с клавиатуры. Некоторые символы (например, одинарные и двойные кавычки) в C++ имеют специальное назначение, поэтому иногда их нельзя ввести напрямую. По этой причине в языке C++ разрешено использовать ряд специальных символьных последовательностей (включающих символ "обратная косая черта"), которые также называются *управляющими последовательностями*. Их список приведен в табл. 2.3.

Таблица 2.3. Управляющие символьные последовательности

Код	Значение
\b	Возврат на одну позицию
\f	Подача страницы (для перехода к началу следующей страницы)
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\"	Двойная кавычка
'	Одинарная кавычка (апостроф)
\\	Обратная косая черта
\v	Вертикальная табуляция
\a	Звуковой сигнал (звонок)
\?	Вопросительный знак
\N	Восьмеричная константа (где <i>N</i> – это сама восьмеричная константа)
\xN	Шестнадцатеричная константа (где <i>N</i> – это сама шестнадцатеричная константа)

Использование управляющих последовательностей демонстрируется на примере следующей программы.

```
// Демонстрация использования управляющих
// последовательностей.

#include <iostream>
using namespace std;

int main()
{
    cout << "one\ttwo\tthree\n";
    cout << "123\b\b45"; // Последовательность \b\b
                        // обеспечит возврат на две позиции
                        // назад. Поэтому цифры 2 и 3 будут
                        // "стерты".
    return 0;
}
```

Эта программа генерирует такие результаты.

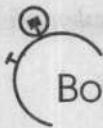
```
one      two      three
145
```

Здесь в первой cout-инструкции для позиционирования слов “two” и “three” используется символ горизонтальной табуляции. Вторая cout-инструкция сначала отображает цифры 123, затем после вывода двух управляющих символов возврата на одну позицию цифры 2 и 3 удаляются. Наконец, выводятся цифры 4 и 5.

### Спросим у опытного программиста

**Вопрос.** Если строка состоит из одного символа, то она эквивалентна символьному литералу? Например, 'k' и 'k' – это одно и то же?

**Ответ.** Нет. Не нужно путать строки с символами. Символьный литерал представляет одну букву, т.е. значение типа `char`. В то же время строка (пусть и состоящая всего из одной буквы) – все равно строка. Хотя строки состоят из символов, это – разные типы данных.



## Вопросы для текущего контроля

1. Какой тип данных используется по умолчанию для литералов 10 и 10.0?
2. Как задать константу 100, чтобы она имела тип данных long int? Как задать константу 100, чтобы она имела тип данных unsigned int?
3. Что такое \b?\*

ВАЖНО!

### 2.3. Создание инициализированных переменных

С переменными мы познакомились в модуле 1. Теперь настало время рассмотреть их более подробно. Общий формат инструкции объявления переменных выглядит так:

*тип имя\_переменной;*

Здесь элемент *тип* означает допустимый в C++ тип данных, а элемент *имя\_переменной* – имя объявляемой переменной. Создавая переменную, мы создаем экземпляр ее типа. Таким образом, возможности переменной определяются ее типом. Например, переменная типа bool может хранить только логические значения, и ее нельзя использовать для хранения значений с плавающей точкой. Тип переменной невозможно изменить во время ее существования. Это значит, что int-переменную нельзя превратить в double-переменную.

### Инициализация переменной

При объявлении переменной можно присвоить некоторое значение, т.е. *инициализировать* ее. Для этого достаточно записать после ее имени знак равенства

1. Для литерала 10 по умолчанию используется тип int, а для литерала 10.0 – тип double.
2. Для того чтобы константа 100 имела тип данных long int, необходимо записать 100L. Для того чтобы константа 100 имела тип данных unsigned int, необходимо записать 100U.
3. Запись \b представляет собой управляющую последовательность, которая означает возврат на одну позицию.

и присваиваемое начальное значение. Общий формат *инициализации переменной* имеет следующий вид:

*тип имя\_переменной = значение;*

Вот несколько примеров.

```
int count = 10; // Присваиваем переменной count начальное
                 // значение 10.
char ch = 'X'; // Инициализируем переменную ch буквой X.
float f = 1.2F // Переменная f инициализируется
               // значением 1,2.
```

При объявлении двух или больше переменных одного типа в форме списка можно одну из них (или несколько) обеспечить начальными значениями. При этом все элементы списка, как обычно, разделяются запятыми. Вот пример.

```
int a, b = 8, c = 19, d; // Переменные b и c
                        // инициализируются при объявлении.
```

## Динамическая инициализация переменных

Несмотря на то что переменные часто инициализируются константами (как показано в предыдущих примерах), C++ позволяет инициализировать переменные динамически, т.е. с помощью любого выражения, действительного на момент инициализации. Например, рассмотрим небольшую программу, которая вычисляет объем цилиндра по радиусу его основания и высоте.

```
// Демонстрация динамической инициализации переменных.
```

```
#include <iostream>
using namespace std;

int main() {
    double radius = 4.0, height = 5.0;

    // Динамически инициализируем переменную volume.
    double volume = 3.1416 * radius * radius * height;

    cout << "Объем цилиндра равен " << volume;

    return 0;
}
```

Здесь объявляются три локальные переменные `radius`, `height` и `volume`. Первые две (`radius` и `height`) инициализируются константами, а третья (`volume`) – результатом выражения, вычисляемым во время выполнения программы. Здесь важно то, что выражение инициализации может использовать любой элемент, действительный на момент инициализации, – обращение к функции, другие переменные или литералы.

## Операторы

В C++ определен широкий набор операторов. *Оператор* (operator) – это символ, который указывает компилятору на выполнение конкретных математических действий или логических манипуляций. В C++ имеется четыре общих класса операторов: *арифметические, поразрядные, логические и операторы отношений*. Помимо них определены другие операторы специального назначения. В этом модуле мы уделим внимание арифметическим, логическим и операторам отношений, а также оператору присваивания. Поразрядные и операторы специального назначения рассматриваются ниже в этой книге.

ВАЖНО!

### 2.4. Арифметические операторы

В C++ определены следующие арифметические операторы.

Оператор	Действие
+	Сложение
-	Вычитание, а также унарный минус
*	Умножение
/	Деление
%	Деление по модулю
--	Декремент
++	Инкремент

Действие операторов `+`, `-`, `*` и `/` совпадает с действием аналогичных операторов в алгебре. Их можно применять к данным любого встроенного числового типа, а также типа `char`.

После применения оператора деления (`/`) к целому числу остаток будет отброшен. Например, результат целочисленного деления `10/3` будет равен 3. Остаток от деления можно получить с помощью *оператора деления по модулю* (%). Например, `10 % 3` равно 1. Это означает, что в C++ оператор “%” нельзя применять к

## 90 Модуль 2. Типы данных и операторы

типам с плавающей точкой (`float` или `double`). Деление по модулю применимо только к операндам целочисленного типа.

Использование оператора деления по модулю демонстрируется в следующей программе.

```
// Демонстрация использования оператора деления по модулю.
```

```
#include <iostream>
using namespace std;

int main()
{
    int x, y;

    x = 10;
    y = 3;
    cout << x << " / " << y << " равно " << x / y <<
        " с остатком " << x % y << "\n";

    x = 1;
    y = 2;
    cout << x << " / " << y << " равно " << x / y << "\n" <<
        x << " % " << y << " равно " << x % y;

    return 0;
}
```

Результаты выполнения этой программы таковы.

```
10 / 3 равно 3 с остатком 1
1 / 2 равно 0
1 % 2 равно 1
```

## Инкремент и декремент

Операторы инкремента (`++`) и декремента (`--`), которые уже упоминались в модуле 1, обладают очень интересными свойствами. Поэтому им следует уделить особое внимание.

Вспомним: оператор инкремента выполняет сложение операнда с числом 1, а оператор декремента вычитает 1 из своего операнда. Это значит, что инструкция `x = x + 1;`

аналогична такой инструкции:

```
++x;
```

А инструкция

```
x = x - 1;
```

аналогична такой инструкции:

```
--x;
```

Операторы инкремента и декремента могут стоять как перед своим операндом (*префиксная форма*), так и после него (*постфиксная форма*). Например, инструкцию

```
x = x + 1;
```

можно переписать в виде префиксной

```
++x; // Префиксная форма оператора инкремента.
```

или постфиксной формы:

```
x++; // Постфиксная форма оператора инкремента.
```

В предыдущем примере не имело значения, в какой форме был применен оператор инкремента: префиксной или постфиксной. Но если оператор инкремента или декремента используется как часть большего выражения, то форма его применения очень важна. Если такой оператор применен в префиксной форме, то C++ сначала выполнит эту операцию, чтобы operand получил новое значение, которое затем будет использовано остальной частью выражения. Если же оператор применен в постфиксной форме, то C# использует в выражении его старое значение, а затем выполнит операцию, в результате которой operand обретет новое значение. Рассмотрим следующий фрагмент кода:

```
x = 10;
```

```
y = ++x;
```

В этом случае переменная y будет установлена равной 11. Но если в этом коде префиксную форму записи заменить постфиксной, переменная y будет установлена равной 10:

```
x = 10;
```

```
y = x++;
```

В обоих случаях переменная x получит значение 11. Разница состоит лишь в том, в какой момент она станет равной 11 (до присвоения ее значения переменной y или после). Для программиста очень важно иметь возможность управлять временем выполнения операции инкремента или декремента.

Арифметические операторы подчиняются следующему порядку выполнения действий.

<u>Приоритет</u>	<u>Операторы</u>
Наивысший	$++ \quad --$ – (унарный минус) $\ast \quad / \quad \%$
Низший	$+ \quad -$

Операторы одного уровня старшинства вычисляются компилятором слева направо. Безусловно, для изменения порядка вычислений можно использовать круглые скобки, которые обрабатываются в C++ так же, как практически во всех других языках программирования. Операции или набор операций, заключенных в круглые скобки, приобретают более высокий приоритет по сравнению с другими операциями выражения.

### Спросим у опытного программиста

**Вопрос.** *Не связан ли оператор инкремента “++” с именем языка C++?*

**Ответ.** Да! Как вы знаете, C++ построен на фундаменте языка C, к которому добавлено множество усовершенствований, большинство из которых предназначено для поддержки объектно-ориентированного программирования. Таким образом, C++ представляет собой *инкрементное* усовершенствование языка C, а результат добавления символов “++” (оператора инкремента) к имени C оказался вполне подходящим именем для нового языка.

Бьерн Страуструп сначала назвал свой язык “C с классами” (C with Classes), но позже он изменил это название на C++. И хотя успех нового языка еще только предполагался, принятие нового названия (C++) практически гарантировало ему видное место в истории, поскольку это имя было узнаваемым для каждого C-программиста.

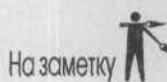
**ВАЖНО!**

## 2.5. Операторы отношений и логические операторы

Операторы отношений и логические (булевы) операторы, которые часто идут “рука об руку”, используются для получения результатов в виде значений ИСТИНА/ЛОЖЬ. Операторы отношений оценивают по “двухбалльной системе” отношения между двумя значениями, а логические определяют различные

способы сочетания истинных и ложных значений. Поскольку операторы отношений генерируют ИСТИНА/ЛОЖЬ-результаты, то они часто выполняются с логическими операторами. Поэтому мы и рассматриваем их в одном разделе.

Операторы отношений и логические (булевы) операторы перечислены в табл. 2.4. Обратите внимание на то, что в языке C++ в качестве оператора отношения “не равно” используется символ “!=”, а для оператора “равно” – двойной символ равенства (==). Согласно стандарту C++ результат выполнения операторов отношений и логических операторов имеет тип `bool`, т.е. при выполнении операций отношений и логических операций получаются значения `true` или `false`.



На заметку

При использовании более старых компиляторов результаты выполнения этих операций имели тип `int` (0 или 1). Это различие в интерпретации значений имеет в основном теоретическую основу, поскольку C++, как упоминалось выше, автоматически преобразует значение `true` в 1, а значение `false` – в 0, и наоборот.

Операнды, участвующие в операциях “выяснения” отношений, могут иметь практически любой тип, главное, чтобы их можно было сравнивать. Что касается логических операторов, то их операнды должны иметь тип `bool`, и результат логической операции всегда будет иметь тип `bool`. Поскольку в C++ любое не-нулевое число оценивается как истинное (`true`), а нуль эквивалентен ложному значению (`false`), то логические операторы можно использовать в любом выражении, которое дает нулевой или ненулевой результат.

Таблица 2.4. Операторы отношений и логические операторы

Операторы отношений	Значение
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
==	Равно
!=	Не равно
Логические операторы	Значение
&&	И
	ИЛИ
!	НЕ

Логические операторы используются для поддержки базовых логических операций И, ИЛИ и НЕ в соответствии со следующей таблицей истинности.

p	q	p И q	p ИЛИ q	НЕ p
ЛОЖЬ	ЛОЖЬ	ЛОЖЬ	ЛОЖЬ	ИСТИНА
ЛОЖЬ	ИСТИНА	ЛОЖЬ	ИСТИНА	ИСТИНА
ИСТИНА	ИСТИНА	ИСТИНА	ИСТИНА	ЛОЖЬ
ИСТИНА	ЛОЖЬ	ЛОЖЬ	ИСТИНА	ЛОЖЬ

Рассмотрим программу, в которой демонстрируется использование нескольких логических операторов и операторов отношений.

```
// Демонстрация использования логических операторов и
// операторов отношений.
```

```
#include <iostream>
using namespace std;

int main() {
    int i, j;
    bool b1, b2;

    i = 10;
    j = 11;
    if(i < j) cout << "i < j\n";
    if(i <= j) cout << "i <= j\n";
    if(i != j) cout << "i != j\n";
    if(i == j) cout << "Это не выполняется!\n";
    if(i >= j) cout << "Это не выполняется!\n";
    if(i > j) cout << "Это не выполняется!\n";

    b1 = true;
    b2 = false;
    if(b1 && b2) cout << "Это не выполняется!\n";
    if(!(b1 && b2)) cout <<
        !(b1 && b2) равно значению ИСТИНА\n";
    if(b1 || b2) cout << "b1 || b2 равно значению ИСТИНА \n";

    return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
i < j
i <= j
```

```
i != j
!(b1 && b2) равно значению ИСТИНА
b1 || b2 равно значению ИСТИНА
```

Как операторы отношений, так и логические операторы имеют более низкий приоритет по сравнению с арифметическими операторами. Это означает, что такое выражение, как  $10 > 1+12$ , будет вычислено так, как если бы оно было записано в виде  $10 > (1+12)$ . Результат этого выражения, конечно же, равен значению ЛОЖЬ.

С помощью логических операторов можно объединить в одном выражении любое количество операций отношений. Например, в этом выражении объединено сразу три такие операции.

```
var > 15 || !(10 < count) && 3 <= item
```

Приоритет операторов отношений и логических операторов показан в следующей таблице.

Приоритет	Операторы
Наивысший	!
	> >= < <=
	== !=
	&&
Низший	

## Проект 2.2

### Проект 2.2. Создание логической операции “исключающее ИЛИ” (XOR)

**XOR.cpp** В языке С++ не определен логический оператор “исключающее ИЛИ” (XOR), хотя это – довольно популярный у программистов бинарный оператор. Операция “исключающее ИЛИ” генерирует значение ИСТИНА, если это значение ИСТИНА имеет один и только один из его operandов. Это утверждение можно выразить в виде следующей таблицы истинности.

p	q	p XOR q
ЛОЖЬ	ЛОЖЬ	ЛОЖЬ
ЛОЖЬ	ИСТИНА	ИСТИНА
ИСТИНА	ЛОЖЬ	ИСТИНА
ИСТИНА	ИСТИНА	ЛОЖЬ

Отсутствие реализации логического оператора “исключающее ИЛИ” как встроенного элемента языка одни программисты считают досадным изъяном

C++. Другие же придерживаются иного мнения: мол, это избавляет язык от избыточности, поскольку логический оператор "исключающее ИЛИ" (XOR) не трудно "создать" на основе встроенных операторов И, ИЛИ и НЕ.

В настоящем проекте мы создадим операцию "исключающее ИЛИ", используя операторы `&&`, `||` и `!`. После этого вы сами решите, как относиться к тому факту, что этот оператор не является встроенным элементом языка C++.

### Последовательность действий

1. Создайте новый файл с именем `XOR.cpp`.
2. Будем исходить из того, что логический оператор "исключающее ИЛИ" (XOR), применяемый к двум значениям булевого типа `p` и `q`, строится так.

$$(p \mid\mid q) \ \&\& \ ! (p \ \&\& \ q)$$

Рассмотрим это выражение. Сначала над операндами `p` и `q` выполняется операция логического ИЛИ (первое подвыражение), результат которой будет равен значению ИСТИНА, если по крайней мере один из операндов будет иметь значение ИСТИНА. Затем над операндами `p` и `q` выполняется операция логического И, результат которой будет равен значению ИСТИНА, если оба операнда будут иметь значение ИСТИНА. Последний результат затем инвертируется с помощью оператора НЕ. Таким образом, результат второго подвыражения `!(p && q)` будет равен значению ИСТИНА, если один из операндов (или оба сразу) будет иметь значение ЛОЖЬ. Наконец, этот результат логически умножается (с помощью операции И) на результат вычисления первого подвыражения `(p || q)`. Таким образом, полное выражение даст значение ИСТИНА, если один из операндов (но не оба одновременно) имеет значение ИСТИНА.

3. Ниже приводится текст всей программы `XOR.cpp`. При ее выполнении демонстрируется результат применения операции "исключающее ИЛИ" для всех четырех возможных комбинаций ИСТИНА/ЛОЖЬ-значений к двум ее operandам.

```
/*
```

```
Проект 2.2.
```

Создание оператора "исключающее ИЛИ" (XOR) на основе встроенных логических C++-операторов.

```
*/
```

```

#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    bool p, q;
    p = true;
    q = true;

    cout << p << " XOR " << q << " равно " <<
        ( (p || q) && !(p && q) ) << "\n";

    p = false;
    q = true;

    cout << p << " XOR " << q << " равно " <<
        ( (p || q) && !(p && q) ) << "\n";

    p = true;
    q = false;

    cout << p << " XOR " << q << " равно " <<
        ( (p || q) && !(p && q) ) << "\n";

    p = false;
    q = false;

    cout << p << " XOR " << q << " равно " <<
        ( (p || q) && !(p && q) ) << "\n";

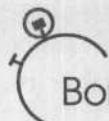
    return 0;
}

```

4. Скомпилируйте и выполните программу. Вы должны получить такие результаты.

```
1 XOR 1 равно 0
0 XOR 1 равно 1
1 XOR 0 равно 1
0 XOR 0 равно 0
```

5. Обратите внимание на внешние круглые скобки, в которые заключены выражения в инструкции `cout`. Без них здесь обойтись нельзя, поскольку операторы `&&` и `||` имеют более низкий приоритет, чем оператор вывода данных (`<<`). Чтобы убедиться в этом, попробуйте убрать внешние круглые скобки. При попытке скомпилировать программу вы получите сообщение об ошибке.



### Вопросы для текущего контроля

1. Каково назначение оператора “%”? К значениям какого типа его можно применять?
2. Как объявить переменную `index` типа `int` с начальным значением 10?
3. Какой тип будет иметь результат логического выражения или выражения, построенного с применением операторов отношений?\*

**ВАЖНО!**

## 2.6. Оператор присваивания

Начиная с модуля 1, мы активно используем оператор присваивания. Настало время для “официального” с ним знакомства. В языке C++ *оператором присваивания* служит одинарный знак равенства (`=`). Его действие во многом подобно действию аналогичных операторов в других языках программирования. Его общий формат имеет следующий вид.

`переменная = выражение;`

1. Оператор “%” предназначен для выполнения операции деления по модулю и возвращает остаток от деления нацело. Его можно применять только к значениям целочисленных типов.
2. `int index = 10;`
3. Результат логического выражения или выражения, построенного с применением операторов отношений, будет иметь тип `bool`.

Эта запись означает, что элементу *переменная* присваивается значение *выражение*.

Оператор присваивания обладает очень интересным свойством: он позволяет создавать цепочку присваиваний. Другими словами, присваивая “общее” значение сразу нескольким переменным, можно “связать воедино” сразу несколько присваиваний. Рассмотрим, например, следующий фрагмент кода.

```
int x, y, z;
x = y = z = 100; // устанавливаем x, y и z равными 100
```

При выполнении этого фрагмента кода с помощью одной инструкции сразу трем переменным *x*, *y* и *z* присваивается значение 100. Почему возможна такая форма записи? Дело в том, что результатом выполнения оператора *=* является значение выражения, стоящего от него с правой стороны. Таким образом, результатом выполнения операции *z = 100* является число 100, которое затем присваивается переменной *y* и которое в свою очередь присваивается переменной *x*. Использование цепочки присваиваний — самый простой способ установить сразу несколько переменных равными “общему” значению.

#### ВАЖНО!

## 2.7. Составные операторы присваивания

В C++ предусмотрены специальные *составные операторы присваивания*, в которых объединено присваивание с еще одной операцией. Начнем с примера и рассмотрим следующую инструкцию.

```
x = x + 10;
```

Используя составной оператор присваивания, ее можно переписать в таком виде.

```
x += 10;
```

Пара операторов *+=* служит указанием компилятору присвоить переменной *x* сумму текущего значения переменной *x* и числа 10.

А вот еще один пример. Инструкция

```
x = x - 100;
```

аналогична такой:

```
x -= 100;
```

Обе эти инструкции присваивают переменной *x* ее прежнее значение, уменьшенное на 100.

Составные версии операторов присваивания существуют для всех бинарных операторов (т.е. для всех операторов, которые работают с двумя операндами). Таким образом, при таком общем формате бинарных операторов присваивания

*переменная = переменная ор выражение;*

общая форма записи их составных версий выглядит так:

*переменная ор = выражение;*

Здесь элемент *ор* означает конкретный арифметический или логический оператор, объединяемый с оператором присваивания.

Поскольку составные операторы присваивания короче своих “несоставных” эквивалентов, составные версии иногда называются *сокращенными* операторами присваивания.

Составные операторы присваивания обладают двумя достоинствами по сравнению со своими “полными формами”. Во-первых, они компактнее. Во-вторых, они позволяют компилятору генерировать более эффективный код (поскольку операнд в этом случае вычисляется только один раз). Именно поэтому их можно часто встретить в профессионально написанных C++-программах, а это значит, что каждый C++-программист должен быть с ними на “ты”.

**ВАЖНО!**

## **2.8. Преобразование типов в операторах присваивания**

Если переменные одного типа смешаны с переменными другого типа, происходит *преобразование типов*. При выполнении инструкции присваивания действует простое правило преобразования типов: значение, расположенное с правой стороны от оператора присваивания, преобразуется в значение типа, соответствующего переменной, указанной слева от оператора присваивания. Рассмотрим пример.

```
int x;
char ch;
float f;

ch = x;      /* строка 1 */
x = f;      /* строка 2 */
f = ch;      /* строка 3 */
f = x;      /* строка 4 */
```

В строке 1 старшие биты целочисленной переменной `x` отбрасываются, оставляя переменной `ch` младшие 8 бит. Если значение переменной `x` лежит в пределах от -128 до 127, то оно будет идентично значению переменной `ch`. В противном случае значение переменной `ch` будет отражать только младшие биты переменной `x`. При выполнении следующей инструкции (строка 2) переменная `x` получит целую часть значения, хранимого в переменной `f`. Присваивание, записанное в строке 3, обеспечит преобразование 8-разрядного целочисленного значения переменной `ch` в формат значения с плавающей точкой. Аналогичное преобразование произойдет и при выполнении строки 4 за исключением того, что в формат вещественного числа в этом случае будет преобразовано не `char`-`int`-значение.

При преобразовании целочисленных значений в символы и длинных `int`-в обычные `int`-значения отбрасывается соответствующее количество старших битов. Во многих 32-разрядных средах это означает, что при переходе от целочисленного значения к символьному будут потеряны 24 бит, а при переходе от `int`-к короткому `int`-значению — 16 бит. При преобразовании вещественного значения в целочисленное теряется дробная часть. Если тип переменной-приемника недостаточно велик для сохранения присваиваемого значения, то в результате этой операции в ней будет сохранен “мусор”.

Несмотря на то что C++ автоматически преобразует значение одного встроенного типа данных в значение другого, результат не всегда будет совпадать с ожидаемым. Чтобы избежать неожиданностей, нужно внимательно относиться к использованию смешанных типов в одном выражении.

## Выражения

Операторы, литералы и переменные — это все составляющие *выражений*. Вероятно, вы уже знакомы с выражениями по предыдущему опыту программирования или из школьного курса алгебры. В следующих разделах мы рассмотрим аспекты выражений, которые касаются их использования в языке C++.

ВАЖНО!

### 2.9. Преобразование типов в выражениях

Если в выражении смешаны различные типы литералов и переменных, компилятор преобразует их в один тип. Во-первых, все `char`- и `short int`-значения автоматически преобразуются (с расширением “типоразмера”) в тип `int`.

Этот процесс называется *целочисленным расширением* (integral promotion). Во-вторых, все операнды преобразуются (также с расширением “типоразмера”) в тип самого большого операнда. Этот процесс называется *расширением типа* (type promotion), причем он выполняется пооперационно. Например, если один операнд имеет тип `int`, а другой — `long int`, то тип `int` расширяется в тип `long int`. Или, если хотя бы один из операндов имеет тип `double`, любой другой операнд приводится к типу `double`. Это означает, что такие преобразования, как из типа `char` в тип `double`, вполне допустимы. После преобразования оба операнда будут иметь один и тот же тип, а результат операции — тип, совпадающий с типом операндов.

## Преобразования, связанные с типом `bool`

Как упоминалось выше, значения типа `bool` при использовании в выражении целочисленного типа автоматически преобразуются в целые числа 0 или 1. При преобразовании целочисленного результата в тип `bool` нуль превращается в `false`, а ненулевое значение — в `true`. И хотя тип `bool` относительно недавно был добавлен в язык C++, выполнение автоматических преобразований, связанных с типом `bool`, означает, что его введение в C++ не имеет негативных последствий для кода, написанного для более ранних версий C++. Более того, автоматические преобразования позволяют C++ поддерживать исходное определение значений ЛОЖЬ и ИСТИНА в виде нуля и ненулевого значения.

**ВАЖНО!**

## 2.10. Приведение типов

В C++ предусмотрена возможность установить для выражения заданный тип. Для этого используется *операция приведения типов* (cast). C++ определяет пять видов таких операций. В этом разделе мы рассмотрим только один из них (самый универсальный и единственный, который поддерживается старыми версиями C++), а остальные четыре описаны ниже в этой книге (после темы создания объектов). Итак, общий формат операции приведения типов таков:

(тип) выражение

Здесь элемент *тип* означает тип, к которому необходимо привести *выражение*. Например, если вы хотите, чтобы выражение `x / 2` имело тип `float`, необходимо написать следующее:

`(float) x / 2`

Приведение типов рассматривается как унарный оператор, и поэтому он имеет такой же приоритет, как и другие унарные операторы.

Иногда операция приведения типов оказывается очень полезной. Например, в следующей программе для управления циклом используется некоторая целочисленная переменная, входящая в состав выражения, результат вычисления которого необходимо получить с дробной частью.

```
// Демонстрация операции приведения типа.
```

```
#include <iostream>
using namespace std;

int main() {
    int i;

    for(i=1; i <= 10; ++i )
        cout << i << " / 2 равно: " << (float) i / 2 << '\n';
        //           ↑
        // Приведение к типу float
        // обеспечит обязательный
        // вывод дробной части.

    return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

```
1 / 2 равно: 0.5
2 / 2 равно: 1
3 / 2 равно: 1.5
4 / 2 равно: 2
5 / 2 равно: 2.5
6 / 2 равно: 3
7 / 2 равно: 3.5
8 / 2 равно: 4
9 / 2 равно: 4.5
10 / 2 равно: 5
```

Без оператора приведения типа (float) выполнилось бы только целочисленное деление. Приведение типов в данном случае гарантирует, что на экране будет отображена и дробная часть результата.

**ВАЖНО!**

## 2.11. Использование пробелов и круглых скобок

Любое выражение в C++ для повышения читабельности может включать пробелы (или символы табуляции). Например, следующие два выражения совершенно одинаковы, но второе прочитать гораздо легче.

```
x=10/y*(127/x);
x = 10 / y * (127/x);
```

Круглые скобки (так же, как в алгебре) повышают приоритет операций, содержащихся внутри них. Использование избыточных или дополнительных круглых скобок не приведет к ошибке или замедлению вычисления выражения. Другими словами, от них не будет никакого вреда, но зато сколько пользы! Ведь они помогут прояснить (для вас самих в первую очередь, не говоря уже о тех, кому придется разбираться в этом без вас) точный порядок вычислений. Скажите, например, какое из следующих двух выражений легче понять?

```
x = y/3-34*temp+127;
x = (y/3) - (34*temp) + 127;
```

### Проект 2.3. Вычисление размера платежей по займу

В этом проекте мы создадим программу, которая вычисляет размер регулярных платежей по займу, скажем, на покупку автомобиля. По таким данным, как основная сумма займа, срок займа, количество выплат в год и процентная ставка, программа вычисляет размер платежа. Имея дело с финансовыми расчетами, нужно использовать типы данных с плавающей точкой. Чаще всего в подобных вычислениях применяется тип `double`, вот и мы не будем отступать от общего правила. Заранее отмечу, что в этом проекте используется еще одна библиотечная C++-функция `pow()`.

Для вычисления размера регулярных платежей по займу мы будем использовать следующую формулу.

$$\text{Payment} = \frac{\text{IntRate} * (\text{principal} / \text{PayPerYear})}{1 - ( \text{IntRate} / \text{PayPerYear} ) + 1^{-\text{PayPerYear} * \text{NumYears}}}$$

Здесь `intRate` — процентная ставка, `principal` — начальное значение остатка (основная сумма займа), `PayPerYear` — количество выплат в год, `NumYears` — срок займа (в годах). Обратите внимание на то, что для вычисления знаменателя

необходимо выполнить операцию возведения в степень. Для этого мы будем использовать функцию `pow()`. Вот как выглядит формат ее вызова.

```
result = pow(base, exp);
```

Функция `pow()` возвращает значение элемента `base`, введенное в степень `exp`. Функция `pow()` принимает аргументы типа `double` и возвращает значение типа `double`.

## Последовательность действий

1. Создайте файл с именем `RegPay.cpp`.
2. Объявите следующие переменные.

```
double Principal;      // исходная сумма займа
double IntRate;        // процентная ставка в виде
                      // числа (например, 0.075)
double PayPerYear;    // количество выплат в год
double NumYears;      // срок займа (в годах)
double Payment;        // размер регулярного платежа
double numer, denom; // временные переменные
double b, e;           // аргументы для вызова функции
Pow()
```

Обратите внимание на то, что после объявления каждой переменной дан комментарий, в котором описывается ее назначение. И хотя мы не снабжаем такими подробными комментариями большинство коротких программ в этой книге, я советую следовать этой практике по мере того, как ваши программы будут становиться все больше и сложнее.

3. Добавьте следующие строки кода, которые обеспечивают ввод необходимой для расчета информации.

```
cout << "Введите исходную сумму займа: ";
cin >> Principal;

cout << "Введите процентную ставку (например, 0.075): ";
cin >> IntRate;

cout << "Введите количество выплат в год: ";
cin >> PayPerYear;

cout << "Введите срок займа (в годах): ";
cin >> NumYears;
```

4. Внесите в программу строки, которые выполняют финансовые расчеты.

```
numer = IntRate * Principal / PayPerYear;
```

```
e = -(PayPerYear * NumYears);
```

```
b = (IntRate / PayPerYear) + 1;
```

```
denom = 1 - pow(b, e);
```

```
Payment = numer / denom;
```

5. Завершите программу выводом результата.

```
cout << "Размер платежа по займу составляет "  
<< Payment;
```

6. Теперь приведем полный текст программы RegPay.cpp.

```
/*
```

```
Проект 2.3.
```

Вычисление размера регулярных платежей по займу.

Назовите этот файл RegPay.cpp.

```
*/
```

```
#include <iostream>  
#include <cmath>  
using namespace std;  
  
int main() {  
    double Principal;      // исходная сумма займа  
    double IntRate;        // процентная ставка в виде  
                          // числа (например, 0.075)  
    double PayPerYear;     // количество выплат в год  
    double NumYears;       // срок займа(в годах)  
    double Payment;        // размер регулярного платежа  
    double numer, denom;  // временные переменные  
    double b, e;           // аргументы для вызова  
                          // функции Pow()  
  
    cout << "Введите исходную сумму займа: ";  
    cin >> Principal;
```

```

cout << "Введите процентную ставку (например, 0.075): ";
cin >> IntRate;

cout << "Введите количество выплат в год: ";
cin >> PayPerYear;

cout << "Введите срок займа (в годах): ";
cin >> NumYears;

numer = IntRate * Principal / PayPerYear;

e = -(PayPerYear * NumYears);
b = (IntRate / PayPerYear) + 1;

denom = 1 - pow(b, e);

Payment = numer / denom;

cout << "Размер платежа по займу составляет "
<< Payment;

return 0;
}

```

Вот как выглядит один из возможных результатов выполнения этой программы.

Введите исходную сумму займа: 10000

Введите процентную ставку (например, 0.075): 0.075

Введите количество выплат в год: 12

Введите срок займа (в годах): 5

Размер платежа по займу составляет 200.379

7. Самостоятельно внесите изменения в программу, чтобы она отображала также общую сумму, выплаченную по процентам за весь период займа.



## **Тест для самоконтроля по модулю 2**

1. Какие типы целочисленных значений поддерживаются в C++?
2. Какой тип по умолчанию будет иметь константа 12.2?
3. Какие значения может иметь переменная типа `bool`?
4. Что представляет собой длинный целочисленный тип данных?
5. Какая управляющая последовательность обеспечивает табуляцию? Какая управляющая последовательность обеспечивает звуковой сигнал?
6. Стока должна быть заключена в двойные кавычки. Это верно?
7. Что представляют собой шестнадцатеричные цифры?
8. Представьте общий формат инициализации переменной при ее объявлении.
9. Какое действие выполняет оператор "%"؟ Можно ли его применять к значениям с плавающей точкой?
10. Поясните различие между префиксной и постфиксной формами оператора инкремента.
11. Какие из перечисленных ниже символов являются логическими операторами в C++?
  - A. `&&`
  - B. `##`
  - C. `||`
  - D. `$$`
  - E. `!`
12. Как по-другому можно переписать следующую инструкцию?  
`x = x + 12;`
13. Что такое приведение типов?
14. Напишите программу, которая находит все простые числа в интервале от 1 до 100.



Ответы на эти вопросы можно найти на Web-странице данной книги по адресу: <http://www.osborne.com>.

# Модуль 3

## Инструкции управления

- 3.1. Инструкция if**
- 3.2. Инструкция switch**
- 3.3. Цикл for**
- 3.4. Цикл while**
- 3.6. Использование инструкции break для выхода из цикла**
- 3.7. Использование инструкции continue**
- 3.8. Вложенные циклы**
- 3.9. Инструкция goto**

В этом модуле вы узнаете, как управлять ходом выполнения C++-программы. Существует три категории управляющих инструкций: *инструкции выбора* (*if*, *switch*), *итерационные инструкции* (состоящие из *for*-, *while*- и *do-while*-циклов) и *инструкции перехода* (*break*, *continue*, *return* и *goto*). За исключением *return*, все остальные перечисленные выше инструкции описаны в этом модуле.

**ВАЖНО!****3.1. Инструкция *if***

Инструкция *if* была представлена в модуле 1, но здесь мы рассмотрим ее более детально. Полный формат ее записи таков.

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

Здесь под элементом *инструкция* понимается одна инструкция языка C++. Часть *else* необязательна. Вместо элемента *инструкция* может быть использован блок инструкций. В этом случае формат записи *if*-инструкции принимает такой вид.

```
if(выражение)
{
    последовательность инструкций
}
else
{
    последовательность инструкций
}
```

Если элемент *выражение*, который представляет собой условное выражение, при вычислении даст значение ИСТИНА, будет выполнена *if*-инструкция; в противном случае – *else*-инструкция (если таковая существует). Обе инструкции никогда не выполняются. Условное выражение, управляющее выполнением *if*-инструкции, может иметь любой тип, действительный для C++-выражений, но главное, чтобы результат его вычисления можно было интерпретировать как значение ИСТИНА или ЛОЖЬ.

Использование *if*-инструкции рассмотрим на примере программы, которая представляет собой упрощенную версию игры “Угадай магическое число”. Программа генерирует случайное число и предлагает вам его угадать. Если вы угадываете число, программа выводит на экран сообщение одобрения \*\* Правильно \*\*. В этой программе представлена еще одна библиотечная функция *rand()*, кото-

рая возвращает случайным образом выбранное целое число. Для использования этой функции необходимо включить в программу заголовок `<cstdlib>`.

// Программа "Угадай магическое число".

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int magic; // магическое число
    int guess; // вариант пользователя

    magic = rand(); // Получаем случайное число.

    cout << "Введите свой вариант магического числа: ";
    cin >> guess;

    // Если значение переменной guess совпадет с
    // "магическим числом", будет выведено сообщение.
    if(guess == magic) cout << "*** Правильно ***";

    return 0;
}
```

В этой программе для проверки того, совпадает ли с "магическим числом" вариант, предложенный пользователем, используется оператор отношения `"=="`. При совпадении чисел на экран выводится сообщение `*** Правильно ***`.

Попробуем усовершенствовать нашу программу и в ее новую версию включим `else`-ветвь для вывода сообщения о том, что предположение пользователя оказалось неверным.

// Программа "Угадай магическое число":

// 1-е усовершенствование.

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
```

## 112 Модуль 3. Инструкции управления

```
{  
    int magic; // магическое число  
    int guess; // вариант пользователя  
  
    magic = rand(); // Получаем случайное число.  
  
    cout << "Введите свой вариант магического числа: ";  
    cin >> guess;  
  
    if(guess == magic)  
        cout << "*** Правильно ***";  
    else  
        cout << "...Очень жаль, но вы ошиблись."; // Индикатор  
                                         // неправильного ответа.  
  
    return 0;  
}
```

### Условное выражение

Иногда новичков в C++ сбивает с толку тот факт, что для управления *if*-инструкцией можно использовать любое действительное C++-выражение. Другими словами, тип выражения необязательно ограничивать операторами отношений и логическими операторами или operandами типа *bool*. Главное, чтобы результат вычисления условного выражения можно было интерпретировать как значение ИСТИНА или ЛОЖЬ. Как вы помните из предыдущего модуля, нуль автоматически преобразуется в *false*, а все ненулевые значения — в *true*. Это означает, что любое выражение, которое дает в результате нулевое или ненулевое значение, можно использовать для управления *if*-инструкцией. Например, следующая программа считывает с клавиатуры два целых числа и отображает частное от деления первого на второе. Чтобы не допустить деления на нуль, в программе используется *if*-инструкция.

```
// Использование int-значения для управления  
// if-инструкцией.
```

```
#include <iostream>  
using namespace std;  
  
int main()
```

```

{
    int a, b;

    cout << "Введите числитель: ";
    cin >> a;
    cout << "Введите знаменатель: ";
    cin >> b;

    if(b)                                // Для управления
        cout << "Результат: " << a / b // этой if-инструкцией
        << '\n'; // достаточно одной
        // переменной b.

    else
        cout << "На нуль делить нельзя.\n";

    return 0;
}

```

Возможные результаты выполнения этой программы выглядят так.

```

Введите числитель: 12
Введите знаменатель: 2
Результат: 6

```

```

Введите числитель: 12
Введите знаменатель: 0
На нуль делить нельзя.

```

Обратите внимание на то, что значение переменной *b* (делимое) сравнивается с нулем с помощью инструкции `if (b)`, а не инструкции `if (b != 0)`. Дело в том, что, если значение *b* равно нулю, условное выражение, управляющее инструкцией `if`, оценивается как ЛОЖЬ, что приводит к выполнению `else`-ветви. В противном случае (если *b* содержит ненулевое значение) условие оценивается как ИСТИНА, и деление благополучно выполняется. Нет никакой необходимости использовать следующую `if`-инструкцию, которая к тому же не свидетельствует о хорошем стиле программирования на C++.

```
if(b != 0) cout << a/b << '\n';
```

Эта форма `if`-инструкции считается устаревшей и потенциально неэффективной.

## Вложенные if-инструкции

Вложенные if-инструкции образуются в том случае, если в качестве элемента инструкция (см. полный формат записи) используется другая if-инструкция. Вложенные if-инструкции очень популярны в программировании. Главное здесь — помнить, что else-инструкция всегда относится к ближайшей if-инструкции, которая находится внутри того же программного блока, но еще не связана ни с какой другой else-инструкцией. Вот пример.

```
if(i) {
    if(j) result = 1;
    if(k) result = 2;
    else result = 3; // Эта else-ветвь связана с if(k).
}
else result = 4; // Эта else-ветвь связана с if(i).
```

Как отмечено в комментариях, последняя else-инструкция не связана с инструкцией if(j), поскольку они не находятся в одном блоке (несмотря на то, что эта if-инструкция — ближайшая, которая не имеет при себе “else-пары”). Внутренняя else-инструкция связана с инструкцией if(k), поскольку она — ближайшая и находится внутри того же блока.

Продемонстрируем использование вложенных инструкций с помощью очередного усовершенствования программы “Угадай магическое число” (здесь игрок получает реакцию программы на неправильный ответ).

```
// Программа “Угадай магическое число”:
// 2-е усовершенствование.

#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
    int magic; // магическое число
    int guess; // вариант пользователя

    magic = rand(); // Получаем случайное число.

    cout << "Введите свой вариант магического числа: ";
    cin >> guess;
```

```

if (guess == magic) {
    cout << "*** Правильно **\n";
    cout << magic
        << " и есть то самое магическое число.\n";
}
else {
    cout << "...Очень жаль, но вы ошиблись.";
    if(guess > magic)
        cout << " Ваш вариант больше магического числа.\n";
    else
        cout << " Ваш вариант меньше магического числа.\n";
}

return 0;
}

```

## “Лестничная” конструкция if-else-if

Очень распространенной в программировании конструкцией, в основе которой лежит вложенная if-инструкция, является “лестница” if-else-if. Ее можно представить в следующем виде.

```

if(условие)
    инструкция;
else if(условие)
    инструкция;
else if(условие)
    инструкция;
.
.
.
else
    инструкция;

```

Здесь под элементом *условие* понимается условное выражение. Условные выражения вычисляются сверху вниз. Как только в какой-нибудь ветви обнаружится истинный результат, будет выполнена инструкция, связанная с этой ветвью, а вся остальная “лестница” опускается. Если окажется, что ни одно из условий не является истинным, будет выполнена последняя *else*-инструкция (можно считать, что она выполняет роль условия, которое действует

## 116 Модуль 3. Инструкции управления

по умолчанию). Если последняя `else`-инструкция не задана, а все остальные оказались ложными, то вообще никакое действие не будет выполнено.

Работа `if-else-if`-“лестницы” демонстрируется в следующей программе.

```
// Демонстрация использования “лестницы” if-else-if.
```

```
#include <iostream>
using namespace std;

int main()
{
    int x;

    for(x=0; x<6; x++) {
        if(x==1) cout << "x равен единице.\n";
        else if(x==2) cout << "x равен двум.\n";
        else if(x==3) cout << "x равен трем.\n";
        else if(x==4) cout << "x равен четырем.\n";
        else cout << "x не попадает в диапазон от 1 до 4.\n";
    }

    return 0;
}
```

Результаты выполнения этой программы таковы.

x не попадает в диапазон от 1 до 4.

x равен единице.

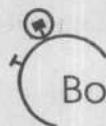
x равен двум.

x равен трем.

x равен четырем.

x не попадает в диапазон от 1 до 4.

Как видите, последняя `else`-инструкция выполняется только в том случае, если все предыдущие `if`-условия дали ложный результат.



## Вопросы для текущего контроля

1. Верно ли то, что условие, управляющее `if`-инструкцией, должно обязательно использовать оператор отношения?
2. С какой `if`-инструкцией всегда связана `else`-инструкция?
3. Что такое `if-else-if`—“лестница”?\*

**ВАЖНО!**

### 3.2. Инструкция `switch`

Теперь познакомимся с еще одной инструкцией выбора – `switch`. Инструкция `switch` обеспечивает многонаправленное ветвление. Она позволяет делать выбор одной из множества альтернатив. Хотя многонаправленное тестирование можно реализовать с помощью последовательности вложенных `if`-инструкций, во многих ситуациях инструкция `switch` оказывается более эффективным решением. Она работает следующим образом. Значение выражения последовательно сравнивается с константами из заданного списка. При обнаружении совпадения для одного из условий сравнения выполняется последовательность инструкций, связанная с этим условием. Общий формат записи инструкции `switch` таков.

```
switch(выражение) {
    case константа1:
        последовательность инструкций
        break;
    case константа2:
        последовательность инструкций
        break;
    case константа3:
        последовательность инструкций
        break;
```

1. Не верно. Условие, управляющее `if`-инструкцией, должно при вычислении давать результат, который можно расценивать как одно из значений ИСТИНА или ЛОЖЬ.
2. Ветвь `else` всегда связана с ближайшей `if`-ветвью, которая находится в том же программном блоке, но еще не связана ни с какой другой `else`-инструкцией.
3. “Лестница” `if-else-if` – это последовательность вложенных `if-else`-инструкций.

```
default:  
    последовательность инструкций  
}
```

Элемент *выражение* инструкции *switch* должен при вычислении давать целочисленное или символьное значение. (Выражения, имеющие, например, тип с плавающей точкой, не разрешены.) Очень часто в качестве управляющего *switch*-выражения используется одна переменная.

Последовательность инструкций *default*-ветви выполняется в том случае, если ни одна из заданных *case*-констант не совпадет с результатом вычисления *switch*-выражения. Ветвь *default* необязательна. Если она отсутствует, то при несовпадении результата выражения ни с одной из *case*-констант никакое действие выполнено не будет. Если такое совпадение все-таки обнаружится, будут выполняться инструкции, соответствующие данной *case*-ветви, до тех пор, пока не встретится инструкция *break* или не будет достигнут конец *switch*-инструкции (либо в *default*-, либо в последней *case*-ветви).

Итак, для применения *switch*-инструкции необходимо знать следующее.

- Инструкция *switch* отличается от инструкции *if* тем, что *switch*-выражение можно тестировать только с использованием условия равенства (т.е. на совпадение *switch*-выражения с одной из заданных *case*-констант), в то время как условное *if*-выражение может быть любого типа.
- Никакие две *case*-константы в одной *switch*-инструкции не могут иметь идентичных значений.
- Инструкция *switch* обычно более эффективна, чем вложенные *if*-инструкции.
- Последовательность инструкций, связанная с каждой *case*-ветвью, *не является блоком*. Однако полная *switch*-инструкция определяет блок. Значимость этого факта станет очевидной после того, как вы больше узнаете о C++.

Использование *switch*-инструкции демонстрируется в следующей программе. После запуска программа предлагает пользователю ввести число от 1 до 3, а затем отображает пословицу, связанную с выбранным числом. При вводе числа, не входящего в диапазон 1–3, выводится сообщение об ошибке.

```
/*
```

Простой “генератор” пословиц, который демонстрирует использование инструкции *switch*.

```
*/
```

```
#include <iostream>
```

```

using namespace std;

int main()
{
    int num;

    cout << "Введите число от 1 до 3: ";
    cin >> num;

    switch(num) { // Значение переменной num определяет
                  // выполняемую case-ветвь.

        case 1:
            cout << "Один пирог два раза не съешь.\n";
            break;
        case 2:
            cout << "Двум головам на одних плечах тесно.\n";
            break;
        case 3:
            cout << "Три раза прости, а в четвертый прихворости.\n";
            break;
        default:
            cout << "Вы должны ввести число 1, 2 или 3.\n";
    }

    return 0;
}

```

Вот как выглядят возможные варианты выполнения этой программы.

Введите число от 1 до 3: 1  
Один пирог два раза не съешь.

Введите число от 1 до 3: 5  
Вы должны ввести число 1, 2 или 3.

Формально инструкция `break` необязательна, хотя в большинстве случаев использования `switch`-конструкций она присутствует. Инструкция `break`, стоящая в последовательности инструкций любой `case`-ветви, приводит к выходу из всей `switch`-конструкции и передает управление инструкции, расположенной сразу после нее. Но если инструкция `break` в `case`-ветви отсутствует, будут выполнены все инструкции, связанные с данной `case`-ветвью, а также все последующие инструкции, расположенные за ней, до тех пор, пока все-таки не встре-

тится инструкция `break`, относящаяся к одной из последующих `case`-ветвей, или не будет достигнут конец `switch`-конструкции. Рассмотрите внимательно текст следующей программы. Попробуйте предугадать, что будет отображено на экране при ее выполнении.

```
// Конструкция switch без инструкций break.

#include <iostream>
using namespace std;

int main()
{
    int i;

    for(i=0; i<5; i++) {
        switch(i) {
            case 0: cout << "меньше 1\n"; // инструкции break нет
            case 1: cout << "меньше 2\n"; // .
            case 2: cout << "меньше 3\n"; // .
            case 3: cout << "меньше 4\n"; // .
            case 4: cout << "меньше 5\n"; // инструкции break нет
        }
        cout << '\n';
    }

    return 0;
}
```

При выполнении программа генерирует такие результаты.

```
меньше 1
меньше 2
меньше 3
меньше 4
меньше 5
```

```
меньше 2
меньше 3
меньше 4
меньше 5
```

```
меньше 3
```

```
меньше 4
меньше 5
```

```
меньше 4
меньше 5
```

```
меньше 5
```

Как видно по результатам, если инструкция `break` в одной `case`-ветви отсутствует, выполняются инструкции, относящиеся к следующей `case`-ветви.

Как показано в следующем примере, в `switch`-конструкцию можно включать "пустые" `case`-ветви.

```
switch(i) {
    case 1:
    case 2: // ← Пустые case-ветви.
    case 3:
        cout << "Значение переменной i меньше 4.";
        break;
    case 4:
        cout << "Значение переменной i равно 4.";
        break;
```

Если переменная `i` в этом фрагменте кода получает значение 1, 2 или 3, отображается одно сообщение.

Значение переменной `i` меньше 4.

Если же значение переменной `i` равно 4, отображается другое сообщение.

Значение переменной `i` равно 4.

Использование "пачки" нескольких пустых `case`-ветвей характерно для случаев, когда при выборе нескольких констант необходимо выполнить один и тот же код.

## Вложенные инструкции `switch`

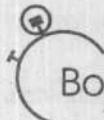
Инструкция `switch` может быть использована как часть `case`-последовательности внешней инструкции `switch`. В этом случае она называется *вложенной* инструкцией `switch`. Необходимо отметить, что `case`-константы внутренних и внешних инструкций `switch` могут иметь одинаковые значения, при этом никаких конфликтов не возникает. Например, следующий фрагмент кода вполне допустим.

```
switch(ch1) {
    case 'A': cout << "Эта A-ветвь - часть внешней switch";
    switch(ch2) { // ← Вложенная инструкция switch.
```

```

case 'A':
    cout << "Эта А-ветвь - часть внутренней switch";
    break;
case 'B': // ...
}
break;
case 'B': // ...

```



## Вопросы для текущего контроля

1. Какого типа должно быть выражение, управляющее инструкцией `switch`?
2. Что происходит в случае, если результат вычисления `switch`-выражения совпадает с `case`-константой?
3. Что происходит, если `case`-последовательность не завершается инструкцией `break`?\*

## Спросим у опытного программиста

**Вопрос.** В каких случаях кодирования многонаправленного ветвления лучше использовать `if-else-if`-“лестницу”, а не инструкцию `switch`?

**Ответ.** Используйте `if-else-if`-“лестницу” в том случае, когда условия, управляющие процессом выбора, нельзя определить одним значением. Рассмотрим, например, следующую `if-else-if`-последовательность.

```

if(x < 10) // ...
    else if(y > 0) // ...
        else if(!done) // ...

```

Эту последовательность нельзя перепрограммировать с помощью инструкции `switch`, поскольку все три условия включают различные переменные и различные типы. Какая переменная управляла бы инструкцией `switch`? Кроме того, `if-else-if`-“лестницу” придется использовать и в случае, когда нужно проверять значения с плавающей точкой или другие объекты, типы которых не подходят для использования в `switch`-выражении.

1. Выражение, управляющее инструкцией `switch`, должно иметь целочисленный или символьный тип.
2. Если результат вычисления `switch`-выражения совпадает с `case`-константой, то выполняется последовательность инструкций, связанная с этой `case`-константой.
3. Если `case`-последовательность не завершается инструкцией `break`, то выполняется следующая `case`-последовательность инструкций.

## Проект 3.1. Построение справочной системы C++

### Help.cpp

Этот проект создает простую справочную систему, которая подсказывает синтаксис управляющих C++-инструкций. После запуска наша программа отображает меню, содержащее инструкции управления, и переходит в режим ожидания до тех пор, пока пользователь не сделает свой выбор. Введенное пользователем значение используется в инструкции `switch` для отображения синтаксиса соответствующей инструкции. В первой версии нашей простой справочной системы можно получить "справку" только по `if`- и `switch`-инструкциям. Описание синтаксиса других управляющих C++-инструкций будет добавлено в последующих проектах.

### Последовательность действий

1. Создайте файл `Help.cpp`.
2. Выполнение программы должно начинаться с отображения следующего меню.

Справка по инструкциям:

1. `if`
2. `switch`

Выберите вариант справки:

Для реализации этого меню используйте следующую последовательность инструкций.

```
cout << "Справка по инструкциям:\n";
cout << " 1. if\n";
cout << " 2. switch\n";
cout << "Выберите вариант справки: ";
```

3. Затем программа должна принять вариант пользователя.
- ```
cin >> choice;
```
4. Получив вариант пользователя, программа применяет инструкцию `switch` для отображения синтаксиса выбранной инструкции.

```
switch(choice) {
    case '1':
        cout << "Синтаксис инструкции if:\n\n";
        cout << "if(условие) инструкция;\n";
        cout << "else инструкция;\n";
        break;
```

## 124 Модуль 3. Инструкции управления

```
case '2':  
    cout << "Синтаксис инструкции switch:\n\n";  
    cout << "switch(выражение) {\n";  
    cout << "    case константа:\n";  
    cout << "        последовательность инструкций\n";  
    cout << "        break;\n";  
    cout << "    // ... \n";  
    cout << "}\n";  
    break;  
default:  
    cout << "Такого варианта нет.\n";  
}
```

### 5. Приведем полный листинг программы Help.cpp.

```
/*  
Проект 3.1.  
  
Простая справочная система.  
*/  
  
#include <iostream>  
using namespace std;  
  
int main() {  
    char choice;  
  
    cout << "Справка по инструкциям:\n";  
    cout << " 1. if\n";  
    cout << " 2. switch\n";  
    cout << "Выберите вариант справки: ";  
    cin >> choice;  
  
    cout << "\n";  
    *  
    switch(choice) {  
        case '1':  
            cout << "Синтаксис инструкции if:\n\n";  
            cout << "if(условие) инструкция;\n";  
            cout << "else инструкция;\n";
```

```

        break;
    case '2':
        cout << "Синтаксис инструкции switch:\n\n";
        cout << "switch(выражение) {\n";
        cout << "    case константа:\n";
        cout << "        последовательность инструкций\n";
        cout << "        break;\n";
        cout << "    // ...\n";
        cout << "}\n";
        break;
    default:
        cout << "Такого варианта нет.\n";
}
}

return 0;
}

```

Вот один из возможных вариантов выполнения этой программы.

Справка по инструкциям:

1. if
2. switch

Выберите вариант справки: 1

Синтаксис инструкции if:

```

if(условие) инструкция;
else инструкция;

```

**ВАЖНО!**

### 3.3. Цикл `for`

В модуле 2 мы уже использовали простую форму цикла `for`. Здесь же мы рассмотрим его более детально, и вы узнаете, насколько мощным и гибким средством программирования он является. Начнем с традиционных форм его использования.

Итак, общий формат записи цикла `for` для многократного выполнения одной инструкции имеет следующий вид.

```
for(инициализация; выражение; инкремент) инструкция;
```

Если цикл `for` предназначен для многократного выполнения не одной инструкции, а программного блока, то его общий формат выглядит так.

## 126 Модуль 3. Инструкции управления

```
for (инициализация; выражение; инкремент)
{
    последовательность инструкций
}
```

Элемент *инициализация* обычно представляет собой инструкцию присваивания, которая устанавливает *управляющую переменную цикла* равной начальному значению. Эта переменная действует в качестве счетчика, который управляет работой цикла. Элемент *выражение* представляет собой условное выражение, в котором тестируется значение управляющей переменной цикла. Результат этого тестирования определяет, выполнится ли цикл *for* еще раз или нет. Элемент *инкремент* – это выражение, которое определяет, как изменяется значение управляющей переменной цикла после каждой итерации. Обратите внимание на то, что все эти элементы цикла *for* должны отделяться точкой с запятой. Цикл *for* будет выполняться до тех пор, пока вычисление элемента *выражение* даст истинный результат. Как только это условное выражение станет ложным, цикл завершится, а выполнение программы продолжится с инструкции, следующей за циклом *for*.

В следующей программе цикл *for* используется для вывода значений квадратного корня, извлеченных из чисел от 1 до 99. В этом примере управляющая переменная цикла называется *num*.

```
// Вывод значений квадратного корня из чисел от 1 до 99.

#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    int num;
    double sq_root;

    for(num=1; num < 100; num++) {
        sq_root = sqrt((double) num);
        cout << num << " " << sq_root << '\n';
    }

    return 0;
}
```

В этой программе использована стандартная C++-функция *sqrt()*. Как разъяснялось в модуле 2, эта функция возвращает значение квадратного корня

из своего аргумента. Аргумент должен иметь тип `double`, и именно поэтому при вызове функции `sqrt()` параметр `num` приводится к типу `double`. Сама функция также возвращает значение типа `double`. Обратите внимание на то, что в программу включен заголовок `<cmath>`, поскольку этот заголовочный файл обеспечивает поддержку функции `sqrt()`.

Управляющая переменная цикла `for` может изменяться как с положительным, так и с отрицательным приращением, причем величина этого приращения также может быть любой. Например, следующая программа выводит числа в диапазоне от 50 до -50 с декрементом, равным 10.

```
// Использование в цикле for отрицательного приращения.
```

```
#include <iostream>
using namespace std;

int main()
{
    int i;

    for(i=50; i >= -50; i = i-10) cout << i << ' ';
    return 0;
}
```

При выполнении программа генерирует такие результаты.

```
50 40 30 20 10 0 -10 -20 -30 -40 -50
```

Важно понимать, что условное выражение всегда тестируется в начале выполнения цикла `for`. Это значит, что если первая же проверка условия даст значение **ЛОЖЬ**, код тела цикла не выполнится ни разу. Вот пример:

```
for(count=10; count < 5; count++)
    cout << count; // Эта инструкция не выполнится.
```

Этот цикл никогда не выполнится, поскольку уже при входе в него значение его управляющей переменной `count` больше пяти. Это делает условное выражение (`count < 5`) ложным с самого начала. Поэтому даже одна итерация этого цикла не будет выполнена.

## Вариации на тему цикла `for`

Цикл `for` — одна из наиболее гибких инструкций в C#, поскольку она позволяет получить широкий диапазон вариантов ее применения. Например, для

## 128 Модуль 3. Инструкции управления

управления циклом `for` можно использовать несколько переменных. Рассмотрим следующий фрагмент кода.

```
for(x=0, y=10; x <= y; ++x, --y) // Сразу несколько
   // управляющих переменных.
    cout << x << ' ' << y << '\n';
```

Здесь запятыми отделяются две инструкции инициализации и два инкрементных выражения. Это делается для того, чтобы компилятор “понимал”, что существует две инструкции инициализации и две инструкции инкремента (декремента). В C++ запятая представляет собой оператор, который, по сути, означает “сделай это и то”. Другие применения оператора “запятая” мы рассмотрим ниже в этой книге, но чаще всего он используется в цикле `for`. В разделах инициализации и инкремента цикла `for` можно использовать любое количество инструкций, но обычно их число не превышает двух.

### Спросим у опытного программиста

**Вопрос.** Поддерживает ли C++, помимо `sqrt()`, другие математические функции?

**Ответ.** Да! Помимо функции `sqrt()`, C++ поддерживает широкий набор математических функций, например `sin()`, `cos()`, `tan()`, `log()`, `ceil()` и `floor()`. Если вас интересует программирование в области математики, вам стоит ознакомиться с составом C++-библиотеки математических функций. Это сделать нетрудно, поскольку все C++-компиляторы их поддерживают, а их описание можно найти в документации, прилагаемой к вашему компилятору. Помните, что все математические функции требуют включения в программу заголовка `<cmath>`.

Условным выражением, которое управляет циклом `for`, может быть любое допустимое C++-выражение. При этом оно необязательно должно включать управляющую переменную цикла. В следующем примере цикл будет выполняться до тех пор, пока функция `rand()` не сгенерирует значение, которое превышает число 20 000.

```
/*
```

Выполнение цикла продолжается здесь до тех пор, пока случайное число не превысит 20000.

```
*/
```

```
#include <iostream>
#include <cstdlib>
using namespace std;
```

```

int main()
{
    int i;
    int r;

    r = rand();

    for(i=0; r <= 20000; i++) // В этом условном выражении
        // управляющая переменная цикла
        // не используется.

    r = rand();

    cout << "Число равно " << r <<
    ". Оно сгенерировано при попытке " << i << ".";

    return 0;
}

```

Вот один из возможных результатов выполнения этой программы.

Число равно 26500. Оно сгенерировано при попытке 3.

При каждом проходе цикла с помощью функции `rand()` генерируется новое случайное число. Когда оно превысит число 20 000, условие продолжения цикла станет ложным, и цикл прекратится.

## Отсутствие элементов в определении цикла

В C++ разрешается опустить любой элемент заголовка цикла (инициализация, условное выражение, инкремент) или даже все сразу. Например, мы хотим написать цикл, который должен выполняться до тех пор, пока с клавиатуры не будет введено число 123. Вот как выглядит такая программа.

// Цикл for без инкремента в заголовке.

```

#include <iostream>
using namespace std;

int main()
{

```

```
int x;

for(x=0; x != 123; ) { // ← Выражения инкремента нет.
    cout << "Введите число: ";
    cin >> x;
}

return 0;
}
```

Здесь в заголовке цикла `for` отсутствует выражение инкремента. Это означает, что при каждом повторении цикла выполняется только одно действие: значение переменной `x` сравнивается с числом 123. Но если ввести с клавиатуры число 123, условное выражение, проверяемое в цикле, станет ложным, и цикл завершится. Поскольку выражение инкремента в заголовке цикла `for` отсутствует, управляющая переменная цикла не модифицируется.

Приведем еще один вариант цикла `for`, в заголовке которого, как показано в следующем фрагменте кода, отсутствует раздел инициализации.

```
x = 0; // ← Переменная x инициализируется вне цикла.
```

```
for( ; x<10; )
{
    cout << x << ' ';
    ++x;
}
```

Здесь пустует раздел инициализации, а управляющая переменная `x` инициализируется до входа в цикл. К размещению выражения инициализации за пределами цикла, как правило, прибегают только в том случае, когда начальное значение генерируется сложным процессом, который неудобно поместить в определение цикла. Обратите внимание также на то, что в этом примере элемент инкремента цикла `for` расположен не в заголовке, а в теле цикла.

### Бесконечный цикл `for`

Оставив пустым условное выражение цикла `for`, можно создать *бесконечный цикл* (цикл, который никогда не заканчивается). Способ создания такого цикла показан на примере следующей конструкции цикла `for`.

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

Этот цикл будет работать без конца. Несмотря на существование некоторых задач программирования (например, командных процессоров операционных систем), которые требуют наличия бесконечного цикла, большинство “бесконечных циклов” — это просто циклы со специальными требованиями к завершению. Ближе к концу этого модуля будет показано, как завершить цикл такого типа. (Подсказка: с помощью инструкции `break`.)

## Циклы без тела

В C++ тело цикла `for` может быть пустым, поскольку в C++ синтаксически допустима *пустая инструкция*. “Бестелесные” циклы не только возможны, но зачастую и полезны. Например, в следующей программе один из таких циклов используется для суммирования чисел от 1 до 10.

```
// Пример использования цикла for с пустым телом.

#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int i;
    int sum = 0;

    // Суммируем числа от 1 до 10.
    for(i=1; i <= 10; sum += i++) ; // ← “Бестелесный” цикл.

    cout << "Сумма чисел равна " << sum;

    return 0;
}
```

Результат выполнения этой программы выглядит так.

Сумма чисел равна 55

Обратите внимание на то, что процесс суммирования полностью реализуется в заголовке инструкции `for`, и поэтому в теле цикла нет никакой необходимости. Обратите также внимание на следующее выражение инкрементирования.

`sum += i++`

Не стоит опасаться инструкций, подобных этой. Они часто встречаются в профессиональных C++-программах. Если разбить такую инструкцию на части, то она окажется совсем несложной. Русским языком выполняемые ею действия можно выразить так: “Прибавим к значению переменной `sum` значение переменной `i` и сохраним результат сложения в переменной `sum`, а затем инкрементируем переменную `i`”. Другими словами, приведенную выше инструкцию можно переписать так.

```
sum = sum + i;  
i++;
```

## Объявление управляющей переменной цикла в заголовке инструкции `for`

Зачастую переменная, которая управляет циклом `for`, нужна только для этого цикла и ни для чего больше. В этом случае такую переменную можно объявить прямо в разделе инициализации заголовка инструкции `for`. Например, в следующей программе, которая вычисляет как сумму чисел от 1 до 5, так и их факториал, управляющая переменная цикла `i` объявляется в самом заголовке инструкции `for`.

```
// Объявление управляющей переменной цикла в заголовке  
// инструкции for.
```

```
#include <iostream>  
using namespace std;  
  
int main() {  
    int sum = 0;  
    int fact = 1;  
  
    // Вычисляем факториал чисел от 1 до 5.  
    for(int i = 1; i <= 5; i++) { // Переменная i объявлена  
                                // в самом цикле for.  
        sum += i; // Переменная i известна в рамках цикла,  
        fact *= i;  
    }  
  
    // но здесь она уже не существует.  
  
    cout << "Сумма чисел равна " << sum << "\n";
```

```

cout << "Факториал равен " << fact;
return 0;
}

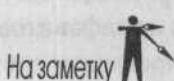
```

При выполнении эта программа генерирует такие результаты.

Сумма чисел равна 15

Факториал равен 120

При объявлении переменной в заголовке цикла важно помнить следующее: эта переменная известна только в рамках инструкции `for`. В этом случае говорят, что *область видимости* такой переменной ограничивается циклом `for`. Вне его она прекращает свое существование. Следовательно, в предыдущем примере переменная `i` недоступна вне цикла `for`. Если же вы собираетесь использовать управляющую переменную цикла в другом месте программы, ее не следует объявлять в заголовке цикла `for`.



В ранних версиях C++ переменная, объявленная в разделе инициализации заголовка инструкции `for`, была доступна и вне цикла `for`. Но в результате процесса стандартизации положение вещей изменилось. В настоящее время стандарт ANSI/ISO ограничивает область видимости такой переменной рамками цикла `for`. Однако в некоторых компиляторах это требование не реализовано. Поэтому вам имеет смысл прояснить этот момент в своей рабочей среде.



### Вопросы для текущего контроля

1. Могут ли некоторые разделы заголовка инструкции `for` быть пустыми?
2. Покажите, как на основе инструкции `for` можно создать бесконечный цикл?
3. Какова область видимости переменной, объявленной в заголовке инструкции `for`?\*

1. Да. Пустыми могут быть все три раздела заголовка инструкции `for`: инициализация, выражение и инкремент.
2. `for ( ; ; )`
3. Область видимости переменной, объявленной в заголовке инструкции `for`, ограничивается рамками цикла `for`. Вне его она неизвестна.

**ВАЖНО!**

### **3.4. Цикл while**

Познакомимся с еще одной инструкцией циклической обработки данных: `while`. Общая форма цикла `while` имеет такой вид:

```
while (выражение) инструкция;
```

Здесь под элементом *инструкция* понимается либо одиночная инструкция, либо блок инструкций. Работой цикла управляет элемент *выражение*, который представляет собой любое допустимое C++-выражение. Элемент *инструкция* выполняется до тех пор, пока условное выражение возвращает значение ИСТИНА. Как только это выражение становится ложным, управление передается инструкции, которая следует за этим циклом.

Использование цикла `while` демонстрируется на примере следующей небольшой программы. Практически все компиляторы поддерживают расширенный набор символов, который не ограничивается символами ASCII. В расширенный набор часто включаются специальные символы и некоторые буквы из алфавитов иностранных языков. ASCII-символы используют значения, не превышающие число 127, а расширенный набор символов — значения из диапазона 128–255. При выполнении этой программы выводятся все символы, значения которых лежат в диапазоне 32–255 (32 — это код пробела). Выполнив эту программу, вы должны увидеть ряд очень интересных символов.

```
/* Эта программа выводит все печатаемые символы,
включая расширенный набор символов, если
таковой существует.
```

```
*/
```

```
#include <iostream>
using namespace std;

int main()
{
    unsigned char ch;

    ch = 32;
    while(ch) { // Начало цикла while.
        cout << ch;
        ch++;
    }
}
```

```
return 0;
}
```

Рассмотрим while-выражение из предыдущей программы. Возможно, вас удивило, что оно состоит всего лишь из одной переменной ch. Но “ларчик” открывается просто. Поскольку переменная ch имеет здесь тип unsigned char, она может содержать значения от 0 до 255. Если ее значение равно 255, то после инкрементирования оно “сбрасывается” в нуль. Следовательно, факт равенства значения переменной ch нулю служит удобным способом завершить while-цикла.

Подобно циклу for, условное выражение проверяется при входе в цикл while, а это значит, что тело цикла (при ложном результате вычисления условного выражения) может не выполниться ни разу. Это свойство цикла устраниет необходимость отдельного тестирования до начала цикла. Следующая программа выводит строку, состоящую из точек. Количество отображаемых точек равно значению, которое вводит пользователь. Программа не позволяет “рисовать” строки, если их длина превышает 80 символов. Проверка на допустимость числа выводимых точек выполняется внутри условного выражения цикла, а не снаружи.

```
#include <iostream>
using namespace std;

int main()
{
    int len;

    cout << "Введите длину строки (от 1 до 79): ";
    cin >> len;

    while(len>0 && len<80) {
        cout << '.';
        len--;
    }

    return 0;
}
```

Если значение переменной len не “вписывается” в заданный диапазон (1–79), то цикл while не выполнится даже один раз. В противном случае его тело будет повторяться до тех пор, пока значение переменной len не станет равным нулю.

Тело while-цикла может вообще не содержать ни одной инструкции. Вот пример:

```
while(rand() != 100) ;
```

Этот цикл выполняется до тех пор, пока случайное число, генерируемое функцией `rand()`, не окажется равным числу 100.

**ВАЖНО!**

### 3.5. Цикл do-while

Настало время рассмотреть последний из C++-циклов: `do-while`. В отличие от циклов `for` и `while`, в которых условие проверяется при входе, цикл `do-while` проверяет условие при выходе из цикла. Это значит, что цикл `do-while` всегда выполняется хотя бы один раз. Его общий формат имеет такой вид.

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

Несмотря на то что фигурные скобки необязательны, если элемент *инструкции* состоит только из одной инструкции, они часто используются для улучшения читабельности конструкции `do-while`, не допуская тем самым путаницы с циклом `while`. Цикл `do-while` выполняется до тех пор, пока остается истинным элемент *выражение*, который представляет собой условное выражение.

В следующей программе цикл `do-while` выполняется до тех пор, пока пользователь не введет число 100.

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int num;  
  
    do { // Цикл do-while всегда выполняется  
        // хотя бы один раз.  
        cout << "Введите число (100 - для выхода): ";  
        cin >> num;  
    } while(num != 100);  
  
    return 0;  
}
```

Используя цикл `do-while`, мы можем еще более усовершенствовать программу "Угадай магическое число". На этот раз программа "не выпустит" вас из цикла угадывания, пока вы не угадаете это число.

```
// Программа "Угадай магическое число":  
// 3-е усовершенствование.  
  
#include <iostream>  
#include <cstdlib>  
using namespace std;  
  
int main()  
{  
    int magic; // магическое число  
    int guess; // вариант пользователя  
  
    magic = rand(); // Получаем случайное число.  
  
    do {  
        cout << "Введите свой вариант магического числа: ";  
        cin >> guess;  
        if(guess == magic) {  
            cout << "*** Правильно ** ";  
            cout << magic <<  
                " и есть то самое магическое число.\n";  
        }  
        else {  
            cout << "...Очень жаль, но вы ошиблись.";  
            if(guess > magic)  
                cout <<  
                    " Ваше число больше магического.\n";  
            else cout <<  
                " Ваше число меньше магического.\n";  
        }  
    } while(guess != magic);  
  
    return 0;  
}
```

Вот как выглядит один из возможных вариантов выполнения этой программы.

## 138 Модуль 3. Инструкции управления

Введите свой вариант магического числа: 10

...Очень жаль, но вы ошиблись. Ваше число меньше магического.

Введите свой вариант магического числа: 100

...Очень жаль, но вы ошиблись. Ваше число больше магического.

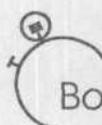
Введите свой вариант магического числа: 50

...Очень жаль, но вы ошиблись. Ваше число больше магического.

Введите свой вариант магического числа: 41

\*\* Правильно \*\* 41 и есть то самое магическое число.

И еще. Подобно циклам `for` и `while`, тело цикла `do-while` может быть пустым, но этот случай редко применяется на практике.



### Вопросы для текущего контроля

1. Какое основное различие между циклами `while` и `do-while`?
2. Может ли тело цикла `while` быть пустым?\*

### Спросим у опытного программиста

**Вопрос.** Учитывая "врожденную" гибкость всех C++-циклов, какой критерий следует использовать при выборе цикла? Другими словами, как выбрать цикл, наиболее подходящий для данной конкретной задачи?

**Ответ.** Используйте цикл `for` для выполнения известного количества итераций. Цикл `do-while` подойдет для тех ситуаций, когда необходимо, чтобы тело цикла выполнилось хотя бы один раз. Цикл `while` лучше всего использовать в случаях, когда его тело должно повториться неизвестное количество раз.

## Проект 3.2. Усовершенствование справочной системы C++

Help2.cpp

Этот проект – попытка усовершенствовать справочную систему C++, которая была создана в проекте 3.1. В этой версии добавлен синтаксис для циклов `for`, `while` и `do-while`. Кроме того, здесь проверя-

1. В цикле `while` условие проверяется при входе, а в цикле `do-while` – при выходе. Это значит, что цикл `do-while` всегда выполняется хотя бы один раз.
2. Да, тело цикла `while` (как и любого другого C++-цикла) может быть пустым.

ется значение, вводимое пользователем при выборе варианта меню, и с помощью цикла `do-while` реализуется прием только допустимого варианта. Следует отметить, что для обеспечения функционирования меню цикл `do-while` применяется очень часто, поскольку именно при обработке меню необходимо, чтобы цикл выполнился хотя бы один раз.

## Последовательность действий

1. Создайте файл `Help2.cpp`.
2. Измените ту часть программы, которая отображает команды меню, причем так, чтобы был использован цикл `do-while`.

```
do {
    cout << "Справка по инструкциям:\n";
    cout << " 1. if\n";
    cout << " 2. switch\n";
    cout << " 3. for\n";
    cout << " 4. while\n";
    cout << " 5. do-while\n";
    cout << "Выберите вариант справки: ";
    cin >> choice;

} while( choice < '1' || choice > '5');
```

3. Расширьте инструкцию `switch`, включив реализацию вывода синтаксиса по циклам `for`, `while` и `do-while`.

```
switch(choice) {
    case '1':
        cout << "Инструкция if:\n\n";
        cout << "if(условие) инструкция;\n";
        cout << "else инструкция;\n";
        break;
    case '2':
        cout << "Инструкция switch:\n\n";
        cout << "switch(выражение) {\n";
        cout << "    case константа:\n";
        cout << "        последовательность инструкций\n";
        cout << "        break;\n";
        cout << "    // ...\n";
        cout << "}\n";
```

```

        break;
    case '3':
        cout << "Инструкция for:\n\n";
        cout << "for(инициализация; условие; инкремент)";
        cout << " инструкция;\n";
        break;
    case '4':
        cout << "Инструкция while:\n\n";
        cout << "while(условие) инструкция;\n";
        break;
    case '5':
        cout << "Инструкция do-while:\n\n";
        cout << "do {\n";
        cout << " инструкция;\n";
        cout << "} while (условие);\n";
        break;
}

```

Обратите внимание на то, что в этой версии switch-инструкции отсутствует ветвь default. Поскольку цикл по обеспечению работы меню гарантирует ввод допустимого варианта, то для обработки некорректных вариантов инструкция default больше не нужна.

4. Приведем полный текст программы Help2.cpp.

```

/*
Проект 3.2.

Усовершенствованная версия справочной системы C++,
в которой для обработки вариантов меню используется
цикл do-while.
*/

#include <iostream>
using namespace std;

int main() {
    char choice;

    do {
        cout << "Справка по инструкциям:\n";
        cout << " 1. if\n";

```

```

cout << " 2. switch\n";
cout << " 3. for\n";
cout << " 4. while\n";
cout << " 5. do-while\n";
cout << "Выберите вариант справки: ";

cin >> choice;

} while( choice < '1' || choice > '5');

cout << "\n\n";

switch(choice) {
    case '1':
        cout << "Инструкция if:\n\n";
        cout << "if(условие) инструкция;\n";
        cout << "else инструкция;\n";
        break;
    case '2':
        cout << "Инструкция switch:\n\n";
        cout << "switch(выражение) {\n";
        cout << "    case константа:\n";
        cout << "        последовательность инструкций\n";
        cout << "        break;\n";
        cout << "    // ...\n";
        cout << "}\n";
        break;
    case '3':
        cout << "Инструкция for:\n\n";
        cout << "for (инициализация; условие; инкремент) ";
        cout << "    инструкция;\n";
        break;
    case '4':
        cout << "Инструкция while:\n\n";
        cout << "while(условие) инструкция;\n";
        break;
    case '5':
        cout << "Инструкция do-while:\n\n";
        cout << "do {\n";
        cout << "    инструкция;\n";
}

```

```

        cout << "} while (условие);\n";
        break;
    }

    return 0;
}

```

## Спросим у опытного программиста

**Вопрос.** Раньше было показано, что переменную можно объявить в разделе инициализации заголовка цикла `for`. Можно ли объявлять переменные внутри любых других управляемых C++-инструкций?

**Ответ.** Да. В C++ можно объявлять переменную в условном выражении `if`- или `switch`-инструкции, в условном выражении цикла `while` либо в разделе инициализации цикла `for`. Область видимости переменной, объявленной в одном из таких мест, ограничена блоком кода, управляемым соответствующей инструкцией.

Вы уже видели пример объявления переменной в цикле `for`. Теперь рассмотрим пример объявления переменной в условном выражении `if`-инструкции.

```

if(int x = 20) {
    x = x - y;
    if(x > 10) y = 0;
}

```

Здесь в `if`-инструкции объявляется переменная `x`, которой присваивается число 20. Поскольку результат присваивания (20) рассматривается как значение ИСТИНА, то будет выполнен блок кода, заключенный в фигурные скобки. Так как область видимости переменной, объявленной в условном выражении `if`-инструкции, ограничена блоком кода, управляемым этой инструкцией, то переменная `x` неизвестна вне этого `if`-блока.

Как упоминалось в разделе, посвященном циклу `for`, в разных компиляторах по-разному реализовано требование ограничения доступа к переменной, объявленной в управляемой инструкции, рамками этой инструкции. Поэтому, хотя в стандарте ANSI/ISO оговорено, что область видимости такой переменной должна быть ограничена рамками соответствующей управляемой инструкции, этот момент необходимо уточнить в документации, прилагаемой к вашему компилятору.

Большинство программистов не объявляет переменные в управляемых инструкциях, за исключением цикла `for`. И в самом деле, объявление переменной в других инструкциях — довольно спорный момент, и многие программисты считают это “дурным тоном” программирования.

ВАЖНО!

### 3.6. Использование инструкции `break` для выхода из цикла

С помощью инструкции `break` можно организовать немедленный выход из цикла, опустив выполнение кода, оставшегося в его теле, и проверку условного выражения. При обнаружении внутри цикла инструкции `break` цикл завершается, а управление передается инструкции, следующей после цикла. Рассмотрим простой пример.

```
#include <iostream>
using namespace std;

int main()
{
    int t;

    // Цикл работает для значений t от 0 до 9, а не до 100!
    for(t=0; t<100; t++) {
        if(t==10) break; // При t = 0 цикл for прекращается.
        cout << t << ' ';
    }

    return 0;
}
```

Результаты выполнения этой программы таковы.

```
0 1 2 3 4 5 6 7 8 9
```

Как подтверждают эти результаты, программа выводит на экран числа от 0 до 9, а не до 100, поскольку инструкция `break` при значении `t`, равном 10, обеспечивает немедленный выход из цикла.

При использовании вложенных циклов (т.е. когда один цикл включает другой) инструкция `break` обеспечивает выход только из наиболее глубоко вложенного цикла. Рассмотрим пример.

```
#include <iostream>
using namespace std;

int main()
{
```

```

int t, count;

for(t=0; t<10; t++) {
    count = 1;
    for(;;) {
        cout << count << ' ';
        count++;
        if(count==10) break;
    }
    cout << '\n';
}

return 0;
}

```

Вот какие результаты генерирует эта программа.

```

1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9

```

Как видите, эта программа выводит на экран числа от 1 до 9 десять раз. Каждый раз, когда во внутреннем цикле встречается инструкция `break`, управление передается во внешний цикл `for`. Обратите внимание на то, что внутренняя `for`-инструкция представляет собой бесконечный цикл, для завершения которого используется инструкция `break`.

Инструкцию `break` можно использовать для завершения любого цикла. Как правило, она предназначена для отслеживания специального условия, при наступлении которого необходимо немедленно прекратить цикл (чаще всего в таких случаях используются бесконечные циклы).

И еще. Если инструкция `break` применяется в инструкции `switch`, которая вложена в некоторый цикл, то она повлияет только на данную инструкцию `switch`, а не на цикл, т.е не обеспечит немедленный выход из этого цикла.

ВАЖНО!

### 3.7. Использование инструкции continue

В C++ существует средство “досрочного” выхода из текущей итерации цикла. Этим средством является инструкция `continue`. Она принудительно выполняет переход к следующей итерации, опуская выполнение оставшегося кода в текущей. Например, в следующей программе инструкция `continue` используется для “ускоренного” поиска четных чисел в диапазоне от 0 до 100.

```
#include <iostream>
using namespace std;

int main()
{
    int x;

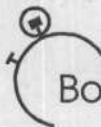
    for(x=0; x<=100; x++) {
        if(x%2) continue; // Если значение x нечетное,
                           // сразу переходим на следующую
                           // итерацию цикла.

        cout << x << ' ';
    }

    return 0;
}
```

Здесь выводятся только четные числа, поскольку при обнаружении нечетного числа происходит преждевременный переход к следующей итерации, и `cout`-инструкция опускается. (Вспомним, что с помощью оператора “%” мы получаем остаток при выполнении целочисленного деления. Это значит, что при нечетном `x` остаток от деления на два будет равен единице, которая (при оценке в качестве результата условного выражения) соответствует истинному значению. В противном случае (при четном `x`) остаток от деления на два будет равен нулю, т.е. значению ЛОЖЬ.)

В циклах `while` и `do-while` инструкция `continue` передает управление непосредственно инструкции, проверяющей условное выражение, после чего циклический процесс продолжает “идти своим чередом”. А в цикле `for` после выполнения инструкции `continue` сначала вычисляется инкрементное выражение, а затем – условное. И только после этого циклический процесс будет продолжен.



### Вопросы для текущего контроля

1. Что происходит при выполнении инструкции `break` в теле цикла?
2. В чем состоит назначение инструкции `continue`?
3. Если цикл включает инструкцию `switch`, то приведет ли к завершению этого цикла инструкция `break`, принадлежащая инструкции `switch`?\*

### Проект 3.3. Завершение построения справочной системы C++

Help3.cpp

Этот проект завершает построение справочной системы C++. Настоящая версия пополняется описанием синтаксиса инструкций `break`, `continue` и `goto`. Здесь пользователь может делать запрос уже не только по одной инструкции. Эта возможность реализована путем использования внешнего цикла, который работает до тех пор, пока пользователь не введет для выхода из программы команду меню `q`.

#### Последовательность действий

1. Скопируйте файл Help2.cpp в новый файл и назовите его Help3.cpp.
2. Заключите весь программный код в бесконечный цикл `for`. Обеспечьте выход из этого цикла с помощью инструкции `break` при вводе пользователем команды меню `q`. Прерывание этого цикла приведет к завершению всей программы в целом.
3. Измените цикл меню следующим образом.

```
do {  
    cout << "Справка по инструкциям:\n";  
    cout << " 1. if\n";
```

1. При выполнении инструкции `break` в теле цикла происходит немедленное завершение этого цикла. Выполнение программы продолжается с инструкции, следующей сразу после цикла.
2. Инструкция `continue` принудительно выполняет переход к следующей итерации цикла, опуская выполнение оставшегося кода в текущей.
3. Нет, если инструкция `break` применяется в инструкции `switch`, которая вложена в некоторый цикл, то она повлияет только на данную инструкцию `switch`, а не на цикл.

```

cout << " 2. switch\n";
cout << " 3. for\n";
cout << " 4. while\n";
cout << " 5. do-while\n";
cout << " 6. break\n";
cout << " 7. continue\n";
cout << " 8. goto\n";
cout << "Выберите вариант справки (q для выхода): ";
cin >> choice;
} while( choice < '1' || choice > '8' && choice != 'q');

```

Обратите внимание на то, что меню сейчас включает дополнительные инструкции: `break`, `continue` и `goto`. Кроме того, в качестве допустимой команды меню принимается вариант `q`.

4. Расширьте инструкцию `switch`, включив реализацию вывода синтаксиса по инструкциям `break`, `continue` и `goto`.

```

case '6':
    cout << "Инструкция break:\n\n";
    cout << "break;\n";
    break;
case '7':
    cout << "Инструкция continue:\n\n";
    cout << "continue;\n";
    break;
case '8':
    cout << "Инструкция goto:\n\n";
    cout << "goto метка;\n";
    break;

```

5. Вот как выглядит полный текст программы `Help3.cpp`.

```

/*
Проект 3.3.

```

Окончательная версия справочной системы C++, которая позволяет обрабатывать несколько запросов пользователя.

```
*/
```

```
#include <iostream>
using namespace std;
```

```
int main() {
    char choice;

    for(;;) {
        do {
            cout << "Справка по инструкциям:\n";
            cout << " 1. if\n";
            cout << " 2. switch\n";
            cout << " 3. for\n";
            cout << " 4. while\n";
            cout << " 5. do-while\n";
            cout << " 6. break\n";
            cout << " 7. continue\n";
            cout << " 8. goto\n";
            cout << "Выберите вариант справки (q для выхода): ";
            cin >> choice;
        } while( choice < '1' || choice > '8' && choice != 'q');

        if(choice == 'q') break;

        cout << "\n\n";

        switch(choice) {
            case '1':
                cout << "Инструкция if:\n\n";
                cout << "if(условие) инструкция;\n";
                cout << "else инструкция;\n";
                break;
            case '2':
                cout << "Инструкция switch:\n\n";
                cout << "switch(выражение) {\n";
                cout << "    case константа:\n";
                cout << "        последовательность инструкций\n";
                cout << "        break;\n";
                cout << "    // ...\n";
                cout << "}\n";
                break;
        }
    }
}
```

```

    case '3':
        cout << "Инструкция for:\n\n";
        cout << "for(инициализация; условие; инкремент) ";
        cout << " инструкция;\n";
        break;
    case '4':
        cout << "Инструкция while:\n\n";
        cout << "while(условие) инструкция;\n";
        break;
    case '5':
        cout << "Инструкция do-while:\n\n";
        cout << "do {\n";
        cout << " инструкция;\n";
        cout << "} while (условие);\n";
        break;
    case '6':
        cout << "Инструкция break:\n\n";
        cout << "break;\n";
        break;
    case '7':
        cout << "Инструкция continue:\n\n";
        cout << "continue;\n";
        break;
    case '8':
        cout << "Инструкция goto:\n\n";
        cout << "goto метка;\n";
        break;
    }
    cout << "\n";
}

return 0;
}

```

6. Вот как выглядит один из возможных вариантов выполнения этой программы.

Справка по инструкциям:

1. if
2. switch
3. for

## 150 Модуль 3. Инструкции управления

- 4. while
- 5. do-while
- 6. break
- 7. continue
- 8. goto

Выберите вариант справки (q для выхода): 1

Инструкция if:

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

Справка по инструкциям:

- 1. if
- 2. switch
- 3. for
- 4. while
- 5. do-while
- 6. break
- 7. continue
- 8. goto

Выберите вариант справки (q для выхода): 6

Инструкция break:

```
break;
```

Справка по инструкциям:

- 1. if
- 2. switch
- 3. for
- 4. while
- 5. do-while
- 6. break
- 7. continue
- 8. goto

Выберите вариант справки (q для выхода): q

ВАЖНО!

### 3.8. Вложенные циклы

Как было продемонстрировано на примере одной из предыдущих программ, один цикл можно вложить в другой. Вложенные циклы используются для решения задач самого разного профиля и часто используются в программировании. Например, в следующей программе вложенный цикл `for` позволяет найти множители для чисел в диапазоне от 2 до 100.

```
/*
Использование вложенных циклов для отыскания
множителей чисел в диапазоне от 2 до 100
*/
```

```
#include <iostream>
using namespace std;

int main() {

    for(int i=2; i <= 100; i++) {
        cout << "Множители числа " << i << ": ";
        for(int j = 2; j < i; j++)
            if((i%j) == 0) cout << j << " ";
        cout << "\n";
    }
    return 0;
}
```

Вот как выглядят часть результатов, генерируемых этой программой.

Множители числа 2:

Множители числа 3:

Множители числа 4: 2

Множители числа 5:

Множители числа 6: 2 3

Множители числа 7:

Множители числа 8: 2 4

Множители числа 9: 3

Множители числа 10: 2 5  
Множители числа 11:  
Множители числа 12: 2 3 4 6  
Множители числа 13:  
Множители числа 14: 2 7  
Множители числа 15: 3 5  
Множители числа 16: 2 4 8  
Множители числа 17:  
Множители числа 18: 2 3 6 9  
Множители числа 19:  
Множители числа 20: 2 4 5 10

В этой программе внешний цикл изменяет свою управляющую переменную *i* от 2 до 100. Во внутреннем цикле тестируются все числа от 2 до значения переменной *i* и выводятся те из них, на которые без остатка делится значение переменной *i*.

**ВАЖНО!**

### 3.9. Инструкция *goto*

Инструкция *goto* — это C++-инструкция безусловного перехода. При ее выполнении управление программой передается инструкции, указанной с помощью метки. Долгие годы эта инструкция находилась в немилости у программистов, поскольку способствовала, с их точки зрения, созданию “спагетти-кода”. Однако инструкция *goto* по-прежнему используется, и иногда даже очень эффективно. В этой книге не делается попытка “реабилитации” законных прав этой инструкции в качестве одной из форм управления программой. Более того, необходимо отметить, что в любой ситуации (в области программирования) можно обойтись без инструкции *goto*, поскольку она не является элементом, обеспечивающим полноту описания языка программирования. Вместе с тем, в определенных ситуациях ее использование может быть очень полезным. В этой книге было решено ограничить использование инструкции *goto* рамками этого раздела, так как, по мнению большинства программистов, она вносит в программу лишь беспорядок и делает ее практически нечитабельной. Но, поскольку использование инструкции *goto* в некоторых случаях может сделать намерение программиста яснее, ей стоит уделить некоторое внимание.

Инструкция *goto* требует наличия в программе метки. *Метка* — это действительный в C++ идентификатор, за которым поставлено двоеточие. При выполнении инструкции *goto* управление программой передается инструкции, указанной с помощью метки. Метка должна находиться в одной функции с инструк-

цией `goto`, которая ссылается на эту метку. Например, с помощью инструкции `goto` и метки можно организовать следующий цикл на 100 итераций.

```
x = 1;
loop1:
    x++;
    if(x < 100) goto loop1; // Выполнение программы будет
                           // продолжено с метки loop1.
```

Иногда инструкцию `goto` стоит использовать для выхода из глубоко вложенных инструкций цикла. Рассмотрим следующий фрагмент кода.

```
for(...) {
    for(...) {
        while(...) {
            if(...) goto stop;
            .
            .
            .
        }
    }
}
stop:
cout << "Ошибка в программе.\n";
```

Чтобы заменить инструкцию `goto`, пришлось бы выполнить ряд дополнительных проверок. В данном случае инструкция `goto` существенно упрощает программный код. Простым применением инструкции `break` здесь не обошлось, поскольку она обеспечила бы выход лишь из наиболее глубокого вложенного цикла.



### Тест для самоконтроля по модулю 3

- Напишите программу, которая считывает с клавиатуры символы до тех пор, пока не будет введен символ “\$”. Организуйте в программе подсчет количества введенных точек. Результаты подсчета должны выводиться по окончании выполнения программы.
- Может ли кодовая последовательность одной `case`-ветви в инструкции `switch` переходить в последовательность инструкций следующей `case`-ветви? Поясните свой ответ.
- Представьте общий формат `if-else-if`—“лестницы”.
- Рассмотрите следующий фрагмент кода.

## 154 Модуль 3. Инструкции управления

```
if(x < 10)
    if(y > 100) {
        if(!done) x = z;
        else y = z;
    }
else cout << "Ошибка!"; // С какой if-инструкцией
                        // связана эта else-ветвь?
```

Вопрос: с какой if-инструкцией связана последняя else-ветвь?

5. Напишите for-инструкцию для цикла, который считает от 1000 до 0 с шагом -2.

6. Допустим ли следующий фрагмент кода?

```
for(int i = 0; i < num; i++)
    sum += i;

count = i;
```

7. Поясните назначение инструкции break.

8. Что будет отображено на экране после выполнения инструкции break в следующем фрагменте кода?

```
for(i = 0; i < 10; i++) {
    while (running) {
        if(x < y) break;
        // ...
    }
    cout << "После инструкции while.\n";
}
cout << "После инструкции for.\n";
```

9. Что будет выведено на экран при выполнении следующего фрагмента кода?

```
for(int i = 0; i < 10; i++) {
    cout << i << " ";
    if(!(i%2)) continue;
    cout << "\n";
}
```

10. Инкрементное выражение в цикле for не обязательно должно изменять управляющую переменную цикла на фиксированную величину. Переменная цикла может изменяться произвольным образом. С учетом этого замечания напишите программу, которая использует цикл for, чтобы сгенерировать и отобразить прогрессию 1, 2, 4, 8, 16, 32 и т.д.

11. Код строчных букв ASCII отличается от кода прописных на 32. Таким образом, чтобы преобразовать строчную букву в прописную, необходимо вычесть из ее кода число 32. Используйте эту информацию при написании программы, которая бы считывала символы с клавиатуры. Перед отображением результата обеспечьте преобразование всех строчных букв в прописные, а всех прописных — в строчные. Другие же символы никаким изменениям подвергаться не должны. Организуйте завершение программы после ввода пользователем символа “точка”. Перед завершением программа должна отобразить количество выполненных преобразований (изменений регистра).

12. В чем состоит назначение C++-инструкции безусловного перехода?

На заметку

Ответы на эти вопросы можно найти на Web-странице данной книги по адресу: <http://www.osborne.com>.

# Модуль 4

## Массивы, строки и указатели

- 4.1.** Одномерные массивы
- 4.2.** Двумерные массивы
- 4.3.** Многомерные массивы
- 4.4.** Строки
- 4.5.** Некоторые библиотечные функции обработки строк
- 4.6.** Инициализация массивов
- 4.7.** Массивы строк
- 4.8.** Указатели
- 4.9.** Операторы, используемые с указателями
- 4.10.** Использование указателей в выражениях
- 4.11.** Указатели и массивы
- 4.12.** Многобуровневая непрямая адресация

В этом модуле мы рассматриваем массивы, строки и указатели. Эти темы могут показаться совсем не связанными одна с другой, но это только на первый взгляд. В C++ они тесно переплетены между собой, и, что важнее всего, если хорошо разобраться в одной из них, у вас не будет проблем и с остальными.

*Массив* (аггай) — это коллекция переменных одинакового типа, обращение к которым происходит с применением общего для всех имени. В C++ массивы могут быть одно- или многомерными, хотя в основном используются одномерные массивы. Массивы представляют собой удобное средство создания списков (группирования) связанных переменных.

Чаще всего в “повседневном” программировании используются символьные массивы, в которых хранятся строки. Как упоминалось выше, в C++ не определен встроенный тип данных для хранения строк. Поэтому *строки* реализуются как массивы символов. Такой подход к реализации строк дает C++-программисту больше “рычагов” управления по сравнению с теми языками, в которых используется отдельный строковый тип данных.

*Указатель* — это объект, который хранит адрес памяти. Обычно указатель используется для доступа к значению другого объекта. Чаще всего таким объектом является массив. В действительности указатели и массивы связаны друг с другом сильнее, чем можно было бы ожидать.

**ВАЖНО!**

## 4.1. Одномерные массивы

*Одномерный массив* — это список связанных переменных. Такие списки довольно популярны в программировании. Например, одномерный массив можно использовать для хранения учетных номеров активных пользователей сети или результатов игры любимой бейсбольной команды. Если потом нужно усреднить какие-то данные (содержащиеся в массиве), то это очень удобно сделать посредством обработки значений массива. Одним словом, массивы — незаменимое средство современного программирования.

Для объявления одномерного массива используется следующая форма записи.  
тип имя\_массива [размер];

Здесь с помощью элемента записи *тип* объявляется базовый тип массива. *Базовый тип* определяет тип данных каждого элемента, составляющего массив. Количество элементов, которые будут храниться в массиве, определяется элементом *размер*. Например, при выполнении приведенной ниже инструкции объявляется *int*-массив (состоящий из 10 элементов) с именем *sample*.

```
int sample[10];
```

Доступ к отдельному элементу массива осуществляется с помощью индекса. Индекс описывает позицию элемента внутри массива. В C++ первый элемент массива имеет нулевой индекс. Поскольку массив `sample` содержит 10 элементов, его индексы изменяются от 0 до 9. Чтобы получить доступ к элементу массива по индексу, достаточно указать нужный номер элемента в квадратных скобках. Так, первым элементом массива `sample` является `sample[0]`, а последним — `sample[9]`. Например, следующая программа помещает в массив `sample` числа от 0 до 9.

```
#include <iostream>
using namespace std;

int main()
{
    int sample[10]; // Эта инструкция резервирует область
                    // памяти для 10 элементов типа int.
    int t;

    // Помещаем в массив значения.
    for(t=0; t<10; ++t) sample[t] = t; // Обратите внимание
   // на то, как индексируется
   // массив sample.

    // Отображаем содержимое массива.
    for(t=0; t<10; ++t)
        cout << "Это элемент sample[" << t << "]: "
   << sample[t] << "\n";

    return 0;
}
```

Результаты выполнения этой программы выглядят так.

```
Это элемент sample[0]: 0
Это элемент sample[1]: 1
Это элемент sample[2]: 2
Это элемент sample[3]: 3
Это элемент sample[4]: 4
Это элемент sample[5]: 5
Это элемент sample[6]: 6
Это элемент sample[7]: 7
```

## 160 Модуль 4. Массивы, строки и указатели

Это элемент sample[8]: 8

Это элемент sample[9]: 9

В C++ все массивы занимают смежные ячейки памяти. (Другими словами, элементы массива в памяти расположены последовательно друг за другом.) Ячейка с наименьшим адресом относится к первому элементу массива, а с наибольшим — к последнему. Например, после выполнения этого фрагмента кода

```
int nums[5];
int i;

for(i=0; i<5; i++) nums[i] = i;
```

массив nums будет выглядеть следующим образом.

| nums[0] | nums[1] | nums[2] | nums[3] | nums[4] |
|---------|---------|---------|---------|---------|
| 0       | 1       | 2       | 3       | 4       |

Массивы часто используются в программировании, поскольку позволяют легко обрабатывать большое количество связанных переменных. Например, в следующей программе создается массив из десяти элементов, каждому элементу присваивается некоторое число, а затем вычисляется и отображается среднее от этих значений, а также минимальное и максимальное.

```
/*
Вычисление среднего и определение минимального
и максимального из набора значений.
*/
```

```
#include <iostream>
using namespace std;

int main()
{
    int i, avg, min_val, max_val;
    int nums[10];

    nums[0] = 10;
    nums[1] = 18;
    nums[2] = 75;
    nums[3] = 0;
    nums[4] = 1;
```

```

nums[5] = 56;
nums[6] = 100;
nums[7] = 12;
nums[8] = -19;
nums[9] = 88;

// Вычисление среднего значения.
avg = 0;
for(i=0; i<10; i++)
    avg += nums[i]; // ← Суммируем значения массива nums.

avg /= 10; // ← Вычисляем среднее значение.

cout << "Среднее значение равно " << avg << '\n';

// Определение минимального и максимального значений.
min_val = max_val = nums[0];
for(i=1; i<10; i++) {
    if(nums[i] < min_val) min_val = nums[i]; // минимум
    if(nums[i] > max_val) max_val = nums[i]; // максимум
}

cout << "Минимальное значение: " << min_val << '\n';
cout << "Максимальное значение: " << max_val << '\n';

return 0;
}

```

При выполнении эта программа генерирует такие результаты.

```

Среднее значение равно 34
Минимальное значение: -19
Максимальное значение: 100

```

Обратите внимание на то, как в программе опрашиваются элементы массива `nums`. Тот факт, что обрабатываемые здесь значения сохранены в массиве, значительно упрощает процесс определения среднего, минимального и максимального значений. Управляющая переменная цикла `for` используется в качестве индекса массива при доступе к очередному его элементу. Подобные циклы очень часто применяются при работе с массивами.

Используя массивы, необходимо иметь в виду одно очень важное ограничение. В C++ нельзя присвоить один массив другому. В следующем фрагменте кода, например, присваивание `a = b;` недопустимо.

```
int a[10], b[10];
```

```
// ...
```

```
a = b; // Ошибка!!!
```

Чтобы поместить содержимое одного массива в другой, необходимо отдельно выполнить присваивание каждого значения. Вот пример.

```
for(i=0; i<10; i++) a[i] = b[i];
```

## На границах массивов без пограничников

В C++ не выполняется никакой проверки “нарушения границ” массивов, т.е. ничего не может помешать программисту обратиться к массиву за его пределами. Другими словами, обращение к массиву (размером  $N$  элементов) за границей  $N$ -го элемента может привести к разрушению программы при отсутствии каких-либо замечаний со стороны компилятора и без выдачи сообщений об ошибках во время работы программы. Например, C++-компилятор “молча” скомпилирует и позволит запустить следующий код на выполнение, несмотря на то, что в нем происходит выход за границы массива `crash`.

```
int crash[10], i;
```

```
for(i=0; i<100; i++) crash[i]=i;
```

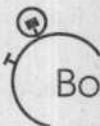
В данном случае цикл `for` выполнит 100 итераций, несмотря на то, что массив `crash` предназначен для хранения лишь десяти элементов. При выполнении этой программы возможна перезапись важной информации, что может привести к аварийному останову программы.

Если “нарушение границы” массива происходит при выполнении инструкции присваивания, могут быть изменены значения в ячейках памяти, выделенных некоторым другим переменным. Если границы массива нарушаются при чтении данных, то неверно считанные данные могут попросту разрушить вашу программу. В любом случае вся ответственность за соблюдение границ массивов лежит только на программах, которые должны гарантировать корректную работу с массивами. Другими словами, программист обязан использовать массивы достаточно большого размера, чтобы в них можно было без осложнений помещать данные, но лучше всего в программе предусмотреть проверку пересечения границ массивов.

## Спросим у опытного программиста

**Вопрос.** Если нарушение границ массива может привести к катастрофическим последствиям, то почему в C++ не предусмотрено операции "граничной" проверки доступа к массиву?

**Ответ.** Вероятно, такая "непредусмотрительность" C++, которая выражается в отсутствии встроенных средств динамической проверки на "неприкосненность" границ массивов, может кого-то удивить. Напомню, однако, что язык C++ предназначен для профессиональных программистов, и его задача — предоставить им возможность создавать максимально эффективный код. Любая проверка корректности доступа средствами C++ существенно замедляет выполнение программы. Поэтому подобные действия оставлены на рассмотрение программистам. Кроме того, при необходимости программист может сам определить тип массива и заложить в него проверку нерушимости границ.



## Вопросы для текущего контроля

- Что представляет собой одномерный массив?
- Индекс первого элемента массива всегда равен нулю. Верно ли это?
- Предусмотрена ли в C++ проверка "нерушимости" границ массива?\*

**ВАЖНО!**

## 4.2. Двумерные массивы

В C++ можно использовать многомерные массивы. Простейший многомерный массив — двумерный. Двумерный массив, по сути, представляет собой список одномерных массивов. Чтобы объявить двумерный массив целочисленных значений размером 10×20 с именем twoD, достаточно записать следующее:

```
int twoD[10][20];
```

Обратите особое внимание на это объявление. В отличие от многих других языков программирования, в которых при объявлении массива значения раз-

- Одномерный массив — это список связанных значений.
- Верно. Первый элемент массива всегда имеет нулевой индекс.
- Нет. В C++ встроенные средства динамической проверки на "неприкосненность" границ массивов отсутствуют.

## 164 Модуль 4. Массивы, строки и указатели

мерностей отделяются запятыми, в C++ каждая размерность заключается в собственную пару квадратных скобок. Так, чтобы получить доступ к элементу массива `twoD` с координатами 3,5, необходимо использовать запись `twoD[3][5]`.

В следующем примере в двумерный массив помещаются последовательные числа от 1 до 12.

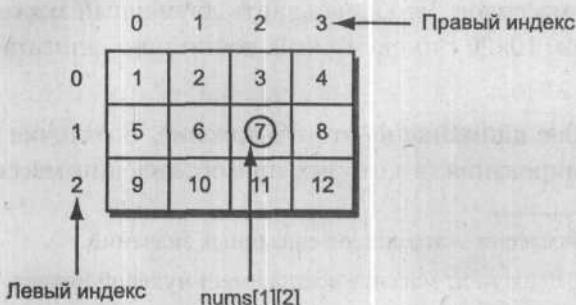
```
#include <iostream>
using namespace std;

int main()
{
    int t,i, nums[3][4];

    for(t=0; t < 3; ++t) {
        for(i=0; i < 4; ++i) {
            nums[t][i] = (t*4)+i+1; // Для индексации массива
                                    // nums требуется два
                                    // индекса.
            cout << nums[t][i] << ' ';
        }
        cout << '\n';
    }

    return 0;
}
```

В этом примере элемент `nums[0][0]` получит значение 1, элемент `nums[0][1]` — значение 2, элемент `nums[0][2]` — значение 3 и т.д. Значение элемента `nums[2][3]` будет равно числу 12. Схематически этот массив можно представить следующим образом.



В двумерном массиве позиция любого элемента определяется двумя индексами. Если представить двумерный массив в виде таблицы данных, то один индекс означает строку, а второй — столбец. Из этого следует, что, если к доступ элементам массива предоставить в порядке, в котором они реально хранятся в памяти, то правый индекс будет изменяться быстрее, чем левый.

Необходимо помнить, что место хранения для всех элементов массива определяется во время компиляции. Кроме того, память, выделенная для хранения массива, используется в течение всего времени существования массива. Для определения количества байтов памяти, занимаемой двумерным массивом, используйте следующую формулу.

число байтов = число строк × число столбцов × размер типа  
в байтах

Следовательно, двумерный целочисленный массив размерностью  $10 \times 5$  занимает в памяти  $10 \times 5 \times 4$ , т.е. 200 байт (если целочисленный тип имеет размер 4 байт).

#### ВАЖНО!

### 4.3. Многомерные массивы

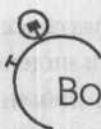
В C++, помимо двумерных, можно определять массивы трех и более измерений. Вот как объявляется многомерный массив.

`тип имя[размер1][размер2]...[размерN];`

Например, с помощью следующего объявления создается трехмерный целочисленный массив размером  $4 \times 10 \times 3$ .

`int multidim[4][10][3];`

Массивы с числом измерений, превышающим три, используются нечасто, хотя бы потому, что для их хранения требуется большой объем памяти. Ведь, как упоминалось выше, память, выделенная для хранения всех элементов массива, используется в течение всего времени существования массива. Например, хранение элементов четырехмерного символьного массива размером  $10 \times 6 \times 9 \times 4$  займет 2 160 байт. А если каждую размерность увеличить в 10 раз, то занимаемая массивом память возрастет до 21 600 000 байт. Как видите, большие многомерные массивы способны “съесть” большой объем памяти, а программа, которая их использует, может очень быстро столкнуться с проблемой нехватки памяти.



### Вопросы для текущего контроля

1. Каждая размерность многомерного массива в C++ заключается в собственную пару квадратных скобок. Верно ли это?
2. Покажите, как объявить двумерный целочисленный массив с именем `list` размерностью  $4 \times 9$ ?
3. Покажите, как получить доступ к элементу 2,3 массива `list` из предыдущего вопроса?\*

## Проект 4.1. Сортировка массива

`Bubble.cpp`

Поскольку одномерный массив обеспечивает организацию данных в виде индексируемого линейного списка, то он представляет собой просто идеальную структуру данных для сортировки. В этом проекте вы познакомитесь с одним из самых простых способов сортировки массивов. Существует много различных алгоритмов сортировки. Широко применяется, например, сортировка перемешиванием и сортировка методом Шелла. Известен также алгоритм быстрой сортировки. Однако самым простым считается алгоритм сортировки *пузырьковым методом*. Несмотря на то что пузырьковая сортировка не отличается высокой эффективностью (его производительность неприемлема для больших массивов), его вполне успешно можно применять для сортировки массивов малого размера.

### Последовательность действий

1. Создайте файл с именем `Bubble.cpp`.
2. Алгоритм сортировки пузырьковым методом получил свое название от способа, используемого для упорядочивания элементов массива. Здесь выполняются повторяющиеся операции сравнения и при необходимости меняются местами смежные элементы. При этом элементы с меньшими значениями постепенно перемещаются к одному концу массива, а элементы с большими значениями — к другому. Этот процесс напоминает поведение пузырьков воздуха
1. Верно: в C++ каждая размерность многомерного массива заключается в собственную пару квадратных скобок.
2. `int list[4][9]`.
3. `list[2][3]`.

в резервуаре с водой. Пузырьковая сортировка выполняется путем нескольких проходов по массиву, во время которых при необходимости осуществляется перестановка элементов, оказавшихся “не на своем месте”. Количество проходов, гарантирующих получение отсортированного массива, равно количеству элементов в массиве, уменьшенному на единицу.

Рассмотрим код, составляющий ядро пузырьковой сортировки. Сортируемый массив носит имя `nums`.

```
// Реализация алгоритма пузырьковой сортировки.
for(a=1; a<size; a++) {
    for(b=size-1; b>=a; b--) {
        if(nums[b-1] > nums[b]) { // Если значения элементов
            // массива расположены не
            // по порядку,
            // то меняем их местами.
            t = nums[b-1];
            nums[b-1] = nums[b];
            nums[b] = t;
        }
    }
}
```

Обратите внимание на то, что сортировка реализована на двух циклах `for`. Во внутреннем цикле организовано сравнение двух смежных элементов массива. Если окажется, что они располагаются “не по порядку”, т.е. не по возрастанию, то их следует поменять местами. При каждом проходе внутреннего цикла самый маленький элемент помещается на “свое законное” место. Внешний цикл обеспечивает повторение этого процесса до тех пор, пока не будет отсортирован весь массив.

### 3. Вот полный текст программы `Bubble.cpp`.

```
/*
Проект 4.1.
Демонстрация метода пузырьковой сортировки
(Bubble sort).

*/
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
```

## 168 Модуль 4. Массивы, строки и указатели

```
int nums[10];
int a, b, t;
int size;

size = 10; // Количество сортируемых элементов.

// Помещаем в массив случайные числа.
for(t=0; t<size; t++) nums[t] = rand();

// Отображаем исходный массив.
cout << "Исходный массив:\n    ";
for(t=0; t<size; t++) cout << nums[t] << ' ';
cout << '\n';

// Реализация алгоритма пузырьковой сортировки.
for(a=1; a<size; a++)
    for(b=size-1; b>=a; b--) {
        if(nums[b-1] > nums[b]) { // Если значения элементов
            // массива расположены не
            // по порядку,
            // то меняем их местами.
            t = nums[b-1];
            nums[b-1] = nums[b];
            nums[b] = t;
        }
    }

// Отображаем отсортированный массив.
cout << "\nОтсортированный массив:\n    ";
for(t=0; t<size; t++) cout << nums[t] << ' ';

return 0;
}
```

Результаты выполнения этой программы таковы.

Исходный массив:

41 18467 6334 26500 19169 15724 11478 29358 26962 24464

Отсортированный массив:

41 6334 11478 15724 18467 19169 24464 26500 26962 29358

4. Хотя алгоритм пузырьковой сортировки пригоден для небольших массивов, для массивов большого размера он становится неэффективным. Более универсальным считается алгоритм Quicksort. Однако этот алгоритм использует средства C++, которые мы еще не изучили. Кроме того, в стандартную библиотеку C++ включена функция `qsort()`, которая реализует одну из версий этого алгоритма. Но, прежде чем использовать ее, вам необходимо изучить больше средств C++.

**ВАЖНО!**

#### 4.4. Строки

Чаще всего одномерные массивы используются для создания символьных строк. В C++ поддерживается два типа строк. Первый (и наиболее популярный) предполагает, что *строка* определяется как символьный массив, который завершается нулевым символом ('\0'). Таким образом, строка с завершающим нулем состоит из символов и конечного нуль-символа. Такие строки широко используются в программировании, поскольку они позволяют программисту выполнять самые разные операции над строками и тем самым обеспечивают высокий уровень эффективности программирования. Поэтому, используя термин *строка*, C++-программист обычно имеет в виду именно строку с завершающим нулем. Однако существует и другой тип представления строк в C++. Он заключается в применении объектов класса `string`, который является частью библиотеки классов C++. Таким образом, класс `string` – не встроенный тип. Он подразумевает объектно-ориентированный подход к обработке строк, но используется не так широко, как строки с завершающим нулем. В этом разделе мы рассматриваем строки с завершающим нулем.

### Основы представления строк

Объявляя символьный массив, предназначенный для хранения строки с завершающим нулем, необходимо учитывать признак ее завершения и задавать длину массива на единицу больше длины самой большой строки из тех, которые предполагается хранить в этом массиве. Например, при объявлении массива `str`, в который предполагается поместить 10-символьную строку, следует использовать следующую инструкцию.

```
char str[11];
```

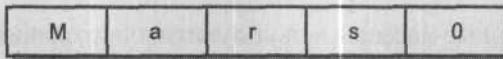
Заданный здесь размер (11) позволяет зарезервировать место для нулевого символа в конце строки.

Как упоминалось выше в этой книге, C++ позволяет определять строковые константы (литералы). Вспомним, что *строковый литерал* — это список символов, заключенный в двойные кавычки. Вот несколько примеров.

"Привет"      "Мне нравится C++"      "Mars"      ""

Строка, приведенная последней (" "), называется *нулевой*. Она состоит только из одного нулевого символа (признака завершения строки). Нулевые строки используются для представления пустых строк.

Вам не нужно вручную добавлять в конец строковых констант нулевые символы. C++-компилятор делает это автоматически. Следовательно, строка "Mars" в памяти размещается так, как показано на этом рисунке:



## Считывание строк с клавиатуры

Проще всего считать строку с клавиатуры, создав массив, который примет эту строку с помощью инструкции `cin`. Считывание строки, введенной пользователем с клавиатуры, отображено в следующей программе.

```
// Использование cin-инструкции для считывания строки
// с клавиатуры.

#include <iostream>
using namespace std;

int main()
{
    char str[80];

    cout << "Введите строку: ";
    cin >> str; // ← Считываем строку с клавиатуры
                 // с помощью инструкции cin.
    cout << "Вот ваша строка: ";
    cout << str;

    return 0;
}
```

Бот возможный результат выполнения этой программы.

Введите строку: Тестирование

Вот ваша строка: Тестирование

Несмотря на то что эта программа формально корректна, она не лишена недостатков. Рассмотрим следующий результат ее выполнения.

Введите строку: Это проверка

Вот ваша строка: Это

Как видите, при выводе строки, введенной с клавиатуры, программа отображает только слово "Это", а не всю строку. Дело в том, что оператор ">>" (в инструкции `cin`) прекращает считывание строки, как только встречает символ пробела, табуляции или новой строки (будем называть эти символы *пробельными*).

Для решения этой проблемы можно использовать еще одну библиотечную функцию `gets()`. Общий формат ее вызова таков.

```
gets(имя_массива);
```

Если в программе необходимо считать строку с клавиатуры, вызовите функцию `gets()`, а в качестве аргумента передайте имя массива, не указывая индекса. После выполнения этой функции заданный массив будет содержать текст, введенный с клавиатуры. Функция `gets()` считывает вводимые пользователем символы до тех пор, пока он не нажмет клавишу <Enter>. Для вызова функции `gets()` в программу необходимо включить заголовок `<cstdio>`.

В следующей версии предыдущей программы демонстрируется использование функции `gets()`, которая позволяет ввести в массив строку символов, содержащую пробелы.

```
// Использование функции gets() для считывания строки
// с клавиатуры.
```

```
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
    char str[80];

    cout << "Введите строку: ";
    gets(str); // ← Считываем строку с клавиатуры
               // с помощью функции gets().
    cout << "Вот ваша строка: ";
    cout << str;
```

## 172 Модуль 4. Массивы, строки и указатели

```
    return 0;  
}
```

На этот раз после запуска новой версии программы на выполнение и ввода с клавиатуры текста “Это простой тест” строка считывается полностью, а затем так же полностью и отображается.

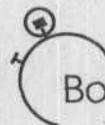
Введите строку: Это простой тест  
Вот ваша строка: Это простой тест

В этой программе следует обратить внимание на следующую инструкцию.

```
cout << str;
```

Здесь (вместо привычного литерала) используется имя строкового массива. Запомните: имя символьного массива, который содержит строку, можно использовать везде, где допустимо применение строкового литерала.

При этом имейте в виду, что ни оператор “>>” (в инструкции `cin`), ни функция `gets()` не выполняют граничной проверки (на отсутствие нарушения границ массива). Поэтому, если пользователь введет строку, длина которой превышает размер массива, возможны неприятности, о которых упоминалось выше. Из сказанного следует, что оба описанных здесь варианта считывания строк с клавиатуры потенциально опасны. Однако после подробного рассмотрения C++-возможностей ввода-вывода мы узнаем способы, позволяющие обойти эту проблему.



### Вопросы для текущего контроля

1. Что представляет собой строка с завершающим нулем?
2. Какой размер должен иметь символьный массив, предназначенный для хранения строки, состоящей из 8 символов?
3. Какую функцию можно использовать для считывания с клавиатуры строки, содержащей пробелы?\*

- 
1. Стока с завершающим нулем представляет собой массив символов, который завершается нулем.
  2. Если символьный массив предназначен для хранения строки, состоящей из 8 символов, то его размер должен быть не меньше 9.
  3. Для считывания с клавиатуры строки, содержащей пробелы, можно использовать функцию `gets()`.

ВАЖНО!

## 4.5. Некоторые библиотечные функции обработки строк

Язык С++ поддерживает множество функций обработки строк. Самыми распространенными из них являются следующие.

```
strcpy()
strcat()
strlen()
strcmp()
```

Для вызова всех этих функций в программу необходимо включить заголовок `<cstring>`. Теперь познакомимся с каждой функцией в отдельности.

### Функция strcpy()

Общий формат вызова функции `strcpy()` таков:

```
strcpy(to, from);
```

Функция `strcpy()` копирует содержимое строки `from` в строку `to`. Помните, что массив, используемый для хранения строки `to`, должен быть достаточно большим, чтобы в него можно было поместить строку из массива `from`. В противном случае массив `to` переполнится, т.е. произойдет выход за его границы, что может привести к разрушению программы.

### Функция strcat()

Обращение к функции `strcat()` имеет следующий формат.

```
strcat(s1, s2);
```

Функция `strcat()` присоединяет строку `s2` к концу строки `s1`; при этом строка `s2` не изменяется. Обе строки должны завершаться нулевым символом. Результат вызова этой функции, т.е. результирующая строка `s1` также будет завершаться нулевым символом. Программист должен позаботиться о том, чтобы строка `s1` была достаточно большой, и в нее поместилось, кроме ее исходного содержимого, содержимое строки `s2`.

### Функция strcmp()

Обращение к функции `strcmp()` имеет следующий формат:

```
strcmp(s1, s2);
```

Функция `strcmp()` сравнивает строку `s2` со строкой `s1` и возвращает значение 0, если они равны. Если строка `s1` лексикографически (т.е. в соответствии с алфавит-

ным порядком) больше строки *s2*, возвращается положительное число. Если строка *s1* лексикографически меньше строки *s2*, возвращается отрицательное число.

При использовании функции `strcmp()` важно помнить, что она возвращает число 0 (т.е. значение `false`), если сравниваемые строки равны. Следовательно, если вам необходимо выполнить определенные действия при условии совпадения строк, вы должны использовать оператор НЕ (`!`). Например, условие, управляющее следующей `if`-инструкцией, даст истинный результат, если строка *str* содержит значение "C++".

```
if(!strcmp(str, "C++")) cout << "Строка str содержит C++";
```

### Функция `strlen()`

Общий формат вызова функции `strlen()` таков:

```
strlen(s);
```

Здесь *s* — строка. Функция `strlen()` возвращает длину строки, указанной аргументом *s*.

### Пример использования строковых функций

В следующей программе показано использование всех четырех строковых функций.

```
// Демонстрация использования строковых функций.
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;

int main()
{
    char s1[80], s2[80];

    strcpy(s1, "C++");
    strcpy(s2, " - это мощный язык.");

    cout << "Длины строк: " << strlen(s1);
    cout << ' ' << strlen(s2) << '\n';

    if(!strcmp(s1, s2))
        cout << "Эти строки равны.\n";
```

```

else cout << "Эти строки не равны.\n";

strcat(s1, s2);
cout << s1 << '\n';

strcpy(s2, s1);
cout << s1 << " и " << s2 << "\n";

if(!strcmp(s1, s2))
    cout << "Строки s1 и s2 теперь одинаковы.\n";

return 0;
}

```

Вот как выглядит результат выполнения этой программы.

```

Длины строк: 3 19
Эти строки не равны.
С++ - это мощный язык.
С++ - это мощный язык. и С++ - это мощный язык.
Строки s1 и s2 теперь одинаковы.

```

## Использование признака завершения строки

Факт завершения нулевыми символами всех C++-строк можно использовать для упрощения различных операций над ними. Следующий пример позволяет убедиться в том, насколько простой код требуется для замены всех символов строки их прописными эквивалентами.

```

// Преобразование символов строки
// в их прописные эквиваленты.
#include <iostream>
#include <cstring>
#include <cctype>
using namespace std;

int main()
{
    char str[80];
    int i;

    strcpy(str, "abcdefg");

```

```
// Этот цикл завершится, когда элемент str[i] будет
// содержать признак завершения строки.
//           ↓
for(i=0; str[i]; i++) str[i] = toupper(str[i]);

cout << str;

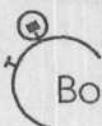
return 0;
}
```

Эта программа генерирует такой результат.

ABCDEG

Здесь используется библиотечная функция `toupper()`, которая возвращает прописной эквивалент своего символьного аргумента. Для вызова функции `toupper()` необходимо включить в программу заголовок `<cctype>`.

Обратите внимание на то, что в качестве условия завершения цикла используется массив `str`, индексируемый управляющей переменной `i` (`str[i]`). Такой способ управления циклом вполне приемлем, поскольку за истинное значение в C++ принимается любое ненулевое значение. Вспомните, что все печатные символы представляются значениями, не равными нулю, и только символ, завершающий строку, равен нулю. Следовательно, этот цикл работает до тех пор, пока индекс не укажет на нулевой признак конца строки, т.е. пока значение `str[i]` не станет нулевым. Поскольку нулевой символ отмечает конец строки, цикл останавливается в точности там, где нужно. В дальнейшем вы увидите множество примеров, в которых нулевой признак конца строки используется подобным образом.



### Вопросы для текущего контроля

1. Какое действие выполняет функция `strcat()`?
2. Что возвращает функция `strcmp()` при сравнении двух эквивалентных строк?
3. Покажите, как получить длину строки с именем `mystr`?\*

- 
1. Функция `strcat()` конкатенирует (т.е. соединяет) две строки.
  2. Функция `strcmp()`, сравнивая две эквивалентные строки, возвращает 0.
  3. `strlen(mystr)`.

## Спросим у опытного программиста

**Вопрос.** Поддерживает ли C++ другие, помимо функции `toupper()`, функции обработки символов?

**Ответ.** Да. Помимо функции `toupper()`, стандартная библиотека C++ содержит много других функций обработки символов. Например, функцию `toupper()` дополняет функция `tolower()`, которая возвращает строчный эквивалент своего символьного аргумента. Например, регистр буквы (т.е. прописная она или строчная) можно определить с помощью функции `isupper()`, которая возвращает значение ИСТИНА, если исследуемая буква является прописной, и функции `islower()`, которая возвращает значение ИСТИНА, если исследуемая буква является строчной. Часто используются такие функции, как `isalpha()`, `isdigit()`, `isspace()` и `ispunct()`, которые принимают символьный аргумент и определяют, принадлежит ли он к соответствующей категории. Например, функция `isalpha()` возвращает значение ИСТИНА, если ее аргументом является буква алфавита.

ВАЖНО!

## 4.6. ИНИЦИАЛИЗАЦИЯ МАССИВОВ

В C++ предусмотрена возможность инициализации массивов. Формат инициализации массивов подобен формату инициализации других переменных.

тип имя\_массива[размер] = {список\_значений};

Здесь элемент `список_значений` представляет собой список значений инициализации элементов массива, разделенных запятыми. Тип каждого значения инициализации должен быть совместим с базовым типом массива (элементом `тип`). Первое значение инициализации будет сохранено в первой позиции массива, второе значение — во второй и т.д. Обратите внимание на то, что точка с запятой ставится после закрывающей фигурной скобки `}`.

Например, в следующем примере 10-элементный целочисленный массив инициализируется числами от 1 до 10.

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

После выполнения этой инструкции элемент `i[0]` получит значение 1, а элемент `i[9]` — значение 10.

Для символьных массивов, предназначенных для хранения строк, предусмотрен сокращенный вариант инициализации, который имеет такую форму.

```
char имя_массива[размер] = "строка";
```

Например, следующий фрагмент кода инициализирует массив str фразой "привет".

```
char str[7] = "привет";
```

Это равнозначно поэлементной инициализации.

```
char str[7] = { 'п', 'р', 'и', 'в', 'е', 'т', '\0' };
```

Поскольку в C++ строки должны завершаться нулевым символом, убедитесь, что при объявлении массива его размер указан с учетом признака конца. Именно поэтому в предыдущем примере массив str объявлен как 7-элементный, несмотря на то, что в слове "привет" только шесть букв. При использовании строкового литерала компилятор добавляет нулевой признак конца строки автоматически.

Многомерные массивы инициализируются по аналогии с одномерными. Например, в следующем фрагменте программы массив sqrs инициализируется числами от 1 до 10 и квадратами этих чисел.

```
int sqrs[10][2] = {
    1, 1,
    2, 4,
    3, 9,
    4, 16,
    5, 25,
    6, 36,
    7, 49,
    8, 64,
    9, 81,
    10, 100
};
```

Теперь рассмотрим, как элементы массива sqrs располагаются в памяти (рис. 4.1).

При инициализации многомерного массива список инициализаторов каждой размерности (подгруппу инициализаторов) можно заключить в фигурные скобки. Вот, например, как выглядит еще один вариант записи предыдущего объявления.

```
int sqrs[10][2] = {
    {1, 1},
    {2, 4},
    {3, 9},
    {4, 16},
    {5, 25},
```

```
{6, 36},  
{7, 49},  
{8, 64},  
{9, 81},  
{10, 100}  
};
```

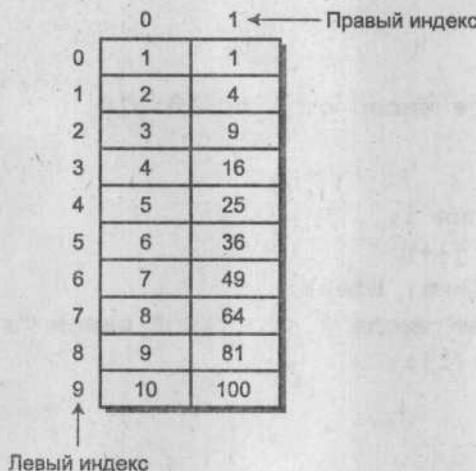


Рис. 4.1. Схематическое представление инициализированного массива sqrs

При использовании подгрупп инициализаторов недостающие члены подгрупп будут инициализированы нулевыми значениями автоматически.

В следующей программе массив sqrs используется для поиска квадрата числа, введенного пользователем. Программа сначала выполняет поиск заданного числа в массиве, а затем выводит соответствующее ему значение квадрата.

```
#include <iostream>  
using namespace std;  
  
int sqrs[10][2] = {  
    {1, 1},  
    {2, 4},  
    {3, 9},  
    {4, 16},  
    {5, 25},  
    {6, 36},  
    {7, 49},  
    {8, 64},  
    {9, 81},  
    {10, 100}
```

```
{8, 64},  
{9, 81},  
{10, 100}  
};  
  
int main()  
{  
    int i, j;  
  
    cout << "Введите число от 1 до 10: ";  
    cin >> i;  
  
    // Поиск значения i.  
    for(j=0; j<10; j++)  
        if(sqr[sqr[j][0]==i]) break;  
    cout << "Квадрат числа " << i << " равен ";  
    cout << sqr[sqr[j][1]];  
  
    return 0;  
}
```

Вот один из возможных результатов выполнения этой программы.

Введите число от 1 до 10: 4

Квадрат числа 4 равен 16

## Инициализация “безразмерных” массивов

Объявляя инициализированный массив, можно позволить C++ автоматически определять его длину. Для этого не нужно задавать размер массива. Компилятор в этом случае определит его путем подсчета количества инициализаторов, после чего создаст массив, размер которого будет достаточным для хранения всех значений инициализаторов. Например, при выполнении этой инструкции

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

будет создан 4-элементный массив `nums`, содержащий значения 1, 2, 3 и 4. Поскольку здесь размер массива `nums` не задан явным образом, его можно назвать “безразмерным”.

Массивы “безразмерного” формата иногда бывают весьма полезными. Например, предположим, что мы используем следующий вариант инициализации массивов для построения таблицы Internet-адресов.

```
char e1[16] = "www.osborn.com";
char e2[16] = "www.weather.com";
char e3[15] = "www.amazon.com";
```

Нетрудно предположить, что вручную неудобно подсчитывать символы в каждом сообщении, чтобы определить корректный размер массива (при этом всегда можно допустить ошибку в подсчетах). Но поскольку в C++ предусмотрена возможность автоматического определения длины массивов путем использования их “безразмерного” формата, то предыдущий вариант инициализации массивов для построения таблицы Internet-адресов можно переписать так.

```
char e1[] = "www.osborn.com";
char e2[] = "www.weather.com";
char e3[] = "www.amazon.com";
```

Помимо удобства в первоначальном определении массивов, метод “безразмерной” инициализации позволяет изменить любое сообщение без пересчета его длины. Тем самым устраняется возможность внесения ошибок, вызванных случайным просчетом.

“Безразмерная” инициализация не ограничивается одномерными массивами. При инициализации многомерных массивов вам необходимо указать все данные, за исключением крайней слева размерности, чтобы C++-компилятор мог должным образом индексировать массив. Используя “безразмерную” инициализацию массивов, можно создавать таблицы различной длины, позволяя компилятору автоматически выделять область памяти, достаточную для их хранения.

В следующем примере массив `sqrs` объявляется как “безразмерный”.

```
int sqrs[][2] = {
    1, 1,
    2, 4,
    3, 9,
    4, 16,
    5, 25,
    6, 36,
    7, 49,
    8, 64,
    9, 81,
    10, 100
};
```

Преимущество такой формы объявления перед “габаритной” (с точным указанием всех размерностей) состоит в том, что программист может удлинять или укорачивать таблицу значений инициализации, не изменяя размерности массива.

**ВАЖНО!**

## 4.7. Массивы строк

Существует специальная форма двумерного символьного массива, которая представляет собой массив строк. В использовании массивов строк нет ничего необычного. Например, в программировании баз данных для выяснения корректности вводимых пользователем команд входные данные сравниваются с содержимым массива строк, в котором записаны допустимые в данном приложении команды. Для создания массива строк используется двумерный символьный массив, в котором размер левого индекса определяет количество строк, а размер правого — максимальную длину каждой строки. Например, при выполнении следующей инструкции объявляется массив, предназначенный для хранения 30 строк длиной 80 символов.

```
char str_array[30][80];
```

Получить доступ к отдельной строке довольно просто: достаточно указать только левый индекс. Например, следующая инструкция вызывает функцию `gets()` для записи третьей строки массива `str_array`.

```
gets(str_array[2]);
```

Для получения доступа к конкретному символу третьей строки используйте инструкцию, подобную следующей.

```
cout << str_array[2][3];
```

При выполнении этой инструкции на экран будет выведен четвертый символ третьей строки.

В следующей программе демонстрируется использование массива строк путем реализации очень простого компьютеризированного телефонного каталога. Двумерный массив `numbers` содержит пары значений: имя и телефонный номер. Чтобы получить телефонный номер, нужно ввести имя. После этого искомая информация (телефонный номер) отобразится на экране.

```
// Простой компьютеризированный телефонный каталог.
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
    int i;
    char str[80];
    char numbers[10][80] = {
```

```

    "Том", "555-3322",
    "Мария", "555-8976",
    "Джон", "555-1037",
    "Раиса", "555-1400",
    "Шура", "555-8873"
};

cout << "Введите имя: ";
cin >> str;

for(i=0; i < 10; i += 2)
    if(!strcmp(str, numbers[i])) {
        cout << "Телефонный номер: " << numbers[i+1] << "\n";
        break;
    }

if(i == 10) cout << "Отсутствует в каталоге.\n";

return 0;
}

```

Вот пример выполнения программы.

```

Введите имя: Джон
Телефонный номер: 555-1037

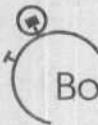
```

Обратите внимание на то, как при каждом проходе цикла `for` управляющая переменная `i` инкрементируется на 2. Такой шаг инкремента объясняется тем, что имена и телефонные номера чередуются в массиве.

#### ВАЖНО!

## 4.8. Указатели

Указатели — один из самых важных и сложных аспектов C++. Вместе с тем они зачастую являются источниками потенциальных проблем. В значительной степени мощь многих средств C++ определяется использованием указателей. Например, благодаря им обеспечивается поддержка связных списков и механизма динамического выделения памяти, и именно они позволяют функциям изменять содержимое своих аргументов. Однако об этом мы поговорим в последующих модулях, а пока (т.е. в этом модуле) мы рассмотрим основы применения указателей и покажем, как с ними обращаться.



## Вопросы для текущего контроля

- Покажите, как инициализировать четырехэлементный массив `list int` значениями 1, 2, 3 и 4.
  - Как можно переписать эту инициализацию?
- ```
char str[6] = {'П', 'р', 'и', 'в', 'е', 'т', '\0'};
```
- Перепишите следующий вариант инициализации в “безразмерном” формате.
- ```
int nums[4] = { 44, 55, 66, 77 };
```

При рассмотрении темы указателей нам придется использовать такие понятия, как размер базовых C++-типов данных. В этой главе мы предположим, что символы занимают в памяти один байт, целочисленные значения — четыре, значения с плавающей точкой типа `float` — четыре, а значения с плавающей точкой типа `double` — восемь (эти размеры характерны для типичной 32-разрядной среды).

## Что представляют собой указатели

Указатель — это объект, который содержит некоторый адрес памяти. Чаще всего этот адрес обозначает местоположение в памяти другого объекта, такого, как переменная. Например, если `x` содержит адрес переменной `y`, то о переменной `x` говорят, что она “указывает” на `y`.

*Переменные-указатели* (или *переменные типа указатель*) должны быть соответственно объявлены. Формат объявления переменной-указателя таков:

тип `*имя_переменной;`

Здесь элемент `тип` означает базовый тип указателя, причем он должен быть допустимым C++-типов. *Базовый тип* определяет, на какой тип данных будет ссылаться объявляемый указатель. Элемент `имя_переменной` представляет собой имя переменной-указателя. Рассмотрим пример. Чтобы объявить переменную `ip` указателем на `int`-значение, используйте следующую инструкцию.

```
int *ip;
```

- `int list[] = { 1, 2, 3, 4 };`
- `char str[] = "Привет";`
- `int nums[] = { 44, 55, 66, 77 };`

Для объявления указателя на float-значение используйте такую инструкцию.

```
float *fp;
```

В общем случае использование символа “звездочка” (\*) перед именем переменной в инструкции объявления превращает эту переменную в указатель.

**ВАЖНО!**

## 4.9. Операторы, используемые с указателями

С указателями используются два оператора: “\*” и “&”. Оператор “&” – унарный. Он возвращает адрес памяти, по которому расположен его operand. (Вспомните: унарному оператору требуется только один operand.) Например, при выполнении следующего фрагмента кода

```
ptr = &total;
```

в переменную `ptr` помещается адрес переменной `total`. Этот адрес соответствует области во внутренней памяти компьютера, которая принадлежит переменной `total`. Выполнение этой инструкции *никак* не повлияло на значение переменной `total`. Назначение оператора “&” можно “перевести” на русский язык как “адрес переменной”, перед которой он стоит. Следовательно, приведенную выше инструкцию присваивания можно выразить так: “переменная `ptr` получает адрес переменной `total`”. Чтобы лучше понять суть этого присваивания, предположим, что переменная `total` расположена в области памяти с адресом 100. Следовательно, после выполнения этой инструкции переменная `ptr` получит значение 100.

Второй оператор работы с указателями (\*) служит дополнением к первому (&). Это также унарный оператор, но он обращается к значению переменной, расположенной по адресу, заданному его operandом. Другими словами, он ссылается на значение переменной, адресуемой заданным указателем. Если (продолжая работу с предыдущей инструкцией присваивания) переменная `ptr` содержит адрес переменной `total`, то при выполнении инструкции

```
val = *ptr;
```

переменной `val` будет присвоено значение переменной `total`, на которую указывает переменная `ptr`. Например, если переменная `total` содержит значение 3200, после выполнения последней инструкции переменная `val` будет содержать значение 3200, поскольку это как раз то значение, которое хранится по адресу 100. Назначение оператора “\*” можно выразить словосочетанием “по адресу”. В данном случае предыдущую инструкцию можно прочитать так: “переменная `val` получает значение (расположенное) по адресу `ptr`”.

Последовательность только что описанных операций выполняется в следующей программе.

```
#include <iostream>
using namespace std;

int main()
{
    int total;
    int *ptr;
    int val;

    total = 3200; // Переменной total присваиваем 3200.

    ptr = &total; // Получаем адрес переменной total.

    val = *ptr;    // Получаем значение, хранимое
                   // по этому адресу.
    cout << "Сумма равна: " << val << '\n';

    return 0;
}
```

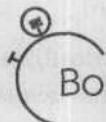
К сожалению, знак умножения (\*) и оператор со значением “по адресу” обозначаются одинаковыми символами “звездочка”, что иногда сбивает с толку новичков в языке C++. Эти операции никак не связаны одна с другой. Имейте в виду, что операторы “\*” и “&” имеют более высокий приоритет, чем любой из арифметических операторов, за исключением унарного минуса, приоритет которого такой же, как у операторов, применяемых для работы с указателями.

Операции, выполняемые с помощью указателей, часто называют *операциями непрямого доступа*, поскольку они позволяют получить доступ к одной переменной посредством некоторой другой переменной.

## О важности базового типа указателя

На примере предыдущей программы была показана возможность присвоения переменной `val` значения переменной `total` посредством операции непрямого доступа, т.е. с использованием указателя. Возможно, при этом у вас возник вопрос: “Как C++-компилятор узнает, сколько необходимо скопировать байтов в переменную `val` из области памяти, адресуемой указателем `ptr`?”. Сформулируем тот же вопрос в более общем виде: как C++-компилятор передает надлежащее

количество байтов при выполнении операции присваивания с использованием указателя? Ответ звучит так. Тип данных, адресуемый указателем, определяется базовым типом указателя. В данном случае, поскольку `ptr` представляет собой указатель на целочисленный тип, C++-компилятор скопирует в переменную `val` из области памяти, адресуемой указателем `ptr`, 4 байт информации (что справедливо для 32-разрядной среды), но если бы мы имели дело с `double`-указателем, то в аналогичной ситуации скопировалось бы 8 байт.



## Вопросы для текущего контроля

1. Что такое указатель?
2. Как объявить указатель с именем `valPtr`, который имеет базовый тип `long int`?
3. Каково назначение операторов “`*`” и “`&`” при использовании с указателями?\*

Переменные-указатели должны всегда указывать на соответствующий тип данных. Например, при объявлении указателя типа `int` компилятор “предполагает”, что все значения, на которые ссылается этот указатель, имеют тип `int`. C++-компилятор попросту не позволит выполнить операцию присваивания с участием указателей (с обеих сторон от оператора присваивания), если типы этих указателей несовместимы (по сути, не одинаковы). Например, следующий фрагмент кода некорректен.

```
int *p;
double f;
// ...
p = &f; // ОШИБКА!
```

Некорректность этого фрагмента состоит в недопустимости присваивания `double`-указателя `int`-указателю. Выражение `&f` генерирует указатель на `double`-значение, а `p` – указатель на целочисленный тип `int`. Эти два типа несовместимы, поэтому компилятор отметит эту инструкцию как ошибочную и не скомпилирует программу.

1. Указатель – это объект, который содержит адрес памяти некоторого другого объекта.
2. `long int *valPtr;`
3. Оператор “`*`” позволяет получить значение, хранимое по адресу, заданному его операндом. Оператор “`&`” возвращает адрес объекта, по которому расположен его операнд.

## 188 Модуль 4. Массивы, строки и указатели

Несмотря на то что, как было заявлено выше, при присваивании два указателя должны быть совместимы по типу, это серьезное ограничение можно преодолеть (правда, на свой страх и риск) с помощью операции приведения типов. Например, следующий фрагмент кода теперь формально корректен.

```
int *p;
double f;
// ...
p = (int *) &f; // Теперь формально все OK!
```

Операция приведения к типу `(int *)` вызовет преобразование `double`-указателю. Все же использование операции приведения в таких целях несколько сомнительно, поскольку именно базовый тип указателя определяет, как компилятор будет обращаться с данными, на которые он ссылается. В данном случае, несмотря на то, что `p` (после выполнения последней инструкции) в действительности указывает на значение с плавающей точкой, компилятор по-прежнему “считает”, что он указывает на целочисленное значение (поскольку `p` по определению – `int`-указатель).

Чтобы лучше понять, почему использование операции приведения типов при присваивании одного указателя другому не всегда приемлемо, рассмотрим следующую программу.

```
// Эта программа не будет выполняться правильно.
#include <iostream>
using namespace std;

int main()
{
    double x, y;
    int *p;

    x = 123.23;
    p = (int *) &x; // Используем операцию приведения типов
                    // для присваивания double-указателя
                    // int-указателю.
    // Следующие две инструкции не дают желаемых результатов.
    y = *p; // Что происходит при выполнении этой инструкции?
    cout << y; // Что выведет эта инструкция?

    return 0;
}
```

Вот какой результат генерирует эта программа. (У вас может получиться другое значение.)

```
1.37439e+009
```

Как видите, в этой программе переменной *p* (точнее, указателю на целочисленное значение) присваивается адрес переменной *x* (которая имеет тип *double*). Следовательно, когда переменной *y* присваивается значение, адресуемое указателем *p*, переменная *y* получает только 4 байт данных (а не все 8, требуемых для *double*-значения), поскольку *p* — указатель на целочисленный тип *int*. Таким образом, при выполнении *cout*-инструкции на экран будет выведено не число 123.23, а, как говорят программисты, “мусор”.

## Присваивание значений с помощью указателей

При присваивании значения области памяти, адресуемой указателем, его (указатель) можно использовать с левой стороны от оператора присваивания. Например, при выполнении следующей инструкции (если *p* — указатель на целочисленный тип)

```
*p = 101;
```

число 101 присваивается области памяти, адресуемой указателем *p*. Таким образом, эту инструкцию можно прочитать так: “по адресу *p* помещаем значение 101”. Чтобы инкрементировать или декрементировать значение, расположенное в области памяти, адресуемой указателем, можно использовать инструкцию, подобную следующей.

```
(*p)++;
```

Круглые скобки здесь обязательны, поскольку оператор “\*” имеет более низкий приоритет, чем оператор “++”.

Присваивание значений с использованием указателей демонстрируется в следующей программе.

```
#include <iostream>
using namespace std;

int main()
{
    int *p, num;
    p = &num;
    *p = 100; // ← Присваиваем переменной num число 100
```

```
//      через указатель p.
cout << num << ' ';
(*p)++; // ← Инкрементируем значение переменной num
//      через указатель p.
cout << num << ' ';
(*p)--; // ← Декрементируем значение переменной num
//      через указатель p.
cout << num << '\n';

return 0;
}
```

Вот такие результаты генерирует эта программа.

100 101 100

**ВАЖНО!**

## **4.10. Использование указателей в выражениях**

Указатели можно использовать в большинстве допустимых в C++ выражений. Но при этом следует применять некоторые специальные правила и не забывать, что некоторые части таких выражений необходимо заключать в круглые скобки, чтобы гарантированно получить желаемый результат.

### **Арифметические операции над указателями**

С указателями можно использовать только четыре арифметических оператора: `++`, `--`, `+` и `-`. Чтобы лучше понять, что происходит при выполнении арифметических действий с указателями, начнем с примера. Пусть `p1` — указатель на `int`-переменную с текущим значением 2000 (т.е. `p1` содержит адрес 2000). После выполнения (в 32-разрядной среде) выражения

`p1++;`

содержимое переменной-указателя `p1` станет равным 2 004, а не 2 001! Дело в том, что при каждом инкрементировании указатель `p1` будет указывать на *следующее int*-значение. Для операции декрементирования справедливо обратное утверждение, т.е. при каждом декрементировании значение `p1` будет уменьшаться на 4. Например, после выполнения инструкции

`p1--;`

указатель `p1` будет иметь значение 1 996, если до этого оно было равно 2 000.

Итак, каждый раз, когда указатель инкрементируется, он будет указывать на область памяти, содержащую следующий элемент базового типа этого указателя. А при каждом декрементировании он будет указывать на область памяти, содержащую предыдущий элемент базового типа этого указателя. Для указателей на символьные значения результат операций инкрементирования и декрементирования будет таким же, как при "нормальной" арифметике, поскольку символы занимают только один байт. Но при использовании любого другого типа указателя при инкрементировании или декрементировании значение переменной-указателя будет увеличиваться или уменьшаться на величину, равную размеру его базового типа.

Арифметические операции над указателями не ограничиваются использованием операторов инкремента и декремента. Со значениями указателей можно выполнять операции сложения и вычитания, используя в качестве второго определяемого целочисленные значения. Выражение

```
p1 = p1 + 9;
```

заставляет `p1` ссылаться на девятый элемент базового типа указателя `p1` относительно элемента, на который `p1` ссылался до выполнения этой инструкции.

Несмотря на то что складывать указатели нельзя, один указатель можно вычесть из другого (если они оба имеют один и тот же базовый тип). Разность покажет количество элементов базового типа, которые разделяют эти два указателя.

Помимо сложения указателя с целочисленным значением и вычитания его из указателя, а также вычисления разности двух указателей, над указателями никакие другие арифметические операции не выполняются. Например, с указателями нельзя складывать `float`- или `double`-значения.

Чтобы понять, как формируется результат выполнения арифметических операций над указателями, выполним следующую короткую программу. Она выводит реальные физические адреса, которые содержат указатель на `int`-значение (`i`) и указатель на `double`-значение (`f`). Обратите внимание на каждое изменение адреса (зависящее от базового типа указателя), которое происходит при повторении цикла. (Для большинства 32-разрядных компиляторов значение `i` будет увеличиваться на 4, а значение `f` — на 8.) Отметьте также, что при использовании указателя в `cout`-инструкции его адрес автоматически отображается в формате адресации, применяемом для текущего процессора и среды выполнения.

```
#include <iostream>
using namespace std;

int main()
{
    int *i, j[10];
```

## 192 Модуль 4. Массивы, строки и указатели

```
double *f, g[10];
int x;

i = j;
f = g;

for(x=0; x<10; x++)
    cout << i+x << ' ' << f+x // Отображаем адреса,
                                // полученные в результате
                                // сложения значения x
                                // с каждым указателем.

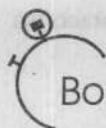
return 0;
}
```

Вот один из возможных результатов выполнения этой программы (у вас могут быть другие значения).

```
0012F724 0012F74C
0012F728 0012F754
0012F72C 0012F75C
0012F730 0012F764
0012F734 0012F76C
0012F738 0012F774
0012F73C 0012F77C
0012F740 0012F784
0012F744 0012F78C
0012F748 0012F794
```

### Сравнение указателей

Указатели можно сравнивать, используя операторы отношения ==, < и >. Однако для того, чтобы результат сравнения указателей поддавался интерпретации, сравниваемые указатели должны быть каким-то образом связаны. Например, если указатели ссылаются на две отдельные и никак не связанные переменные, то любое их сравнение в общем случае не имеет смысла. Но если они указывают на переменные, между которыми существует некоторая связь (как, например, между элементами одного и того же массива), то результат сравнения этих указателей может иметь определенный смысл (пример такого сравнения вы увидите в проекте 4.2). При этом можно говорить о еще одном типе сравнения: любой указатель можно сравнить с нулевым указателем, который, по сути, равен нулю.



## Вопросы для текущего контроля

- С каким типом связаны все арифметические действия над указателями?
- На какую величину возрастет значение `double`-указателя при его инкрементировании, если предположить, что тип `double` занимает 8 байт?
- В каком случае может иметь смысл сравнение двух указателей?\*

**ВАЖНО!**

## 4.11 Указатели и массивы

В C++ указатели и массивы тесно связаны между собой, причем настолько, что зачастую понятия “указатель” и “массив” взаимозаменяемы. В этом разделе мы попробуем проследить эту связь. Для начала рассмотрим следующий фрагмент программы.

```
char str[80];
char *p1;
```

```
p1 = str;
```

Здесь `str` представляет собой имя массива, содержащего 80 символов, а `p1` — указатель на тип `char`. Особый интерес представляет третья строка, при выполнении которой переменной `p1` присваивается адрес первого элемента массива `str`. (Другими словами, после этого присваивания `p1` будет указывать на элемент `str[0]`.) Дело в том, что в C++ использование имени массива без индекса генерирует указатель на первый элемент этого массива. Таким образом, при выполнении присваивания

```
p1 = str
```

адрес `str[0]` присваивается указателю `p1`. Это и есть ключевой момент, который необходимо четко понимать: неиндексированное имя массива, использованное в выражении, означает указатель на начало этого массива.

Поскольку после рассмотренного выше присваивания `p1` будет указывать на начало массива `str`, указатель `p1` можно использовать для доступа к элементам

- Все арифметические действия над указателями связаны с базовым типом указателя.
- Значение указателя увеличится на 8.
- Сравнение двух указателей может иметь смысл в случае, когда они ссылаются на один и тот же объект (например, массив).

## 194 Модуль 4. Массивы, строки и указатели

этого массива. Например, чтобы получить доступ к пятому элементу массива `str`, используйте одно из следующих выражений:

`str[4]`

или

`* (p1+4)`

В обоих случаях будет выполнено обращение к пятому элементу. Помните, что индексирование массива начинается с нуля, поэтому при индексе, равном четырем, обеспечивается доступ к пятому элементу. Точно такой же эффект производит суммирование значения исходного указателя (`p1`) с числом 4, поскольку `p1` указывает на первый элемент массива `str`.

Необходимость использования круглых скобок, в которые заключено выражение `p1+4`, обусловлена тем, что оператор `"*"` имеет более высокий приоритет, чем оператор `"+"`. Без этих круглых скобок выражение бы свелось к получению значения, адресуемого указателем `p1`, т.е. значения первого элемента массива, которое затем было бы увеличено на 4.

В действительности в C++ предусмотрено два способа доступа к элементам массивов: с помощью *индексирования массивов* и *арифметики указателей*. Дело в том, что арифметические операции над указателями иногда выполняются быстрее, чем индексирование массивов, особенно при доступе к элементам, расположение которых отличается строгой упорядоченностью. Поскольку быстродействие часто является определяющим фактором при выборе тех или иных решений в программировании, то использование указателей для доступа к элементам массива — характерная особенность многих C++-программ. Кроме того, иногда указатели позволяют написать более компактный код по сравнению с использованием индексирования массивов.

Чтобы лучше понять различие между использованием индексирования массивов и арифметических операций над указателями, рассмотрим две версии одной и той же программы, которая реверсирует регистр букв в строке (т.е. строчные буквы преобразует в прописные и наоборот). В первой версии используется индексирование массивов, а во второй — арифметика указателей. Вот текст первой версии программы.

```
// Реверсирование регистра букв с помощью
// индексирования массивов.
```

```
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
```

```

int i;
char str[80] = "This Is A Test";

cout << "Исходная строка: " << str << "\n";

for(i = 0; str[i]; i++) {
    if(isupper(str[i]))
        str[i] = tolower(str[i]);
    else if(islower(str[i]))
        str[i] = toupper(str[i]);
}

cout << "Строка с инвертированным регистром букв: "
    << str;

return 0;
}

```

Результаты выполнения этой программы таковы.

Исходная строка: This Is A Test

Строка с инвертированным регистром букв: tHIS iS a tEST

Обратите внимание на то, что в программе для определения регистра букв используются библиотечные функции `isupper()` и `islower()`. Функция `isupper()` возвращает значение ИСТИНА, если ее аргумент представляет собой прописную букву, а функция `islower()` возвращает значение ИСТИНА, если ее аргумент представляет собой строчную букву. На каждом проходе цикла `for` с помощью индексирования массива `str` осуществляется доступ к очередному его элементу, который подвергается проверке регистра с последующим изменением его на "противоположный". Цикл останавливается при доступе к завершающему нулю массива `str`, поскольку нуль означает ложное значение.

А вот версия той же программы, переписанной с использованием арифметики указателей.

```

// Реверсирование регистра букв с помощью
// арифметики указателей.

#include <iostream>
#include <cctype>
using namespace std;

int main()

```

## 196 Модуль 4. Массивы, строки и указатели

```
{  
    char *p;  
    char str[80] = "This Is A Test";  
  
    cout << "Исходная строка: " << str << "\n";  
  
    p = str; // Присваиваем указателю p адрес начала массива.  
  
    while(*p) {  
        if(isupper(*p))  
            *p = tolower(*p);  
        else if(islower(*p)) // Доступ к массиву str через  
                            // указатель.  
            *p = toupper(*p);  
        p++;  
    }  
  
    cout << "Строка с инвертированным регистром букв: "  
        << str;  
  
    return 0;  
}
```

В этой версии указатель p устанавливается на начало массива str. Затем буква, на которую указывает p, проверяется и изменяется соответствующим образом в цикле while. Цикл остановится, когда p будет указывать на завершающий нуль массива str.

У этих программ может быть различное быстродействие, что обусловлено особенностями генерирования кода C++-компиляторами. Как правило, при использовании индексирования массивов генерируется более длинный код (с большим количеством машинных команд), чем при выполнении арифметических действий над указателями. Поэтому неудивительно, что в профессионально написанном C++-коде чаще встречаются версии, ориентированные на обработку указателей. Но если вы — начинающий программист, смело используйте индексирование массивов, пока не научитесь свободно обращаться с указателями.

### Индексирование указателя

Как было показано выше, доступ к массиву можно получить путем выполнения арифметических действий над указателями. Интересно то, что в C++ указа-

тель, который ссылается на массив, можно индексировать так, как если бы это было имя массива (это говорит о тесной связи между указателями и массивами). Рассмотрим пример, который представляет собой третью версию программы изменения регистра букв.

```
// Индексирование указателя подобно массиву.
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
    char *p;
    int i;
    char str[80] = "This Is A Test";

    cout << "Исходная строка: " << str << "\n";

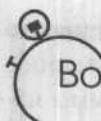
    p = str; // Присваиваем указателю p адрес начала массива.

    // now, index p
    for(i = 0; p[i]; i++) {
        if(isupper(p[i]))
            p[i] = tolower(p[i]); // ← Используем p как если бы
        else if(islower(p[i])) // это было имя массива.
            p[i] = toupper(p[i]);
    }

    cout << "Строка с инвертированным регистром букв: "
        << str;

    return 0;
}
```

При выполнении эта программа создает `char`-указатель с именем `p`, а затем присваивает ему адрес первого элемента массива `str`. В цикле `for` указатель `p` индексируется с использованием обычного синтаксиса индексирования массивов, что абсолютно допустимо, поскольку в C++ выражение `p[i]` по своему действию идентично выражению `*(p+1)`. Этот пример прекрасно иллюстрирует тесную связь между указателями и массивами.



### Вопросы для текущего контроля

1. Можно ли к массиву получить доступ через указатель?
2. Можно ли индексировать указатель подобно массиву?
3. Какое назначение имеет само имя массива, использованное без индекса?\*

## Строковые константы

Возможно, вас удивит способ обработки C++-компиляторами строковых констант, подобных следующей.

```
cout << strlen("C++-компилятор");
```

Если C++-компилятор обнаруживает строковый литерал, он сохраняет его в *таблице строк* программы и генерирует указатель на нужную строку. Поэтому следующая программа совершенно корректна и при выполнении выводит на экран фразу Работа с указателями – сплошное удовольствие!.

```
#include <iostream>
using namespace std;

int main()
{
    char *ptr;

    // Указателю ptr присваивается адрес строковой
    // константы.
    ptr = "Работа с указателями – сплошное удовольствие!\n";

    cout << ptr;

    return 0;
}
```

1. Да, к массиву можно получить доступ через указатель.
2. Да, указатель можно индексировать подобно массиву.
3. Само имя массива, использованное без индекса, генерирует указатель на первый элемент этого массива.

## Спросим у опытного программиста

**Вопрос.** Можно ли утверждать, что указатели и массивы взаимозаменяемы?

**Ответ.** Указатели и массивы очень тесно связаны и во многих случаях взаимозаменяемы. Например, с помощью указателя, который содержит адрес начала массива, можно получить доступ к элементам этого массива либо посредством арифметических действий над указателем, либо посредством индексирования массива. Однако утверждать, что указатели и массивы взаимозаменяемыми в общем случае нельзя. Рассмотрим, например, такой фрагмент кода.

```
int num[10];
    int i;

    for(i=0; i<10; i++) {
        *num = i; // Здесь все в порядке.
        num++;    // ОШИБКА! Переменную num
                    // модифицировать нельзя.
    }
```

Здесь используется массив целочисленных значений с именем num. Как отмечено в комментарии, несмотря на то, что совершенно приемлемо применить к имени num оператор "\*" (который обычно применяется к указателям), абсолютно недопустимо модифицировать значение num. Дело в том, что num — это константа, которая указывает на начало массива. И ее, следовательно, инкрементировать никак нельзя. Другими словами, несмотря на то, что имя массива (без индекса) действительно генерирует указатель на начало массива, его значение изменению не подлежит. Хотя имя массива генерирует константу-указатель, его, тем не менее, можно (подобно указателям) включать в выражения, если, конечно, оно при этом не модифицируется. Например, следующая инструкция, при выполнении которой элементу num[3] присваивается значение 100, вполне допустима.

```
* (num+3) = 100; // Здесь все в порядке,
                  // поскольку num не изменяется.
```

При выполнении этой программы символы, образующие строковую константу, сохраняются в таблице строк, а переменной ptr присваивается указатель на соответствующую строку в этой таблице.

Поскольку указатель на таблицу строк конкретной программы при использовании строкового литерала генерируется автоматически, то можно попытаться использовать этот факт для модификации содержимого данной таблицы. Однако такое решение вряд ли можно назвать удачным. Дело в том, что C++-компиля-

торы создают оптимизированные таблицы, в которых один строковый литерал может использоваться в двух (или больше) различных местах программы. Поэтому “насильственное” изменение строки может вызвать нежелательные побочные эффекты.

## Проект 4.2 Реверсирование строки

**StrRev.cpp**

Выше отмечалось, что сравнение указателей имеет смысл, если сравниваемые указатели ссылаются на один и тот же объект, например массив. Допустим, даны два указателя (с именами A и B), которые ссылаются на один и тот же массив. Теперь, когда вы понимаете, как могут быть связаны указатели и массивы, можно применить сравнение указателей для упрощения некоторых алгоритмов. В этом проекте мы рассмотрим один такой пример.

Разрабатываемая здесь программа реверсирует содержимое строки, т.е. меняет на обратный порядок следования ее символов. Но вместо копирования строки в обратном порядке (от конца к началу) в другой массив, она реверсирует содержимое строки внутри массива, который ее содержит. Для реализации вышесказанного программа использует две переменные типа указатель. Одна указывает на начало массива (его первый элемент), а вторая — на его конец (его последний элемент). Цикл, используемый в программе, продолжается до тех пор, пока указатель на начало строки меньше указателя на ее конец. На каждой итерации цикла символы, на которые ссылаются эти указатели, меняются местами, после чего обеспечивается настройка указателей (путем инкрементирования и декрементирования) на следующие символы. Когда указатель на начало строки в результате сравнения становится больше или равным указателю на ее конец, строка оказывается инвертированной.

### Последовательность действий

1. Создайте файл с именем **StrRev.cpp**.
2. Введите в этот файл следующие строки кода.

```
/*
Проект 4.2.
Реверсирование содержимого строки "по месту".
*/
#include <iostream>
#include <cstring>
using namespace std;
```

```

int main()
{
    char str[] = "это простой тест";
    char *start, *end;
    int len;
    char t;

```

Строка, подлежащая реверсированию, содержится в массиве `str`. Для доступа к нему используются указатели `start` и `end`.

3. Введите в файл программы код, предназначенный для отображения исходной строки, получения ее длины и установки начальных значений указателей `start` и `end`.

```
cout << "Исходная строка: " << str << "\n";
```

```
len = strlen(str);
```

```
start = str;
end = &str[len-1];
```

Обратите внимание на то, что указатель `end` ссылается на последний символ строки, а не на признак ее завершения (нулевой символ).

4. Добавьте код, который обеспечивает реверсирование символов в строке.

```

while(start < end) {
    // обмен символов
    t = *start;
    *start = *end;
    *end = t;

    // настройка указателей на следующие символы
    start++;
    end--;
}

```

Этот процесс работает следующим образом. До тех пор пока указатель `start` ссылается на область памяти, адрес которой меньше адреса, содержащегося в указателе `end`, цикл `while` продолжает работать, т.е. менять местами символы, адресуемые указателями `start` и `end`. Каждая итерация цикла заканчивается инкрементированием указателя `start` и декрементированием указателя `end`. В тот момент, когда значение указателя `start`

станет больше или равным значению указателя end, можно утверждать, что все символы строки реверсированы. Поскольку указатели start и end ссылаются на один и тот же массив, их сравнение имеет смысл.

### 5. Приведем полный текст программы StrRev.cpp.

```
/*
Проект 4.2.
Реверсирование содержимого строки "по месту".
*/
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char str[] = "это простой тест";
    char *start, *end;
    int len;
    char t;

    cout << "Исходная строка: " << str << "\n";

    len = strlen(str);

    start = str;
    end = &str[len-1];

    while(start < end) {
        // обмен символов
        t = *start;
        *start = *end;
        *end = t;

        // настройка указателей на следующие символы
        start++;
        end--;
    }

    cout << "Строка после реверсирования: " << str << "\n";
}
```

```
n";
    return 0;
}
```

Результаты выполнения этой программы таковы.

Исходная строка: это простой тест

Строка после реверсирования: тсет йотсорп отэ

## Массивы указателей

Указатели, подобно данным других типов, могут храниться в массивах. Вот, например, как выглядит объявление 10-элементного массива указателей на int-значения.

```
int *pi[10];
```

Здесь каждый элемент массива pi содержит указатель на целочисленное значение.

Чтобы присвоить адрес int-переменной с именем var третьему элементу этого массива указателей, запишите следующее.

```
int var;
pi[2] = &var;
```

Помните, что здесь pi — массив указателей на целочисленные значения. Элементы этого массива могут содержать только значения, которые представляют собой *адреса* переменных целочисленного типа, а не сами значения. Вот поэтому переменная var предваряется оператором "&".

Чтобы узнать значение переменной var с помощью массива pi, используйте такой синтаксис.

```
*pi[2]
```

Поскольку адрес переменной var хранится в элементе pi[2], применение оператора "\*" к этой индексированной переменной и позволяет получить значение переменной var.

Подобно другим массивам, массивы указателей можно инициализировать. Как правило, инициализированные массивы указателей используются для хранения указателей на строки. Рассмотрим пример использования двумерного массива символьных указателей для создания небольшого толкового словаря.

```
// Использование двумерного массива указателей
// для создания словаря.
```

```
#include <iostream>
#include <cstring>
```

## 204 Модуль 4. Массивы, строки и указатели

```
using namespace std;

int main() {
    // Двумерный массив char-указателей для адресации
    // пар строк.
    char *dictionary[][2] = {
        "карандаш", "Инструмент для письма или рисования.",
        "клавиатура", "Устройство ввода данных.",
        "винтовка", "Огнестрельное оружие.",
        "самолет", "Воздушное судно с неподвижным крылом.",
        "сеть", "Взаимосвязанная группа компьютеров.",
        "", ""
    };
    char word[80];
    int i;

    cout << "Введите слово: ";
    cin >> word;

    for(i = 0; *dictionary[i][0]; i++) {
        // Для отыскания определения в массиве dictionary
        // выполняется поиск значения строки word. Если
        // обнаруживается совпадение, отображается определение,
        // соответствующее найденному слову.
        if(!strcmp(dictionary[i][0], word)) {
            cout << dictionary[i][1] << "\n";
            break;
        }
    }

    if(!*dictionary[i][0])
        cout << word << " не найдено.\n";
}

return 0;
}
```

Вот пример выполнения этой программы.

Введите слово: сеть

Взаимосвязанная группа компьютеров.

При создании массива `dictionary` инициализируется набором слов и их значений (определений). Вспомните: C++ запоминает все строковые константы в таблице строк, связанной с конкретной программой, поэтому этот массив нужен только для хранения указателей на используемые в программе строки. Функционирование словаря заключается в сравнении слова, введенного пользователем, со строками, хранимыми в словаре. При обнаружении совпадения отображается строка, которая является его определением. В противном случае (когда заданное слово в словаре не найдено) выводится сообщение об ошибке.

Обратите внимание на то, что инициализация массива `dictionary` завершается двумя пустыми строками. Они служат в качестве признака конца массива. Вспомните, что пустые строки содержат только завершающие нулевые символы. Используемый в программе цикл `for` продолжается до тех пор, пока первым символом строки не окажется нуль-символ. Это условие реализуется с помощью такого выражения.

```
*dictionary[i][0]
```

Индексы массива задают указатель на строку. Оператор "\*" позволяет получить символ по указанному адресу. Если этот символ окажется нулевым, то выражение `*dictionary[i][0]` будет оценено как ложное, и цикл завершится. В противном случае это выражение (как истинное) дает "добро" на продолжение цикла.

## Соглашение о нулевых указателях

Объявленный, но не инициализированный указатель будет содержать произвольное значение. При попытке использовать указатель до присвоения ему конкретного значения можно разрушить не только собственную программу, но даже и операционную систему (отвратительнейший, надо сказать, тип ошибки!). Поскольку не существует гарантированного способа избежать использования неинициализированного указателя, C++-программисты приняли процедуру, которая позволяет избегать таких ужасных ошибок. По соглашению, если указатель содержит нулевое значение, считается, что он ни на что не ссылается. Это значит, что, если всем неиспользуемым указателям присваивать нулевые значения и избегать применения нулевых указателей, можно предотвратить случайное использование неинициализированного указателя. Вам также следует придерживаться этой практики программирования.

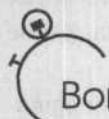
При объявлении указатель любого типа можно инициализировать нулевым значением, например, как это делается в следующей инструкции.

```
float *p = 0; // p теперь нулевой указатель.
```

Для тестирования указателя используется инструкция `if` (любой из следующих ее вариантов).

`if(p) // Выполняем что-то, если p – не нулевой указатель.`

`if(!p) // Выполняем что-то, если p – нулевой указатель.`



### Вопросы для текущего контроля

1. Используемые в программе строковые константы сохраняются в \_\_\_\_\_.
2. Что создает объявление `float *fpa[18];?`
3. По соглашению указатель, содержащий нулевое значение, считается неиспользуемым. Так ли это?\*

**ВАЖНО!**

## 4.2 Многоуровневая непрямая адресация

Можно создать указатель, который будет ссылаться на другой указатель, а тот – на конечное значение. Этую ситуацию называют цепочкой указателей, *многоуровневой непрямой адресацией* (multiple indirection) или использованием *указателя на указатель*. Идея многоуровневой непрямой адресации схематично проиллюстрирована на рис. 4.2. Как видите, значение обычного указателя (при одноуровневой непрямой адресации) представляет собой адрес переменной, которая содержит некоторое значение. В случае применения указателя на указатель первый содержит адрес второго, а тот ссылается на переменную, содержащую определенное значение.

При использовании непрямой адресации можно организовать любое желаемое количество уровней, но, как правило, ограничиваются лишь двумя, поскольку увеличение числа уровней часто ведет к возникновению концептуальных ошибок.

1. Используемые в программе строковые константы сохраняются в таблице строк.
2. Это объявление создает 18-элементный массив указателей на `float`-значения.
3. Да, указатель, содержащий нулевое значение, считается неиспользуемым.

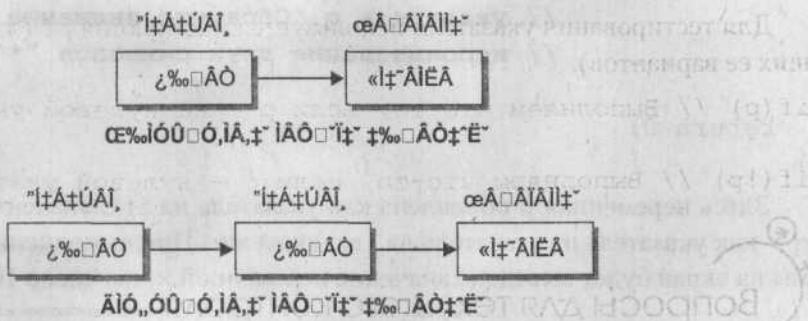


Рис. 4.2. Одноуровневая и многоуровневая непрямая адресация

Переменную, которая является указателем на указатель, нужно объявить соответствующим образом. Для этого достаточно перед ее именем поставить дополнительный символ “звездочка” (\*). Например, следующее объявление сообщает компилятору о том, что balance – это указатель на указатель на значение типа int.

```
int **balance;
```

Необходимо помнить, что переменная balance здесь – не указатель на целочисленное значение, а указатель на указатель на int-значение.

Чтобы получить доступ к значению, адресуемому указателем на указатель, необходимо дважды применить оператор “\*”, как показано в следующем примере.

```
// Использование многоуровневой непрямой адресации.
```

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int x, *p, **q;
    x = 10;
    p = &x; // Присваиваем переменной p адрес переменной x.

    q = &p; // Присваиваем переменной q адрес переменной p.

    cout << **q; // Выводим значение переменной x. (Доступ к
                  // значению переменной x мы получаем через
```

```
// указатель q. Обратите внимание на  
// использование двух символов "**".)  
  
return 0;  
}
```

Здесь переменная `p` объявлена как указатель на `int`-значение, а переменная `q` — как указатель на указатель на `int`-значение. При выполнении этой программы на экран будет выведено значение переменной `x`, т.е. число 10.

### Спросим у опытного программиста

**Вопрос.** *“Неосторожное” обращение с указателями, учитывая их огромные возможности, может иметь разрушительные последствия для программы. Как можно избежать ошибок при работе с указателями?*

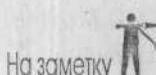
**Ответ.** Во-первых, перед использованием указателя необходимо убедиться, что он инициализирован, т.е. действительно указывает на существующее значение. Во-вторых, следует позаботиться о том, чтобы тип объекта, адресуемого указателем, совпадал с его базовым типом. В-третьих, не выполняйте операции с использованием нулевых указателей. Не забывайте: нулевой указатель означает, что он ни на что не указывает. Наконец, не выполняйте с указателями операции приведения типа “только для того, чтобы программа скомпилировалась”. Обычно ошибки, связанные с несогласием типов указателей, свидетельствуют о явной недоработке программиста. Практика показывает, что приведение одного типа указателя к другому требуется лишь в особых обстоятельствах.



### Тест для самоконтроля по модулю 4

- Покажите, как объявить массив `hightemps` для хранения 31 элемента типа `short int`?
- В C++ индексирование всех массивов начинается с \_\_\_\_.
- Напишите программу, которая находит и отображает значения-дубликаты в 10-элементном массиве целочисленных значений (если такие в нем присутствуют).
- Что такое строка с завершающим нулем?
- Напишите программу, которая предлагает пользователю ввести две строки, а затем сравнивает их, игнорируя “регистровые” различия, т.е. прописные и строчные буквы ваша программа должна воспринимать как одинаковые.

6. Какой размер должен иметь массив-приемник при использовании функции `strcat()`?
7. Как задается каждый индекс в многомерном массиве?
8. Покажите, как инициализировать `int`-массив `nums` значениями 5, 66 и 88?
9. Какое принципиальное преимущество имеет объявление безразмерного массива?
10. Что такое указатель? Какие вы знаете два оператора, используемые с указателями?
11. Можно ли индексировать указатель подобно массиву? Можно ли получить доступ к элементам массива посредством указателя?
12. Напишите программу, которая подсчитывает и отображает на экране количество прописных букв в строке.
13. Как называется ситуация, когда используется указатель, который ссылается на другой?
14. Что означает нулевой указатель в C++?



Ответы на эти вопросы можно найти на Web-странице данной книги по адресу: <http://www.osborne.com>.

# Модуль 5

## Введение в функции

- 5.1.** Общий формат C++-функций
- 5.2.** Создание функции
- 5.3.** Использование аргументов
- 5.4.** Использование инструкции `return`
- 5.5.** Использование функций в выражениях
- 5.6.** Локальная область видимости
- 5.7.** Глобальная область видимости
- 5.8.** Передача указателей и массивов в качестве аргументов функций
- 5.9.** Возвращение функциями указателей
- 5.10.** Передача аргументов командной строки функции `main()`
- 5.11.** Прототипы функций
- 5.12.** Рекурсия

В этом модуле мы приступаем к углубленному рассмотрению функций. Функции – это строительные блоки C++, а потому без полного их понимания невозможно стать успешным C++-программистом. Здесь вы узнаете, как создать функцию и передать ей аргументы, а также о локальных и глобальных переменных, об областях видимости функций, их прототипах и рекурсии.

## Основы использования функций

*Функция* – это подпрограмма, которая содержит одну или несколько C++-инструкций и выполняет определенную задачу. Каждая из приведенных выше программ содержала одну функцию `main()`. Функции называются строительными блоками C++, поскольку программа в C++, как правило, представляет собой коллекцию функций. Все действия программы реализованы в виде функций. Таким образом, функция содержит инструкции, которые с точки зрения человека составляют исполняемую часть программы.

Очень простые программы (как представленные здесь до сих пор) содержат только одну функцию `main()`, в то время как большинство других включает несколько функций. Однако коммерческие программы насчитывают сотни функций.

**ВАЖНО!**

### 5.1 Общий формат C++-функций

Все C++-функции имеют такой общий формат.

```
тип_возвращаемого_значения имя (список_параметров) {  
    . // тело функции  
    .  
}
```

С помощью элемента `тип_возвращаемого_значения` указывается тип значения, возвращаемого функцией. Это может быть практически любой тип, за исключением массива. Если функция не возвращает никакого значения, необходимо указать тип `void`. Если функция действительно возвращает значение, оно должно иметь тип, совместимый с указанным в определении функции. Каждая функция имеет `имя`. В качестве имени можно использовать любой допустимый идентификатор, который еще не был задействован в программе. После имени функции в круглых скобках указывается `список параметров`, который представляет собой последовательность пар (состоящих из типа данных и имени), разделенных запятыми. Параметры – это, по сути, переменные, которые получают

значение аргументов, передаваемых функции при вызове. Если функция не имеет параметров, элемент *список\_параметров* отсутствует, т.е. круглые скобки остаются пустыми.

В фигурные скобки заключено тело функции. Тело функции составляют C++-инструкции, которые определяют действия функции. Функция завершается (и управление передается вызывающей процедуре) при достижении закрывающей фигурной скобки или инструкции *return*.

**ВАЖНО!**

## 5.2. Создание функции

Функцию создать очень просто. Поскольку все функции используют один и тот же общий формат, их структура подобна структуре функции *main()*, с которой нам уже приходилось иметь дело. Итак, начнем с простого примера, который содержит две функции: *main()* и *myfunc()*. Еще до выполнения этой программы (или чтения последующего описания) внимательно изучите ее текст и попытайтесь предугадать, что она должна отобразить на экране.

```
/* Эта программа содержит две функции: main()
   и myfunc().

*/
#include <iostream>
using namespace std;

void myfunc(); // Прототип функции myfunc().

int main()
{
    cout << "В функции main().\n";
    myfunc(); // Вызываем функцию myfunc().

    cout << "Снова в функции main().\n";

    return 0;
}

// Определение функции myfunc().
void myfunc()
```

```

{
    cout << " В функции myfunc().\n";
}

```

Программа работает следующим образом. Вызывается функция `main()` и выполняется ее первая `cout`-инструкция. Затем из функции `main()` вызывается функция `myfunc()`. Обратите внимание на то, как этот вызов реализуется в программе: указывается имя функции `myfunc`, за которым следуют пара круглых скобок и точка с запятой. Вызов любой функции представляет собой C++-инструкцию и поэтому должен завершаться точкой с запятой. Затем функция `myfunc()` выполняет свою единственную `cout`-инструкцию и передает управление назад функции `main()`, причем той строке кода, которая расположена непосредственно за вызовом функции. Наконец, функция `main()` выполняет свою вторую `cout`-инструкцию, которая завершает всю программу. Итак, на экране мы должны увидеть такие результаты.

В функции `main()`.

В функции `myfunc()`.

Снова в функции `main()`.

То, как организован вызов функции `myfunc()` и ее возврат, представляет собой конкретный вариант общего процесса, который применяется ко всем функциям. В общем случае, чтобы вызвать функцию, достаточно указать ее имя с парой круглых скобок. После вызова управление программой переходит к функции. Выполнение функции продолжается до обнаружения закрывающей фигурной скобки. Когда функция завершается, управление передается инициатору ее вызова.

В этой программе обратите внимание на следующую инструкцию.

`void myfunc(); // Прототип функции myfunc().`

Как отмечено в комментарии, это — *прототип* функции `myfunc()`. Хотя подробнее прототипы будут рассмотрены ниже, без кратких пояснений здесь не обойтись. Прототип функции объявляет функцию до ее определения. Прототип позволяет компилятору узнать тип значения, возвращаемого этой функцией, а также количество и тип параметров, которые она может иметь. Компилятору нужно знать эту информацию до первого вызова функции. Поэтому прототип располагается до функции `main()`. Единственной функцией, которая не требует прототипа, является `main()`, поскольку она встроена в язык C++.

Ключевое слово `void`, которое предваряет как прототип, так и определение функции `myfunc()`, формально заявляет о том, что функция `myfunc()` не возвращает никакого значения. В C++ функции, не возвращающие значений, объявляются с использованием ключевого слова `void`.

ВАЖНО!

### 5.3. Использование аргументов

Функции можно передать одно или несколько значений. Значение, передаваемое функции, называется *аргументом*. Таким образом, аргументы представляют собой средство передачи инициализации в функцию.

При создании функции, которая принимает один или несколько аргументов, необходимо объявить переменные, которые получат значения этих аргументов. Эти переменные называются *параметрами функции*. Рассмотрим пример определения функции с именем `box()`, которая вычисляет объем параллелепипеда и отображает полученный результат. Функция `box()` принимает три параметра.

```
void box(int length, int width, int height)
{
    cout << "Объем параллелепипеда равен "
         << length * width * height << "\n";
}
```

В общем случае при каждом вызове функции `box()` будет вычислен объем параллелепипеда путем умножения значений, переданных ее параметрам `length`, `width` и `height`. Обратите внимание на то, что объявлены эти параметры в виде списка, заключенного в круглые скобки, расположенные после имени функции. Объявление каждого параметра отделяется от следующего запятой. Так объявляются параметры для всех функций (если они их используют).

При вызове функции необходимо указать три аргумента. Вот примеры.

```
box(7, 20, 4);
box(50, 3, 2);
box(8, 6, 9);
```

Значения, заданные в круглых скобках, являются аргументами, передаваемыми функции `box()`. Каждое из этих значений копируется в соответствующий параметр. Так, при первом вызове функции `box()` число 7 копируется в параметр `length`, 20 – в параметр `width`, а 4 – в параметр `height`. При выполнении второго вызова 50 копируется в параметр `length`, 3 – в параметр `width`, а 2 – в параметр `height`. Во время третьего вызова в параметр `length` будет скопировано число 8, в параметр `width` – число 6 и в параметр `height` – число 9.

Использование функции `box()` демонстрируется в следующей программе.

```
// Программа демонстрации использования функции box().
```

```
#include <iostream>
using namespace std;
```

## 216 Модуль 5. Введение в функции

```
void box(int length, int width, int height); // Прототип
  // функции box().

int main()
{
    box(7, 20, 4); // Передаем аргументы функции box().
    box(50, 3, 2);
    box(8, 6, 9);

    return 0;
}

// Вычисление объема параллелепипеда.

void box(int length, int width, int height) // Параметры
  // принимают значения аргументов,
  // переданных функции box().
{
    cout << "Объем параллелепипеда равен "
        << length * width * height << "\n";
}
```

При выполнении программы генерируются такие результаты.

Объем параллелепипеда равен 560

Объем параллелепипеда равен 300

Объем параллелепипеда равен 432



Помните, что термин *аргумент* относится к значению, которое используется при вызове функции. Переменная, которая принимает значение аргумента, называется *параметром*. Функции, принимающие аргументы, называются *параметризованными*.

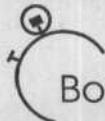
**ВАЖНО!**

### 5.4. Использование инструкции

#### **return**

В предыдущих примерах возврат из функции к инициатору ее вызова происходил при обнаружении закрывающей фигурной скобки. Однако это приемлемо

не для всех функций. Иногда требуется более гибкое средство управления возвратом из функции. Таким средством служит инструкция `return`.



## Вопросы для текущего контроля

1. Как выполняется программа при вызове функции?
2. Чем аргумент отличается от параметра?
3. Если функция использует параметр, то где он объявляется?\*

Инструкция `return` имеет две формы применения. Первая позволяет возвращать значение, а вторая — нет. Начнем с версии инструкции `return`, которая не возвращает значение. Если тип значения, возвращаемого функцией, определяется ключевым словом `void` (т.е. функция не возвращает значения вообще), то для выхода из функции достаточно использовать такую форму инструкции `return`.

При обнаружении инструкции `return` управление программой немедленно передается инициатору ее вызова. Любой код в функции, расположенный за инструкцией `return`, игнорируется. Рассмотрим, например, эту программу.

```
// Использование инструкции return.
```

```
#include <iostream>
using namespace std;

void f();
int main()
{
    cout << "До вызова функции.\n";
    f();
}
```

1. После вызова управление программой переходит к функции. Когда выполнение функции завершается, управление передается инициатору ее вызова, причем той строке кода, которая расположена непосредственно за вызовом функции.
2. Аргумент — это значение, передаваемое функции. Параметр — это переменная, которая принимает это значение.
3. Параметр объявляется в круглых скобках, стоящих после имени функции.

## 218 Модуль 5. Введение в функции

```
cout << "После вызова функции.\n";  
  
return 0;  
}  
  
// Определение void-функции, которая использует  
// инструкцию return.  
void f()  
{  
    cout << "В функции f().\n";  
  
    return; // Немедленное возвращение к инициатору вызова  
            // без выполнения следующей cout-инструкции.  
  
    cout <<"Этот текст не будет отображен.\n";  
}
```

Вот как выглядят результаты выполнения этой программы.

До вызова функции.

В функции f().

После вызова функции.

Как видно из результатов выполнения этой программы, функция `f()` возвращается в функцию `main()` сразу после обнаружения инструкции `return`. Следующая после `return` инструкция `cout` никогда не выполнится.

Теперь рассмотрим более практический пример использования инструкции `return`. Функция `power()`, показанная в следующей программе, отображает результат возведения целочисленного значения в положительную целую степень. Если показатель степени окажется отрицательным, инструкция `return` немедленно завершит эту функцию еще до попытки вычислить результат.

```
#include <iostream>  
using namespace std;  
  
void power(int base, int exp);  
  
int main()  
{  
    power(10, 2);  
    power(10, -2);
```

```

    return 0;
}

// Возводим целое число в положительную степень.
void power(int base, int exp)
{
    int i;

    if(exp < 0) return; /* Отрицательные показатели степени
                        не обрабатываются. */

    i = 1;

    for( ; exp; exp--) i = base * i;
    cout << "Ответ равен: " << i;
}

```

Результат выполнения этой программы таков.

Ответ равен: 100

Если значение параметра `exp` отрицательно (как при втором вызове функции `power()`), вся оставшаяся часть функции опускается.

Функция может содержать несколько инструкций `return`. В этом случае возврат из функции будет выполнен при обнаружении одной из них. Например, следующий фрагмент кода вполне допустим.

```

void f()
{
    // ...

    switch(c) {
        case 'a': return;
        case 'b': // ...
        case 'c': return;
    }
    if(count < 100) return;
    // ...
}

```

Однако следует иметь в виду, что слишком большое количество инструкций `return` может ухудшить ясность алгоритма и ввести в заблуждение тех, кто бу-

дет в нем разбираться. Несколько инструкций `return` стоит использовать только в том случае, если они способствуют ясности функции.

## Возврат значений

Функция может возвращать значение инициатору своего вызова. Таким образом, возвращаемое функцией значение — это средство получения информации из функции. Чтобы вернуть значение, используйте вторую форму инструкции `return`.

```
return значение;
```

Эту форму инструкции `return` можно использовать только с не `void`-функциями.

Функция, которая возвращает значение, должна определить тип этого значения. Тип возвращаемого значения должен быть совместимым с типом данных, используемых в инструкции `return`. В противном случае неминуема ошибка во время компиляции программы. Функция может возвращать данные любого допустимого в C++ типа, за исключением массива.

Чтобы проиллюстрировать процесс возврата функцией значений, перепишем уже известную вам функцию `box()`. В этой версии функция `box()` возвращает объем параллелепипеда. Обратите внимание на расположение функции с правой стороны от оператора присваивания. Это значит, что переменной, расположенной с левой стороны, присваивается значение, возвращаемое функцией.

```
// Возврат значения.
```

```
#include <iostream>
using namespace std;

int box(int length, int width, int height); // Функция
  // возвращает объем параллелепипеда.

int main()
{
    int answer;

    answer = box(10, 11, 3); // Значение, возвращаемое
                           // функцией, присваивается
                           // переменной answer.

    cout << "Объем параллелепипеда равен " << answer;
```

```

    return 0;
}

// Эта функция возвращает значение.
int box(int length, int width, int height)
{
    return length * width * height ; // Возвращается объем.
}

```

Вот результат выполнения этой программы.

Объем параллелепипеда равен 330

В этом примере функция `box()` с помощью инструкции `return` возвращает значение выражения `length * width * height`, которое затем присваивается переменной `answer`. Другими словами, значение, возвращаемое инструкцией `return`, становится значением выражения `box()` в вызывающем коде.

Поскольку функция `box()` теперь возвращает значение, ее прототип и определение не предваряется ключевым словом `void`. (Вспомните: ключевое слово `void` используется только для функции, которая *не* возвращает значение.) На этот раз функция `box()` объявляется как возвращающая значение типа `int`.

Безусловно, тип `int` – не единственный тип данных, которые может возвращать функция. Как упоминалось выше, функция может возвращать данные любого допустимого в C++ типа, за исключением массива. Например, в следующей программе мы переделаем функцию `box()` так, чтобы она принимала параметры типа `double` и возвращала значение типа `double`.

```

// Возврат значения типа double.

#include <iostream>
using namespace std;

// Использование параметров типа double.
double box(double length, double width, double height);

int main()
{
    double answer;

    answer = box(10.1, 11.2, 3.3); // Присваиваем значение,
                                    // возвращаемое функцией.
    cout << "Объем параллелепипеда равен " << answer;
}

```

```

    // Непосредственное использование значения,
    return 0;
}

// Эта версия функции box() использует данные типа double.
double box(double length, double width, double height)
{
    return length * width * height ;
}

```

Вот результат выполнения этой программы.

Объем параллелепипеда равен 373.296

Необходимо также отметить следующее. Если не void-функция завершается из-за обнаружения закрывающейся фигурной скобки, инициатору вызова функции вернется неопределенное (т.е. неизвестное) значение. Из-за особенностей формального синтаксиса C++ не void-функция не обязана в действительности выполнять инструкцию `return`. Но, поскольку функция объявлена как возвращающая значение, значение должно быть возвращено “во что бы то ни стало” — пусть даже это будет “мусор”. Конечно, хорошая практика программирования подразумевает, что не void-функция должна возвращать значение посредством явно выполняемой инструкции `return`.

**ВАЖНО!**

## 5.5. Использование функций в выражениях

В предыдущем примере значение, возвращаемое функцией `box()`, присваивалось переменной, а затем значение этой переменной отображалось с помощью инструкции `cout`. Этую программу можно было бы переписать в более эффективном варианте, используя возвращаемое функцией значение непосредственно в инструкции `cout`. Например, функцию `main()` из предыдущей программы можно было бы представить в таком виде.

```

int main()
{
    // Непосредственное использование значения,
    // возвращаемого функцией box().
    cout << "Объем параллелепипеда равен "
        << box(10.1, 11.2, 3.3); // Использование значения,

```

```

    // возвращаемого функцией
    // box(), непосредственно
    // в инструкции cout.

return 0;
}

```

При выполнении этой инструкции cout автоматически вызывается функция box() для получения возвращаемого ею значения, которое и выводится на экран. Выходит, совершенно необязательно присваивать его сначала некоторой переменной.

В общем случае void-функцию можно использовать в выражении любого типа. При вычислении выражения функция, которая в нем содержится, автоматически вызывается с целью получения возвращаемого ею значения. Например, в следующей программе суммируется объем трех параллелепипедов, а затем отображается среднее значение объема.

```

// Использование функции box() в выражении.

#include <iostream>
using namespace std;

// Используем параметры типа double.
double box(double length, double width, double height);

int main()
{
    double sum;

    sum = box(10.1, 11.2, 3.3) + box(5.5, 6.6, 7.7)
        + box(4.0, 5.0, 8.0);

    cout << "Суммарный объем всех параллелепипедов равен "
        << sum << "\n";
    cout << "Средний объем равен " << sum / 3.0 << "\n";

    return 0;
}

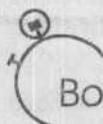
// В этой версии функции box() используются параметры
// типа double.

```

```
double box(double length, double width, double height)
{
    return length * width * height ;
}
```

При выполнении эта программа генерирует такие результаты.

Суммарный объем всех параллелепипедов равен 812.806  
Средний объем равен 270.935



### Вопросы для текущего контроля

1. Покажите две формы применения инструкции `return`.
2. Может ли `void`-функция возвращать значение?
3. Может ли функция быть частью выражения?\*

## Правила действия областей видимости функций

До сих пор мы использовали переменные, не рассматривая формально, где их можно объявить, как долго они существуют и какие части программы могут получить к ним доступ. Эти атрибуты определяются правилами действия областей видимости, определенными в C++.

*Правила действия областей видимости* любого языка программирования – это правила, которые позволяют управлять доступом к объекту из различных частей программы. Другими словами, правила действия областей видимости определяют, какой код имеет доступ к той или иной переменной. Эти правила также определяют продолжительность “жизни” переменной. В C++ определено две основные области видимости: *локальные* и *глобальные*. В любой из них можно объявлять переменные. В этом разделе мы рассмотрим, чем переменные, объявленные в разных областях, отличаются.

1. Вот как выглядят две формы инструкции `return`.

```
METHOD return;
        return значение;
```

2. Нет, `void`-функция не может возвращать значение.
3. Вызов не `void`-функции можно использовать в выражении любого типа. В этом случае функция, которая в нем содержится, автоматически выполняется с целью получения возвращаемого ею значения.

ленные в локальной области, отличаются от переменных, объявленных в глобальной, и как каждая из них связана с функциями.

**ВАЖНО!**

## 5.6. Локальная область видимости

Локальная область видимости создается блоком. (Вспомните, что блок начинается с открывающей фигурной скобки и завершается закрывающей.) Таким образом, каждый раз при создании нового блока программист создает новую область видимости. Переменная, объявленная внутри любого блока кода, называется *локальной* (по отношению к этому блоку).

Локальную переменную могут использовать лишь инструкции, включенные в блок, в котором эта переменная объявлена. Другими словами, локальная переменная неизвестна за пределами собственного блока кода. Следовательно, инструкции вне блока не могут получить доступ к объекту, определенному внутри блока. По сути, объявляя локальную переменную, вы защищаете ее от несанкционированного доступа и/или модификации. Более того, можно сказать, что правила действия областей видимости обеспечивают фундамент для инкапсуляции.

Важно понимать, что локальные переменные существуют только во время выполнения программного блока, в котором они объявлены. Это означает, что локальная переменная создается при входе в “свой” блок и разрушается при выходе из него. А поскольку локальная переменная разрушается при выходе из “своего” блока, ее значение теряется.

Самым распространенным программным блоком является функция. В C++ каждая функция определяет блок кода, который начинается с открывающей фигурной скобки этой функции и завершается ее закрывающей фигурной скобкой. Код функции и ее данные – это ее “частная собственность”, и к ней не может получить доступ ни одна инструкция из любой другой функции, за исключением инструкции ее вызова. (Например, невозможно использовать инструкцию `goto` для перехода в середину кода другой функции.) Тело функции надежно скрыто от остальной части программы, и она не может оказать никакого влияния на другие части программы, равно, как и те на нее. Таким образом, содержимое одной функции совершенно независимо от содержимого другой. Другими словами, код и данные, определенные в одной функции, не могут взаимодействовать с кодом и данными, определенными в другой, поскольку две функции имеют различные области видимости.

Так как каждая функция определяет собственную область видимости, переменные, объявленные в одной функции, не оказывают никакого влияния на пере-

менные, объявленные в другой, причем даже в том случае, если эти переменные имеют одинаковые имена. Рассмотрим, например, следующую программу.

```
#include <iostream>
using namespace std;

void f1();

int main()
{
    int val = 10; // Эта переменная val локальна по
                  // отношению к функции main().

    cout << "Значение переменной val в функции main(): "
        << val << '\n';
    f1();
    cout << "Значение переменной val в функции main(): "
        << val << '\n';

    return 0;
}

void f1()
{
    int val = 88; // Эта переменная val локальна по
                  // отношению к функции f1().

    cout << "Значение переменной val в функции f1(): "
        << val << "\n";
}
```

Вот результаты выполнения этой программы.

Значение переменной val в функции main(): 10

Значение переменной val в функции f1(): 88

Значение переменной val в функции main(): 10

Целочисленная переменная val объявляется здесь дважды: первый раз в функции main() и еще раз — в функции f1(). При этом переменная val, объявленная в функции main(), не имеет никакого отношения к одноименной переменной из функции f1(). Как разъяснялось выше, каждая переменная (в данном случае val) известна только блоку кода, в котором она объявлена. Чтобы убе-

диться в этом, достаточно выполнить приведенную выше программу. Как видите, несмотря на то, что при выполнении функции `f1()` переменная `val` устанавливается равной числу 88, значение переменной `val` в функции `main()` остается равным 10.

Поскольку локальные переменные создаются с каждым входом и разрушаются с каждым выходом из программного блока, в котором они объявлены, они не хранят своих значений между активизациями блоков. Это особенно важно помнить в отношении функций. При вызове функции ее локальные переменные создаются, а при выходе из нее — разрушаются. Это означает, что локальные переменные не сохраняют своих значений между вызовами функций.

Если объявление локальной переменной включает инициализатор, то такая переменная инициализируется при каждом входе в соответствующий блок. Рассмотрим пример.

```
/*
Локальная переменная инициализируется
при каждом входе в "свой" блок.
*/
#include <iostream>
using namespace std;

void f();

int main()
{
    for(int i=0; i < 3; i++) f();

    return 0;
}

// Переменная num инициализируется при каждом вызове
// функции f().
void f()
{
    int num = 99; // При каждом вызове функции f()
                  // переменная num устанавливается равной 99.

    cout << num << "\n";
}
```

```
    num++; // Это действие не имеет устойчивого эффекта.
}
```

Результаты выполнения этой программы подтверждают тот факт, что переменная num заново устанавливается при каждом вызове функции f().

```
99  
99  
99
```

Содержимое неинициализированной локальной переменной будет неизвестно до тех пор, пока ей не будет присвоено некоторое значение.

### Локальные переменные можно объявлять внутри любого блока

Обычно все переменные, которые необходимы для выполнения функции, объявляются в начале блока кода этой функции. Это делается, в основном, для того, чтобы тому, кто станет разбираться в коде этой функции, было проще понять, какие переменные здесь используются. Однако начало блока функции – не единственное место, где можно объявлять локальные переменные. Локальные переменные могут быть объявлены в любом месте блока кода. Локальная переменная, объявленная внутри блока, локальна по отношению к этому блоку. Это означает, что переменная не существует до входа в этот блок, и разрушается при выходе из блока. Более того, никакой код вне этого блока – включая иной код в той же функции – не может получить доступ к этой переменной. Чтобы убедиться в вышесказанном, рассмотрим следующую программу.

```
// Переменные могут быть локальными по отношению к блоку.

#include <iostream>
using namespace std;

int main() {
    int x = 19; // Переменная x известна всему коду функции.

    if(x == 19) {
        int y = 20; // Переменная y локальна для if-блока.

        cout << "x + y равно " << x + y << "\n";
    }
}
```

```
// y = 100; // Ошибка! Переменная у здесь неизвестна.

return 0;
}
```

Переменная `x` объявляется в начале области видимости функции `main()` и поэтому доступна для всего последующего кода этой функции. В блоке `if`-инструкции объявляется переменная `y`. Поскольку рамками блока и определяется область видимости, переменная `y` видима только для кода внутри этого блока. Поэтому строка

```
y = 100;
```

(она находится за пределами этого блока) отмечена как комментарий. Если убрать символ комментария (`//`) в начале этой строки, компилятор отреагирует сообщением об ошибке, поскольку переменная `y` невидима вне этого блока. Внутри `if`-блока вполне можно использовать переменную `x`, поскольку код в рамках блока имеет доступ к переменным, объявленным во включающем блоке.

Несмотря на то что локальные переменные обычно объявляются в начале своего блока, это не является обязательным требованием. Локальная переменная может быть объявлена в любом месте блока кода — главное, чтобы это произошло до ее использования. Например, следующая программа абсолютно допустима.

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Введите первое число: ";
    int a; // Объявляем одну переменную.
    cin >> a;

    cout << "Введите второе число: ";
    int b; // Объявляем еще одну переменную.
    cin >> b;

    cout << "Произведение равно: " << a*b << '\n';

    return 0;
}
```

В этом примере переменные *a* и *b* объявляются “по мере необходимости”, т.е. не объявляются до тех пор, пока в них нет нужды. Справедливо ради отмечу, что большинство программистов объявляют локальные переменные все-таки в начале функции, в которой они используются, но этот вопрос — чисто стилистический.

## Скрытие имен

Если имя переменной, объявленной во внутреннем блоке, совпадает с именем переменной, объявленной во внешнем блоке, то “внутренняя” переменная скрывает, или переопределяет, “внешнюю” в пределах области видимости внутреннего блока. Рассмотрим пример.

```
#include <iostream>
using namespace std;

int main()
{
    int i, j;

    i = 10;
    j = 100;

    if(j > 0) {
        int i; // Эта переменная i отделена от
               // внешней переменной i.

        i = j / 2;
        cout << "Внутренняя переменная i: " << i << '\n';
    }

    cout << "Внешняя переменная i: " << i << '\n';

    return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

```
Внутренняя переменная i: 50
Внешняя переменная i: 10
```

Здесь переменная *i*, объявленная внутри *if*-блока, скрывает внешнюю переменную *i*. Изменения, которым подверглась внутренняя переменная *i*, не ока-

зывают никакого влияния на внешнюю `i`. Более того, вне `if`-блока внутренняя переменная `i` больше не существует, и поэтому внешняя переменная `i` снова становится видимой.

## Параметры функции

Параметры функции существуют в пределах области видимости функции. Таким образом, они локальны по отношению к функции. Если не считать получения значений аргументов при вызове функции, то поведение параметров ничем не отличается от поведения любых других локальных переменных внутри функции.

### Спросим у опытного программиста

**Вопрос.** *Каково назначение ключевого слова `auto`? Мне известно, что оно используется для объявления локальных переменных. Так ли это?*

**Ответ.** Действительно, язык C++ содержит ключевое слово `auto`, которое можно использовать для объявления локальных переменных. Но поскольку все локальные переменные являются по умолчанию `auto`-переменными, то к этому ключевому слову практически никогда не прибегают. Поэтому вы не найдете в этой книге ни одного примера с его использованием. Но если вы захотите все-таки применить его в своей программе, то знайте, что размещать его нужно непосредственно перед типом переменной, как показано ниже.

```
auto char ch;
```

Еще раз напомню, что ключевое слово `auto` использовать необязательно, и в этой книге вы его больше не встретите.

**ВАЖНО!**

## 5.7. Глобальная область видимости

Поскольку локальные переменные известны только в пределах функции, в которой они объявлены, то у вас может возникнуть вопрос: а как создать переменную, которую могли бы использовать сразу несколько функций? Ответ звучит так: необходимо создать переменную в глобальной области видимости. *Глобальная область видимости* – это декларативная область, которая “заполняет” пространство вне всех функций. При объявлении переменной в глобальной области видимости создается *глобальная переменная*.

Глобальные переменные известны на протяжении всей программы, их можно использовать в любом месте кода, и они поддерживают свои значения во время

## 232 Модуль 5. Введение в функции

выполнения всего кода программы. Следовательно, их область видимости расширяется до объема всей программы. Глобальная переменная создается путем ее объявления вне какой бы то ни было функции. Благодаря их глобальности доступ к этим переменным можно получить из любого выражения, независимо от функции, в которой это выражение находится.

Использование глобальной переменной демонстрируется в следующей программе. Как видите, переменная `count` объявлена вне всех функций (в данном случае до функции `main()`), следовательно, она глобальная. Из обычных практических соображений лучше объявлять глобальные переменные поближе к началу программы. Но формально они просто должны быть объявлены до их первого использования.

```
// Использование глобальной переменной.

#include <iostream>
using namespace std;

void func1();
void func2();

int count; // Это глобальная переменная.

int main()
{
    int i; // Это локальная переменная.

    for(i=0; i<10; i++) {
        count = i * 2; // Здесь результат сохраняется
                        // в глобальной переменной count.
        func1();
    }

    return 0;
}

void func1()
{
    cout << "count: " << count; // Доступ к глобальной
                                // переменной count.
    cout << '\n'; // Вывод символа новой строки.
```

```

func2();
}

void func2()
{
    int count; // Это локальная переменная.

    // Здесь используется локальная переменная count.
    //
    for(count=0; count<3; count++) cout << '.';
}

```

Результаты выполнения этой программы таковы.

```

count: 0
...count: 2
...count: 4
...count: 6
...count: 8
...count: 10
...count: 12
...count: 14
...count: 16
...count: 18
...

```

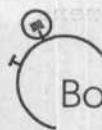
При внимательном изучении этой программы вам должно быть ясно, что как функция `main()`, так и функция `func1()` используют глобальную переменную `count`. Но в функции `func2()` объявляется локальная переменная `count`. Поэтому здесь при использовании имени `count` подразумевается именно локальная, а не глобальная переменная `count`. Помните, что, если в функции глобальная и локальная переменные имеют одинаковые имена, то при обращении к этому имени используется локальная переменная, не оказывая при этом никакого влияния на глобальную. Это и означает, что локальная переменная скрывает глобальную с таким же именем.

Глобальные переменные инициализируются при запуске программы на выполнение. Если объявление глобальной переменной включает инициализатор, то переменная инициализируется заданным значением. В противном случае (при отсутствии инициализатора) глобальная переменная устанавливается равной нулю.

Хранение глобальных переменных осуществляется в некоторой определенной области памяти, специально выделяемой программой для этих целей. Глобаль-

ные переменные полезны в том случае, когда в нескольких функциях программы используются одни и те же данные, или когда переменная должна хранить свое значение на протяжении выполнения всей программы. Однако без особой необходимости следует избегать использования глобальных переменных, и на это есть три причины.

- Они занимают память в течение всего времени выполнения программы, а не только тогда, когда действительно необходимы.
- Использование глобальной переменной в "роли", с которой легко бы "справилась" локальная переменная, делает такую функцию менее универсальной, поскольку она полагается на необходимость определения данных вне этой функции.
- Использование большого количества глобальных переменных может привести к появлению ошибок в работе программы, поскольку при этом возможно проявление неизвестных и нежелательных побочных эффектов. Основная проблема, характерная для разработки больших C++-программ, — случайная модификация значения переменной в каком-то другом месте программы. Чем больше глобальных переменных в программе, тем больше вероятность ошибки.



### Вопросы для текущего контроля

1. Каковы основные различия между локальными и глобальными переменными?
2. Можно ли локальную переменную объявить в любом месте внутри блока?
3. Сохраняет ли локальная переменная свое значение между вызовами функции, в которой она объявлена?\*

- 
1. Локальная переменная известна только внутри блока, в котором она определена. Она создается при входе в блок и разрушается при выходе из него. Глобальная переменная объявляется вне всех функций. Ее могут использовать все функции, и она существует на протяжении времени выполнения всей программы.
  2. Да, локальную переменную можно объявить в любом месте блока, но до ее использования.
  3. Нет, локальные переменные разрушаются при выходе из функций, в которых они объявлены.

ВАЖНО!

## 5.8. Передача указателей и массивов в качестве аргументов функций

В предыдущих примерах функциям передавались значения простых переменных типа `int` или `double`. Но возможны ситуации, когда в качестве аргументов необходимо использовать указатели и массивы. Рассмотрению особенностей передачи аргументов этого типа и посвящены следующие разделы.

### Передача функции указателя

Чтобы передать функции указатель, необходимо объявить параметр типа указатель. Рассмотрим пример.

```
// Передача функции указателя.

#include <iostream>
using namespace std;

void f(int *j); // Функция f() объявляет параметр-указатель
                // на int-значение.

int main()
{
    int i;
    int *p;

    p = &i; // Указатель p теперь ссылается на переменную i.

    f(p); // Передаем указатель. Функция f() вызывается с
           // указателем на int-значение.

    cout << i; // Переменная i теперь содержит число 100.

    return 0;
}

// Функция f() принимает указатель на int-значение.
void f(int *j)
```

## 236 Модуль 5. Введение в функции

```
*j = 100; // Переменной, адресуемой указателем j,  
           // присваивается число 100.  
}
```

Как видите, в этой программе функция `f()` принимает один параметр: указатель на целочисленное значение. В функции `main()` указателю `p` присваивается адрес переменной `i`. Затем из функции `main()` вызывается функция `f()`, а указатель `p` передается ей в качестве аргумента. После того как параметр-указатель `j` получит значение аргумента `p`, он (так же, как и `p`) будет указывать на переменную `i`, определенную в функции `main()`. Таким образом, при выполнении операции присваивания

```
*j = 100;
```

переменная `i` получает значение 100. Поэтому программа отобразит на экране число 100. В общем случае приведенная здесь функция `f()` присваивает число 100 переменной, адрес которой был передан этой функции в качестве аргумента.

В предыдущем примере необязательно было использовать переменную-указатель `p`. Вместо нее при вызове функции `f()` достаточно взять переменную `i`, предварив ее оператором "&" (при этом, как вы знаете, генерируется адрес переменной `i`). После внесения оговоренного изменения предыдущая программа приобретает такой вид.

```
// Передача указателя функции (исправленная версия).  
#include <iostream>  
using namespace std;  
  
void f(int *j);  
  
int main()  
{  
    int i;  
  
    f(&i); // В переменной-указателе p нет необходимости.  
           // Так напрямую передается адрес переменной i.  
  
    cout << i;  
  
    return 0;  
}  
  
void f(int *j)  
{
```

```
*j = 100; // Переменной, адресуемой указателем j,  
          // присваивается число 100.  
}
```

Передавая указатель функции, необходимо понимать следующее. При выполнении некоторой операции в функции, которая использует указатель, эта операция фактически выполняется над переменной, адресуемой этим указателем. Это значит, что такая функция может изменить значение объекта, адресуемого ее параметром.

## Передача функции массива

Если массив является аргументом функции, то необходимо понимать, что при вызове такой функции ей передается только адрес первого элемента массива, а не полная его копия. (Помните, что в C++ имя массива без индекса представляет собой указатель на первый элемент этого массива.) Это означает, что объявление параметра должно иметь тип, совместимый с типом аргумента. Вообще существует три способа объявить параметр, который принимает указатель на массив. Во-первых, параметр можно объявить как массив, тип и размер которого совпадает с типом и размером массива, используемого при вызове функции. Этот вариант объявления параметра-массива продемонстрирован в следующем примере.

```
#include <iostream>  
using namespace std;  
  
void display(int num[10]);  
  
int main()  
{  
    int t[10], i;  
  
    for(i=0; i<10; ++i) t[i]=i;  
  
    display(t); // Передаем функции массив t.  
  
    return 0;  
}  
  
// Функция выводит все элементы массива.  
void display(int num[10]) // Параметр здесь объявлен как  
                      // массив заданного размера.  
{
```

```

int i;
for(i=0; i<10; i++) cout << num[i] << ' ';
}

```

Несмотря на то что параметр num объявлен здесь как целочисленный массив, состоящий из 10 элементов, C++-компилятор автоматически преобразует его в указатель на целочисленное значение. Необходимость этого преобразования объясняется тем, что никакой параметр в действительности не может принять массив целиком. А так как будет передан один лишь указатель на массив, то функция должна иметь параметр, способный принять этот указатель.

Второй способ объявления параметра-массива состоит в его представлении в виде безразмерного массива, как показано ниже.

```

void display(int num[]) // Параметр здесь объявлен как
                       // безразмерный массив.
{
    int i;

    for(i=0; i<10; i++) cout << num[i] << ' ';
}

```

Здесь параметр num объявляется как целочисленный массив неизвестного размера. Поскольку C++ не обеспечивает проверку нарушения границ массива, то реальный размер массива — нерелевантный фактор для подобного параметра (но, безусловно, не для программы в целом). Целочисленный массив при таком способе объявления также автоматически преобразуется C++-компилятором в указатель на целочисленное значение.

Наконец, рассмотрим третий способ объявления параметра-массива. При передаче массива функции ее параметр можно объявить как указатель. Как раз этот вариант чаще всего используется профессиональными программистами. Вот пример:

```

void display(int *num) // Параметр здесь объявлен как
                      // указатель.
{
    int i;

    for(i=0; i<10; i++) cout << num[i] << ' ';
}

```

Возможность такого объявления параметра объясняется тем, что любой указатель (подобно массиву) можно индексировать с помощью символов квадратных скобок ([ ]). Таким образом, все три способа объявления параметра-массива приводятся к одному результату, который можно выразить одним словом — *указатель*.

Важно помнить, что, если массив используется в качестве аргумента функции, то функции передается адрес этого массива. Это означает, что код функции может потенциально изменить реальное содержимое массива, используемого при вызове функции. Например, в следующей программе функция `cube()` преобразует значение каждого элемента массива в куб этого значения. При вызове функции `cube()` в качестве первого аргумента необходимо передать адрес массива значений, подлежащих преобразованию, а в качестве второго — его размер.

```
// Изменение содержимого массива с помощью функции.
```

```
#include <iostream>
using namespace std;

void cube(int *n, int num);

int main()
{
    int i, nums[10];

    for(i=0; i<10; i++) nums[i] = i+1;

    cout << "Исходное содержимое массива: ";
    for(i=0; i<10; i++) cout << nums[i] << ' ';
    cout << '\n';

    // Вычисляем кубы значений.
    cube(nums, 10); // Передаем функции cube() адрес
                    // массива nums.

    cout << "Измененное содержимое: ";
    for(i=0; i<10; i++) cout << nums[i] << ' ';

    return 0;
}

// Возводим в куб элементы массива.
void cube(int *n, int num)
{
    while(num) {
        *n = *n * *n * *n; // Это выражение изменяет
                            // значение элемента массива,
        n++;
        num--;
    }
}
```

```
//      адресуемого указателем p.  
num--;  
n++;  
}  
}
```

Результаты выполнения этой программы таковы.

Исходное содержимое массива: 1 2 3 4 5 6 7 8 9 10

Измененное содержимое: 1 8 27 64 125 216 343 512 729 1000

Как видите, после обращения к функции `cube()` содержимое массива `nums` изменилось: каждый элемент стал равным кубу исходного значения. Другими словами, элементы массива `nums` были модифицированы инструкциями, составляющими тело функции `cube()`, поскольку ее параметр `p` указывает на массив `nums`.

## Передача функциям строк

Поскольку строки в C++ – это обычные символьные массивы, которые завершаются нулевым символом, то при передаче функции строки реально передается только указатель (типа `char *`) на начало этой строки. Рассмотрим, например, следующую программу. В ней определяется функция `strInvertCase()`, которая инвертирует регистр букв строки, т.е. заменяет строчные символы прописными, а прописные – строчными.

```
// Передача функции строки.  
  
#include <iostream>  
#include <cstring>  
#include <cctype>  
using namespace std;  
  
void strInvertCase(char *str);  
  
int main()  
{  
    char str[80];  
  
    strcpy(str, "This Is A Test");  
  
    strInvertCase(str);  
  
    cout << str; // Отображаем модифицированную строку.
```

```

    return 0;
}

// Инвертируем регистр букв строки.
void strInvertCase(char *str)
{
    while(*str) {

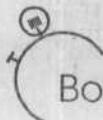
        // Инверсия регистра букв.
        if(isupper(*str)) *str = tolower(*str);
        else if(islower(*str)) *str = toupper(*str);

        str++; // Переход к следующей букве.
    }
}

```

Вот как выглядит результат выполнения этой программы.

THIS iS a tEST



### Вопросы для текущего контроля

- Покажите, как можно объявить void-функцию с именем count, принимающую один параметр ptr, который представляет собой указатель на значение типа long int.
- Если функции передается указатель, то может ли эта функция изменить содержимое объекта, адресуемого этим указателем?
- Можно ли функции в качестве аргумента передать массив? Поясните ответ.\*

- 
- void count(long int \*ptr)
  - Да, объект, адресуемый параметром-указателем, может быть модифицирован кодом функции.
  - Массивы как таковые функции передать нельзя. Но можно передать функции указатель на массив. В этом случае изменения, вносимые в массив кодом функции, окажут влияние на массив, используемый в качестве аргумента.

**ВАЖНО!**

## 5.9. Возвращение функциями указателей

Функции могут возвращать указатели. Указатели возвращаются подобно значениям любых других типов данных и не создают при этом особых проблем. Но, поскольку указатель представляет собой одно из самых сложных (или небезопасных) средств языка C++, имеет смысл посвятить этой теме отдельный раздел.

Чтобы вернуть указатель, функция должна объявить его тип в качестве типа возвращаемого значения. Вот как, например, объявляется тип возвращаемого значения для функции `f()`, которая должна возвращать указатель на целое число.

```
int *f();
```

Если функция возвращает указатель, то значение, используемое в ее инструкции `return`, также должно быть указателем. (Как и для всех функций, `return`-значение должно быть совместимым с типом возвращаемого значения.)

В следующей программе демонстрируется использование указателя в качестве типа возвращаемого значения. Функция `get_substr()` возвращает указатель на первую подстроку (найденную в строке), которая совпадает с заданной. Если заданная подстрока не найдена, возвращается нулевой указатель. Например, если в строке "Я люблю C++" выполняется поиск подстроки "люблю", то функция `get_substr()` возвратит указатель на букву `л` в слове "люблю".

```
// Возвращение функцией указателя.
```

```
#include <iostream>
using namespace std;

char *get_substr(char *sub, char *str);

int main()
{
    char *substr;

    substr = get_substr("три", "один два три четыре");

    cout << "Заданная подстрока найдена: " << substr;

    return 0;
}
```

```

// Возвращаем указатель на подстроку или нуль,
// если таковая не найдена.
char *get_substr(char *sub, char *str) // Эта функция
   // возвращает
   // указатель на
   // char-значение.
{
    int t;
    char *p, *p2, *start;

    for(t=0; str[t]; t++) {
        p = &str[t]; // установка указателей
        start = p;
        p2 = sub;
        while(*p2 && *p2==*p) { // проверка на наличие подстроки
            p++;
            p2++;
        }
        /* Если обнаружен конец p2-строки (т.е. подстроки), то
           искомая подстрока найдена. */
        if(!*p2)
            return start; // Возвращаем указатель на начало
                           // подстроки.
    }
    return 0; // Совпадения не обнаружено.
}

```

Вот результаты выполнения этой программы.

Заданная подстрока найдена: три четыре

## ФУНКЦИЯ `main()`

Как вы уже знаете, функция `main()` — специальная, поскольку это первая функция, которая вызывается при выполнении программы. В отличие от некоторых других языков программирования, в которых выполнение всегда начинается с “сверху”, т.е. с первой строки кода, каждая C++-программа всегда начинается с вызова функции `main()` независимо от ее расположения в программе. (Все же обычно функцию `main()` размещают первой, чтобы ее было легко найти.)

В программе может быть только одна функция `main()`. Если попытаться включить в программу несколько функций `main()`, она “не будет знать”, с какой

из них начать работу. В действительности большинство компиляторов легко обнаружат ошибку этого типа и сообщат о ней. Как упоминалось выше, поскольку функция `main()` встроена в язык C++, она не требует прототипа.

ВАЖНО!

## 5.10. Передача аргументов командной строки функции `main()`

Иногда возникает необходимость передать информацию программе при ее запуске. Как правило, это реализуется путем передачи *аргументов командной строки* функции `main()`. Аргумент командной строки представляет собой информацию, указываемую в команде (командной строке), предназначеннной для выполнения операционной системой, после имени программы. (В Windows команда "Run" (Выполнить) также использует командную строку.) Например, C++-программы можно компилировать путем выполнения следующей команды.

```
c1 prog_name
```

Здесь элемент `prog_name` — имя программы, которую мы хотим скомпилировать. Имя программы передается C++-компилятору в качестве аргумента командной строки.

В C++ для функции `main()` определено два встроенных, но необязательных параметра, `argc` и `argv`, которые получают свои значения от аргументов командной строки. В конкретной операционной среде могут поддерживаться и другие аргументы (такую информацию необходимо уточнить по документации, прилагаемой к вашему компилятору). Рассмотрим параметры `argc` и `argv` более подробно.



Формально для имен параметров командной строки можно выбирать любые идентификаторы, однако имена `argc` и `argv` используются по соглашению уже в течение нескольких лет, и поэтому имеет смысл не прибегать к другим идентификаторам, чтобы любой программист, которому придется разбираться в вашей программе, смог быстро идентифицировать их как параметры командной строки.

Параметр `argc` имеет целочисленный тип и предназначен для хранения количества аргументов командной строки. Его значение всегда не меньше единицы, поскольку имя программы также является одним из учитываемых аргументов.

Параметр `argv` представляет собой указатель на массив символьных указателей. Каждый указатель в массиве `argv` ссылается на строку, содержащую аргумент командной строки. Элемент `argv[0]` указывает на имя программы; элемент `argv[1]` — на первый аргумент, элемент `argv[2]` — на второй и т.д. Все аргументы

командной строки передаются программе как строки, поэтому числовые аргументы необходимо преобразовать в программе в соответствующий внутренний формат.

Важно правильно объявить параметр `argv`. Обычно это делается так.

```
char *argv[];
```

Доступ к отдельным аргументам командной строки можно получить путем индексации массива `argv`. Как это сделать, показано в следующей программе. При ее выполнении на экран выводятся все аргументы командной строки.

```
// Отображение аргументов командной строки.
```

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    for(int i = 0; i < argc; i++) {
        cout << argv[i] << "\n";
    }

    return 0;
}
```

Например, присвоим этой программе имя `ComLine` и вызовем ее так.

```
ComLine
one
two
three
```

В C++ точно не оговорено, как должны быть представлены аргументы командной строки, поскольку среди выполнения (операционные системы) имеют здесь большие различия. Однако чаще всего используется следующее соглашение: каждый аргумент командной строки должен быть отделен пробелом или символом табуляции. Как правило, запятые, точки с запятой и тому подобные знаки не являются допустимыми разделителями аргументов. Например, строка

один, два и три

состоит из четырех строковых аргументов, в то время как строка

один, два, три

включает только два, поскольку запятая не является допустимым разделителем.

Если необходимо передать в качестве одного аргумента командной строки набор символов, который содержит пробелы, то его нужно заключить в кавычки. Например, этот набор символов будет воспринят как один аргумент командной строки: "это лишь один аргумент"

Следует иметь в виду, что представленные здесь примеры применимы к широкому диапазону сред, но это не означает, что ваша среда входит в их число.

Обычно аргументы `argc` и `argv` используются для ввода в программу начальных параметров, исходных значений, имен файлов или вариантов (режимов) работы программы. В C++ можно ввести столько аргументов командной строки, сколько допускает операционная система. Использование аргументов командной строки придает программе профессиональный вид и позволяет использовать ее в командном файле (исполнением текстовом файле, содержащем одну или несколько команд).

## Передача числовых аргументов командной строки

При передаче программе числовых данных в качестве аргументов командной строки эти данные принимаются в строковой форме. В программе должно быть предусмотрено их преобразование в соответствующий внутренний формат с помощью одной из стандартных библиотечных функций, поддерживаемых C++. Перечислим три из них.

- `atof()` Преобразует строку в значение типа `double` и возвращает результат.
- `atol()` Преобразует строку в значение типа `long int` и возвращает результат.
- `atoi()` Преобразует строку в значение типа `int` и возвращает результат.

Каждая из перечисленных функций использует при вызове в качестве аргумента строку, содержащую числовое значение. Для их вызова необходимо включить в программу заголовочный файл `<cstdlib>`.

При выполнении следующей программы выводится сумма двух чисел, которые указываются в командной строке после имени программы. Для преобразования аргументов командной строки во внутреннее представление здесь используется стандартная библиотечная функция `atof()`. Она преобразует число из строкового формата в значение типа `double`.

```
/* Эта программа отображает сумму двух числовых
   аргументов командной строки.
*/
```

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
```

```

{
    double a, b;

    if(argc!=3) {
        cout << "Использование: add число число\n";
        return 1;
    }
    // Использование функции atof() для преобразования
    // строковых аргументов командной строки в числовые
    // значения типа double.
    a = atof(argv[1]); // Преобразование второго аргумента
                        // командной строки.
    b = atof(argv[2]); // Преобразование третьего аргумента
                        // командной строки.

    cout << a + b;

    return 0;
}

```

Чтобы сложить два числа, используйте командную строку такого вида (предполагая, что эта программа имеет имя `add`).

C>`add 100.2 231`



### Вопросы для текущего контроля

1. Как обычно называются два параметра функции `main()`? Поясните их назначение.
  2. Что всегда содержит первый аргумент командной строки?
  3. Числовой аргумент командной строки передается в виде строки. Верно ли это?\*
- 
1. Два параметра функции `main()` обычно называются `argc` и `argv`. Параметр `argc` предназначен для хранения количества аргументов командной строки. Параметр `argv` представляет собой указатель на массив символьных указателей, а каждый указатель в массиве `argv` ссылается на строку, содержащую соответствующий аргумент командной строки.
  2. Первый аргумент командной строки всегда содержит имя самой программы.
  3. Верно, числовые аргументы командной строки принимаются в строковой форме.

**ВАЖНО!**

## 5.11. Прототипы функций

В начале этого модуля мы кратко коснулись темы использования прототипов функций. Теперь настало время поговорить о них подробно. В C++ все функции должны быть объявлены до их использования. Обычно это реализуется с помощью прототипа функции. Прототипы содержат три вида информации о функции:

- тип возвращаемого ею значения;
- тип ее параметров;
- количество параметров.

Прототипы позволяют компилятору выполнить следующие три важные операции.

- Они сообщают компилятору, код какого типа необходимо генерировать при вызове функции. Различия в типах параметров и значении, возвращаемом функцией, обеспечивают разную обработку компилятором.
- Они позволяют C++ обнаружить недопустимые преобразования типов аргументов, используемых при вызове функции, в тип, указанный в объявлении ее параметров, и сообщить о них.
- Они позволяют компилятору выявить различия между количеством аргументов, используемых при вызове функции, и количеством параметров, заданных в определении функции.

Общая форма прототипа функции аналогична ее определению за исключением того, что в прототипе не представлено тело функции.

```
тип имя_функции(тип имя_параметра1, тип имя_параметра2, ...,
                  тип имя_параметраN);
```

Использование имен параметров в прототипе необязательно, но позволяет компилятору идентифицировать любое несовпадение типов при возникновении ошибки, поэтому лучше имена параметров все же включать в прототип функции.

Чтобы лучше понять полезность прототипов функций, рассмотрим следующую программу. Если вы попытаетесь ее скомпилировать, то получите от компилятора сообщение об ошибке, поскольку в этой программе делается попытка вызвать функцию `sqr_it()` с целочисленным аргументом, а не с указателем на целочисленное значение (согласно прототипу функции). Ошибка состоит в невозможности автоматического преобразования целочисленного значения в указатель.

```
/* Эта программа использует прототип функции для
   осуществления контроля типов.
*/
```

```

void sqr_it(int *i); // прототип функции

int main()
{
    int x;
    x = 10;
    // Благодаря прототипу компилятор выявит несоответствие
    // типов. Ведь функция sqr_it() должна принимать
    // указатель, а она вызвана с целочисленным аргументом.
    sqr_it(x); // Ошибка! Несоответствие типов!

    return 0;
}

void sqr_it(int *i)
{
    *i = *i * *i;
}

```

Определение функции может также служить в качестве своего прототипа, если оно размещено до первого ее использования в программе. Например, следующая программа абсолютно допустима.

```

// Использование определения функции в качестве ее прототипа.

#include <iostream>
using namespace std;

// Определяем, является ли заданное число четным.
bool isEven(int num) { // Поскольку функция isEven()
                      // определяется до ее использования,
                      // ее определение одновременно
                      // служит и ее прототипом.
    if(!(num % 2)) return true; // Значение num - четное число.
    return false;
}

int main()
{

```

## 250 Модуль 5. Введение в функции

```
if(isEven(4)) cout << "Число 4 четное.\n";
if(isEven(3)) cout << "Эта строка не будет выведена.";

return 0;
}
```

Здесь функция `isEven()` определяется до ее использования в функции `main()`. Таким образом, ее определение служит и ее прототипом, поэтому нет необходимости в отдельном включении прототипа в программу.

Обычно все же проще объявить прототип для всех функций, используемых в программе, вместо определения до их вызова. Особенно это касается больших программ, в которых трудно отследить последовательность вызова одних функций другими. Более того, возможны ситуации, когда две функции вызывают друг друга. В этом случае необходимо использовать именно прототипы.

### Стандартные заголовки содержат прототипы функций

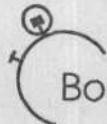
Ранее вы узнали о существовании стандартных заголовков C++, которые содержат информацию, необходимую для ваших программ. Несмотря на то что все выше сказанное — истинная правда, это еще *не вся* правда. Заголовки C++ содержат прототипы стандартных библиотечных функций, а также различные значения и определения, используемые этими функциями. Подобно функциям, создаваемым программистами, стандартные библиотечные функции также должны “заявить о себе” в форме прототипов до их использования. Поэтому любая программа, в которой используется библиотечная функция, должна включать заголовок, содержащий прототип этой функции.

Чтобы узнать, какой заголовок необходим для той или иной библиотечной функции, следует обратиться к справочному руководству, прилагаемому к вашему компилятору. Помимо описания каждой функции, там должно быть указано имя заголовка, который необходимо включить в программу для использования выбранной функции.

**ВАЖНО!**

### 5.12. Рекурсия

Рекурсия — это последняя тема, которую мы рассмотрим в этом модуле. *Рекурсия*, которую иногда называют *циклическим определением*, представляет собой процесс определения чего-либо на собственной основе. В области программирования под рекурсией понимается процесс вызова функцией самой себя. Функцию, которая вызывает саму себя, называют *рекурсивной*.



## Вопросы для текущего контроля

- Что представляет собой прототип функции? Каково его назначение?
- Все ли функции (за исключением `main()`) должны иметь прототипы?
- Почему необходимо включать в программу заголовок при использовании стандартной библиотечной функции?\*

Классическим примером рекурсии является вычисление факториала числа с помощью функции `factr()`. Факториал числа  $N$  представляет собой произведение всех целых чисел от 1 до  $N$ . Например, факториал числа 3 равен  $1 \times 2 \times 3$ , или 6. Рекурсивный способ вычисления факториала числа демонстрируется в следующей программе. Для сравнения сюда же включен и его нерекурсивный (итеративный) эквивалент.

```
// Демонстрация рекурсии.

#include <iostream>
using namespace std;

int factr(int n);
int fact(int n);

int main()
{
    // Использование рекурсивной версии.
    cout << "Факториал числа 4 равен " << factr(4);
    cout << '\n';

    // Использование итеративной версии.
    cout << "Факториал числа 4 равен " << fact(4);
    cout << '\n';
}
```

- Прототип объявляет имя функции, тип возвращаемого ею значения и параметры. Прототип сообщает компилятору, как генерировать код при обращении к функции, и гарантирует, что она будет вызвана корректно.
- Да, все функции, за исключением `main()`, должны иметь прототипы.
- Помимо прочего, заголовок включает прототип библиотечной функции.

## 252 Модуль 5. Введение в функции

```
    return 0;
}

// Рекурсивная версия.
int factr(int n)
{
    int answer;

    if(n==1) return(1);
    answer = factr(n-1)*n; // Выполнение рекурсивного вызова
                           // функции factr().

    return(answer);
}

// Итеративная версия.
int fact(int n)
{
    int t, answer;

    answer = 1;
    for(t=1; t<=n; t++) answer = answer*(t);
    return(answer);
}
```

Нерекурсивная версия функции `fact()` довольно проста и не требует расширенных пояснений. В ней используется цикл, в котором организовано перемножение последовательных чисел, начиная с 1 и заканчивая числом, заданным в качестве параметра: на каждой итерации цикла текущее значение управляющей переменной цикла умножается на текущее значение произведения, полученное в результате выполнения предыдущей итерации цикла.

Рекурсивная функция `factr()` несколько сложнее. Если она вызывается с аргументом, равным 1, то сразу возвращает значение 1. В противном случае она возвращает произведение `factr(n-1) * n`. Для вычисления этого выражения вызывается функция `factr()` с аргументом `n-1`. Этот процесс повторяется до тех пор, пока аргумент не станет равным 1, после чего вызванные ранее функции начнут возвращать значения. Например, при вычислении факториала числа 2 первое обращение к функции `factr()` приведет ко второму обращению к той же функции, но с аргументом, равным 1. Второй вызов функции `factr()` возвратит значение 1, которое будет умножено на 2 (исходное значение параметра `n`). Возможно, вам будет интересно вставить в функцию `factr()` инструкции `cout`, чтобы показать уровень каждого вызова и промежуточные результаты.

Когда функция вызывает сама себя, в системном стеке выделяется память для новых локальных переменных и параметров, и код функции с самого начала выполняется с этими новыми переменными. При возвращении каждого рекурсивного вызова из стека извлекаются старые локальные переменные и параметры, и выполнение функции возобновляется с "внутренней" точки ее вызова. О рекурсивных функциях можно сказать, что они "выдвигаются" и "задвигаются".

Следует иметь в виду, что в большинстве случаев использование рекурсивных функций не дает значительного сокращения объема кода. Кроме того, рекурсивные версии многих процедур выполняются медленнее, чем их итеративные эквиваленты, из-за дополнительных затрат системных ресурсов, связанных с многократными вызовами функций. Слишком большое количество рекурсивных обращений к функции может вызвать переполнение стека. Поскольку локальные переменные и параметры сохраняются в системном стеке и каждый новый вызов создает новую копию этих переменных, может настать момент, когда память стека будет исчерпана. В этом случае могут быть разрушены другие ("ни в чем не повинные") данные. Но если рекурсия построена корректно, об этом вряд ли стоит волноваться.

Основное достоинство рекурсии состоит в том, что некоторые типы алгоритмов рекурсивно реализуются проще, чем их итеративные эквиваленты. Например, алгоритм сортировки QuickSort довольно трудно реализовать итеративным способом. Кроме того, некоторые задачи (особенно те, которые связаны с искусственным интеллектом) просто созданы для рекурсивных решений.

При написании рекурсивной функции необходимо включить в нее инструкцию проверки условия (например, `if`-инструкцию), которая бы обеспечивала выход из функции без выполнения рекурсивного вызова. Если этого не сделать, то, вызвав однажды такую функцию, из нее уже нельзя будет вернуться. При работе с рекурсией это самый распространенный тип ошибки. Поэтому при разработке программ с рекурсивными функциями не стоит скучиться на инструкции `cout`, чтобы быть в курсе того, что происходит в конкретной функции, и иметь возможность прервать ее работу в случае обнаружения ошибки.

Рассмотрим еще один пример рекурсивной функции. Функция `reverse()` использует рекурсию для отображения своего строкового аргумента в обратном порядке.

```
// Отображение строки в обратном порядке с помощью рекурсии.

#include <iostream>
using namespace std;

void reverse(char *s);
int main()
```

```
{  
    char str[] = "Это тест";  
  
    reverse(str);  
  
    return 0;  
}  
  
// Вывод строки в обратном порядке.  
void reverse(char *s)  
{  
    if(*s)  
        reverse(s+1);  
    else  
        return;  
  
    cout << *s;  
}
```

Функция `reverse()` проверяет, не передан ли ей в качестве параметра указатель на нуль, которым завершается строка. Если нет, то функция `reverse()` вызывает саму себя с указателем на следующий символ в строке. Этот “закручивающийся” процесс повторяется до тех пор, пока той же функции не будет передан указатель на нуль. Когда, наконец, обнаружится символ конца строки, пойдет процесс “раскручивания”, т.е. вызванные ранее функции начнут возвращать значения, и каждый возврат будет сопровождаться “довыполнением” метода, т.е. отображением символа `s`. В результате исходная строка посимвольно отобразится в обратном порядке.

И еще. Создание рекурсивных функций часто вызывает трудности у начинающих программистов. Но с приходом опыта использование рекурсии становится для многих обычной практикой.

## Проект 5.1. Алгоритм “быстрой” сортировки QuickSort

QSDemo.cpp

В модуле 4 был рассмотрен простой метод пузырьковой сортировки, а также упоминалось о существовании гораздо более эффективных методов сортировки. Здесь же мы запрограммируем версию одного из лучших алгоритмов, известного под названием QuickSort. Метод

QuickSort, изобретенный С.А.Р. Hoare, — лучший из известных ныне алгоритмов сортировки общего назначения. Мы не рассматривали этот алгоритм в модуле 4, поскольку самая эффективная его реализация опирается на использование рекурсии. Версия, которую мы будем разрабатывать, предназначена для сортировки символьного массива, но ее логику вполне можно адаптировать под сортировку объектов любого типа.

Алгоритм QuickSort основан на идее разделений. Общая процедура состоит в выборе значения, именуемого *компарапондом* (операнд в операции сравнения), с последующим разделением массива на две части. Все элементы, которые больше компаранда или равны ему, помещаем в одну часть, остальные (которые меньше компаранда) — в другую. Этот процесс затем повторяется для каждой части до тех пор, пока массив не станет полностью отсортированным. Например, возьмем массив `fedabc` и используем значение `d` в качестве компаранда. Тогда в результате первого прохода алгоритма QuickSort наш массив будет реорганизован следующим образом.

|                 |                      |
|-----------------|----------------------|
| Исходный массив | <code>fedabc</code>  |
| Проход 1        | <code>bcaodef</code> |

Этот процесс затем повторяется отдельно для частей `bca` и `def`. Как видите, этот процесс по своей природе рекурсивен, поэтому его проще всего реализовать в виде рекурсивной функции.

Значение компаранда можно выбрать двумя способами: либо случайным образом, либо путем усреднения небольшого набора значений, взятых из исходного массива. Для получения оптимального результата сортировки следует взять значение, расположенное в середине массива. Но для большинства наборов данных это сделать нелегко. В самом худшем случае выбранное вами значение будет находиться на одном из концов массива. Но даже в этом случае алгоритм QuickSort выполнится корректно. В нашей версии в качестве компаранда мы выбираем средний элемент массива.

Следует отметить, что в библиотеке C++ содержится функция `qsort()`, которая также выполняет сортировку методом QuickSort. Вам было бы интересно сравнить ее с версией, представленной в этом проекте.

## Последовательность действий

1. Создайте файл `QSDemo.cpp`.
2. Алгоритм QuickSort реализуется здесь в виде двух функций. Одна из них, `quicksort()`, обеспечивает удобный интерфейс для пользователя и подготавливает обращение к другой, `qs()`, которая, по сути, и выполняет сортировку. Сначала создадим функцию `quicksort()`.

## 256 Модуль 5. Введение в функции

```
// Функция обеспечивает вызов реальной функции сортировки.  
void quicksort(char *items, int len)  
{  
    qs(items, 0, len-1);  
}
```

Здесь параметр `items` указывает на массив, подлежащий сортировке, а параметр `len` задает количество элементов в массиве. Как показано в следующем действии (см. пункт 3), функция `qs()` принимает область массива, предлагаемую ей для сортировки функцией `quicksort()`. Преимущество использования функции `quicksort()` состоит в том, что ее можно вызвать, передав ей в качестве параметров лишь указатель на сортируемый массив и его длину. Из этих параметров затем формируются начальный и конечный индексы сортируемой области.

3. Добавим в программу функцию реальной сортировки массива `qs()`.

```
// Рекурсивная версия алгоритма Quicksort для сортировки  
// символов.  
void qs(char *items, int left, int right)  
{  
    int i, j;  
    char x, y;  
  
    i = left; j = right;  
    x = items[(left+right) / 2];  
  
    do {  
        while((items[i] < x) && (i < right)) i++;  
        while((k < items[j]) && (j > left)) j--;  
  
        if(i <= j) {  
            y = items[i];  
            items[i] = items[j];  
            items[j] = y;  
            i++; j--;  
        }  
    } while(i <= j);  
  
    if(left < j) qs(items, left, j);  
    if(i < right) qs(items, i, right);  
}
```

Эту функцию необходимо вызывать, передавая ей в качестве параметров индексы сортируемой области. Параметр `left` должен содержать начало (левую границу) области, а параметр `right` — ее конец (правую границу). При первом вызове сортируемая область представляет собой весь массив. С каждым рекурсивным вызовом сортируется все меньшая область.

4. Для использования алгоритма Quicksort достаточно вызвать функцию `quicksort()`, передав ей имя сортируемого массива и его длину. По завершении этой функции заданный массив будет отсортирован. Помните, эта версия подходит только для символьных массивов, но ее логику можно адаптировать для сортировки массивов любого типа.
5. Приведем полную программу реализации алгоритма Quicksort.

```
/*
Проект 5.1.
Версия реализации алгоритма Quicksort для сортировки
символов.

*/
#include <iostream>
#include <cstring>

using namespace std;

void quicksort(char *items, int len);

void qs(char *items, int left, int right);

int main() {

    char str[] = "jfmckldoelazlkper";

    cout << "Массив в исходном порядке: " << str << "\n";

    quicksort(str, strlen(str));

    cout << "Отсортированный массив: " << str << "\n";

    return 0;
}
```

## 258 Модуль 5. Введение в функции

```
}

// Функция обеспечивает вызов реальной функции сортировки.
void quicksort(char *items, int len)
{
    qs(items, 0, len-1);
}

// Рекурсивная версия алгоритма Quicksort для сортировки
// символов.
void qs(char *items, int left, int right)
{
    int i, j;
    char x, y;

    i = left; j = right;
    x = items[(left+right) / 2 ];

    do {
        while((items[i] < x) && (i < right)) i++;
        while((x < items[j]) && (j > left)) j--;

        if(i <= j) {
            y = items[i];
            items[i] = items[j];
            items[j] = y;
            i++; j--;
        }
    } while(i <= j);

    if(left < j) qs(items, left, j);
    if(i < right) qs(items, i, right);
}
```

При выполнении эта программа генерирует такие результаты.

Массив в исходном порядке: jfmckldoelazlkper

Отсортированный массив: acdeefjkklllmoprz

## Спросим у опытного программиста

**Вопрос.** Я слышал, что существует правило “по умолчанию тип int”. В чем оно состоит и применяется ли к C++?

**Ответ.** В языке С (да и в ранних версиях C++) , если в каком-нибудь объявлении не был указан спецификатор типа, то подразумевался тип `int`. Например, в С-коде (или в старом C++-коде) следующая функция была бы допустима и возвращала результат типа `int`.

```
f() { // По умолчанию для значения,
       // возвращаемого функцией, подразумевается
       // тип int.
{
    int x;
    // ...
    return x;
}
```

Здесь тип значения, возвращаемого функцией `f()`, принимается равным `int`, поскольку никакой другой тип не задан. Однако правило “по умолчанию тип `int`” (или правило “неявного типа `int`”) не поддерживается современными версиями C++. И хотя большинство компиляторов должно поддерживать это правило ради обратной совместимости, вам следует явно задавать тип значений, возвращаемых всеми функциями, которые вы определяете. При модернизации старого кода это обстоятельство также следует иметь в виду.



## Тест для самоконтроля по модулю 5

- Представьте общий формат функции.
- Создайте функцию с именем `hypot()`, которая вычисляет длину гипотенузы прямоугольного треугольника, заданного длинами двух противоположных сторон. Продемонстрируйте применение этой функции в программе. Для решения этой задачи воспользуйтесь стандартной библиотечной функцией, которая возвращает значение квадратного корня из аргумента. Вот ее прототип.

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

Использование функции `sqrt()` требует включения в программу заголовка `<cmath>`.

3. Может ли функция возвращать указатель? Может ли функция возвращать массив?
4. Создайте собственную версию стандартной библиотечной функции `strlen()`. Назовите свою версию `mystrlen()` и продемонстрируйте ее применение в программе.
5. Поддерживает ли локальная переменная свое значение между вызовами функции, в которой она объявлена?
6. Назовите одно достоинство и один недостаток глобальных переменных.
7. Создайте функцию с именем `byThrees()`, которая возвращает ряд чисел, в котором каждое следующее значение на 3 больше предыдущего. Пусть этот ряд начинается с числа 0. Таким образом, первыми пятью членами ряда, которые возвращает функция `byThrees()`, должны быть числа 0, 3, 6, 9 и 12. Создайте еще одну функцию с именем `reset()`, которая заставит функцию `byThrees()` снова начинать генерируемый ею ряд чисел с нуля. Продемонстрируйте применение этих функций в программе. Подсказка: здесь нужно использовать глобальную переменную.
8. Напишите программу, которая запрашивает пароль, задаваемый в командной строке. Ваша программа не должна реально выполнять какие-либо действия, за исключением выдачи сообщения о том, корректно ли был введен пароль или нет.
9. Прототип функции предотвращает ее вызов с неверно заданным количеством аргументов. Так ли это?
10. Напишите рекурсивную функцию, которая выводит числа от 1 до 10. Продемонстрируйте ее применение в программе.

На заметку 

Ответы на эти вопросы можно найти на Web-странице данной книги по адресу: <http://www.osborne.com>.

# Модуль 6

## О функциях подробнее

- 6.1. Два способа передачи аргументов
- 6.2. Как в C++ реализована передача аргументов
- 6.3. Использование указателя для обеспечения вызова по ссылке
- 6.4. Ссылочные параметры
- 6.5. Возврат ссылок
- 6.6. Независимые ссылки
- 6.7. Перегрузка функций
- 6.8. Аргументы, передаваемые функции по умолчанию
- 6.9. Перегрузка функций и неоднозначность

В этом модуле мы продолжим изучение функций, рассматривая три самые важные темы, связанные с функциями C++: ссылки, перегрузка функций и использование аргументов по умолчанию. Эти три средства в значительной степени расширяют возможности функций. Как будет показано ниже, ссылка — это неявный указатель. Перегрузка функций представляет собой средство, которое позволяет одну функцию реализовать несколькими способами, причем в каждом случае возможно выполнение отдельной задачи. Поэтому есть все основания считать перегрузку функций одним из способов поддержки полиморфизма в C++. Используя возможность задания аргументов по умолчанию, можно определить значение для параметра, которое будет автоматически применено в случае, если соответствующий аргумент не задан.

Начнем с рассмотрения двух способов передачи аргументов функциям и результатов их использования. Это важно для понимания назначения ссылок, поскольку их применение к параметрам функций — основная причина их существования.

**ВАЖНО!**

## **6.1. Два способа передачи аргументов**

В языках программирования, как правило, предусматривается два способа, которые позволяют передавать аргументы в подпрограммы (функции, методы, процедуры). Первый называется *вызовом по значению* (call-by-value). В этом случае значение аргумента копируется в формальный параметр подпрограммы. Следовательно, изменения, внесенные в параметры подпрограммы, не влияют на аргументы, используемые при ее вызове.

Второй способ передачи аргумента подпрограмме называется *вызовом по ссылке* (call-by-reference). В этом случае в параметр копируется *адрес* аргумента (а не его значение). В пределах вызываемой подпрограммы этот адрес используется для доступа к реальному аргументу, заданному при ее вызове. Это значит, что изменения, внесенные в параметр, окажут воздействие на аргумент, используемый при вызове подпрограммы.

**ВАЖНО!**

## **6.2. Как в C++ реализована передача аргументов**

По умолчанию для передачи аргументов в C++ используется *метод вызова по значению*. Это означает, что в общем случае код функции не может изменить аргументы, используемые при вызове функции. Во всех программах этой книги,

представленных до сих пор, использовался метод вызова по значению. Рассмотрим, например, функцию `reciprocal()` в следующей программе.

```
// Изменение параметра при вызове по функции по значению
// не влияет на аргумент.
```

```
#include <iostream>
using namespace std;

double reciprocal(double x);

int main()
{
    double t = 10.0;

    cout << "Обратная величина от числа 10.0 равна "
        << reciprocal(t) << "\n";

    cout << "Значение переменной t по-прежнему равно: "
        << t << "\n";

    return 0;
}

// Функция возвращает обратную величину от
// заданного значения.
double reciprocal(double x)
{
    x = 1 / x; // Вычисляем обратную величину. (Это действие
                // никак не отражается на значении переменной t
                // из функции main().)

    return x;
}
```

При выполнении эта программа генерирует такие результаты.

Обратная величина от числа 10.0 равна 0.1

Значение переменной t по-прежнему равно: 10

Как подтверждают результаты выполнения этой программы, при выполнении присваивания

```
x = 1 / x;
```

в функции `reciprocal()` модифицируется только переменная `x`. Переменная `t`, используемая в качестве аргумента, по-прежнему содержит значение 10 и не подвержена влиянию действий, выполняемых в функции `reciprocal()`.

**ВАЖНО!**

### **6.3. Использование указателя для обеспечения вызова по ссылке**

Несмотря на то что в качестве C++-соглашения о передаче параметров по умолчанию действует вызов по значению, существует возможность “вручную” заменить его вызовом по ссылке. В этом случае функции будет передаваться адрес аргумента (т.е. указатель на аргумент). Это позволит внутреннему коду функции изменить значение аргумента, которое хранится вне функции. Пример такого “дистанционного” управления значениями переменных представлен в предыдущем модуле при рассмотрении возможности вызова функции с указателями. Как вы знаете, указатели передаются функциям подобно значениям любого другого типа. Безусловно, для этого необходимо объявить параметры с типом указателей.

Чтобы понять, как передача указателя позволяет вручную обеспечить вызов по ссылке, рассмотрим функцию `swap()`. (Она меняет значения двух переменных, на которые указывают ее аргументы.) Вот как выглядит один из способов ее реализации.

```
// Обмен значениями двух переменных, адресуемых
// параметрами x и y.
void swap(int *x, int *y)
{
    int temp;

    temp = *x; // Временно сохраняем значение,
               // расположенное по адресу x.
    *x = *y;   // Помещаем значение, хранимое по адресу y,
               // по адресу x.
    *y = temp; // Помещаем значение, которое раньше
               // хранилось по адресу x, по адресу y.
}
```

Функция `swap()` объявляет два параметра-указателя (`x` и `y`). Она использует эти параметры для обмена значений переменных, адресуемых аргумента-

ми (переданными функции). Не забывайте, что выражения `*x` и `*y` обозначают переменные, адресуемые указателями `x` и `y` соответственно. Таким образом, при выполнении инструкции

```
*x = *y;
```

значение объекта, адресуемого указателем `y`, помещается в объект, адресуемый указателем `x`. Следовательно, по завершении этой функции содержимое переменных, используемых при ее вызове, “поменяется местами”.

Поскольку функция `swap()` ожидает получить два указателя, вы должны помнить, что функцию `swap()` необходимо вызывать с *адресами* переменных, значения которых вы хотите обменять. Корректный вызов этой функции продемонстрирован в следующей программе.

```
// Демонстрируется версия функции swap() с использованием
// указателей.
```

```
#include <iostream>
using namespace std;

// Объявляем функцию swap(), которая использует указатели.
void swap(int *x, int *y);

int main()
{
    int i, j;

    i = 10;
    j = 20;

    cout << "Исходные значения переменных i и j: ";
    cout << i << ' ' << j << '\n';

    swap(&j, &i); // Вызываем swap() с адресами
                  // переменных i и j.

    cout << "Значения переменных i и j после обмена: ";
    cout << i << ' ' << j << '\n';

    return 0;
}
```

## 266 Модуль 6. О функциях подробнее

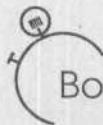
```
// Обмен значениями двух переменных, адресуемых
// параметрами x и y.
void swap(int *x, int *y)
{
    int temp;

    // Меняем значения переменных, адресуемых
    // параметрами x и y, используя явно заданные
    // операции с указателями.
    temp = *x; // Временно сохраняем значение,
                // расположенное по адресу x.
    *x = *y; // Помещаем значение, хранимое по адресу y,
              // по адресу x.
    *y = temp; // Помещаем значение, которое раньше
                // хранилось по адресу x, по адресу y.
}
```

В функции `main()` переменной `i` было присвоено начальное значение 10, а переменной `j` – 20. Затем была вызвана функция `swap()` с адресами переменных `i` и `j`. Для получения адресов здесь используется унарный оператор “`&`”. Следовательно, функции `swap()` при вызове были переданы адреса переменных `i` и `j`, а не их значения. После выполнения функции `swap()` переменные `i` и `j` обменялись своими значениями, что подтверждается результатами выполнения этой программы.

Исходные значения переменных `i` и `j`: 10 20

Значения переменных `i` и `j` после обмена: 20 10



### Вопросы для текущего контроля

- Поясните суть передачи функции параметров по значению.
- Поясните суть передачи функции параметров по ссылке.
- Какой механизм передачи параметров используется в C++ по умолчанию?\*

- 
- При передаче функции параметров по значению (т.е. при вызове по значению) значения аргументов копируются в параметры подпрограммы.
  - При передаче функции параметров по ссылке (т.е. при вызове по ссылке) в параметры подпрограммы копируются адреса аргументов.
  - По умолчанию в C++ используется вызов по значению.

ВАЖНО!

## 6.4. Ссылочные параметры

Несмотря на возможность “вручную” организовать вызов по ссылке с помощью оператора получения адреса, такой подход не всегда удобен. Во-первых, он вынуждает программиста выполнять все операции с использованием указателей. Во-вторых, вызывая функцию, программист должен не забыть передать ей адреса аргументов, а не их значения. К счастью, в C++ можно сориентировать компилятор на автоматическое использование вызова по ссылке (вместо вызова по значению) для одного или нескольких параметров конкретной функции. Такая возможность реализуется с помощью *ссылочного параметра* (reference parameter). При использовании ссылочного параметра функции автоматически передается адрес (а не значение) аргумента. При выполнении кода функции (при выполнении операций над ссылочным параметром) обеспечивается его автоматическое разыменование, и поэтому программисту не нужно прибегать к операторам, используемым с указателями.

Ссылочный параметр объявляется с помощью символа “&”, который должен предшествовать имени параметра в объявлении функции. Операции, выполняемые над ссылочным параметром, оказывают влияние на аргумент, используемый при вызове функции, а не на сам ссылочный параметр.

Чтобы лучше понять механизм действия ссылочных параметров, рассмотрим для начала простой пример. В следующей программе функция `f()` принимает один ссылочный параметр типа `int`.

```
// Использование ссылочного параметра.
```

```
#include <iostream>
using namespace std;

void f(int &i); // Здесь i объявляется как ссылочный параметр.

int main()
{
    int val = 1;
    cout << "Старое значение переменной val: " << val
        << '\n';
    f(val); // Передаем адрес переменной val функции f().
    cout << "Новое значение переменной val: " << val
        << '\n';
    return 0;
}
```

```
void f(int &i)
{
    i = 10; // Модификация аргумента, заданного при вызове.
}
```

При выполнении этой программы получаем такой результат.

Старое значение переменной val: 1

Новое значение переменной val: 10

Обратите особое внимание на определение функции f().

```
void f(int &i)
{
    i = 10; // Модификация аргумента, заданного при вызове.
}
```

Итак, рассмотрим объявление параметра *i*. Его имени предшествует символ "&", который "превращает" переменную *i* в ссылочный параметр. (Это объявление также используется в прототипе функции.) Инструкция

*i* = 10;

(в данном случае она одна составляет тело функции) *не* присваивает переменной *i* значение 10. В действительности значение 10 присваивается переменной, на которую ссылается переменная *i* (в нашей программе ею является переменная *val*). Обратите внимание на то, что в этой инструкции нет оператора "\*", который используется при работе с указателями. Применяя ссылочный параметр, вы тем самым уведомляете C++-компилятор о передаче адреса (т.е. указателя), и компилятор автоматически разыменовывает его за вас. Более того, если бы вы попытались "помочь" компилятору, использовав оператор "\*", то сразу же получили бы сообщение об ошибке.

Поскольку переменная *i* была объявлена как ссылочный параметр, компилятор автоматически передает функции *f()* *адрес* аргумента, с которым вызывается эта функция. Таким образом, в функции *main()* инструкция

*f(val); // Передаем адрес переменной val функции f().*

передает функции *f()* адрес переменной *val* (а не ее значение). Обратите внимание на то, что при вызове функции *f()* не нужно предварять переменную *val* оператором "&". (Более того, это было бы ошибкой.) Поскольку функция *f()* получает адрес переменной *val* в форме ссылки, она может модифицировать значение этой переменной.

Чтобы проиллюстрировать реальное применение ссылочных параметров (и тем самым продемонстрировать их достоинства), перепишем нашу старую знакомую функцию *swap()* с использованием ссылок. В следующей программе обратите внимание на то, как функция *swap()* объявляется и вызывается.

```

// Использование ссылочных параметров в функции swap().

#include <iostream>
using namespace std;

// Объявляем функцию swap() с использованием ссылочных
// параметров.
void swap(int &x, int &y);

int main()
{
    int i, j;

    i = 10;
    j = 20;

    cout << " Исходные значения переменных i и j: ";
    cout << i << ' ' << j << '\n';

    swap(j, i); // Здесь функции swap() автоматически
                 // передаются адреса переменных i и j.

    cout << " Значения переменных i и j после обмена: ";
    cout << i << ' ' << j << '\n';

    return 0;
}

/* Здесь функция swap() определяется с использованием
вызыва по ссылке, а не вызова по значению. Поэтому она
может выполнить обмен значениями двух аргументов, с
которыми она вызывается.

*/
void swap(int &x, int &y)
{
    int temp;

    // Для обмена значениями аргументов используем
    // ссылки на них.
    temp = x;
    x = y;      // Теперь обмен значениями происходит
                // автоматически посредством ссылок.
    y = temp;
}

```

Результаты выполнения этой программы совпадают с результатами запуска предыдущей версии. Опять таки, обратите внимание на то, что объявление `x` и `y` ссылочными параметрами избавляет вас от необходимости использовать оператор `*` при организации обмена значениями. Как уже упоминалось, такая "навязчивость" с вашей стороны стала бы причиной ошибки. Поэтому запомните, что компилятор автоматически генерирует адреса аргументов, используемых при вызове функции `swap()`, и автоматически разыменовывает ссылки `x` и `y`.

Итак, подведем некоторые итоги. После создания ссылочный параметр автоматически ссылается (т.е. неявно указывает) на аргумент, используемый при вызове функции. Более того, при вызове функции не нужно применять к аргументу оператор `&`. Кроме того, в теле функции ссылочный параметр используется непосредственно, т.е. без оператора `*`. Все операции, включающие ссылочный параметр, автоматически выполняются над аргументом, используемым при вызове функции. Наконец, присваивая некоторое значение параметру-ссылке, вы в действительности присваиваете это значение переменной, на которую указывает эта ссылка. Поэтому, применяя ссылку в качестве параметра функции, при вызове функции вы в действительности используете переменную.

И еще одно замечание. В языке С ссылки не поддерживаются. Поэтому единственный способ организации в С вызова по ссылке — использовать указатели, как показано в первой версии функции `swap()`. При переводе С-программы в C++ код необходимо преобразовать эти типы параметров в ссылки, где это возможно.



### Вопросы для текущего контроля

1. Как объявить параметр-ссылку?
2. Нужно ли предварять аргумент символом `&` при вызове функции, которая принимает ссылочный параметр?
3. Нужно ли предварять параметр символами `*` или `&` при выполнении операций над ними в функции, которая принимает ссылочный параметр?\*

1. Для объявления ссылочного параметра необходимо предварить его имя символом `&`.
2. Нет, при использовании ссылочного параметра аргумент автоматически передается по ссылке.
3. Нет, параметр обрабатывается обычным способом. Но изменения, которым подвергается параметр, влияют на аргумент, используемый при вызове функции.

## Спросим у опытного программиста

**Вопрос.** В некоторых C++-программах мне приходилось встречать стиль объявления ссылок, в соответствии с которым оператор "&" связывается с именем типа (`int& i;`), а не с именем переменной (`int &i;`). Есть ли разница между этими двумя объявлениями?

**Ответ.** Если кратко ответить, то никакой разницы нет. Например, вот как выглядит еще один способ записи прототипа функции `swap()`.

```
void swap(int& x, int& y);
```

Нетрудно заметить, что в этом объявлении символ "&" прилегает вплотную к имени типа `int`, а не к имени переменной `x`. Более того, некоторые программисты определяют в таком стиле и указатели, без пробела связывая символ "\*" с типом, а не с переменной, как в этом примере.

```
float* p;
```

Приведенные объявления отражают желание некоторых программистов иметь в C++ отдельный тип ссылки или указателя. Но дело в том, что подобное связывание символа "&" или "\*" с типом (а не с переменной) не распространяется, в соответствии с формальным синтаксисом C++, на весь список переменных, приведенных в объявлении, что может вызвать путаницу. Например, в следующем объявлении создается один указатель (а не два) на целочисленную переменную.

```
int* a, b;
```

Здесь `b` объявляется как целочисленная переменная (а не как указатель на целочисленную переменную), поскольку, как определено синтаксисом C++, используемый в объявлении символ "\*" или "&" связывается с конкретной переменной, которой он предшествует, а не с типом, за которым он следует.

Важно понимать, что для C++-компилятора абсолютно безразлично, как именно вы напишете объявление: `int *p` или `int* p`. Таким образом, если вы предпочитаете связывать символ "\*" или "&" с типом, а не переменной, поступайте, как вам удобно. Но, чтобы избежать в дальнейшем каких-либо недоразумений, в этой книге мы будем связывать символ "\*" или "&" с именем переменной, а не с именем типа.

### ВАЖНО!

## 6.5. Возврат ссылок

Функция может возвращать ссылку. В программировании на C++ предусмотрено несколько применений для ссылочных значений, возвращаемых функция-

ми. Сейчас мы продемонстрируем только некоторые из них, а другие рассмотрим ниже в этой книге, когда познакомимся с перегрузкой операторов.

Если функция возвращает ссылку, это означает, что она возвращает неявный указатель на значение, передаваемое ею в инструкции `return`. Этот факт открывает поразительные возможности: функцию, оказывается, можно использовать в левой части инструкции присваивания! Например, рассмотрим следующую простую программу.

```
// Возврат ссылки.

#include <iostream>
using namespace std;

double &f(); // ← Эта функция возвращает ссылку
             // на double-значение.
double val = 100.0;

int main()
{
    double x;

    cout << f() << '\n';      // Отображаем значение val.

    x = f(); // Присваиваем значение val переменной x.
    cout << x << '\n'; // Отображаем значение переменной x.

    f() = 99.1;           // Изменяем значение глобальной
                          // переменной val.
    cout << f() << '\n'; // Отображаем новое значение val.

    return 0;
}

// Эта функция возвращает ссылку на double-значение.
double &f()
{
    return val; // Возвращаем ссылку на глобальную
                // переменную val.
}
```

Вот как выглядят результаты выполнения этой программы.

```
100
100
99.1
```

Рассмотрим эту программу подробнее. Судя по прототипу функции `f()`, она должна возвращать ссылку на `double`-значение. За объявлением функции `f()` следует объявление глобальной переменной `val`, которая инициализируется значением 100. При выполнении следующей инструкции в функции `main()` выводится исходное значение переменной `val`.

```
cout << f() << '\n'; // Отображаем значение val.
```

После вызова функция `f()` при выполнении инструкции `return` возвращает ссылку на переменную `val`.

```
return val; // Возвращаем ссылку на val.
```

Таким образом, при выполнении приведенной выше строки автоматически возвращается ссылка на глобальную переменную `val`. Эта ссылка затем используется инструкцией `cout` для отображения значения `val`.

При выполнении строки

```
x = f(); // Присваиваем значение val переменной x.
```

ссылка на переменную `val`, возвращенная функцией `f()`, используется для присвоения значения `val` переменной `x`.

А вот самая интересная строка в программе.

```
f() = 99.1; // Изменяем значение глобальной переменной val.
```

При выполнении этой инструкции присваивания значение переменной `val` становится равным числу 99.1. И вот почему: поскольку функция `f()` возвращает ссылку на переменную `val`, эта ссылка и является приемником инструкции присваивания. Таким образом, значение 99.1 присваивается переменной `val` косвенно, через ссылку (на нее), которую возвращает функция `f()`.

Приведем еще один пример программы, в которой в качестве значения, возвращаемого функцией, используется ссылка (или значение ссылочного типа).

```
// Пример функции, возвращающей ссылку на элемент массива.
```

```
#include <iostream>
```

```
using namespace std;
```

```
double &change_it(int i); // Функция возвращает ссылку.
```

```
double vals[] = {1.1, 2.2, 3.3, 4.4, 5.5};
```

## 274 Модуль 6. О функциях подробнее

```
int main()
{
    int i;

    cout << "Вот исходные значения: ";
    for(i=0; i<5; i++)
        cout << vals[i] << ' ';
    cout << '\n';

    change_it(1) = 5298.23; // Изменяем 2-й элемент.
    change_it(3) = -98.8;   // Изменяем 4-й элемент.

    cout << "Вот измененные значения: ";
    for(i=0; i<5; i++)
        cout << vals[i] << ' ';
    cout << '\n';

    return 0;
}

double &change_it(int i)
{
    return vals[i]; // Возвращаем ссылку на i-й элемент.
}
```

Эта программа изменяет значения второго и четвертого элементов массива `vals`. Результаты ее выполнения таковы.

Вот исходные значения: 1.1 2.2 3.3 4.4 5.5

Вот измененные значения: 1.1 5298.23 3.3 -98.8 5.5

Давайте разберемся, как они были получены.

Функция `change_it()` объявлена как возвращающая ссылку на значение типа `double`. Говоря более конкретно, она возвращает ссылку на элемент массива `vals`, который задан ей в качестве параметра `i`. Таким образом, при выполнении следующей инструкции функции `main()`

`change_it(1) = 5298.23; // Изменяем 2-й элемент массива.`

элементу `vals[1]` присваивается значение 5298.23. Поскольку функция `change_it()` возвращает ссылку на конкретный элемент массива `vals`, ее можно использовать в левой части инструкции для присвоения нового значения соответствующему элементу массива.

Однако, организуя возврат функцией ссылки, необходимо позаботиться о том, чтобы объект, на который она ссылается, не выходил за пределы действующей области видимости. Например, рассмотрим такую функцию.

```
// Здесь ошибка: нельзя возвращать ссылку
// на локальную переменную.
int &f()
{
    int i=10;
    return i;
}
```

При завершении функции `f()` локальная переменная `i` выйдет за пределы области видимости. Следовательно, ссылка на переменную `i`, возвращаемая функцией `f()`, будет неопределенной. В действительности некоторые компиляторы не скомпилируют функцию `f()` в таком виде, и именно по этой причине. Однако проблема такого рода может быть создана опосредованно, поэтому нужно внимательно отнестись к тому, на какой объект будет возвращать ссылку ваша функция.

#### ВАЖНО!

## 6. Независимые ссылки

Понятие ссылки включено в C++ главным образом для поддержки способа передачи параметров “по ссылке” и для использования в качестве ссылочного типа значения, возвращаемого функцией. Несмотря на это, можно объявить независимую переменную ссылочного типа, которая и называется *независимой ссылкой*. Однако справедливости ради необходимо сказать, что эти независимые ссылочные переменные используются довольно редко, поскольку они могут “сбить с пути истинного” вашу программу. Сделав (для очистки совести) эти замечания, мы все же можем уделить независимым ссылкам некоторое внимание.

Независимая ссылка должна указывать на некоторый объект. Следовательно, независимая ссылка должна быть инициализирована при ее объявлении. В общем случае это означает, что ей будет присвоен адрес некоторой ранее объявленной переменной. После этого имя такой ссылочной переменной можно применять везде, где может быть использована переменная, на которую она ссылается. И в самом деле, между ссылкой и переменной, на которую она ссылается, практически нет никакой разницы. Рассмотрим, например, следующую программу.

```
// Использование независимой ссылки.
```

```
#include <iostream>
using namespace std;
```

## 276 Модуль 6. О функциях подробнее

```
int main()
{
    int j, k;
    int &i = j; // независимая ссылка

    j = 10;

    cout << j << " " << i; // Выводится: 10 10

    k = 121;
    i = k; // Копирует в переменную j значение
           // переменной k, а не адрес переменной k.

    cout << "\n" << j; // Выводится: 121

    return 0;
}
```

При выполнении эта программа выводит следующие результаты.

```
10 10
121
```

Адрес, который содержит ссылочная переменная, фиксирован и его изменить нельзя. Следовательно, при выполнении инструкции

```
i = k;
```

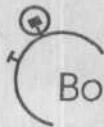
в переменную j (адресуемую ссылкой i) копируется значение переменной k, а не ее адрес.

Как было отмечено выше, независимые ссылки лучше не использовать, поскольку чаще всего им можно найти замену, а их неаккуратное применение может исказить ваш код. Согласитесь: наличие двух имен для одной и той же переменной, по сути, уже создает ситуацию, потенциально порождающую недоразумения.

### Ограничения при использовании ссылок

На применение ссылочных переменных налагается ряд следующих ограничений.

- Нельзя ссылаться на ссылочную переменную.
- Нельзя создавать массивы ссылок.
- Нельзя создавать указатель на ссылку, т.е. нельзя к ссылке применять оператор "&".



## Вопросы для текущего контроля

1. Может ли функция возвращать ссылку?
2. Что представляет собой независимая ссылка?
3. Можно ли создать ссылку на ссылку?\*

**ВАЖНО!**

## 6.7. Перегрузка функций

В этом разделе мы узнаем об одной из самых удивительных возможностей языка C++ — перегрузке функций. В C++ несколько функций могут иметь одинаковые имена, но при условии, что их параметры будут различными. Такую ситуацию называют *перегрузкой функций* (function overloading), а функции, которые в ней задействованы, — *перегруженными* (overloaded). Перегрузка функций — один из способов реализации полиморфизма в C++.

Чтобы перегрузить функцию, достаточно объявить различные ее версии. Об остальном же позаботится компилятор. При этом важно помнить, что тип и/или количество параметров каждой перегруженной функции должны быть уникальными, поскольку именно эти факторы позволяют компилятору определить, какую версию перегруженной функции следует вызвать. Таким образом, перегруженные функции должны отличаться типами и/или числом параметров. Несмотря на то что перегруженные функции *могут* отличаться и типами возвращаемых значений, этого вида информации недостаточно для C++, чтобы во всех случаях компилятор мог решить, какую именно функцию нужно вызвать.

Рассмотрим простой пример перегрузки функций.

```
// "Трехкратная" перегрузка функции f().
```

```
#include <iostream>
using namespace std;

// Для каждой версии перегруженной функции необходимо
// включить в программу отдельный прототип.
```

1. Да, функция может возвращать ссылку.
2. Независимая ссылка — эквивалент, по сути, еще одно имя для объекта.
3. Нет, ссылку на ссылку создать нельзя.

## 278 Модуль 6. О функциях подробнее

```
void f(int i);           // один целочисленный параметр
void f(int i, int j);   // два целочисленных параметра
void f(double k);       // один параметр типа double

int main()
{
    f(10);      // вызов функции f(int)

    f(10, 20); // вызов функции f(int, int)

    f(12.23);  // вызов функции f(double)

    return 0;
}

// Первая реализация функции f().
void f(int i)
{
    cout << "В функции f(int), i равно " << i << '\n';
}

// Вторая реализация функции f().
void f(int i, int j)
{
    cout << "В функции f(int, int), i равно " << i;
    cout << ", j равно " << j << '\n';
}

// Третья реализация функции f().
void f(double k)
{
    cout << "В функции f(double), k равно " << k << '\n';
}
```

При выполнении эта программа генерирует следующие результаты.

В функции f(int), i равно 10

В функции f(int, int), i равно 10, j равно 20

В функции f(double), k равно 12.23

Как видите, функция `f()` перегружается три раза. Первая версия принимает один целочисленный параметр, вторая — два целочисленных параметра, а третья — один `double`-параметр. Поскольку списки параметров для всех трех версий различны, компилятор обладает достаточной информацией, чтобы вызывать правильную версию каждой функции.

Чтобы лучше понять выигрыш от перегрузки функций, рассмотрим функцию `neg()`, которая возвращает результат отрицания (умножения на  $-1$ ) своего аргумента. Например, если функцию `neg()` вызвать с аргументом  $-10$ , то она возвратит число  $10$ . А если при вызове ей передать число  $9$ , она возвратит число  $-9$ . Если бы вам понадобились функции (и при этом вы не имели возможности их перегружать) отрицания для данных типа `int`, `double` и `long`, то вам бы пришлось создавать три различные функции с разными именами, например, `ineg()`, `dneg()` и `lneg()`. Но благодаря перегрузке функций можно обойтись только одним именем (например, `neg()`), которое подойдет для всех функций, возвращающих результат отрицания аргумента. Таким образом, перегрузка функций поддерживает принцип полиморфизма “один интерфейс — множество методов”. Этот принцип демонстрируется на примере следующей программы.

```
// Создание различных версий функции neg().

#include <iostream>
using namespace std;

int neg(int n);           // neg() для int-аргумента.
double neg(double n);    // neg() для double-аргумента.
long neg(long n);        // neg() для long-аргумента.

int main()
{
    cout << "neg(-10): " << neg(-10) << "\n";
    cout << "neg(9L): " << neg(9L) << "\n";
    cout << "neg(11.23): " << neg(11.23) << "\n";

    return 0;
}

// Функция neg() для int-аргумента.
int neg(int n)
```

```
{  
    return -n;  
}  
  
// Функция neg() для double-аргумента.  
double neg(double n)  
{  
    return -n;  
}  
  
// Функция neg() для long-аргумента.  
long neg(long n)  
{  
    return -n;  
}
```

Результаты выполнения этой программы таковы.

```
neg(-10): 10  
neg(9L): -9  
neg(11.23): -11.23
```

При выполнении эта программа создает три похожие, но все же различные функции, вызываемые с использованием “общего” (одного на всех) имени neg. Каждая из них возвращает результат отрицания своего аргумента. Во всех ситуациях вызова компилятор “знает”, какую именно функцию ему использовать. Для принятия решения ему достаточно “взглянуть” на тип аргумента, передаваемого функции.

Принципиальная значимость перегрузки состоит в том, что она позволяет обращаться к связанным функциям посредством одного, общего для всех, имени. Следовательно, имя neg представляет *общее действие*, которое выполняется во всех случаях. Компилятору остается правильно выбрать *конкретную* версию при конкретных обстоятельствах. Благодаря полиморфизму программисту нужно помнить не три различных имени, а только одно. Несмотря на простоту приведенного примера, он позволяет понять, насколько перегрузка способна упростить процесс программирования.

Еще одно достоинство перегрузки состоит в том, что она позволяет определить версии одной функции с небольшими различиями, которые зависят от типа обрабатываемых этой функцией данных. Рассмотрим, например, функцию min(), которая определяет минимальное из двух значений. Можно создать версии этой функции, поведение которых было бы различным для разных типов данных. При сравнении двух целых чисел функция min() возвращала бы наименьшее из них,

а при сравнении двух букв — ту, которая стоит по алфавиту раньше другой (независимо от ее регистра). Согласно ASCII-кодам символов прописные буквы представляются значениями, которые на 32 меньше значений, которые имеют строчные буквы. Поэтому при упорядочении букв по алфавиту важно игнорировать способ их начертания (регистр букв). Создавая версию функции `min()`, принимающую в качестве аргументов указатели, можно обеспечить сравнение значений, на которые они указывают, и возврат указателя, адресующего меньшее из них. Теперь рассмотрим программу, которая реализует эти версии функции `min()`.

```
// Создание различных версий функции min().
```

```
#include <iostream>
using namespace std;

int min(int a, int b);      // min() для int-значений
char min(char a, char b);  // min() для char-значений
int * min(int *a, int *b); // min() для указателей
                           // на int-значения

int main()
{
    int i=10, j=22;

    cout << "min('X', 'a'): " << min('X', 'a') << "\n";
    cout << "min(9, 3): " << min(9, 3) << "\n";
    cout << "*min(&i, &j): " << *min(&i, &j) << "\n";

    return 0;
}

// Функция min() для int-значений. Возвращает
// наименьшее значение.
int min(int a, int b)
{
    if(a < b) return a;
    else return b;
}

// Функция min() для char-значений (игнорирует регистр).
char min(char a, char b)
```

```
{  
    if(tolower(a) < tolower(b)) return a;  
    else return b;  
}  
  
/*  
Функция min() для указателей на int-значения.  
Сравнивает адресуемые ими значения и возвращает  
указатель на наименьшее значение.  
*/  
int * min(int *a, int *b)  
{  
    if(*a < *b) return a;  
    else return b;  
}  
  
Вот как выглядят результаты выполнения этой программы.  
min('X', 'a'): a  
min(9, 3): 3  
*min(&i, &j): 10
```

Каждая версия перегруженной функции может выполнять любые действия. Другими словами, не существует правила, которое бы обязывало программиста связывать перегруженные функции общими действиями. Однако с точки зрения стилистики перегрузка функций все-таки подразумевает определенное "родство" его версий. Таким образом, несмотря на то, что одно и то же имя можно использовать для перегрузки не связанных общими действиями функций, этого делать не стоит. Например, в принципе можно использовать имя `sqr` для создания функции, которая возвращает *квадрат* целого числа, и функции, которая возвращает значение *квадратного корня* из вещественного числа (типа `double`). Но, поскольку эти операции фундаментально различны, применение механизма перегрузки методов в этом случае сводит на нет его первоначальную цель. (Такой стиль программирования, наверное, подошел бы лишь для того, чтобы ввести в заблуждение конкурента.) На практике перегружать имеет смысл только тесно связанные операции.

## Автоматическое преобразование типов и перегрузка

Как упоминалось в модуле 2, в C++ предусмотрено автоматическое преобразование типов. Эти преобразования также применяются к параметрам перегруженных функций. Рассмотрим, например, следующую программу.

```

/*
Автоматические преобразования могут оказывать влияние на
выбор выполняемой функции из перегруженных ее версий.
*/

#include <iostream>
using namespace std;

void f(int x);

void f(double x);

int main() {
    int i = 10;
    double d = 10.1;
    short s = 99;
    float r = 11.5F;

    f(i); // вызов версии f(int)
    f(d); // вызов версии f(double)

    // При следующих вызовах функций происходит
    // автоматическое преобразование типов.
    f(s); // вызов версии f(int) - преобразование типов
    f(r); // вызов версии f(double) - преобразование типов

    return 0;
}

void f(int x) {
    cout << "В функции f(int): " << x << "\n";
}

void f(double x) {
    cout << "В функции f(double): " << x << "\n";
}

```

При выполнении эта программа генерирует такие результаты.

В функции f(int): 10

В функции f(double): 10.1

## 284 Модуль 6. О функциях подробнее

В функции f(int): 99

В функции f(double): 11.5

В этом примере определены две версии функции f(): с int- и double-параметром. При этом функции f() можно передать short- или float-значение. При передаче short-параметра C++ автоматически преобразует его в int-значение, и поэтому вызывается версия f(int). А при передаче float-параметра его значение преобразуется в double-значение, и поэтому вызывается версия f(double).

Однако важно понимать, что автоматические преобразования применяются только в случае, если нет непосредственного совпадения в типах параметра и аргумента. Рассмотрим предыдущую программу с добавлением версии функции f(), которая имеет short-параметр.

// Предыдущая программа с добавлением версии f(short).

```
#include <iostream>
using namespace std;

void f(int x);
void f(short x); // Добавляем версию f(short).
void f(double x);

int main() {
    int i = 10;
    double d = 10.1;
    short s = 99;
    float r = 11.5F;

    f(i); // Вызов версии f(int).
    f(d); // Вызов версии f(double).

    f(s); // Теперь будет вызвана версия f(short). Здесь
           // преобразования типов нет, поскольку определена
           // версия f(short).

    f(r); // Вызов версии f(double) - преобразование типов.

    return 0;
}

void f(int x) {
```

```

cout << "В функции f(int): " << x << "\n";
}

void f(short x) {
    cout << "В функции f(short): " << x << "\n";
}

void f(double x) {
    cout << "В функции f(double): " << x << "\n";
}

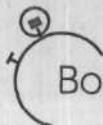
```

Теперь результаты выполнения этого варианта программы таковы.

```

В функции f(int): 10
В функции f(double): 10.1
В функции f(short): 99
В функции f(double): 11.5

```



### Вопросы для текущего контроля

1. Какие условия должны соблюдаться при перегрузке функции?
2. Почему перегруженные функции должны выполнять близкие по значению действия?
3. Учитывается ли тип значения, возвращаемого функцией, в решении о выборе нужной версии перегруженной функции?\*

## Проект 6.1. Создание перегруженных функций вывода

`Print.cpp`

В этом проекте мы создадим коллекцию перегруженных функций, которые предназначены для вывода на экран данных различного типа. Несмотря на то что во многих случаях можно обойтись инструк-

Проект  
6.1

Создание перегруженных функций вывода

1. Перегруженные функции должны иметь одинаковые имена, но объявление их параметров должно быть различным.
2. Перегруженные функции должны выполнять близкие по значению действия, поскольку перегрузка функций подразумевает связь между ними.
3. Нет, типы значений, возвращаемых функциями, могут различаться, но это различие не влияет на решение о выборе нужной версии.

цией `cout`, программисты всегда предпочитают иметь альтернативу. Создаваемая здесь коллекция функций вывода и послужит такой альтернативой. (Вот в языках Java и C# как раз используются функции вывода данных, а не операторы вывода.) Создав перегруженную функцию вывода, вы получите возможность выбора и сможете воспользоваться более подходящим для вас вариантом. Более того, средство перегрузки позволяет настроить функцию вывода под конкретные требования. Например, при выводе результатов булевого типа вместо значений 0 и 1 можно выводить слова “ложь” и “истина” соответственно.

Мы создадим два набора функций с именами `println()` и `print()`. Функция `println()` выводит после заданного аргумента символ новой строки, а функция `print()` – только аргумент. Например, при выполнении этих строк кода

```
print(1);
println('X');
println("Перегрузка функций – мощное средство! ");
print(18.22);
```

будут выведены такие данные.

1X  
Перегрузка функций – мощное средство! 18.22

В этом проекте функции `println()` и `print()` перегружаются для данных типа `bool`, `char`, `int`, `long`, `char*` и `double`, но вы сами можете добавить возможность вывода данных и других типов.

## Последовательность действий

1. Создайте файл с именем `Print.cpp`.
2. Начните проект с таких строк кода.

```
/*
Проект 6.1.
```

Создание перегруженных функций `print()` и `println()`  
для отображения данных различного типа.

\*/

```
#include <iostream>
using namespace std;
```

3. Добавьте прототипы функций `println()` и `print()`.

```
// Эти функции обеспечивают после вывода аргумента
// переход на новую строку.
```

```

void println(bool b);
void println(int i);
void println(long i);
void println(char ch);
void println(char *str);
void println(double d);

// Эти функции не обеспечивают после вывода аргумента
// переход на новую строку.
void print(bool b);
void print(int i);
void print(long i);
void print(char ch);
void print(char *str);
void print(double d);

```

#### 4. Обеспечьте реализацию функций println().

```

// Набор перегруженных функций println().
void println(bool b)
{
    if(b) cout << "истина\n";
    else cout << "ложь\n";
}

void println(int i)
{
    cout << i << "\n";
}

void println(long i)
{
    cout << i << "\n";
}

void println(char ch)
{
    cout << ch << "\n";
}

```

```
void println(char *str)
{
    cout << str << "\n";
}

void println(double d)
{
    cout << d << "\n";
}
```

Обратите внимание на то, что каждая функция из этого набора после аргумента выводит символ новой строки. Кроме того, отметьте, что версия `println(bool)` при выводе значения булевого типа отображает слова "ложь" или "истина". При желании эти слова можно выводить прописными буквами. Это доказывает, насколько легко можно настроить функции вывода под конкретные требования.

5. Реализуйте функции `print()` следующим образом.

```
// Набор перегруженных функций print().
void print(bool b)
{
    if(b) cout << "истина";
    else cout << "ложь";
}

void print(int i)
{
    cout << i;
}

void print(long i)
{
    cout << i;
}

void print(char ch)
{
    cout << ch;
}
```

```

void print(char *str)
{
    cout << str;
}

void print(double d)
{
    cout << d;
}

```

Эти функции аналогичны функциям `println()` за исключением того, что они не выводят символа новой строки. Поэтому следующий выводимый символ появится на той же строке.

6. Вот как выглядит полный текст программы `Print.cpp`.

```

/*
    Проект 6.1.

    Создание перегруженных функций print() и println()
    для отображения данных различного типа.

*/
#include <iostream>
using namespace std;

// Эти функции обеспечивают после вывода аргумента
// переход на новую строку.
void println(bool b);
void println(int i);
void println(long i);
void println(char ch);
void println(char *str);
void println(double d);

// Эти функции не обеспечивают после вывода аргумента
// переход на новую строку.
void print(bool b);
void print(int i);
void print(long i);
void print(char ch);
void print(char *str);

```

```
void print(double d);

int main()
{
    println(true);
    println(10);
    println("Это простой тест");
    println('x');
    println(99L);
    println(123.23);

    print("Вот несколько значений: ");
    print(false);
    print(' ');
    print(88);
    print(' ');
    print(100000L);
    print(' ');
    print(100.01);

    println(" Выполнено!");
}

// Набор перегруженных функций println().
void println(bool b)
{
    if(b) cout << "истина\n";
    else cout << "ложь\n";
}

void println(int i)
{
    cout << i << "\n";
}

void println(long i)
{
```

```

    cout << i << "\n";
}

void println(char ch)
{
    cout << ch << "\n";
}

void println(char *str)
{
    cout << str << "\n";
}

void println(double d)
{
    cout << d << "\n";
}

// Набор перегруженных функций print().
void print(bool b)
{
    if(b) cout << "истина";
    else cout << "ложь";
}

void print(int i)
{
    cout << i;
}

void print(long i)
{
    cout << i;
}

void print(char ch)
{
    cout << ch;
}

```

```

void print(char *str)
{
    cout << str;
}

void print(double d)
{
    cout << d;
}

```

При выполнении этой программы получаем такие результаты.

```

истина
10
Это простой тест
x
99
123.23

```

Вот несколько значений: ложь 88 100000 100.01 Выполнено!

**ВАЖНО!**

## 6.8. Аргументы, передаваемые функции по умолчанию

В C++ мы можем придать параметру некоторое значение, которое будет автоматически использовано, если при вызове функции не задается аргумент, соответствующий этому параметру. Аргументы, передаваемые функции *по умолчанию*, можно использовать, чтобы упростить обращение к сложным функциям, а также в качестве "сокращенной формы" перегрузки функций.

Задание аргументов, передаваемых функции по умолчанию, синтаксически аналогично инициализации переменных. Рассмотрим следующий пример, в котором объявляется функция `myfunc()`, принимающая два аргумента типа `int` с действующими по умолчанию значениями 0 и 100.

```
void myfunc(int x = 0, int y = 100);
```

После такого объявления функцию `myfunc()` можно вызвать одним из трех следующих способов.

```
myfunc(1, 2); // Передаем явно заданные значения.
myfunc(10);   // Передаем для параметра x значение 10,
               // а для параметра y позволяем применить
```

```
// значение, задаваемое по умолчанию (100).
myfunc(); // Для обоих параметров x и y позволяем
           // применить значения, задаваемые по умолчанию.
```

При первом вызове параметру x передается число 1, а параметру y — число 2. Во время второго вызова параметру x передается число 10, а параметр y по умолчанию устанавливается равным числу 100. Наконец, в результате третьего вызова как параметр x, так и параметр y по умолчанию устанавливаются равными значениям, заданным в объявлении функции. Этот процесс демонстрируется в следующей программе.

```
// Демонстрация использования аргументов, передаваемых
// по умолчанию.
```

```
#include <iostream>
using namespace std;

void myfunc(int x = 0, int y = 100); // В прототипе функции
                                       // заданы аргументы по
                                       // умолчанию для обоих
                                       // параметров.

int main()
{
    myfunc(1, 2);

    myfunc(10);

    myfunc();

    return 0;
}

void myfunc(int x, int y)
```

Результаты выполнения этой программы подтверждают принципы использования аргументов по умолчанию.

```
x: 1, y: 2
x: 10, y: 100
x: 0, y: 100
```

При создании функций, имеющих значения аргументов, передаваемых по умолчанию, необходимо помнить, что эти значения должны быть заданы только однажды и причем при первом объявлении функции в файле. В предыдущем примере аргумент по умолчанию был задан в прототипе функции `myfunc()`. При попытке определить новые (или даже те же) передаваемые по умолчанию значения аргументов в определении функции `myfunc()` компилятор отобразит сообщение об ошибке и не скомпилирует вашу программу.

Несмотря на то что передаваемые по умолчанию аргументы должны быть определены только один раз, для каждой версии перегруженной функции для передачи по умолчанию можно задавать различные аргументы. Таким образом, разные версии перегруженной функции могут иметь различные значения аргументов, действующие по умолчанию.

Важно помнить, что все параметры, которые принимают значения по умолчанию, должны быть расположены справа от остальных. Например, следующий прототип функции содержит ошибку.

```
// Неверно!
void f(int a = 1, int b);
```

Если вы начали определять параметры, которые принимают значения по умолчанию, нельзя после них указывать параметры, задаваемые при вызове функции только явным образом. Поэтому следующее объявление также неверно и не будет скомпилировано.

```
int myfunc(float f, char *str, int i=10, int j);
```

Поскольку для параметра `i` определено значение по умолчанию, для параметра `j` также нужно задать значение по умолчанию.

Включение в C++ возможности передачи аргументов по умолчанию позволяет программистам упрощать код программ. Чтобы предусмотреть максимально возможное количество ситуаций и обеспечить их корректную обработку, функции часто объявляются с большим числом параметров, чем необходимо в наиболее распространенных случаях. Поэтому благодаря применению аргументов по умолчанию программисту нужно указывать не все аргументы (используемые в общем случае), а только те, которые имеют смысл для определенной ситуации.

## Сравнение возможности передачи аргументов по умолчанию с перегрузкой функций

Одним из применений передачи аргументов по умолчанию является “сокращенная форма” перегрузки функций. Чтобы понять это, представьте, что вам нужно создать две специальные версии стандартной функции `strcat()`. Одна версия должна

присоединять все содержимое одной строки к концу другой. Вторая же принимает третий аргумент, который задает количество конкатенируемых (присоединяемых) символов. Другими словами, эта версия должна конкатенировать только заданное количество символов одной строки к концу другой. Допустим, что вы назвали свои функции именем `mystrcat()` и предложили такой вариант их прототипов.

```
void mystrcat(char *s1, char *s2, int len);
void mystrcat(char *s1, char *s2);
```

Первая версия должна скопировать `len` символов из строки `s2` в конец строки `s1`. Вторая версия копирует всю строку, адресуемую указателем `s2`, в конец строки, адресуемой указателем `s1`, т.е. действует подобно стандартной функции `strcat()`.

Несмотря на то что для достижения поставленной цели можно реализовать две версии функции `mystrcat()`, есть более простой способ решения этой задачи. Используя возможность передачи аргументов по умолчанию, можно создать только одну функцию `mystrcat()`, которая заменит обе задуманные ее версии. Реализация этой идеи продемонстрирована в следующей программе.

```
// Создание пользовательской версии функции strcat().

#include <iostream>
#include <cstring>
using namespace std;

void mystrcat(char *s1, char *s2, int len = 0); // Здесь
  // параметр len по умолчанию
  // устанавливается равным нулю.

int main()
{
    char str1[80] = "Это тест.";
    char str2[80] = "0123456789";

    mystrcat(str1, str2, 5); // Присоединяем только 5 символов.
    cout << str1 << '\n';

    strcpy(str1, "Это тест."); // Восстанавливаем str1.

    mystrcat(str1, str2); // Присоединяем всю строку,
                          // поскольку параметр len по
                          // умолчанию равен нулю.
    cout << str1 << '\n';
```

```
    return 0;
}

// Пользовательская версия функции strcat().
void mystrcat(char *s1, char *s2, int len)
{
    // Находим конец строки s1.
    while(*s1) s1++;

    if(len == 0) len = strlen(s2);

    while(*s2 && len) {
        *s1 = *s2; // Копируем символы.
        s1++;
        s2++;
        len--;
    }
    *s1 = '\0'; // Завершаем строку s1 нулевым символом.
}
```

Результаты выполнения этой программы выглядят так.

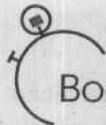
```
Это тест.01234
Это тест.0123456789
```

Здесь функция `mystrcat()` присоединяет `len` символов строки, адресуемой параметром `s2`, к концу строки, адресуемой параметром `s1`. Но если значение `len` равно нулю, как и в случае разрешения передачи этого аргумента по умолчанию, функция `mystrcat()` присоединит к строке `s1` всю строку, адресуемую параметром `s2`. (Другими словами, если значение `len` равно 0, функция `mystrcat()` действует подобно стандартной функции `strcat()`.) Используя для параметра `len` возможность передачи аргумента по умолчанию, обе операции можно объединить в одной функции. Этот пример позволил продемонстрировать, как аргументы, передаваемые функции по умолчанию, обеспечивают основу для сокращенной формы объявления перегруженных функций.

## Об использовании аргументов, передаваемых по умолчанию

Несмотря на то что аргументы, передаваемые функции по умолчанию, — очень мощное средство программирования (при их корректном использовании), с ними

могут иногда возникать проблемы. Их назначение — позволить функции эффективно выполнять свою работу, обеспечивая при всей простоте этого механизма значительную гибкость. В этом смысле все передаваемые по умолчанию аргументы должны отражать способ наиболее общего использования функции или альтернативного ее применения. Если не существует некоторого единого значения, которое обычно присваивается тому или иному параметру, то и нет смысла объявлять соответствующий аргумент по умолчанию. На самом деле объявление аргументов, передаваемых функции по умолчанию, при недостаточном для этого основании приводит к деструктуризации кода, поскольку такие аргументы способны сбить с толку любого, кому придется разбираться в такой программе. Наконец, в основе использования аргументов по умолчанию должен лежать, как у врачей, принцип “не навредить”. Другими словами, случайное использование аргумента по умолчанию не должно привести к необратимым отрицательным последствиям. Ведь такой аргумент можно просто забыть указать при вызове некоторой функции, и, если это случится, подобный промах не должен вызвать, например, потерю важных данных!



### Вопросы для текущего контроля

- Покажите, как объявить void-функцию с именем `count ()`, которая принимает два int-параметра (`a` и `b`), устанавливая их по умолчанию равными нулю.
- Можно ли устанавливаемые по умолчанию аргументы объявить как в прототипе функции, так и в ее определении?
- Корректно ли это объявление? Если нет, то почему?

```
int f(int x=10, double b); *
```

- 
- `void count(int a=0, int b=0);`
  - Нет, устанавливаемые по умолчанию аргументы должны быть объявлены лишь в первом определении функции, которым обычно служит ее прототип.
  - Нет, параметр, который не устанавливается по умолчанию, не может стоять после параметра, определяемого по умолчанию.

ВАЖНО!

## 6.9. Перегрузка функций и неоднозначность

Прежде чем завершить этот модуль, мы должны исследовать вид ошибок, уникальный для C++: неоднозначность. Возможны ситуации, в которых компилятор не способен сделать выбор между двумя (или более) корректно перегруженными функциями. Такие ситуации и называют *неоднозначными*. Инструкции, создающие неоднозначность, являются ошибочными, а программы, которые их содержат, скомпилированы не будут.

Основной причиной неоднозначности в C++ является автоматическое преобразование типов. В C++ делается попытка автоматически преобразовать тип аргументов, используемых для вызова функции, в тип параметров, определенных функцией. Рассмотрим пример.

```
int myfunc(double d);
// ...
cout << myfunc('c'); // Ошибки нет, выполняется
                      // преобразование типов.
```

Как отмечено в комментарии, ошибки здесь нет, поскольку C++ автоматически преобразует символ 'c' в его double-эквивалент. Вообще говоря, в C++ запрещено довольно мало видов преобразований типов. Несмотря на то что автоматическое преобразование типов – это очень удобно, оно, тем не менее, является главной причиной неоднозначности. Рассмотрим следующую программу.

// Неоднозначность вследствие перегрузки функций.

```
#include <iostream>
using namespace std;

float myfunc(float i);
double myfunc(double i);

int main()
{
    // Неоднозначности нет, вызывается
    // функция myfunc(double).
    cout << myfunc(10.1) << " ";

    // Здесь присутствует неоднозначность.
```

```

cout << myfunc(10); // Ошибка! (Какую версию функции
                     // myfunc() следует здесь вызвать?)

return 0;
}

float myfunc(float i)
{
    return i;
}

double myfunc(double i)
{
    return -i;
}

```

Здесь благодаря перегрузке функция `myfunc()` может принимать аргументы либо типа `float`, либо типа `double`. При выполнении строки кода

```
cout << myfunc(10.1) << " ";
```

не возникает никакой неоднозначности: компилятор “уверенно” обеспечивает вызов функции `myfunc(double)`, поскольку, если не задано явным образом иное, все литералы (константы) с плавающей точкой в C++ автоматически получают тип `double`. Но при вызове функции `myfunc()` с аргументом, равным целому числу 10, в программу вносится неоднозначность, поскольку компилятору неизвестно, в какой тип ему следует преобразовать этот аргумент: `float` или `double`. Оба преобразования допустимы. В такой неоднозначной ситуации будет выдано сообщение об ошибке, и программа не скомпилируется.

На примере предыдущей программы хотелось бы подчеркнуть, что неоднозначность в ней вызвана не перегрузкой функции `myfunc()`, объявленной дважды для приема `double`- и `float`-аргумента, а использованием при конкретном вызове функции `myfunc()` аргумента неопределенного для преобразования типа. Другими словами, ошибка состоит не в перегрузке функции `myfunc()`, а в конкретном ее вызове.

А вот еще один пример неоднозначности, вызванной автоматическим преобразованием типов в C++.

```
// Еще один пример ошибки, вызванной неоднозначностью.
```

```
#include <iostream>
using namespace std;
```

## 300 Модуль 6. О функциях подробнее

```
char myfunc(unsigned char ch);
char myfunc(char ch);

int main()
{
    cout << myfunc('c'); // Здесь вызывается myfunc(char).
    cout << myfunc(88) << " "; // Ошибка! Здесь вносится
        // неоднозначность, поскольку неясно, в
        // значение какого типа следует преобразовать
        // число 88: char или unsigned char?

    return 0;
}

char myfunc(unsigned char ch)
{
    return ch-1;
}

char myfunc(char ch)
{
    return ch+1;
}
```

В C++ типы `unsigned char` и `char` не являются существенно неоднозначными. (Это различные типы.) Но при вызове функции `myfunc()` с целочисленным аргументом 88 компилятор “не знает”, какую функцию ему выполнить, т.е. в значение какого типа ему следует преобразовать число 88: типа `char` или типа `unsigned char`? Ведь оба преобразования здесь вполне допустимы!

Неоднозначность может быть также вызвана использованием в перегруженных функциях аргументов, передаваемых по умолчанию. Для примера рассмотрим следующую программу.

```
// И еще один пример внесения неоднозначности.
```

```
#include <iostream>
using namespace std;

int myfunc(int i);
int myfunc(int i, int j=1);
```

```

int main()
{
    cout << myfunc(4, 5) << " "; // неоднозначности нет
    cout << myfunc(10); // Неоднозначность, поскольку неясно,
                        // то ли считать параметр j задаваемым
                        // по умолчанию, то ли вызывать версию
                        // функции myfunc() с одним параметром?

    return 0;
}

int myfunc(int i)
{
    return i;
}

int myfunc(int i, int j)
{
    return i*j;
}

```

Здесь в первом обращении к функции `myfunc()` задается два аргумента, поэтому у компилятора нет никаких сомнений в выборе нужной функции, а именно `myfunc(int i, int j)`, т.е. никакой неоднозначности в этом случае не привносится. Но при втором обращении к функции `myfunc()` мы сталкиваемся с неоднозначностью, поскольку компилятор “не знает”, то ли ему вызвать версию функции `myfunc()`, которая принимает один аргумент, то ли использовать возможность передачи аргумента по умолчанию применительно к версии, которая принимает два аргумента.

Программируя на языке C++, вам еще не раз придется столкнуться с ошибками неоднозначности, которые, к сожалению, очень легко “проникают” в программы, и только опыт и практика помогут вам избавиться от них.



## Тест для самоконтроля по модулю 6

1. Какие существуют два способа передачи аргумента в подпрограмму?
2. Что представляет собой ссылка в C++? Как создается ссылочный параметр?
3. Как вызвать функцию `f()` с аргументами `ch` и `i` в программе, содержащей следующий фрагмент кода?

```

int f(char &c, int *i);
// ...

```

```
char ch = 'x';
int i = 10;
```

4. Создайте void-функцию `round()`, которая округляет значение своего `double`-аргумента до ближайшего целого числа. Позаботьтесь о том, чтобы функция `round()` использовала ссылочный параметр и возвращала результат округления в этом параметре. Можете исходить из предположения, что все округляемые значения положительны. Продемонстрируйте использование функции `round()` в программе. Для решения этой задачи используйте стандартную библиотечную функцию `modf()`. Вот как выглядит ее прототип.

```
double modf(double num, double *i);
```

Функция `modf()` разбивает число `num` на целую и дробную части. При этом она возвращает дробную часть, а целую размещает в переменной, адресуемой параметром `i`. Для использования функции `modf()` необходимо включить в программу заголовок `<cmath>`.

5. Модифицируйте ссылочную версию функции `swap()`, чтобы она, помимо обмена значениями двух своих аргументов, возвращала ссылку на меньший из них. Назовите эту функцию `min_swap()`.

6. Почему функция не может возвращать ссылку на локальную переменную?  
 7. Чем должны отличаться списки параметров двух перегруженных функций?  
 8. В проекте 6.1 мы создали `print()`- и `println()`-коллекции функций. Добавьте в эти функции второй параметр, который бы задавал уровень отступа. Например, функцию `print()` можно было бы в этом случае вызвать так.

```
print("тест", 18);
```

При выполнении этого вызова был бы сделан отступ шириной в 18 пробелов, а затем выведено слово "тест". Обеспечьте параметр отступа нулевым значением, устанавливаемым по умолчанию, если он не задан при вызове. Нетрудно догадаться, что в этом случае отступ будет попросту отсутствовать.

9. При заданном прототипе функции

```
bool myfunc(char ch, int a=10, int b=20);
```

продемонстрируйте возможные способы вызова функции `myfunc()`.

10. Поясните кратко, как перегрузка функции может внести в программу неоднозначность.



Ответы на эти вопросы можно найти на Web-странице данной книги по адресу: <http://www.osborne.com>.

# Модуль 7

## Еще о типах данных и операторах

- 7.1. Спецификаторы типа `const` и `volatile`
- 7.2. Спецификатор класса памяти `extern`
- 7.3. Статические переменные
- 7.4. Регистровые переменные
- 7.5. Перечисления
- 7.6. Ключевое слово `typedef`
- 7.7. Поразрядные операторы
- 7.8. Операторы сдвига
- 7.9. Оператор “знак вопроса”
- 7.10. Оператор “запятая”
- 7.11. Составные операторы присваивания
- 7.12. Использование ключевого слова `sizeof`

В этом модуле мы возвращаемся к теме типов данных и операторов. Кроме уже рассмотренных нами типов данных, в C++ определены и другие. Одни типы данных состоят из модификаторов, добавляемых к уже известным вам типам. Другие включают перечисления, а третья используют ключевое слово `typedef`. C++ также поддерживает ряд операторов, которые значительно расширяют область применения языка и позволяют решать задачи программирования в весьма широком диапазоне. Речь идет о поразрядных операторах, операторах сдвига, а также операторах “?” и `sizeof`.

**ВАЖНО!**

## 7.1. Спецификаторы типа `const` и `volatile`

В C++ определено два спецификатора типа, которые оказывают влияние на то, каким образом можно получить доступ к переменным или модифицировать их. Это спецификаторы `const` и `volatile`. Официально они именуются *cv-спецификаторами* и должны предшествовать базовому типу при объявлении переменной.

### Спецификатор типа `const`

Переменные, объявленные с использованием спецификатора `const`, не могут изменить свои значения во время выполнения программы. Выходит, `const`-переменную нельзя назвать “настоящей” переменной. Тем не менее ей можно присвоить некоторое начальное значение. Например, при выполнении инструкции

```
const int max_users = 9;
```

создается `int`-переменная `max_users`, которая содержит значение 9, и это значение программа изменить уже не может. Но саму переменную можно использовать в других выражениях.

Чаще всего спецификатор `const` используется для создания *именованных констант*. Иногда в программах многократно применяется одно и то же значение для различных целей. Например, в программе необходимо объявить несколько различных массивов таким образом, чтобы все они имели одинаковый размер. Такой общий для всех массивов размер имеет смысл реализовать в виде `const`-переменной. Затем вместо реального значения можно использовать имя этой переменной, а если это значение придется впоследствии изменить, вы измените его только в одном месте программы (а не в объявлении каждого массива). Такой подход позволяет избежать ошибок и упростить программу. Следующий пример предлагает вам попробовать этот вид применения спецификатора `const` “на вкус”.

```
#include <iostream>
using namespace std;

const int num_employees = 100; // ← Создаем именованную
                             // константу num_employees, которая
                             // содержит значение 100.

int main()
{
    int empNums[num_employees]; // Именованная константа
    double salary[num_employees]; // num_employees здесь
    char *names[num_employees]; // используется для задания
                               // размеров массивов.

    // ...
}
```

Если в этом примере понадобится использовать новый размер для массивов, вам потребуется изменить только объявление переменной num\_employees и перекомпилировать программу. В результате все три массива автоматически получат новый размер.

Спецификатор const также используется для защиты объекта от модификации посредством указателя. Например, с помощью спецификатора const можно не допустить, чтобы функция изменила значение объекта, адресуемого ее параметром-указателем. Для этого достаточно объявить такой параметр-указатель с использованием спецификатора const. Другими словами, если параметр-указатель предваряется ключевым словом const, никакая инструкция этой функции не может модифицировать переменную, адресуемую этим параметром. Например, функция negate() в следующей короткой программе возвращает результат отрицания значения, на которое указывает параметр. Использование спецификатора const в объявлении параметра не позволяет коду функции модифицировать объект, адресуемый этим параметром.

```
// Использование const-параметра-указателя.
```

```
#include <iostream>
using namespace std;

int negate(const int *val);
```

## 306 Модуль 7. Еще о типах данных и операторах

```
int main()
{
    int result;
    int v = 10;

    result = negate(&v);

    cout << "Результат отрицания " << v << " равен " << result;
    cout << "\n";

    return 0;
}

int negate(const int *val) // ← Так параметр val
                         // объявляется const-указателем.
{
    return - *val;
}
```

Поскольку параметр `val` является `const`-указателем, функция `negate()` не может изменить значение, на которое он указывает. Так как функция `negate()` не пытается изменить значение, адресуемое параметром `val`, программа скомпилируется и выполнится корректно. Но если функцию `negate()` переписать так, как показано в следующем примере, компилятор выдаст ошибку.

```
// Этот вариант работать не будет!
int negate(const int *val)
{
    *val = - *val; // Ошибка: значение, адресуемое
                  // параметром val, изменять нельзя.
    return *val;
}
```

В этом случае программа попытается изменить значение переменной, на которую указывает параметр `val`, но это не будет реализовано, поскольку `val` объявлен `const`-параметром.

Спецификатор `const` можно также использовать для ссылочных параметров, чтобы не допустить в функции модификацию переменных, на которые ссылаются эти параметры. Например, следующая версия функции `negate()` некорректна, поскольку в ней делается попытка модифицировать переменную, на которую ссылается параметр `val`.

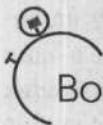
```
// Этот вариант также не будет работать!
int negate(const int &val)
{
    val = -val; // Ошибка: значение, на которое ссылается
                // параметр val, изменять нельзя.
    return val;
}
```

## Спецификатор типа volatile

Спецификатор `volatile` сообщает компилятору о том, что значение соответствующей переменной может быть изменено в программе неявным образом. Например, адрес некоторой глобальной переменной может передаваться управляемой прерываниями подпрограмме тактирования, которая обновляет эту переменную с приходом каждого импульса сигнала времени. В такой ситуации содержимое переменной изменяется без использования явно заданных инструкций программы. Существуют веские основания для того, чтобы сообщить компилятору о внешних факторах изменения переменной. Дело в том, что C++-компилятору разрешается автоматически оптимизировать определенные выражения в предположении, что содержимое той или иной переменной остается неизменным, если оно не находится в левой части инструкции присваивания. Но если некоторые факторы (внешние по отношению к программе) изменят значение этого поля, такое предположение окажется неверным, в результате чего могут возникнуть проблемы. Для решения подобных проблем необходимо объявлять такие переменные с ключевым словом `volatile`.

```
volatile int current_users;
```

Теперь значение переменной `current_users` будет опрашиваться при каждом ее использовании.



### Вопросы для текущего контроля

1. Может ли значение `const`-переменной быть изменено программой?
2. Как следует объявить переменную, которая может изменить свое значение в результате воздействия внешних факторов?\*

- 
1. Нет, значение `const`-переменной программа изменить не может.
  2. Переменную, которая может изменить свое значение в результате воздействия внешних факторов, нужно объявить с использованием спецификатора `volatile`.

## Спецификаторы классов памяти

C++ поддерживает пять спецификаторов *классов памяти*:

```
auto
extern
register
static
mutable
```

С помощью этих ключевых слов компилятор получает информацию о том, как должна храниться переменная. Спецификатор классов памяти необходимо указывать в начале объявления переменной.

Спецификатор `mutable` применяется только к объектам классов, о которых речь впереди. Остальные спецификаторы мы рассмотрим в этом разделе.

### Спецификатор класса памяти `auto`

Спецификатор `auto` (достался языку C++ от C "по наследству") объявляет локальную переменную. Но он используется довольно редко (возможно, вам никогда и не доведется применить его), поскольку локальные переменные являются "автоматическими" по умолчанию. Вряд ли вам попадется это ключевое слово и в чужих программах.

**ВАЖНО!**

### 7.2. Спецификатор класса памяти `extern`

Все программы, которые мы рассматривали до сих пор, имели довольно скромный размер. Реальные же компьютерные программы гораздо больше. По мере увеличения размера файла, содержащего программу, время компиляции становится иногда раздражающе долгим. В этом случае следует разбить программу на несколько отдельных файлов. После этого небольшие изменения, вносимые в один файл, не потребуют перекомпиляции всей программы. При разработке больших проектов такой многофайловый подход может сэкономить существенное время. Реализовать этот подход позволяет ключевое слово `extern`.

В программах, которые состоят из двух или более файлов, каждый файл должен "знать" имена и типы глобальных переменных, используемых программой в целом. Однако нельзя просто объявить копии глобальных переменных в каждом файле. Дело в том, что в C++ программа может включать только одну копию каждой глобальной переменной. Следовательно, если вы попытаетесь объявить необходимые глобальные переменные в каждом файле, возникнут проблемы. Когда компоновщик попытается объединить эти файлы, он обнаружит дублированные глобальные

переменные, и компоновка программы не состоится. Чтобы выйти из этого затруднительного положения, достаточно объявить все глобальные переменные в одном файле, а в других использовать `extern`-объявления, как показано на рис. 7.1.

| Файл F1                                                                                 | Файл F2                                                                                                     |
|-----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <pre>int x, y; char ch;  int main() {     // ... }  void func1() {     x = 123; }</pre> | <pre>extern int x, y; extern char ch;  void func22() {     x = y/10; }  void func23() {     y = 10; }</pre> |
|                                                                                         |                                                                                                             |

Рис. 7.1. Использование глобальных переменных в отдельно компилируемых модулях

В файле F1 объявляются и определяются переменные `x`, `y` и `ch`. В файле F2 используется скопированный из файла F1 список глобальных переменных, к объявлению которых добавлено ключевое слово `extern`. Спецификатор `extern` делает переменную известной для модуля, но в действительности не создает ее. Другими словами, ключевое слово `extern` предоставляет компилятору информацию о типе и имени глобальных переменных, повторно не выделяя для них памяти. Во время компоновки этих двух модулей все ссылки на внешние переменные будут определены.

До сих пор мы не уточняли, в чем состоит различие между объявлением и определением переменной, но здесь это очень важно. При *объявлении* переменной присваивается имя и тип, а посредством *определения* для переменной выделяется память. В большинстве случаев объявление переменных одновременно являются определениями. Предварив имя переменной спецификатором `extern`, можно объявить переменную, не определяя ее.

### Спецификатор компоновки `extern`

Спецификатор `extern` позволяет также указать, как та или иная функция связывается с программой (т.е. каким образом функция должна быть обработана

компоновщиком). По умолчанию функции компонуются как C++-функции. Но, используя *спецификацию компоновки* (специальную инструкцию для компилятора), можно обеспечить компоновку функций, написанных на других языках программирования. Общий формат спецификатора компоновки выглядит так:

```
extern "язык" прототип_функции
```

Здесь элемент *язык* означает нужный язык программирования. Например, эта инструкция позволяет скомпоновать функцию *myCfunc()* как C-функцию.

```
extern "C" void myCfunc();
```

Все C++-компиляторы поддерживают как C-, так и C++-компонентовку. Некоторые компиляторы также позволяют использовать спецификаторы компоновки для таких языков, как Fortran, Pascal или BASIC. (Эту информацию необходимо уточнить в документации, прилагаемой к вашему компилятору.) Используя следующий формат спецификации компоновки, можно задать не одну, а сразу несколько функций.

```
extern "язык" {  
    прототипы_функций  
}
```

Спецификации компоновки используются довольно редко, и вам, возможно, никогда не придется их применять.

### ВАЖНО!

## 7.3. Статические переменные

Переменные типа *static* – это переменные “долговременного” хранения, т.е. они хранят свои значения в пределах своей функции или файла. От глобальных они отличаются тем, что за рамками своей функции или файла они неизвестны. Поскольку спецификатор *static* по-разному определяет “судьбу” локальных и глобальных переменных, мы рассмотрим их в отдельности.

### Локальные static-переменные

Если к локальной переменной применен модификатор *static*, то для нее выделяется постоянная область памяти практически так же, как и для глобальной переменной. Это позволяет статической переменной поддерживать ее значение между вызовами функций. (Другими словами, в отличие от обычной локальной переменной, значение *static*-переменной не теряется при выходе из функции.) Ключевое различие между статической локальной и глобальной переменными

состоит в том, что статическая локальная переменная известна только блоку, в котором она объявлена.

Чтобы объявить статическую переменную, достаточно предварить ее тип ключевым словом `static`. Например, при выполнении этой инструкции переменная `count` является статической.

```
static int count;
```

Статической переменной можно присвоить некоторое начальное значение. Например, в этой инструкции переменной `count` присваивается начальное значение 200:

```
static int count = 200;
```

Локальные `static`-переменные инициализируются только однажды, в начале выполнения программы, а не при каждом входе в функцию, в которой они объявлены.

Возможность использования статических локальных переменных важна для создания функций, которые должны сохранять их значения между вызовами. Если бы статические переменные не были предусмотрены в C++, пришлось бы использовать вместо них глобальные, что открыло бы "лазейку" для всевозможных побочных эффектов.

Рассмотрим пример использования `static`-переменной. Она служит для хранения текущего среднего значения от чисел, вводимых пользователем.

```
/* Вычисляем текущее среднее значение от чисел, вводимых
пользователем.
```

```
*/
```

```
#include <iostream>
using namespace std;

int running_avg(int i);

int main()
{
    int num;

    do {
        cout << "Введите числа (-1 означает выход): ";
        cin >> num;
        if(num != -1)
            cout << "Текущее среднее равно: "
```

```
        << running_avg(num);
    cout << '\n';
} while(num > -1);

return 0;
}

// Вычисляем текущее среднее.
int running_avg(int i)
{
    static int sum = 0, count = 0; // Поскольку переменные
                                // sum и count являются статическими,
                                // они сохраняют свои значения между
                                // вызовами функции running_avg().

    sum = sum + i;

    count++;

    return sum / count;
}
```

Здесь обе локальные переменные `sum` и `count` объявлены статическими и инициализированы значением 0. Помните, что для статических переменных инициализация выполняется только один раз (при первом выполнении функции), а не при каждом входе в функцию. В этой программе функция `running_avg()` используется для вычисления текущего среднего значения от чисел, вводимых пользователем. Поскольку обе переменные `sum` и `count` являются статическими, они поддерживают свои значения между вызовами функции `running_avg()`, что позволяет нам получить правильный результат вычислений. Чтобы убедиться в необходимости модификатора `static`, попробуйте удалить его из программы. После этого программа не будет работать корректно, поскольку промежуточная сумма будет теряться при каждом выходе из функции `running_avg()`.

### Глобальные `static`-переменные

Если модификатор `static` применен к глобальной переменной, то компилятор создаст глобальную переменную, которая будет известна только для файла, в котором она объявлена. Это означает, что, хотя эта переменная является глобальной, другие функции в других файлах не имеют о ней "ни малейшего поня-

тия" и не могут изменить ее содержимое. Поэтому она и не может стать "жертвой" несанкционированных изменений. Следовательно, для особых ситуаций, когда локальная статичность оказывается бессильной, можно создать небольшой файл, который будет содержать лишь функции, использующие глобальные static-переменные, отдельно скомпилировать этот файл и работать с ним, не опасаясь вреда от побочных эффектов "всеобщей глобальности".

Рассмотрим пример, который представляет собой переработанную версию программы (из предыдущего подраздела), вычисляющей текущее среднее значение. Эта версия состоит из двух файлов и использует глобальные static-переменные для хранения значений промежуточной суммы и счетчика вводимых чисел. В эту версию программы добавлена функция `reset()`, которая обнуляет ("сбрасывает") значения глобальных static-переменных.

```
// ----- Первый файл -----
#include <iostream>
using namespace std;

int running_avg(int i);
void reset();

int main()
{
    int num;

    do {
        cout <<
            "Введите числа (-1 для выхода, -2 для сброса): ";
        cin >> num;
        if(num== -2) {
            reset();
            continue;
        }
        if(num != -1)
            cout << "Среднее значение равно: "
                << running_avg(num);
        cout << '\n';
    } while(num != -1);

    return 0;
}
```

```
}

// ----- Второй файл -----
static int sum=0, count=0; // Эти переменные известны
                         // только в файле, в котором
                         // они объявлены.

int running_avg(int i)
{
    sum = sum + i;

    count++;

    return sum / count;
}

void reset()
{
    sum = 0;
    count = 0;
}
```

В этой версии программы переменные `sum` и `count` являются глобально статическими, т.е. их глобальность ограничена вторым файлом. Итак, они используются функциями `running_avg()` и `reset()`, причем обе они расположены во втором файле. Этот вариант программы позволяет сбрасывать накопленную сумму (путем установки в исходное положение переменных `sum` и `count`), чтобы можно было усреднить другой набор чисел. Но ни одна из функций, расположенных вне второго файла, не может получить доступ к этим переменным. Работая с данной программой, можно обнулить предыдущие накопления, введя число `-2`. В этом случае будет вызвана функция `reset()`. Проверьте это. Кроме того, попытайтесь получить из первого файла доступ к любой из переменных `sum` или `count`. (Вы получите сообщение об ошибке.)

Итак, имя локальной `static`-переменной известно только функции или блоку кода, в котором она объявлена, а имя глобальной `static`-переменной — только файлу, в котором она “обитает”. По сути, модификатор `static` позволяет переменным существовать так, что о них знают только функции, использующие их, тем самым “держа в узде” и ограничивая возможности негативных побочных эффектов. Переменные типа `static` позволяют программисту “скрывать” одни части своей программы от других частей. Это может оказаться просто супердостижением, когда вам придется разрабатывать очень большую и сложную программу.

## Спросим у опытного программиста

**Вопрос.** Я слышал, что некоторые C++-программисты не используют глобальные static-переменные. Так ли это?

**Ответ.** Несмотря на то что глобальные static-переменные по-прежнему допустимы и широко используются в C++-коде, стандарт C++ не предусматривает их применения. Для управления доступом к глобальным переменным рекомендуется другой метод, который заключается в использовании пространств имен. Этот метод описан ниже в данной книге. При этом глобальные static-переменные широко используются С-программистами, поскольку в С не поддерживаются пространства имен. Поэтому вам еще долго придется встречаться с глобальными static-переменными.

ВАЖНО!

## 7.4. Регистровые переменные

Возможно, чаще всего используется спецификатор класса памяти `register`. Для компилятора модификатор `register` означает предписание обеспечить такое хранение соответствующей переменной, чтобы доступ к ней можно было получить максимально быстро. Обычно переменная в этом случае будет храниться либо в регистре центрального процессора (ЦП), либо в кэш-памяти (быстродействующей буферной памяти небольшой емкости). Вероятно, вы знаете, что доступ к регистрам ЦП (или к кэш-памяти) принципиально быстрее, чем доступ к основной памяти компьютера. Таким образом, переменная, сохраняемая в регистре, будет обслужена гораздо быстрее, чем переменная, сохраняемая, например, в оперативной памяти (ОЗУ). Поскольку скорость, с которой к переменным можно получить доступ, определяет, по сути, скорость выполнения вашей программы, для получения удовлетворительных результатов программирования важно разумно использовать спецификатор `register`.

Формально спецификатор `register` представляет собой лишь запрос, который компилятор вправе проигнорировать. Это легко объяснить: ведь количество регистров (или устройств памяти с малым временем выборки) ограничено, причем для разных сред оно может быть различным. Поэтому, если компилятор исчерпает память быстрого доступа, он будет хранить `register`-переменные обычным способом. В общем случае неудовлетворенный `register`-запрос не приносит вреда, но, конечно же, и не дает никаких преимуществ хранения в регистровой памяти. Как правило, программист может рассчитывать на удовлетворение `register`-запроса по крайней мере для двух переменных, обработка которых действительно будет оптимизирована с точки зрения максимальной скорости.

## 316 Модуль 7. Еще о типах данных и операторах

Поскольку быстрый доступ можно обеспечить на самом деле только для ограниченного количества переменных, важно тщательно выбрать, к каким из них применить модификатор `register`. (Лишь правильный выбор может повысить быстродействие программы.) Как правило, чем чаще к переменной требуется доступ, тем большая выгода будет получена в результате оптимизации кода с помощью спецификатора `register`. Поэтому объявлять регистровыми имеет смысл управляющие переменные цикла или переменные, к которым выполняется доступ в теле цикла.

На примере следующей функции показано, как используются `register`-переменные для повышения быстродействия функции `summation()`, которая вычисляет сумму значений элементов массива. Здесь как раз и предполагается, что реально для получения скоростного эффекта будут оптимизированы только две переменные.

```
// Демонстрация использования register-переменных.

#include <iostream>
using namespace std;

int summation(int nums[], int n);

int main()
{
    int vals[] = { 1, 2, 3, 4, 5 };
    int result;

    result = summation(vals, 5);

    cout << "Сумма элементов массива равна " << result << "\n";
}

return 0;
}

// Функция возвращает сумму int-элементов массива.
int summation(int nums[], int n)
{
    register int i;          // Эти переменные оптимизированы для
    register int sum = 0;    // для получения максимальной скорости.
```

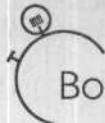
```

for(i = 0; i < n; i++)
    sum = sum + nums[i];

return sum;
}

```

Здесь переменная `i`, которая управляет циклом `for`, и переменная `sum`, к которой осуществляется доступ в теле цикла, определены с использованием спецификатора `register`. Поскольку обе они используются внутри цикла, можно рассчитывать на то, что их обработка будет оптимизирована для реализации быстрого к ним доступа. В этом примере предполагалось, что реально для получения скоростного эффекта будут оптимизированы только две переменные, поэтому `nums` и `n` не были определены как `register`-переменные, ведь доступ к ним реализуется не столь часто, как к переменным `i` и `sum`. Но если среда позволяет оптимизацию более двух переменных, то имело бы смысл переменные `nums` и `n` также объявить с использованием спецификатора `register`, что еще больше повысило бы быстродействие программы.



## Вопросы для текущего контроля

- Локальная переменная, объявленная с использованием модификатора `static`, \_\_\_\_\_ свое значение между вызовами функции.
- Спецификатор `extern` используется для объявления переменной без ее определения. Верно ли это?
- Какой спецификатор служит для компилятора запросом на оптимизацию обработки переменной с целью повышения быстродействия программы?\*

- Локальная переменная, объявленная с использованием модификатора `static`, сохраняет свое значение между вызовами функции.
- Верно, спецификатор `extern` действительно используется для объявления переменной без ее определения.
- Запросом на оптимизацию обработки переменной с целью повышения быстродействия программы служит спецификатор `register`.

## Спросим у опытного программиста

**Вопрос.** Когда я добавил в программу спецификатор `register`, то не заметил никаких изменений в быстродействии. В чем причина?

**Ответ.** Большинство компиляторов (благодаря прогрессивной технологии их разработки) автоматически оптимизируют программный код. Поэтому во многих случаях внесение спецификатора `register` в объявление переменной не ускорит выполнение программы, поскольку обработка этой переменной уже оптимизирована. Но в некоторых случаях использование спецификатора `register` оказывается весьма полезным, так как он позволяет сообщить компилятору, какие именно переменные вы считаете наиболее важными для оптимизации. Это особенно ценно для функций, в которых используется большое количество переменных, и ясно, что всех их невозможно оптимизировать. Следовательно, несмотря на прогрессивную технологию разработки компиляторов, спецификатор `register` по-прежнему играет важную роль для эффективного программирования.

### ВАЖНО!

## 7.5. Перечисления

В C++ можно определить список именованных целочисленных констант. Такой список называется *перечислением* (enumeration). Эти константы можно затем использовать везде, где допустимы целочисленные значения (например, в целочисленных выражениях). Перечисления определяются с помощью ключевого слова `enum`, а формат их определения имеет такой вид:

```
enum имя_типа { список_перечисления } список_переменных;
```

Под элементом `список_перечисления` понимается список разделенных запятыми имен, которые представляют значения перечисления. Элемент `список_переменных` необязателен, поскольку переменные можно объявить позже, используя `имя_типа` перечисления.

В следующем примере определяется перечисление `transport` и две переменные типа `transport` с именами `t1` и `t2`.

```
enum transport { car, truck, airplane, train, boat } t1, t2;
```

Определив перечисление, можно объявить другие переменные этого типа, используя имя типа перечисления. Например, с помощью следующей инструкции объявляется одна переменная `how` перечисления `transport`.

```
transport how;
```

Эту инструкцию можно записать и так.

```
enum transport how;
```

Однако использование ключевого слова enum здесь излишне. В языке С (который также поддерживает перечисления) обязательной была вторая форма, поэтому в некоторых программах вы можете встретить подобную запись.

С учетом предыдущих объявлений при выполнении следующей инструкции переменной how присваивается значение airplane.

```
how = airplane;
```

Важно понимать, что каждый символ списка перечисления означает целое число, причем каждое следующее число (представленное идентификатором) на единицу больше предыдущего. По умолчанию значение первого символа перечисления равно нулю, следовательно, значение второго — единице и т.д. Поэтому при выполнении этой инструкции

```
cout << car << ' ' << train;
```

на экран будут выведены числа 0 и 3.

Несмотря на то что перечислимые константы автоматически преобразуются в целочисленные, обратное преобразование автоматически не выполняется. Например, следующая инструкция некорректна.

```
how = 1; // ошибка
```

Эта инструкция вызовет во время компиляции ошибку, поскольку автоматического преобразования целочисленных значений в значения типа transport не существует. Откорректировать предыдущую инструкцию можно с помощью операции приведения типов.

```
fruit = (transport) 1; // Теперь все в порядке,
                        // но стиль не совершенен.
```

Теперь переменная how будет содержать значение truck, поскольку эта transport-константа связывается со значением 1. Как отмечено в комментарии, несмотря на то, что эта инструкция стала корректной, ее стиль оставляет желать лучшего, что простительно лишь в особых обстоятельствах.

Используя инициализатор, можно указать значение одной или нескольких перечислимых констант. Это делается так: после соответствующего элемента списка перечисления ставится знак равенства и нужное целое число. При использовании инициализатора следующему (после инициализированного) элементу списка присваивается значение, на единицу превышающее предыдущее значение инициализатора. Например, при выполнении следующей инструкции константе airplane присваивается значение 10.

```
enum transport { car, truck, airplane = 10, train, boat };
```

Теперь все символы перечисления `transport` имеют следующие значения.

|          |    |
|----------|----|
| car      | 0  |
| truck    | 1  |
| airplane | 10 |
| train    | 11 |
| boat     | 12 |

Часто ошибочно предполагается, что символы перечисления можно вводить и выводить как строки. Например, следующий фрагмент кода выполнен не будет.

```
// Слово "train" на экран таким образом не попадет.  
how = train;  
cout << how;
```

Незабывайте, что символ `train` — это просто имя для некоторого целочисленного значения, а не строка. Следовательно, при выполнении предыдущего кода на экране отобразится числовое значение константы `train`, а не строка `"train"`. Конечно, можно создать код ввода и вывода символов перечисления в виде строк, но он выходит несколько громоздким. Вот, например, как можно отобразить на экране названия транспортных средств, связанных с переменной `how`.

```
switch (how) {  
    case car:  
        cout << "Automobile";  
        break;  
    case truck:  
        cout << "Truck";  
        break;  
    case airplane:  
        cout << "Airplane";  
        break;  
    case train:  
        cout << "Train";  
        break;  
    case boat:  
        cout << "Boat";  
        break;  
}
```

Иногда для перевода значения перечисления в соответствующую строку можно объявить массив строк и использовать значение перечисления в качестве индекса. Например, следующая программа выводит названия трех видов транспорта.

```
// Демонстрация использования перечисления.

#include <iostream>
using namespace std;

enum transport { car, truck, airplane, train, boat };

char name[][20] = {
    "Automobile",
    "Truck",
    "Airplane",
    "Train",
    "Boat"
};

int main()
{
    transport how;

    how = car;
    cout << name[how] << '\n';

    how = airplane;
    cout << name[how] << '\n';

    how = train;
    cout << name[how] << '\n';

    return 0;
}
```

Вот результаты выполнения этой программы.

```
Automobile
Airplane
Train
```

Использованный в этой программе метод преобразования значения перечисления в строку можно применить к перечислению любого типа, если оно не содержит инициализаторов. Для надлежащего индексирования массива строк перечислимые константы должны начинаться с нуля, быть строго упорядочен-

ными по возрастанию, и каждая следующая константа должна быть больше предыдущей точно на единицу.

Из-за того, что значения перечисления необходимо вручную преобразовывать в удобные для восприятия человеком строки, они, в основном, используются там, где такое преобразование не требуется. Для примера рассмотрите перечисление, используемое для определения таблицы символов компилятора.

**ВАЖНО!**

## 7.6. Ключевое слово `typedef`

В C++ разрешается определять новые имена типов данных с помощью ключевого слова `typedef`. При использовании `typedef`-имени не создается новый тип данных, а лишь определяется новое имя для уже существующего типа. Благодаря `typedef`-именам можно сделать машинозависимые программы более переносимыми: для этого иногда достаточно изменить `typedef`-инструкции. Это средство также позволяет улучшить читабельность кода, поскольку для стандартных типов данных с его помощью можно использовать описательные имена. Общий формат записи инструкции `typedef` таков.

```
typedef тип имя;
```

Здесь элемент `тип` означает любой допустимый тип данных, а элемент `имя` — новое имя для этого типа. При этом заметьте: новое имя определяется вами в качестве дополнения к существующему имени типа, а не для его замены.

Например, с помощью следующей инструкции можно создать новое имя для типа `float`.

```
typedef float balance;
```

Эта инструкция является предписанием компилятору распознавать идентификатор `balance` как еще одно имя для типа `float`. После этой инструкции можно создавать `float`-переменные с использованием имени `balance`.

```
balance over_due;
```

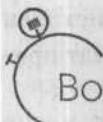
Здесь объявлена переменная с плавающей точкой `over_due` типа `balance`, который представляет собой стандартный тип `float` с другим названием..

**ВАЖНО!**

## 7.7. Поразрядные операторы

Поскольку язык C++ сориентирован так, чтобы позволить полный доступ к аппаратным средствам компьютера, важно, чтобы он имел возможность непо-

средственно воздействовать на отдельные биты в рамках байта или машинного слова. Именно поэтому C++ и содержит поразрядные операторы. *Поразрядные операторы* предназначены для тестирования, установки или сдвига реальных битов в байтах или словах, которые соответствуют символьным или целочисленным C++-типам. Поразрядные операторы не используются для operandов типа `bool`, `float`, `double`, `long double`, `void` или других еще более сложных типов данных. Поразрядные операторы (табл. 7.1) очень часто используются для решения широкого круга задач программирования системного уровня, например, при опросе информации о состоянии устройства или ее формировании. Теперь рассмотрим каждый оператор этой группы в отдельности.



### Вопросы для текущего контроля

1. Перечисление – это список именованных \_\_\_\_\_ констант.
2. Какое целочисленное значение по умолчанию имеет первый символ перечисления?
3. Покажите, как объявить идентификатор `BigInt`, чтобы он стал еще одним именем для типа `long int`.\*

Таблица 7.1. Поразрядные операторы

| Оператор | Значение                              |
|----------|---------------------------------------|
| &        | Поразрядное И (AND)                   |
|          | Поразрядное ИЛИ (OR)                  |
| ^        | Поразрядное исключающее ИЛИ (XOR)     |
| ~        | Дополнение до 1 (унарный оператор НЕ) |
| >>       | Сдвиг вправо                          |
| <<       | Сдвиг влево                           |

## Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ

Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ (обозначаемые символами `&`, `|`, `^` и `~` соответственно) выполняют те же операции, что и их ло-

- 
1. Перечисление – это список именованных целочисленных констант.
  2. По умолчанию первый символ перечисления имеет значение 0.
  3. `typedef long int BigInt;`.

гические эквиваленты (т.е. они действуют согласно той же таблице истинности). Различие состоит лишь в том, что поразрядные операции работают на побитовой основе. В следующей таблице показан результат выполнения каждой поразрядной операции для всех возможных сочетаний operandов (нулей и единиц).

| p | q | $p \& q$ | $p   q$ | $p ^ q$ | $\sim p$ |
|---|---|----------|---------|---------|----------|
| 0 | 0 | 0        | 0       | 0       | 1        |
| 1 | 0 | 0        | 1       | 1       | 0        |
| 0 | 1 | 0        | 1       | 1       | 1        |
| 1 | 1 | 1        | 1       | 0       | 0        |

Как видно из таблицы, результат применения оператора XOR (исключающее ИЛИ) будет равен значению ИСТИНА (1) только в том случае, если истинен (равен значению 1) лишь один из operandов; в противном случае результат принимает значение ЛОЖЬ (0).

Поразрядный оператор И можно представить как способ подавления побитовой информации. Это значит, что 0 в любом operandе обеспечит установку в 0 соответствующего бита результата. Вот пример.

```

1101 0011
& 1010 1010
-----
1000 0010

```

Использование оператора “`&`” демонстрируется в следующей программе. Она преобразует любой строчный символ в его прописной эквивалент путем установки шестого бита равным значению 0. Набор символов ASCII определен так, что строчные буквы имеют почти такой же код, что и прописные, за исключением того, что код первых отличается от кода вторых ровно на 32. Следовательно, как показано в этой программе, чтобы из строчной буквы сделать прописную, достаточно обнулить ее шестой бит.

```
// Получение прописных букв с использованием поразрядного
// оператора "И".
```

```
#include <iostream>
using namespace std;

int main()
{
    char ch;

    for(int i = 0 ; i < 10; i++)  {

```

```

ch = 'a' + i;
cout << ch;

// Эта инструкция обнуляет 6-й бит.
ch = ch & 223; // В переменной ch теперь
                // прописная буква.

cout << ch << " ";
}

cout << "\n";

return 0;
}

```

Вот как выглядят результаты выполнения этой программы.

```
aA bB cC dD eE fF gG hH iI jJ
```

Значение 223, используемое в инструкции поразрядного оператора “И”, является десятичным представлением двоичного числа 1101 1111. Следовательно, эта операция “И” оставляет все биты в переменной ch нетронутыми, за исключением шестого (он сбрасывается в нуль).

Оператор “И” также полезно использовать, если нужно определить, установлен интересующий вас бит (т.е. равен ли он значению 1) или нет. Например, при выполнении следующей инструкции вы узнаете, установлен ли 4-й бит в переменной `status`.

```
if(status & 8) cout << "Бит 4 установлен";
```

Чтобы понять, почему для тестирования четвертого бита используется число 8, вспомните, что в двоичной системе счисления число 8 представляется как 0000 1000, т.е. в числе 8 установлен только четвертый разряд. Поэтому условное выражение инструкции `if` даст значение ИСТИНА только в том случае, если четвертый бит переменной `status` также установлен (равен 1). Интересное использование этого метода показано на примере функции `disp_binary()`. Она отображает в двоичном формате конфигурацию битов своего аргумента. Мы будем использовать функцию `show_binary()` ниже в этой главе для исследования возможностей других поразрядных операций.

```
// Отображение конфигурации битов в байте.
```

```
void show_binary(unsigned int u)
{
```

```

int t;

for(t=128; t > 0; t = t/2)
    if(u & t) cout << "1 ";
    else cout << "0 ";

cout << "\n";
}

```

Функция `show_binary()`, используя поразрядный оператор “И”, последовательно тестирует каждый бит младшего байта переменной `u`, чтобы определить, установлен он или сброшен. Если он установлен, отображается цифра 1, в противном случае – цифра 0.

Поразрядный оператор “ИЛИ” (в противоположность поразрядному “И”) удобно использовать для установки нужных битов равными единице. При выполнении операции “ИЛИ” наличие в любом операнде бита, равного 1, означает, что в результате соответствующий бит также будет равен единице. Вот пример.

```

1101 0011
1010 1010
1111 1011

```

Оператор “ИЛИ” можно использовать для превращения рассмотренной выше программы (которая преобразует строчные символы в их прописные эквиваленты) в ее “противоположность”, т.е. теперь, как показано ниже, она будет преобразовывать прописные буквы в строчные.

```
// Получение строчных букв с использованием поразрядного
// оператора “ИЛИ”.
```

```

#include <iostream>
using namespace std;

int main()
{
    char ch;

    for(int i = 0 ; i < 10; i++) {
        ch = 'A' + i;
        cout << ch;
    }
}

```

```

// Эта инструкция делает букву строчной,
// устанавливая ее 6-й бит.
ch = ch | 32; // В переменной ch теперь
                // строчная буква.

cout << ch << " ";
}

cout << "\n";

return 0;
}

```

Вот результаты выполнения этой программы.

Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj

Итак, вы убедились, что установка шестого бита превращает прописную букву в ее строчный эквивалент.

Поразрядное исключающее “ИЛИ” (XOR) устанавливает бит результата равным единице только в том случае, если соответствующие биты операндов отличаются один от другого, т.е. не равны. Вот пример:

$$\begin{array}{r}
 0111\ 1111 \\
 ^{\wedge}\ 1011\ 1001 \\
 \hline
 1100\ 0110
 \end{array}$$

Оператор “исключающее ИЛИ” (XOR) обладает интересным свойством, которое дает нам простой способ кодирования сообщений. Если применить операцию “исключающее ИЛИ” к некоторому значению X и заранее известному значению Y, а затем проделать то же самое с результатом предыдущей операции и значением Y, то мы снова получим значение X. Это означает, что после выполнения этих операций

```

R1 = X ^ Y;
R2 = R1 ^ Y;

```

R2 будет иметь значение X. Таким образом, результат последовательного выполнения двух операций “XOR” с использованием одного и того же значения дает исходного значение. Этот принцип можно применить для создания простой шифровальной программы, в которой некоторое целое число используется в качестве ключа для шифрования и дешифровки сообщения путем выполнения операции XOR над символами этого сообщения. Первый раз мы используем операцию XOR, чтобы закодировать сообщение, а после второго ее применения мы получа-

## 328 Модуль 7. Еще о типах данных и операторах

ем исходное (декодированное) сообщение. Рассмотрим простой пример использования этого метода для шифрования и дешифровки короткого сообщения.

```
// Использование оператора XOR для кодирования и  
// декодирования сообщения.
```

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    char msg[] = "Это простой тест";  
    char key = 88;  
  
    cout << "Исходное сообщение: " << msg << "\n";  
  
    for(int i = 0 ; i < strlen(msg); i++)  
        msg[i] = msg[i] ^ key;  
  
    cout << "Закодированное сообщение: " << msg << "\n";  
  
    for(int i = 0 ; i < strlen(msg); i++)  
        msg[i] = msg[i] ^ key;  
  
    cout << "Декодированное сообщение: " << msg << "\n";  
  
    return 0;  
}
```

Эта программа генерирует такие результаты.

Исходное сообщение: Это простой тест

Закодированное сообщение: +!Ўхў¬Ў!;Ўёх!¤!!

Декодированное сообщение: Это простой тест

Унарный оператор “НЕ” (или оператор дополнения до 1) инвертирует состояние всех битов своего операнда. Например, если целочисленное значение (хранимое в переменной A) представляет собой двоичный код 1001 0110, то в результате операции ~A получим двоичный код 0110 1001.

В следующей программе демонстрируется использование оператора “НЕ” посредством отображения некоторого числа и его дополнения до 1 в двоичном коде с помощью приведенной выше функции `show_binary()`.

```
#include <iostream>
using namespace std;

void show_binary(unsigned int u);

int main()
{
    unsigned u;

    cout << "Введите число между 0 и 255: ";
    cin >> u;

    cout << "Исходное число в двоичном коде: ";
    show_binary(u);

    cout << "Его дополнение до единицы: ";
    show_binary(~u);

    return 0;
}

// Отображение битов, составляющих байт.
void show_binary(unsigned int u)
{
    register int t;

    for(t=128; t>0; t = t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";

    cout << "\n";
}
```

Вот как выглядят результаты выполнения этой программы.

Введите число между 0 и 255: 99

Исходное число в двоичном коде: 0 1 1 0 0 0 1 1

Его дополнение до единицы: 1 0 0 1 1 1 0 0

Операторы `&`, `|` и `~` применяются непосредственно к каждому биту значения в отдельности. Поэтому поразрядные операторы нельзя использовать вместо их

логических эквивалентов в условных выражениях. Например, если значение *x* равно 7, то выражение *x && 8* имеет значение ИСТИНА, в то время как выражение *x & 8* дает значение ЛОЖЬ.

**ВАЖНО!**

## 7.8. Операторы сдвига

Операторы сдвига, “`>>`” и “`<<`”, сдвигают все биты в значении переменной вправо или влево. Общий формат использования оператора сдвига вправо выглядит так.

*переменная >> число\_битов*

А оператор сдвига влево используется так.

*переменная << число\_битов*

Здесь элемент *число\_битов* указывает, на сколько позиций должно быть сдвинуто значение элемента *переменная*. При каждом сдвиге влево все биты, составляющие значение, сдвигаются влево на одну позицию, а в младший разряд записывается нуль. При каждом сдвиге вправо все биты сдвигаются, соответственно, вправо. Если сдвигу вправо подвергается значение без знака, в старший разряд записывается нуль. Если же сдвигу вправо подвергается значение со знаком, значение знакового разряда сохраняется. Как вы помните, отрицательные целые числа представляются установкой старшего разряда числа равным единице. Таким образом, если сдвигаемое значение отрицательно, при каждом сдвиге вправо в старший разряд записывается единица, а если положительно – нуль. Не забывайте: сдвиг, выполняемый операторами сдвига, не является циклическим, т.е. при сдвиге как вправо, так и влево крайние биты теряются, и содержимое потерянного бита узнать невозможно.

Операторы сдвига работают только со значениями целочисленных типов, например, символами, целыми числами и длинными целыми числами (`int`, `char`, `long int` или `short int`). Они не применимы к значениям с плавающей точкой.

Побитовые операции сдвига могут оказаться весьма полезными для декодирования входной информации, получаемой от внешних устройств (например, цифроаналоговых преобразователей), и обработки информации о состоянии устройств. Поразрядные операторы сдвига можно также использовать для выполнения ускоренных операций умножения и деления целых чисел. С помощью сдвига влево можно эффективно умножать на два, сдвиг вправо позволяет не менее эффективно делить на два.

Следующая программа наглядно иллюстрирует результат использования операторов сдвига.

```

// Демонстрация выполнения поразрядного сдвига.

#include <iostream>
using namespace std;

void show_binary(unsigned int u);

int main()
{
    int i=1, t;

    // Сдвиг влево.
    for(t=0; t < 8; t++) {
        show_binary(i);
        i = i << 1; // ← Сдвиг влево переменной i на 1 позицию.
    }

    cout << "\n";

    // Сдвиг вправо.
    for(t=0; t < 8; t++) {
        i = i >> 1; // ← Сдвиг вправо переменной i на 1 позицию.
        show_binary(i);
    }

    return 0;
}

// Отображение битов, составляющих байт.
void show_binary(unsigned int u)
{
    int t;

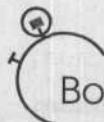
    for(t=128; t>0; t=t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";

    cout << "\n";
}

```

Результаты выполнения этой программы таковы.

```
0 0 0 0 0 0 0 1  
0 0 0 0 0 0 1 0  
0 0 0 0 0 1 0 0  
0 0 0 0 1 0 0 0  
0 0 0 1 0 0 0 0  
0 0 1 0 0 0 0 0  
0 1 0 0 0 0 0 0  
1 0 0 0 0 0 0 0  
  
1 0 0 0 0 0 0 0  
0 1 0 0 0 0 0 0  
0 0 1 0 0 0 0 0  
0 0 0 1 0 0 0 0  
0 0 0 0 1 0 0 0  
0 0 0 0 0 1 0 0  
0 0 0 0 0 0 1 0  
0 0 0 0 0 0 0 1
```



### Вопросы для текущего контроля

1. Какими символами обозначаются поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ?
2. Поразрядный оператор работает на побитовой основе. Верно ли это?
3. Покажите, как выполнить сдвиг значения переменной *x* влево на два разряда.\*

## Проект 7.1. Создание функций поразрядного циклического сдвига

rotate.cpp

Несмотря на то что язык C++ обеспечивает два оператора сдвига, в нем не определен оператор *циклического сдвига*. Оператор ци-

1. Поразрядные операторы обозначаются такими символами: &, |, ^ и ~.
2. Верно, поразрядный оператор работает на побитовой основе.
3. *x << 2*

клического сдвига отличается от оператора обычного сдвига тем, что бит, выдвигаемый с одного конца, "вдвигается" в другой. Таким образом, выдвинутые биты не теряются, а просто перемещаются "по кругу". Возможен циклический сдвиг как вправо, так и влево. Например, значение 1010 0000 после циклического сдвига влево на один разряд даст значение 0100 0001, а после сдвига вправо — число 0101 0000. В каждом случае бит, выдвинутый с одного конца, "вдвигается" в другой. Хотя отсутствие операторов циклического сдвига может показаться упущением, на самом деле это не так, поскольку их очень легко создать на основе других поразрядных операторов.

В этом проекте предполагается создание двух функций: `rrotate()` и `lrotate()`, которые сдвигают байт вправо или влево. Каждая функция принимает два параметра и возвращает результат. Первый параметр содержит значение, подлежащее сдвигу, второй — количество сдвигаемых разрядов. Этот проект включает ряд действий над битами и отображает возможность применения поразрядных операторов.

## Последовательность действий

1. Создайте файл с именем `rotate.cpp`.
2. Определите функцию `lrotate()`, предназначенную для выполнения циклического сдвига влево.

```
// Функция циклического сдвига влево байта на n разрядов.
unsigned char lrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

    for(int i=0; i < n; i++) {
        t = t << 1;

        /* Если выдвигаемый бит (8-й разряд значения t)
           содержит единицу, то устанавливаем младший бит. */
        if(t & 256)
            t = t | 1; // Устанавливаем 1 на правом конце.
    }

    return t; // Возвращаем младшие 8 бит.
}
```

Вот как работает функция `lrotate()`. Ей передается значение, которое нужно сдвинуть, в параметре `val`, и количество разрядов, на которое нужно сдвинуть, в параметре `n`. Функция присваивает значение параметра `val` переменной `t`, которая имеет тип `unsigned int`. Необходимость присваивания `unsigned char`-значения переменной типа `unsigned int` обусловлена тем, что в этом случае мы не теряем биты, выдвинутые влево. Дело в том, что бит, выдвинутый влево из байтового значения, просто становится восьмым битом целочисленного значения (поскольку его длина больше длины байта). Значение этого бита можно затем скопировать в нулевой бит байтового значения, тем самым выполнив циклический сдвиг.

В действительности циклический сдвиг выполняется следующим образом. Настраивается цикл на количество итераций, равное требуемому числу сдвигов. В теле цикла значение переменной `t` сдвигается влево на один разряд. При этом справа будет "вдинут" нуль. Но если значение восьмого бита результата (т.е. бита, который был выдвинут из байтового значения) равно единице, то и нулевой бит необходимо установить равным 1. В противном случае нулевой бит остается равным 0.

Значение восьмого бита тестируется с использованием `if`-инструкции:

```
if(t & 256)
```

Значение 256 представляет собой десятичное значение, в котором установлен только 8-й бит. Таким образом, выражение `t & 256` будет истинным лишь в случае, если в переменной `t` 8-й бит равен единице.

После выполнения циклического сдвига функция `lrotate()` возвращает значение `t`. Но поскольку она объявлена как возвращающая значение типа `unsigned char`, то фактически вернутся только младшие 8 бит значения `t`.

3. Определите функцию `rrotate()`, предназначенную для выполнения циклического сдвига вправо.

```
// Функция циклического сдвига вправо байта на n разрядов.
unsigned char rrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

    // Сначала сдвигаем значение на 8 бит влево.
    t = t << 8;
```

```

for(int i=0; i < n; i++) {
    t = t >> 1; // Сдвиг вправо на 1 разряд.

    /* После сдвига бита вправо на 1 разряд он становится
       седьмым битом целочисленного значения t. Если 7-й
       бит

       равен единице, то нужно установить крайний слева
       бит. */
    if(t & 128)
        t = t | 32768; // Устанавливаем "1" с левого кон-
       ца.
}

/* Наконец, помещаем результат назад в младшие 8 бит
   значения t. */
t = t >> 8;

return t;
}

```

Циклический сдвиг вправо немного сложнее циклического сдвига влево, поскольку значение, переданное в параметре val, должно быть первоначально сдвинуто во второй байт значения t, чтобы “не упустить” биты, сдвигаемые вправо. После завершения циклического сдвига значение необходимо сдвинуть назад в младший байт значения t, подготовив его тем самым для возврата из функции. Поскольку сдвигаемый вправо бит становится седьмым битом, то для проверки его значения используется следующая инструкция.

```
if(t & 128)
```

В десятичном значении 128 установлен только седьмой бит. Если анализируемый бит оказывается равным единице, то в значении t устанавливается 15-й бит путем применения к значению t операции “ИЛИ” с числом 32768. В результате выполнения этой операции 15-й бит значения t устанавливается равным 1, а все остальные не меняются.

4. Приведем полный текст программы, которая демонстрирует использование функций `rrotate()` и `lrotate()`. Для отображения результатов выполнения каждого циклического сдвига используется функция `show_binary()`.

### 336 Модуль 7. Еще о типах данных и операторах

```
/*
Проект 7.1.

Функции циклического сдвига вправо и влево для байтовых
значений.

*/
#include <iostream>
using namespace std;

unsigned char rrotate(unsigned char val, int n);
unsigned char lrotate(unsigned char val, int n);
void show_binary(unsigned int u);

int main()
{
    char ch = 'T';

    cout << "Исходное значение в двоичном коде:\n";
    show_binary(ch);

    cout << "Результат 8-кратного циклического сдвига вправо:\n";
    for(int i=0; i < 8; i++) {
        ch = rrotate(ch, 1);
        show_binary(ch);
    }

    cout << "Результат 8-кратного циклического сдвига влево:\n";
    for(int i=0; i < 8; i++) {
        ch = lrotate(ch, 1);
        show_binary(ch);
    }

    return 0;
}

// Функция циклического сдвига влево байта на n разрядов.
```

```

unsigned char lrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

    for(int i=0; i < n; i++) {
        t = t << 1;

        /* Если выдвигаемый бит (8-й разряд значения t)
           содержит единицу, то устанавливаем младший бит. */
        if(t & 256)
            t = t | 1; // Устанавливаем 1 на правом конце.

    }

    return t; // Возвращаем младшие 8 бит.
}

// Функция циклического сдвига вправо байта на n разрядов.
unsigned char rrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

    // Сначала сдвигаем значение на 8 бит влево.
    t = t << 8;

    for(int i=0; i < n; i++) {
        t = t >> 1; // Сдвиг вправо на 1 разряд.

        /* После сдвига бита вправо на 1 разряд он становится
           седьмым битом целочисленного значения t.
           Если 7-й бит равен единице, то нужно установить
           крайний слева бит. */
        if(t & 128)
            t = t | 32768; // Устанавливаем "1" с левого конца.

    }
}

```

### 338 Модуль 7. Еще о типах данных и операторах

```
/* Наконец, помещаем результат назад в младшие 8 бит
   значения t. */
t = t >> 8;

return t;
}

// Отображаем биты, которые составляют байт.
void show_binary(unsigned int u)
{
    int t;

    for(t=128; t>0; t = t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";

    cout << "\n";
}
```

5. Вот как выглядят результаты выполнения этой программы.

Исходное значение в двоичном коде:

0 1 0 1 0 1 0 0

Результат 8-кратного циклического сдвига вправо:

0 0 1 0 1 0 1 0  
0 0 0 1 0 1 0 1  
1 0 0 0 1 0 1 0  
0 1 0 0 0 1 0 1  
1 0 1 0 0 0 1 0  
0 1 0 1 0 0 0 1  
1 0 1 0 1 0 0 0  
0 1 0 1 0 1 0 0

Результат 8-кратного циклического сдвига влево:

1 0 1 0 1 0 0 0  
0 1 0 1 0 0 0 1  
1 0 1 0 0 0 1 0  
0 1 0 0 0 1 0 1  
1 0 0 0 1 0 1 0  
0 0 0 1 0 1 0 1  
0 0 1 0 1 0 1 0  
0 1 0 1 0 1 0 0

ВАЖНО!

## 7.9. Оператор “знак вопроса”

Одним из самых замечательных операторов C++ является оператор “?”. Оператор “?” можно использовать в качестве замены if-else-инструкций, употребляемых в следующем общем формате.

```
if (условие)
    переменная = выражение1;
else
    переменная = выражение2;
```

Здесь значение, присваиваемое переменной, зависит от результата вычисления элемента *условие*, управляющего инструкцией *if*.

Оператор “?” называется *тернарным*, поскольку он работает с тремя операндами. Вот его общий формат записи:

```
Выражение1 ? Выражение2 : Выражение3;
```

Все элементы здесь являются выражениями. Обратите внимание на использование и расположение двоеточия.

Значение ?-выражения определяется следующим образом. Вычисляется *Выражение1*. Если оно оказывается истинным, вычисляется *Выражение2*, и результат его вычисления становится значением всего ?-выражения. Если результат вычисления элемента *Выражение1* оказывается ложным, значением всего ?-выражения становится результат вычисления элемента *Выражение3*. Рассмотрим следующий пример.

```
absval = val < 0 ? -val : val; // Получение абсолютного
                                // значения переменной val.
```

Здесь переменной *absval* будет присвоено значение *val*, если значение переменной *val* больше или равно нулю. Если значение *val* отрицательно, переменной *absval* будет присвоен результат отрицания этого значения, т.е. будет получено положительное значение. Аналогичный код, но с использованием if-else-инструкции, выглядел бы так.

```
if(val < 0) absval = -val;
else absval = val;
```

А вот еще один пример практического применения оператора ?. Следующая программа делит два числа, но не допускает деления на нуль.

```
/* Эта программа использует оператор "?" для предотвращения
деления на нуль. */
```

```
#include <iostream>
using namespace std;

int div_zero();

int main()
{
    int i, j, result;

    cout << "Введите делимое и делитель: ";
    cin >> i >> j;

    // Эта инструкция не допустит возникновения ошибки
    // деления на нуль.
    result = j ? i/j : div_zero(); // Использование ?:оператора
                                    // для предотвращения
                                    // деления на нуль.

    cout << "Результат: " << result;

    return 0;
}

int div_zero()
{
    cout << "Нельзя делить на нуль.\n";
    return 0;
}
```

Здесь, если значение переменной *j* не равно нулю, выполняется деление значения переменной *i* на значение переменной *j*, а результат присваивается переменной *result*. В противном случае вызывается обработчик ошибки деления на нуль *div\_zero()*, и переменной *result* присваивается нулевое значение.

**ВАЖНО!****7.10. Оператор “запятая”**

Не менее интересным, чем описанные выше операторы, является такой оператор C++, как “запятая”. Вы уже видели несколько примеров его использования в цикле *for*, где с его помощью была организована инициализация сразу нескольких переменных. Но оператор “запятая” также может составлять часть любого выражения.

Его назначение в этом случае — связать определенным образом несколько выражений. Значение списка выражений, разделенных запятыми, определяется в этом случае значением крайнего справа выражения. Значения других выражений отбрасываются. Следовательно, значение выражения справа становится значением всего выражения-списка. Например, при выполнении этой инструкции

```
var = (count=19, incr=10, count+1);
```

переменной `count` сначала присваивается число 19, переменной `incr` — число 10, а затем к значению переменной `count` прибавляется единица, после чего переменной `var` присваивается значение крайнего справа выражения, т.е. `count+1`, которое равно 20. Круглые скобки здесь обязательны, поскольку оператор “запятая” имеет более низкий приоритет, чем оператор присваивания.

Чтобы понять назначение оператора “запятая”, попробуем выполнить следующую программу.

```
#include <iostream>
using namespace std;

int main()
{
    int i, j;

    j = 10;

    i = (j++, j+100, 999+j); // Оператор "запятая" означает
                           // "сделать это, и то, и другое".
                           // cout << i;

    return 0;
}
```

Эта программа выводит на экран число 1010. И вот почему: сначала переменной `j` присваивается число 10, затем переменная `j` инкрементируется до 11. После этого вычисляется выражение `j+100`, которое нигде не применяется. Наконец, выполняется сложение значения переменной `j` (оно по-прежнему равно 11) с числом 999, что в результате дает число 1010.

По сути, назначение оператора “запятая” — обеспечить выполнение заданной последовательности операций. Если эта последовательность используется в правой части инструкции присваивания, то переменной, указанной в ее левой

части, присваивается значение последнего выражения из списка выражений, разделенных запятыми. Оператор “запятая” по его функциональной нагрузке можно сравнить со словом “и”, используемым в фразе: “сделай это, и то, и другое...”.



### Вопросы для текущего контроля

1. Какое значение будет иметь переменная `x` после вычисления этого выражения?  
`x = 10 > 11 ? 1 : 0;`
2. Оператор “?” называется *тернарным*, поскольку он работает с \_\_\_\_\_ операндами.
3. Каково назначение оператора “запятая”?\*

## Несколько присваиваний “в одном”

Язык C++ позволяет применить очень удобный метод одновременного присваивания многим переменным одного и того же значения. Речь идет об объединении сразу нескольких присваиваний в одной инструкции. Например, при выполнении этой инструкции переменным `count`, `incr` и `index` будет присвоено число 10.

```
count = incr = index = 10;
```

Этот формат присвоения нескольким переменным общего значения можно часто встретить в профессионально написанных программах.

ВАЖНО!

### 7.11 Составные операторы присваивания

В C++ предусмотрены специальные составные операторы присваивания, в которых объединено присваивание с еще одной операцией. Начнем с примера и рассмотрим следующую инструкцию.

```
x = x + 10;
```

1. Переменная `x` после вычисления этого выражения получит значение 0.
2. Оператор “?” называется тернарным, поскольку он работает с тремя operandами.
3. Оператор “запятая” обеспечивает выполнение заданной последовательности операций.

Используя составной оператор присваивания, ее можно переписать в таком виде.

```
x += 10;
```

Пара операторов `+=` служит указанием компилятору присвоить переменной `x` сумму текущего значения переменной `x` и числа 10. Составные версии операторов присваивания существуют для всех бинарных операторов (т.е. для всех операторов, которые работают с двумя операндами). Таким образом, при таком общем формате бинарных операторов присваивания

*переменная = переменная op выражение;*

общая форма записи их составных версий выглядит так:

*переменная op = выражение;*

Здесь элемент `op` означает конкретный арифметический или логический оператор, объединяемый с оператором присваивания.

А вот еще один пример. Инструкция

```
x = x - 100;
```

аналогична такой:

```
x -= 100;
```

Обе эти инструкции присваивают переменной `x` ее прежнее значение, уменьшенное на 100.

Эти примеры служат иллюстрацией того, что составные операторы присваивания упрощают программирование определенных инструкций присваивания. Кроме того, они позволяют компилятору генерировать более эффективный код. Составные операторы присваивания можно очень часто встретить в профессионально написанных C++-программах, поэтому каждый C++-программист должен быть с ними на "ты".

#### ВАЖНО!

## 7.12 Использование ключевого слова `sizeof`

Иногда полезно знать размер (в байтах) одного из типов данных. Поскольку размеры встроенных C++-типов данных в разных вычислительных средах могут быть различными, знать заранее размер переменной во всех ситуациях не представляется возможным. Для решения этой проблемы в C++ включен оператор `sizeof` (действующий во время компиляции программы), который используется в двух следующих форматах.

```
sizeof (тип)
sizeof имя_переменной
```

Первая версия возвращает размер заданного типа данных, а вторая — размер заданной переменной. Если вам нужно узнать размер некоторого типа данных (например, `int`), заключите название этого типа в круглые скобки. Если же вас интересует размер области памяти, занимаемой конкретной переменной, можно обойтись без круглых скобок, хотя при желании их можно использовать.

Чтобы понять, как работает оператор `sizeof`, испытайте следующую короткую программу. Для многих 32-разрядных сред она должна отобразить значения 1, 4, 4 и 8.

```
// Демонстрация использования оператора sizeof.
```

```
#include <iostream>
using namespace std;

int main()
{
    char ch;
    int i;

    cout << sizeof ch << ' ';           // размер типа char
    cout << sizeof i << ' ';           // размер типа int
    cout << sizeof (float) << ' ';     // размер типа float
    cout << sizeof (double) << ' ';   // размер типа double

    return 0;
}
```

Оператор `sizeof` можно применить к любому типу данных. Например, в случае применения к массиву он возвращает количество байтов, занимаемых массивом. Рассмотрим следующий фрагмент кода.

```
int nums[4];
cout << sizeof nums; // Будет выведено число 16.
```

Для 4-байтных значений типа `int` при выполнении этого фрагмента кода на экране отобразится число 16 (которое получается в результате умножения 4 байт на 4 элемента массива).

Как упоминалось выше, оператор `sizeof` действует во время компиляции программы. Вся информация, необходимая для вычисления размера указанной переменной или заданного типа данных, известна уже во время компиляции. Оператор `sizeof` главным образом используется при написании кода, который зависит от размера C++-типов данных. Помните: поскольку размеры типов дан-

ных в C++ определяются конкретной реализацией, не стоит полагаться на размеры типов, определенные в реализации, в которой вы работаете в данный момент.



### Вопросы для текущего контроля

- Покажите, как присвоить переменным `t1`, `t2` и `t3` значение 10, используя только одну инструкцию присваивания.
- Как по-другому можно переписать эту инструкцию?  
 $x = x + 100;$
- Оператор `sizeof` возвращает размер заданной переменной или типа в   
`*`

## Сводная таблица приоритетов C++-операторов

В табл. 7.2 показан приоритет выполнения всех C++-операторов (от высшего до самого низкого). Большинство операторов ассоциированы слева направо. Но унарные операторы, операторы присваивания и оператор "?" ассоциированы справа налево. Обратите внимание на то, что эта таблица включает несколько операторов, которые мы пока не использовали в наших примерах, поскольку они относятся к объектно-ориентированному программированию (и описаны ниже).

- 
- `t1 = t2 = t3 = 10;`
  - `x += 100;`
  - Оператор `sizeof` возвращает размер заданной переменной или типа в байтах.

Таблица 7.2. Приоритет C++-операторов

|           |                                                                                                                                                                                                                           |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Наивысший | ( ) [ ] -> :: .<br>! ~ ++ -- - * & sizeof new delete typeid<br>операторы приведения типа<br>. * -> *<br>* / %<br>+ -<br><< >><br>< <= > >=<br>== !=<br>&<br>^<br> <br>&&<br>  <br>?:<br>= += -= *= /= %= >>= <<= &= ^=  = |
| Низший    | ,                                                                                                                                                                                                                         |



## Тест для самоконтроля по модулю 7

- Покажите, как объявить int-переменную с именем `test`, которую невозможно изменить в программе. Присвойте ей начальное значение 100.
- Спецификатор `volatile` сообщает компилятору о том, что значение переменной может быть изменено за пределами программы. Так ли это?
- Какой спецификатор применяется в одном из файлов многофайловой программы, чтобы сообщить об использовании глобальной переменной, объявленной в другом файле?
- Каково наиглавнейшее свойство статической локальной переменной?
- Напишите программу, содержащую функцию с именем `counter()`, которая просто подсчитывает количество вызовов. Позаботьтесь о том, чтобы функция возвращала текущее значение счетчика вызовов.
- Дан следующий фрагмент кода.

```
int myfunc()
{
    int x;
    int y;
    int z;
```

```

z = 10;
y = 0;

for(x=z; x < 15; x++)
    y += x;

return y;
}

```

Какую переменную следовало бы объявить с использованием спецификатора `register` с точки зрения получения наибольшей эффективности?

7. Чем оператор “`&`” отличается от оператора “`&&`”?
8. Какие действия выполняет эта инструкция?

```
x *= 10;
```

9. Используя функции `rrotate()` и `lrotate()` из проекта 7.1, можно закодировать и декодировать строку. Чтобы закодировать строку, выполните циклический сдвиг влево каждого символа на некоторое количество разрядов, которое задается ключом. Чтобы декодировать строку, выполните циклический сдвиг вправо каждого символа на то же самое количество разрядов. Используйте ключ, который состоит из некоторой строки символов. Существует множество способов для вычисления количества сдвигов по ключу. Проявите свои способности к творчеству. Решение, представленное в разделе ответов, является лишь одним из многих.
10. Самостоятельно расширьте возможности функции `show_binary()`, чтобы она отображала все биты значения типа `unsigned int`, а не только первые восемь.



Ответы на эти вопросы можно найти на Web-странице данной книги по адресу: <http://www.osborne.com>.



# Модуль 8

## Классы и объекты

- 8.1. Общий формат объявления класса**
- 8.2. Определение класса и создание объектов**
- 8.3. Добавление в класс функций-членов**
- 8.4. Конструкторы и деструкторы**
- 8.5. Параметризованные конструкторы**
- 8.6. Встраиваемые функции**
- 8.7. Массивы объектов**
- 8.8. Инициализация массивов объектов**
- 8.9. Указатели на объекты**

До сих пор мы писали программы, в которых не использовались какие бы то ни было C++-средства объектно-ориентированного программирования. Таким образом, программы в предыдущих модулях отражали структурированное, а не объектно-ориентированное программирование. Для написания же объектно-ориентированных программ необходимо использовать классы. Класс — это базовая C++-единица инкапсуляции. Классы используются для создания объектов. При этом классы и объекты настолько существенны для C++, что остальная часть книги в той или иной мере посвящена им.

## Основы понятия класса

Начнем с определения терминов класса и объекта. Класс можно представить как некий шаблон, который определяет формат *объекта*. Класс включает как данные, так и код, предназначенный для выполнения над этими данными. В C++ спецификация класса используется для построения объектов. Объекты — это *экземпляры* класса. По сути, класс представляет собой набор планов, которые определяют, как строить объект. Важно понимать, что класс — это логическая абстракция, которая реально не существует до тех пор, пока не будет создан объект этого класса, т.е. то, что станет физическим представлением этого класса в памяти компьютера.

Определяя класс, вы объявляете данные, которые он содержит, и код, который выполняется над этими данными. Хотя очень простые классы могут содержать только код или только данные, большинство реальных классов содержат оба компонента. В классе данные объявляются в виде переменных, а код оформляется в виде функций. Функции и переменные, составляющие класс, называются его *членами*. Таким образом, переменная, объявленная в классе, называется *членом данных*, а функция, объявленная в классе, называется *функцией-членом*. Иногда вместо термина *член данных* используется термин *переменная экземпляра* (или *переменная реализации*).

**ВАЖНО!**

### 8.1. Общий формат объявления класса

Класс создается с помощью ключевого слова `class`. Общий формат объявления класса имеет следующий вид.

```
class имя_класса {  
    закрытые данные и функции  
public:  
    открытые данные и функции  
} список_объектов;
```

Здесь элемент *имя\_класса* означает имя класса. Это имя становится именем нового типа, которое можно использовать для создания объектов класса. Объекты класса можно создать, указав их имена непосредственно за закрывающейся фигурной скобкой объявления класса (в качестве элемента *список\_объектов*), но это необязательно. После объявления класса его элементы можно создавать по мере необходимости.

Любой класс может содержать как закрытые, так и открытые члены. По умолчанию все элементы, определенные в классе, являются закрытыми. Это означает, что к ним могут получить доступ только другие члены их класса; никакие другие части программы этого сделать не могут. В этом состоит одно из проявлений инкапсуляции: программист в полной мере может управлять доступом к определенным элементам данных.

Чтобы сделать части класса открытыми (т.е. доступными для других частей программы), необходимо объявить их после ключевого слова `public`. Все переменные или функции, определенные после спецификатора `public`, доступны для всех других функций программы. Обычно в программе организуется доступ к закрытым членам класса через его открытые функции. Обратите внимание на то, что после ключевого слова `public` стоит двоеточие.

Хорошо разработанный класс должен определять одну и только одну логическую сущность, хотя такое синтаксическое правило отсутствует в "конституции" C++. Например, класс, в котором хранятся имена лиц и их телефонные номера, не стоит использовать для хранения информации о фондовой бирже, среднем количестве осадков, циклах солнечной активности и прочих не связанных с конкретными лицами данных. Основная мысль здесь состоит в том, что хорошо разработанный класс должен включать логически связанную информацию. Попытка поместить не связанные между собой данные в один класс быстро деструктуризирует ваш код!

Подведем первые итоги: в C++ класс создает новый тип данных, который можно использовать для создания объектов. В частности, класс создает логическую конструкцию, которая определяет отношения между ее членами. Объявляя переменную класса, мы создаем объект. Объект характеризуется физическим существованием и является конкретным экземпляром класса. Другими словами, объект занимает некоторую область памяти, а определение типа — нет.

#### ВАЖНО!

## 8.2. Определение класса и создание объектов

Для иллюстрации мы создадим класс, который инкапсулирует информацию о транспортных средствах (автомобилях, автофургонах и грузовиках). В этом клас-

се (назовем его `Vehicle`) будут храниться три элемента информации о транспортных средствах: количество пассажиров, которые могут там поместиться, общая вместимость топливных резервуаров (в литрах) и среднее значение расхода горючего (в километрах на литр).

Ниже представлена первая версия класса `Vehicle`. В нем определены три переменные экземпляра: `passengers`, `fuelcap` и `mpg`. Обратите внимание на то, что класс `Vehicle` не содержит ни одной функции. Поэтому пока его можно считать классом данных. (В следующих разделах мы дополним его функциями.)

```
class Vehicle {
public:
    int passengers; // количество пассажиров
    int fuelcap; // вместимость топливных резервуаров
                  // в литрах
    int mpg; // расход горючего в километрах на литр
};
```

Переменные экземпляра, определенные в классе `Vehicle`, иллюстрируют общий формат их объявления. Итак, формат объявления переменной экземпляра таков: *тип имя\_переменной*;

Здесь элемент *тип* определяет тип переменной экземпляра, а элемент *имя\_переменной* — ее имя. Таким образом, переменная экземпляра объявляется так же, как локальная переменная. В классе `Vehicle` все переменные экземпляра объявлены с использованием модификатора доступа `public`, который, как упоминалось выше, позволяет получать к ним доступ со стороны кода, расположенного даже вне класса `Vehicle`.

Определение `class` создает новый тип данных. В данном случае этот новый тип данных называется `Vehicle`. Это имя можно использовать для объявления объектов типа `Vehicle`. Помните, что объявление `class` — это лишь описание типа; оно не создает реальных объектов. Таким образом, предыдущий код не означает существования объектов типа `Vehicle`.

Чтобы реально создать объект класса `Vehicle`, используйте, например, такую инструкцию:

```
Vehicle minivan; // Создаем объект minivan типа Building.
```

После выполнения этой инструкции объект `minivan` станет экземпляром класса `Vehicle`, т.е. обретет “физическую” реальность.

При каждом создании экземпляра класса создается объект, который содержит собственную копию каждой переменной экземпляра, определенной этим классом. Таким образом, каждый объект класса `Vehicle` будет содержать собственные копии переменных экземпляра `passengers`, `fuelcap` и `mpg`. Для доступа

к этим переменным используется *оператор "точка"* (.). Оператор "точка" связывает имя объекта с именем его члена. Общий формат этого оператора имеет такой вид:

**объект.член**

Как видите, объект указывается слева от оператора "точка", а его член — справа. Например, чтобы присвоить переменной fuelcap объекта minivan значение 16, используйте следующую инструкцию.

```
minivan.fuelcap = 16;
```

В общем случае оператор "точка" можно использовать для доступа как к переменным экземпляров, так и к функциям.

Теперь рассмотрим программу, в которой используется класс Vehicle.

```
// Программа, в которой используется класс Vehicle.
```

```
#include <iostream>
using namespace std;

// Объявление класса Vehicle.
class Vehicle {
public:
    int passengers; // количество пассажиров
    int fuelcap; // вместимость топливных резервуаров
                  // в литрах
    int mpg; // расход горючего в километрах на литр
};

int main() {
    Vehicle minivan; // Создание объекта minivan (экземпляра
                      // класса Vehicle).
    int range;

    // Присваиваем значения полям в объекте minivan.
    minivan.passengers = 7; // Обратите внимание на использо-
    вание
    minivan.fuelcap = 16; // оператора "точка" для доступа к
    minivan.mpg = 21; // членам объекта.

    // Вычисляем расстояние, которое может проехать транспортное
    // средство после заливки полного бака горючего.
```

## 354 Модуль 8. Классы и объекты

```
range = minivan.fuelcap * minivan.mpg;

cout << "Минифургон может перевезти " << minivan.passengers
    << " пассажиров на расстояние " << range
    << " километров." << "\n";

return 0;
}
```

Рассмотрим внимательно эту программу. В функции `main()` мы сначала создаем экземпляр класса `Vehicle` с именем `minivan`, а затем осуществляем доступ к переменным этого экземпляра `minivan`, присваивая им конкретные значения и используя эти значения в вычислениях. Функция `main()` получает доступ к членам класса `Vehicle`, поскольку они объявлены открытыми, т.е. `public`-членами. Если бы в их объявлении не было спецификатора доступа `public`, доступ к ним ограничивался бы рамками класса `Vehicle`, а функция `main()` не имела бы возможности использовать их.

При выполнении этой программы получаем такие результаты:

```
Минифургон может перевезти 7 пассажиров на расстояние 336
километров.
```

Прежде чем идти дальше, имеет смысл вспомнить основной принцип программирования классов: каждый объект класса имеет собственные копии переменных экземпляра, определенных в этом классе. Таким образом, содержимое переменных в одном объекте может отличаться от содержимого аналогичных переменных в другом. Между двумя объектами нет связи, за исключением того, что они являются объектами одного и того же типа. Например, если у вас есть два объекта типа `Vehicle` и каждый объект имеет свою копию переменных `passengers`, `fuelcap` и `mpg`, то содержимое соответствующих (одноименных) переменных этих двух экземпляров может быть разным. Следующая программа демонстрирует это.

```
// Эта программа создает два объекта класса Vehicle.
```

```
#include <iostream>
using namespace std;

// Объявление класса Vehicle.
class Vehicle {
public:
    int passengers; // количество пассажиров
    int fuelcap; // вместимость топливных резервуаров
```

```

        // в литрах
int mpg;           // расход горючего в километрах на литр
};

int main() {
    Vehicle minivan; // Создаем первый объект типа Vehicle.
    Vehicle sportscar; // Создаем второй объект типа Vehicle.
    // Теперь объекты minivan и sportscar имеют собственные
    // копии переменных реализации класса Vehicle.

    int range1, range2;

    // Присваиваем значения полям объекта minivan.
    minivan.passengers = 7;
    minivan.fuelcap = 16;
    minivan.mpg = 21;

    // Присваиваем значения полям объекта sportscar.
    sportscar.passengers = 2;
    sportscar.fuelcap = 14;
    sportscar.mpg = 12;

    // Вычисляем расстояние, которое может проехать каждое
    // транспортное средство после заливки полного бака
    // горючего.
    range1 = minivan.fuelcap * minivan.mpg;
    range2 = sportscar.fuelcap * sportscar.mpg;

    cout << "Минифургон может перевезти " << minivan.passengers
        << " пассажиров на расстояние " << range1
        << " километров." << "\n";

    cout << "Спортивный автомобиль может перевезти "
        << sportscar.passengers
        << " пассажира на расстояние " << range2
        << " километров." << "\n";

    return 0;
}

```

## 356 Модуль 8. Классы и объекты

Эта программа генерирует такие результаты.

Минифургон может перевезти 7 пассажиров на расстояние 336 километров.

Спортивный автомобиль может перевезти 2 пассажира на расстояние 168 километров.

Как видите, данные о минифургоне (содержащиеся в объекте `minivan`) совершенно не связаны с данными о спортивном автомобиле (содержащимися в объекте `sportscar`). Эта ситуация отображена на рис. 8.1.

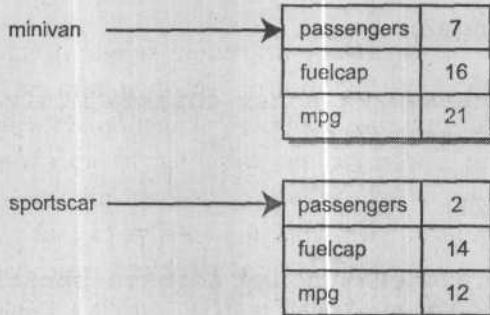
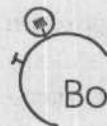


Рис. 8.1. Переменные одного экземпляра не связаны с переменными другого экземпляра



### Вопросы для текущего контроля

1. Какие два компонента может содержать класс?
2. Какой оператор используется для доступа к членам класса посредством объекта?
3. Каждый объект имеет собственные копии \_\_\_\_\_.\*

- 
1. Класс может содержать данные и код.
  2. Для доступа к членам класса посредством объекта используется оператор “точка”.
  3. Каждый объект имеет собственные копии переменных экземпляра.

ВАЖНО!

8.3.

## Добавление в класс функций-членов

Пока класс `Vehicle` содержит только данные и ни одной функции. И хотя такие “чисто информационные” классы вполне допустимы, большинство классов все-таки имеют функции-члены. Как правило, функции-члены класса обрабатывают данные, определенные в этом классе, и во многих случаях обеспечивают доступ к этим данным. Другие части программы обычно взаимодействуют с классом через его функции.

Чтобы проиллюстрировать использование функций-членов, добавим функцию-член в класс `Vehicle`. Вспомните, что функция `main()` вычисляет расстояние, которое может проехать заданное транспортное средство после заливки полного бака горючего, путем умножения вместимости топливных резервуаров (в литрах) на расход горючего (в километрах на литр). Несмотря на формальную корректность, эти вычисления выполнены не самым удачным образом. Вычисление этого расстояния лучше бы предоставить самому классу `Vehicle`, поскольку обе величины, используемые в вычислении, инкапсулированы классом `Vehicle`. Внеся в класс `Vehicle` функцию, которая вычисляет произведение членов данных этого класса, мы значительно улучшим его объектно-ориентированную структуру.

Чтобы добавить функцию в класс `Vehicle`, необходимо задать ее прототип внутри объявления этого класса. Например, следующая версия класса `Vehicle` содержит функцию с именем `range()`, которая возвращает значение максимального расстояния, которое может проехать заданное транспортное средство после заливки полного бака горючего.

```
// Новое объявление класса Vehicle.
class Vehicle {
public:
    int passengers; // количество пассажиров
    int fuelcap; // вместимость топливных резервуаров
                  // в литрах
    int mpg; // расход горючего в километрах на литр

    // Объявление функции-члена range().
    int range(); // функция вычисления максимального пробега
};
```

Поскольку функции-члены обеспечены своими прототипами в определении класса (в данном случае речь идет о функции-члене `range()`), их не нужно помещать больше ни в какое другое место программы.

Чтобы реализовать функцию, которая является членом класса, необходимо сообщить компилятору, какому классу она принадлежит, квалифицировав имя этой функции с именем класса. Например, вот как можно записать код функции `range()`.

```
// Реализация функции-члена range().
int Vehicle::range() {
    return mpg * fuelcap;
}
```

Обратите внимание на оператор, обозначаемый символом “`::`”, который отделяет имя класса `Vehicle` от имени функции `range()`. Этот оператор называется *оператором разрешения области видимости* или *оператором разрешения контекста*. Он связывает имя класса с именем его члена, чтобы сообщить компилятору, какому классу принадлежит данная функция. В данном случае он связывает функцию `range()` с классом `Vehicle`. Другими словами, оператор `::` заявляет о том, что функция `range()` находится в области видимости класса `Vehicle`. Это очень важно, поскольку различные классы могут использовать одинаковые имена функций. Компилятор же определит, к какому классу принадлежит функция, именно с помощью оператора разрешения области видимости и имени класса.

Тело функции `range()` состоит из одной строки.

```
return mpg * fuelcap;
```

Эта инструкция возвращает значение расстояния как результат умножения значений переменных `mpg` и `fuelcap`. Поскольку каждый объект типа `Vehicle` имеет собственную копию переменных `mpg` и `fuelcap`, при выполнении функции `range()` используются копии этих переменных, принадлежащих вызывающему объекту.

В теле функции `range()` обращение к переменным `mpg` и `fuelcap` происходит напрямую, т.е. без указания имени объекта и оператора “точка”. Если функция-член использует переменную экземпляра, которая определена в том же классе, не нужно делать явную ссылку на объект (с участием оператора “точка”). Ведь компилятор в этом случае уже точно знает, какой объект подвергается обработке, поскольку функция-член всегда вызывается относительно некоторого объекта “ее” класса. Коль вызов функции произошел, значит, объект известен. Поэтому в функции-члене нет необходимости указывать объект еще раз. Следовательно, члены `mpg` и `fuelcap` в функции `range()` неявно обращаются к копиям этих переменных, относящихся к объекту, который вызвал функцию `range()`. Одна-

ко код, расположенный вне класса `Vehicle`, должен обращаться к переменным `mpg` и `fuelcap`, используя имя объекта и оператор “точка”.

Функции-члены можно вызывать только относительно заданного объекта. Это можно реализовать одним из двух способов. Во-первых, функцию-член может вызвать код, расположенный вне класса. В этом случае (т.е. при вызове функции-члена из части программы, которая находится вне класса) необходимо использовать имя объекта и оператор “точка”. Например, при выполнении этого кода будет вызвана функция `range()` для объекта `minivan`.

```
rangel = minivan.range();
```

Выражение `minivan.range()` означает вызов функции `range()`, которая будет обрабатывать копии переменных объекта `minivan`. Поэтому функция `range()` возвратит максимальную дальность пробега для объекта `minivan`.

Во-вторых, функцию-член можно вызвать из другой функции-члена того же класса. В этом случае вызов также происходит напрямую, без оператора “точка”, поскольку компилятору уже известно, какой объект подвергается обработке. И только тогда, когда функция-член вызывается кодом, который не принадлежит классу, необходимо использовать имя объекта и оператор “точка”.

Использование класса `Vehicle` и функции `range()` демонстрируется в приведенной ниже программе (для этого объединены все уже знакомые вам части кода и добавлены недостающие детали).

```
// Программа использования класса Vehicle.

#include <iostream>
using namespace std;

// Объявление класса Vehicle.
class Vehicle {
public:
    int passengers; // количество пассажиров
    int fuelcap; // вместимость топливных резервуаров
                  // в литрах
    int mpg; // расход горючего в километрах на литр

    // Объявление функции-члена range().
    int range(); // функция вычисления максимального пробега
};

// Реализация функции-члена range().
int Vehicle::range() {
```

## 360 Модуль 8. Классы и объекты

```
    return mpg * fuelcap;
}

int main() {
    Vehicle minivan; // Создаем первый объект типа Vehicle.
    Vehicle sportscar; // Создаем второй объект типа Vehicle

    int range1, range2;

    // Присваиваем значения полям объекта minivan.
    minivan.passengers = 7;
    minivan.fuelcap = 16;
    minivan.mpg = 21;

    // Присваиваем значения полям объекта sportscar.
    sportscar.passengers = 2;
    sportscar.fuelcap = 14;
    sportscar.mpg = 12;

    // Вычисляем расстояние, которое может проехать каждое
    // транспортное средство после заливки полного бака
    // горючего.
    range1 = minivan.range(); // Вызов функции range() для
    range2 = sportscar.range(); // объектов класса Vehicle.

    cout << "Минифургон может перевезти " << minivan.passengers
        << " пассажиров на расстояние " << range1
        << " километров." << "\n";

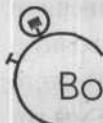
    cout << "Спортивный автомобиль может перевезти "
        << sportscar.passengers
        << " пассажира на расстояние " << range2
        << " километров." << "\n";

    return 0;
}
```

При выполнении программа отображает такие результаты.

Минифургон может перевезти 7 пассажиров на расстояние 336 километров.

Спортивный автомобиль может перевезти 2 пассажира на расстояние 168 километров.



## Вопросы для текущего контроля

1. Как называется оператор “::”?
2. Каково назначение оператора “::”?
3. Если функция-член вызывается извне ее класса, то обращение к ней должно быть организовано с использованием имени объекта и оператора “точка”. Правда ли это?\*

### Проект 8.1 Создание класса справочной информации

`HelpClass.cpp`

Если бы мы попытались резюмировать суть класса одним предложением, оно приблизительно прозвучало бы так: класс инкапсулирует функциональность. Иногда главное – знать, где заканчивается одна “функциональность” и начинается другая. Как правило, назначение классов – быть строительными блоками “большого” приложения. Для этого каждый класс должен представлять одну функциональную “единицу”, которая выполняет четко очерченные действия. Таким образом, свои классы имеет смысл делать настолько мелкими, насколько это возможно. Другими словами, классы, которые содержат постороннюю информацию, дезорганизуют и деструктуризируют код, но классы, которые содержат недостаточную функциональность, можно упрекнуть в раздробленности. Истина, как известно, лежит посередине. Соблюсти баланс – после решения именно этой сверхзадачи *наука* программирования становится *искусством* программирования. К счастью, большинство программистов считают, что с опытом находить этот самый баланс становится все легче и легче.

Итак, начнем приобретать опыт. Для этого попытаемся преобразовать справочную систему из проекта 3.3 (см. модуль 3) в класс справочной информации. Почему эта идея заслуживает внимания? Во-первых, справочная система определяет одну

Проект  
8.1

Создание класса справочной информации

1. Оператор “::” называется оператором разрешения области видимости.
2. Оператор “::” связывает имя класса с именем его члена.
3. Правда. При вызове функции-члена извне ее класса необходимо указывать имя объекта и использовать оператор “точка”.

логическую “единицу”. Она просто отображает синтаксис C++-инструкций управления. Таким образом, ее функциональность компактна и хорошо определена. Во-вторых, оформление справочной системы в виде класса кажется удачным решением даже с эстетической точки зрения. Ведь для того, чтобы предложить справочную систему любому пользователю, достаточно реализовать конкретный объект этого класса. Наконец, поскольку при “классовом” подходе к справочной системе мы соблюдаем принцип инкапсуляции, то любое ее усовершенствование или изменение не вызовет нежелательных побочных эффектов в программе, которая ее использует.

### Последовательность действий

1. Создайте новый файл с именем HelpClass.cpp. Скопируйте файл из проекта 3.3, Help3.cpp, в файл HelpClass.cpp.
2. Чтобы преобразовать справочную систему в класс, сначала необходимо точно определить ее составляющие. Например, файл Help3.cpp содержит код, который отображает меню, вводит команду от пользователя, проверяет ее допустимость и отображает информацию о выбранной инструкции. Кроме того, в программе создан цикл, выход из которого организован после ввода пользователем буквы “q”. Очевидно, что меню, проверка допустимости введенного запроса и отображение запрошенной информации являются неотъемлемыми частями справочной системы, чего нельзя сказать о приеме команды от пользователя. Таким образом, можно уверенно создать класс, который будет отображать справочную информацию, меню и проверять допустимость введенного запроса. Назовем функции, “отвечающие” за эти действия, helpon(), showmenu() и isvalid() соответственно.
3. Объявите класс Help таким образом.

```
// Класс, который инкапсулирует справочную систему.
class Help {
public:
    void helpon(char what);
    void showmenu();
    bool isvalid(char ch);
};
```

Обратите внимание на то, что этот класс включает лишь функции. В нем нет ни одной переменной реализации. Как упоминалось выше, класс такого вида (как и класс без функций) вполне допустим. ( В вопросе 9 раздела “Тест для самоконтроля по модулю 8” вам предлагается внести в класс Help переменную экземпляра.)

## 4. Создайте функцию helpon().

```
// Отображение справочной информации.
void Help::helpon(char what) {
    switch(what) {
        case '1':
            cout << "Инструкция if:\n\n";
            cout << "if(условие) инструкция;\n";
            cout << "else инструкция;\n";
            break;
        case '2':
            cout << "Инструкция switch:\n\n";
            cout << "switch(выражение) {\n";
            cout << "    case константа:\n";
            cout << "        последовательность инструкций\n";
            cout << "        break;\n";
            cout << "    // ...\n";
            cout << "}\n";
            break;
        case '3':
            cout << "Инструкция for:\n\n";
            cout << "for(инициализация; условие; инкремент) ";
            cout << " инструкция;\n";
            break;
        case '4':
            cout << "Инструкция while:\n\n";
            cout << "while(условие) инструкция;\n";
            break;
        case '5':
            cout << "Инструкция do-while:\n\n";
            cout << "do {\n";
            cout << "    инструкция;\n";
            cout << "} while (условие);\n";
            break;
        case '6':
            cout << "Инструкция break:\n\n";
            cout << "break;\n";
            break;
        case '7':
            cout << "Инструкция continue:\n\n";
    }
}
```

## 364 Модуль 8. Классы и объекты

```
        cout << "continue;\n";
        break;
    case '8':
        cout << "Инструкция goto:\n\n";
        cout << "goto метка;\n";
        break;
    }
    cout << "\n";
}
```

5. Создайте функцию showmenu().

```
// Отображение меню справочной системы.
void Help::showmenu() {
    cout << "Справка по инструкциям:\n";
    cout << " 1. if\n";
    cout << " 2. switch\n";
    cout << " 3. for\n";
    cout << " 4. while\n";
    cout << " 5. do-while\n";
    cout << " 6. break\n";
    cout << " 7. continue\n";
    cout << " 8. goto\n";
    cout << "Выберите вариант справки (q для выхода): ";
}
```

6. Создайте функцию isvalid().

```
// Проверка допустимости команды пользователя (функция
// возвращает значение ИСТИНА, если команда допустима).
bool Help::isvalid(char ch) {
    if(ch < '1' || ch > '8' && ch != 'q')
        return false;
    else
        return true;
}
```

7. Перепишите функцию main() из проекта 3.3 так, чтобы она использовала класс Help. Вот как выглядит полный текст файла HelpClass.cpp.

```
/*
Проект 8.1.
```

Преобразование справочной системы из проекта 3.3 в

```

    класс Help.

*/
#include <iostream>
using namespace std;

// Класс, который инкапсулирует справочную систему.
class Help {
public:
    void helpon(char what);
    void showmenu();
    bool isvalid(char ch);
};

// Отображение справочной информации.
void Help::helpon(char what) {
    switch(what) {
        case '1':
            cout << "Инструкция if:\n\n";
            cout << "if(условие) инструкция;\n";
            cout << "else инструкция;\n";
            break;
        case '2':
            cout << "Инструкция switch:\n\n";
            cout << "switch(выражение) {\n";
            cout << "    case константа:\n";
            cout << "        последовательность инструкций\n";
            cout << "        break;\n";
            cout << "    // ...\n";
            cout << "}\n";
            break;
        case '3':
            cout << "Инструкция for:\n\n";
            cout << "for(инициализация; условие; инкремент) ";
            cout << "    инструкция;\n";
            break;
        case '4':
            cout << "Инструкция while:\n\n";
            cout << "while(условие) инструкция;\n";
            break;
    }
}

```

## 366 Модуль 8. Классы и объекты

```
case '5':  
    cout << "Инструкция do-while:\n\n";  
    cout << "do {\n";  
    cout << "    инструкция;\n";  
    cout << "} while (условие);\n";  
    break;  
case '6':  
    cout << "Инструкция break:\n\n";  
    cout << "break;\n";  
    break;  
case '7':  
    cout << "Инструкция continue:\n\n";  
    cout << "continue;\n";  
    break;  
case '8':  
    cout << "Инструкция goto:\n\n";  
    cout << "goto метка;\n";  
    break;  
}  
cout << "\n";  
}  
  
// Отображение меню справочной системы.  
void Help::showmenu() {  
    cout << "Справка по инструкциям:\n";  
    cout << " 1. if\n";  
    cout << " 2. switch\n";  
    cout << " 3. for\n";  
    cout << " 4. while\n";  
    cout << " 5. do-while\n";  
    cout << " 6. break\n";  
    cout << " 7. continue\n";  
    cout << " 8. goto\n";  
    cout << "Выберите вариант справки (q для выхода): ";  
}  
  
// Проверка допустимости команды пользователя (функция  
// возвращает значение ИСТИНА, если команда допустима).  
bool Help::isValid(char ch) {  
    if(ch < '1' || ch > '8' && ch != 'q')  
        return false;  
    return true;  
}
```

```

        return false;
    else
        return true;
}

int main()
{
    char choice;
    Help hlpob; // Создаем экземпляр класса Help.

    // Используем объект класса Help для отображения информации.
    for(;;) {
        do {
            hlpob.showmenu();
            cin >> choice;
        } while(!hlpob.isvalid(choice));

        if(choice == 'q') break;
        cout << "\n";

        hlpob.helpon(choice);
    }

    return 0;
}

```

Опробовав эту программу, нетрудно убедиться, что она функционирует подобно той, которая представлена в модуле 3. Преимущество этого подхода состоит в том, что теперь у вас есть компонент справочной системы, который при необходимости можно использовать во многих других приложениях.

**ВАЖНО!****8.4. Конструкторы и деструкторы**

В предыдущих примерах переменные каждого Vehicle-объекта устанавливались "вручную" с помощью следующей последовательности инструкций:

```

minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;

```

Профессионал никогда бы не использовал подобный подход. И дело не столько в том, что таким образом можно попросту “забыть” об одном или нескольких данных, сколько в том, что существует гораздо более удобный способ это сделать. Этот способ – использование конструктора.

Конструктор инициализирует объект при его создании. Он имеет такое же имя, что и сам класс, и синтаксически подобен функции. Однако в определении конструкторов не указывается тип возвращаемого значения. Формат записи конструктора такой:

```
имя_класса() {  
    // код конструктора  
}
```

Обычно конструктор используется, чтобы придать переменным экземпляра, определенным в классе, начальные значения или выполнить исходные действия, необходимые для создания полностью сформированного объекта.

Дополнением к конструктору служит *деструктор*. Во многих случаях при разрушении объекту необходимо выполнить некоторое действие или даже некоторую последовательность действий. Локальные объекты создаются при входе в блок, в котором они определены, и разрушаются при выходе из него. Глобальные объекты разрушаются при завершении программы. Существует множество факторов, обусловливающих необходимость деструктора. Например, объект должен освободить ранее выделенную для него память или закрыть ранее открытый для него файл. В C++ именно деструктору поручается обработка процессаdezактивизации объекта. Имя деструктора совпадает с именем конструктора, но предваряется символом “~”. Подобно конструкторам деструкторы не возвращают значений, а следовательно, в их объявлениях отсутствует тип возвращаемого значения.

Рассмотрим простой пример, в котором используется конструктор и деструктор.

```
// Использование конструктора и деструктора.
```

```
#include <iostream>  
using namespace std;  
  
class MyClass {  
public:  
    int x;  
  
    // Объявляем конструктор и деструктор для класса MyClass.
```

```

MyClass(); // конструктор
~MyClass(); // деструктор
};

// Реализация конструктора класса MyClass.
MyClass::MyClass() {
    x = 10;
}

// Реализация деструктора класса MyClass.
MyClass::~MyClass() {
    cout << "Разрушение объекта...\n";
}

int main() {
    MyClass ob1;
    MyClass ob2;

    cout << ob1.x << " " << ob2.x << "\n";

    return 0;
}

```

При выполнении эта программа генерирует такие результаты.

```

10 10
Разрушение объекта...
Разрушение объекта...

```

В этом примере конструктор для класса MyClass выглядит так.

```

// Реализация конструктора класса MyClass.
MyClass::MyClass() {
    x = 10;
}

```

Обратите внимание на `public`-определение конструктора, которое позволяет вызывать его из кода, определенного вне класса MyClass. Этот конструктор присваивает переменной экземпляра `x` значение 10. Конструктор `MyClass()` вызывается при создании объекта класса MyClass. Например, при выполнении строки

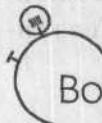
```
MyClass ob1;
```

для объекта `ob1` вызывается конструктор `MyClass()`, который присваивает переменной экземпляра `ob1.x` значение 10. То же самое справедливо и в отношении объекта `ob2`, т.е. в результате создания объекта `ob2` значение переменной экземпляра `ob2.x` также станет равным 10.

Деструктор для класса `MyClass` имеет такой вид.

```
// Реализация деструктора класса MyClass.  
MyClass::~MyClass() {  
    cout << "Разрушение объекта...\n";  
}
```

Этот деструктор попросту отображает сообщение, но в реальных программах деструктор используется для освобождения одного или нескольких ресурсов (файловых дескрипторов или памяти), используемых классом.



### Вопросы для текущего контроля

---

1. Что представляет собой конструктор и в каком случае он выполняется?
  2. Имеет ли конструктор тип возвращаемого значения?
  3. В каком случае вызывается деструктор?\*
- 

ВАЖНО!

## 8.5. Параметризованные конструкторы

В предыдущем примере использовался конструктор без параметров. Но чаще приходится иметь дело с конструкторами, которые принимают один или несколько параметров. Параметры вносятся в конструктор точно так же, как в функцию: для этого достаточно объявить их внутри круглых скобок после имени конструктора. Например, вот как мог бы выглядеть параметризованный конструктор для класса `MyClass`.

```
MyClass::MyClass(int i) {  
    x = i;  
}
```

- 
1. Конструктор – это функция, которая выполняется при создании объекта. Конструктор используется для инициализации создаваемого объекта.
  2. Нет, конструктор не имеет типа возвращаемого значения.
  3. Деструктор вызывается при разрушении объекта.

Чтобы передать аргумент конструктору, необходимо связать этот аргумент с объектом при объявлении объекта. C++ поддерживает два способа реализации такого связывания. Вот как выглядит первый способ.

```
MyClass ob1 = MyClass(101);
```

В этом объявлении создается объект класса `MyClass` (с именем `ob1`), которому передается значение 101. Но эта форма (в таком контексте) используется редко, поскольку второй способ имеет более короткую запись и удобнее для использования. Во втором способе аргумент (или аргументы) должен следовать за именем объекта и заключаться в круглые скобки. Например, следующая инструкция эквивалентна предыдущему объявлению.

```
MyClass ob1(101);
```

Это самый распространенный способ объявления параметризованных объектов. Опираясь на этот метод, приведем общий формат передачи аргументов конструкторам.

*тип\_класса имя\_переменной(список\_аргументов);*

Здесь элемент `список_аргументов` представляет собой список разделенных запятыми аргументов, передаваемых конструктору.



Формально между двумя приведенными выше формами инициализации существует небольшое различие, которое вы поймете при дальнейшем чтении этой книги. Но это различие не влияет на результаты выполнения программ, представленных в этом модуле, и большинства других программ.

Рассмотрим полную программу, в которой демонстрируется использование параметризованного конструктора класса `MyClass`.

```
// Использование параметризованного конструктора.
```

```
#include <iostream>
using namespace std;

class MyClass {
public:
    int x;

    // Объявляем конструктор и деструктор.
    MyClass(int i); // конструктор (добавляем параметр)
    ~MyClass(); // деструктор
```

## 372 Модуль 8. Классы и объекты

```
};

// Реализуем параметризованный конструктор.
 MyClass::MyClass(int i) {
    x = i;
}

// Реализуем деструктор класса MyClass.
 MyClass::~MyClass() {
    cout << "Разрушение объекта, в котором значение x равно "
        << x << "\n";
}

int main() {
    MyClass t1(5); // Передаем аргументы в конструктор MyClass().
    MyClass t2(19);

    cout << t1.x << " " << t2.x << "\n";

    return 0;
}
```

Результаты выполнения этой программы таковы.

5 19

Разрушение объекта, в котором значение x равно 19

Разрушение объекта, в котором значение x равно 5

В этой версии программы конструктор MyClass() определяет один параметр, именуемый i, который используется для инициализации переменной реализации x. Таким образом, при выполнении строки кода

MyClass ob1(5);

значение 5 передается параметру i, а затем присваивается переменной реализации x.

В отличие от конструкторов, деструкторы не могут иметь параметров. Причину понять нетрудно: не существует средств передачи аргументов объекту, который разрушается. Если же у вас возникнет та редкая ситуация, когда при вызове деструктора вашему объекту необходимо передать некоторые данные, определяемые только во время выполнения программы, создайте для этой цели специальную переменную. Затем непосредственно перед разрушением объекта установите эту переменную равной нужному значению.

## Добавление конструктора в класс Vehicle

Мы можем улучшить класс Vehicle, добавив в него конструктор, который при создании объекта автоматически инициализирует поля (т.е. переменные экземпляра) passengers, fuelcap и mpg. Обратите особое внимание на то, как создаются объекты класса Vehicle.

```
// Добавление конструктора в класс vehicle.

#include <iostream>
using namespace std;

// Объявление класса Vehicle.
class Vehicle {
public:
    int passengers; // количество пассажиров
    int fuelcap; // вместимость топливных резервуаров в литрах
    int mpg; // расход горючего в километрах на литр

    // Объявление конструктора для класса Vehicle.
    Vehicle(int p, int f, int m);

    int range(); // функция вычисления максимального пробега
};

// Реализация конструктора для класса Vehicle.
Vehicle::Vehicle(int p, int f, int m) { // Этот конструктор
    passengers = p; // инициализирует члены данных
    fuelcap = f; // passengers, fuelcap и mpg.
    mpg = m;
}

// Реализация функции-члена range().
int Vehicle::range() {
    return mpg * fuelcap;
}

int main() {
    // Передаем данные о транспортном средстве
    // конструктору Vehicle().
    Vehicle minivan(7, 16, 21);
```

```
Vehicle sportscar(2, 14, 12);

int range1, range2;

// Вычисляем расстояние, которое может проехать каждое
// транспортное средство после заливки полного бака
// горючего.
range1 = minivan.range();
range2 = sportscar.range();

cout << "Минифургон может перевезти "
    << minivan.passengers
    << " пассажиров на расстояние " << range1
    << " километров." << "\n";

cout << "Спортивный автомобиль может перевезти "
    << sportscar.passengers
    << " пассажира на расстояние " << range2
    << " километров." << "\n";

return 0;
}
```

Оба объекта, `minivan` и `sportscar`, в момент создания инициализируются в программе конструктором `Vehicle()`. Каждый объект инициализируется в соответствии с тем, как заданы параметры, передаваемые конструктору. Например, при выполнении строки

```
Vehicle minivan(7, 16, 21);
```

конструктору `Vehicle()` передаются значения 7, 16 и 21. В результате этого копии переменных `passengers`, `fuelcap` и `mpg`, принадлежащие объекту `minivan`, будут содержать значения 7, 16 и 21 соответственно.

## Альтернативный вариант инициализации объекта

Если конструктор принимает только один параметр, можно использовать альтернативный способ инициализации членов объекта. Рассмотрим следующую программу.

```
// Альтернативный вариант инициализации объекта.
```

```
#include <iostream>
```

```

using namespace std;

class MyClass {
public:
    int x;

    // Объявляем конструктор и деструктор.
    MyClass(int i); // конструктор
    ~MyClass(); // деструктор
};

// Реализуем параметризованный конструктор.
MyClass::MyClass(int i) {
    x = i;
}

// Реализуем деструктор класса MyClass.
MyClass::~MyClass() {
    cout << "Разрушение объекта, в котором значение x равно "
       << x << "\n";
}

int main() {
    MyClass ob = 5; // Вызывается конструктор MyClass(5).

    cout << ob.x << "\n";

    return 0;
}

```

Здесь конструктор для объектов класса `MyClass` принимает только один параметр. Обратите внимание на то, как в функции `main()` объявляется объект `ob`. Для этого используется такой формат объявления:

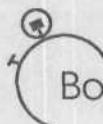
```
MyClass ob = 5;
```

В этой форме инициализации объекта число 5 автоматически передается параметру `i` при вызове конструктора `MyClass()`. Другими словами, эта инструкция объявления обрабатывается компилятором так, как если бы она была записана следующим образом.

```
myclass ob (5);
```

В общем случае, если у вас есть конструктор, который принимает только один аргумент, для инициализации объекта вы можете использовать либо вариант `ob(x)`, либо вариант `ob = x`. Дело в том, что при создании конструктора с одним аргументом неявно создается преобразование из типа этого аргумента в тип этого класса.

Помните, что показанный здесь альтернативный способ инициализации объектов применяется только к конструкторам, которые имеют лишь один параметр.



### Вопросы для текущего контроля

1. Покажите, как для класса с именем `Test` объявить конструктор, который принимает один `int`-параметр `count`?
2. Как можно переписать эту инструкцию?

```
Test ob = Test(10);
```

3. Как еще можно переписать объявление из второго вопроса?\*

### Спросим у опытного программиста

**Вопрос.** *Можно ли один класс объявить внутри другого?*

**Ответ.** Да, можно. В этом случае создается **вложенный класс**. Поскольку объявление класса, по сути, определяет область видимости, вложенный класс действителен только в рамках области видимости внешнего класса. Справедливости ради хочу сказать, что благодаря богатству и гибкости других средств C++ (например, наследованию), о которых речь еще впереди, необходимости в создании вложенных классов практически не существует.

**ВАЖНО!**

### 8.6. Встраиваемые функции

Прежде чем мы продолжим освоение класса, сделаем небольшое, но важное отступление. Оно не относится конкретно к объектно-ориентированному программированию, но является очень полезным средством C++, которое довольно часто

1. `Test::Test(int count) { ... }`
2. `Test ob(10);`
3. `Test ob = 10;`

используется в определениях классов. Речь идет о *встраиваемой*, или *подставляемой*, функции (*inline function*). Встраиваемой называется функция, код которой подставляется в то место строки программы, из которого она вызывается, т.е. вызов такой функции заменяется ее кодом. Существует два способа создания встраиваемой функции. Первый состоит в использовании модификатора *inline*. Например, чтобы создать встраиваемую функцию *f*, которая возвращает *int*-значение и не принимает ни одного параметра, достаточно объявить ее таким образом.

```
inline int f()
{
    // ...
}
```

Модификатор *inline* должен предварять все остальные аспекты объявления функции.

Причиной существования встраиваемых функций является эффективность. Ведь при каждом вызове обычной функции должна быть выполнена целая последовательность инструкций, связанных с обработкой самого вызова, включая помещение ее аргументов в стек, и с возвратом из функции. В некоторых случаях значительное количество циклов центрального процессора используется именно для выполнения этих действий. Но если функция встраивается в строку программы, подобные системные затраты попросту отсутствуют, и общая скорость выполнения программы возрастает. Если же встраиваемая функция оказывается не такой уж маленькой, общий размер программы может существенно увеличиться. Поэтому лучше всего в качестве встраиваемых использовать только очень маленькие функции, а те, что побольше, — оформлять в виде обычных.

Продемонстрируем использование встраиваемой функции на примере следующей программы.

```
// Демонстрация использования inline-функции.

#include <iostream>
using namespace std;

class cl {
    int i; // Этот член данных закрыт (private) по умолчанию.
public:
    int get_i();
    void put_i(int j);
};

inline int cl::get_i() // Эта функция встраивается в строку.
```

```

    {
        return i;
    }

inline void cl::put_i(int j) // И эта функция встраиваемая.
{
    i = j;
}

int main()
{
    cl s;

    s.put_i(10);
    cout << s.get_i();

    return 0;
}

```

Важно понимать, что в действительности использование модификатора `inline` является *запросом*, а не *командой*, по которой компилятор генерирует встраиваемый (`inline-`) код. Существуют различные ситуации, которые могут не позволить компилятору удовлетворить наш запрос. Вот несколько примеров.

- Некоторые компиляторы не генерируют встраиваемый код, если соответствующая функция содержит цикл, конструкцию `switch` или инструкцию `goto`.
- Чаще всего встраиваемыми не могут быть рекурсивные функции.
- Как правило, встраивание “не проходит” для функций, которые содержат статические (`static`) переменные.

Помните, что ограничения на использование встраиваемых функций зависят от конкретной реализации системы, поэтому, чтобы узнать, какие ограничения имеют место в вашем случае, обратитесь к документации, прилагаемой к вашему компилятору.

## Создание встраиваемых функций в объявлении класса

Существует еще один способ создания встраиваемой функции. Он состоит в определении кода для функции-члена класса в самом объявлении класса. Любая функция, которая определяется в объявлении класса, автоматически становится встраиваемой.

ваемой. В этом случае необязательно предварять ее объявление ключевым словом `inline`. Например, предыдущую программу можно переписать в таком виде.

```
#include <iostream>
using namespace std;

class cl {
    int i; // Этот член данных закрыт (private) по умолчанию.
public:
    // Автоматически встраиваемые функции, поскольку
    // они определены внутри своего класса.
    int get_i() { return i; }
    void put_i(int j) { i = j; }
}

int main()
{
    cl s;

    s.put_i(10);
    cout << s.get_i();

    return 0;
}
```

Обратите внимание на то, как выглядит код функций, определенных “внутри” класса `cl`. Для очень небольших по объему функций такое представление кода отражает обычный стиль языка C++. Однако можно сформатировать эти функции и таким образом.

```
class cl {
    int i; // закрытый член по умолчанию
public:
    // встраиваемые функции
    int get_i()
    {
        return i;
    }

    void put_i(int j)
    {
```

```
i = j;
}
};
```

Небольшие функции (как представленные в этом примере) обычно определяются в объявлении класса. Использование “внутриклассовых” встраиваемых `public`-функций – обычная практика в C++-программировании, поскольку с их помощью обеспечивается доступ к закрытым (`private`) переменным. Такие функции называются *аксессорными* или *функциями доступа*. Успех объектно-ориентированного программирования зачастую определяется умением управлять доступом к данным посредством функций-членов. Так как большинство C++-программистов определяют функции доступа внутри своих классов, это соглашение применяется и к остальным примерам данной книги. То же самое я рекомендую делать и вам.

Итак, перепишем класс `Vehicle`, чтобы конструктор, деструктор и функция `range()` определялись внутри класса. Кроме того, сделаем поля `passengers`, `fuelcap` и `mpg` закрытыми, а для считывания их значений создадим специальные функции доступа.

```
// Определение конструктора, деструктора и функции range()
// в качестве встраиваемых функций.

#include <iostream>
using namespace std;

// Объявление класса Vehicle.
class Vehicle {
    // Теперь эти переменные закрыты (private).
    int passengers; // количество пассажиров
    int fuelcap; // вместимость топливных резервуаров в литрах
    int mpg; // расход горючего в километрах на литр
public:
    // Определение встраиваемого конструктора класса Vehicle.
    Vehicle(int p, int f, int m) {
        passengers = p;
        fuelcap = f;
        mpg = m;
    }
    // Функция вычисления максимального пробега теперь
```

```

// встраиваемая.

int range() { return mpg * fuelcap; }

// Функции доступа к членам класса теперь встраиваемые.
int get_passengers() { return passengers; }
int get_fuelcap() { return fuelcap; }
int get_mpg() { return mpg; }

};

int main() {
    // Передаем значения в конструктор класса Vehicle.
    Vehicle minivan(7, 16, 21);
    Vehicle sportscar(2, 14, 12);

    int range1, range2;

    // Вычисляем расстояние, которое может проехать каждое
    // транспортное средство после заливки полного бака
    // горючего:
    range1 = minivan.range();
    range2 = sportscar.range();

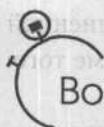
    cout << "Минифургон может перевезти "
        << minivan.get_passengers()
        << " пассажиров на расстояние " << range1
        << " километров." << "\n";

    cout << "Спортивный автомобиль может перевезти "
        << sportscar.get_passengers()
        << " пассажира на расстояние " << range2
        << " километров." << "\n";

    return 0;
}

```

Поскольку члены данных класса `Vehicle` теперь объявлены закрытыми, то для получения количества пассажиров, которых может перевезти то или иное транспортное средство, в функции `main()` необходимо использовать аксессорную функцию `get_passengers()`.



## Вопросы для текущего контроля

1. Каково назначение модификатора `inline`?
2. Можно ли определить встраиваемую функцию в объявлении класса?
3. Что представляет собой функция доступа?\*

## Проект 8.2. Создание класса очереди

Queue.cpp

Возможно, вы знаете, что под *структурой данных* понимается средство организации данных. В качестве примера самой простой структуры данных можно привести *массив* (*array*), представляющий собой линейный список, в котором поддерживается произвольный доступ к его элементам. Массивы часто используются как “фундамент”, на который опираются такие более сложные структуры, как стеки или очереди. *Стек* (*stack*) – это список, доступ к элементам которого возможен только по принципу “первым прибыл, последним обслужен” (*First in Last Out* – FILO). *Очередь* (*queue*) – это список, доступ к элементам которого возможен только по принципу “первым прибыл, первым обслужен” (*First in First Out* – FIFO). Вероятно, очередь не нуждается в примерах: вспомните, как вам приходилось стоять к какому-нибудь окошечку кассы. А стек можно представить в виде множества тарелок, поставленных одна на другую (к первой, или самой нижней, можно добраться только в самом конце, когда будут сняты все верхние).

В стеках и очередях интересно то, что в них объединяются как средства хранения информации, так и функции доступа к этой информации. Таким образом, стеки и очереди представляют собой специальные механизмы обслуживания данных, в которых хранение и доступ к ним обеспечивается самой структурой, а не программой, в которой они используются.

Как правило, очереди поддерживают две основные операции: чтение и запись. При каждой операции записи новый элемент помещается в конец очереди, а при каждой операции чтения считывается следующий элемент с начала очереди. При

1. Модификатор `inline` обеспечивает замену инструкции вызова функции в программе кодом этой функции.
2. Да, встраиваемую функцию можно определить в объявлении класса.
3. Функция доступа – это, как правило, короткая функция, которая считывает значение закрытого члена данных класса или устанавливает его.

этом считанный элемент снова считать нельзя. Очередь может быть заполненной "до отказа", если для приема нового элемента нет свободного места. Кроме того, очередь может быть пустой, если из нее удалены (считаны) все элементы.

Существует два основных типа очередей: циклические и нециклические. В циклической очереди при удалении очередного элемента повторно используется "ячейка" базового массива, на котором построена эта очередь. В нециклической повторного использования элементов массива не предусмотрено, и потому такая очередь в конце концов исчерпывается. В нашем проекте с целью упрощения задачи рассматривается нециклическая очередь, но при небольших дополнительных усилиях ее легко превратить в циклическую.

## Последовательность действий

1. Создайте файл с именем Queue.cpp.
2. В качестве фундамента для нашей очереди мы берем массив (хотя возможны и другие варианты). Доступ к этому массиву реализуется с помощью двух индексов: индекса записи (put-индекса) и индекса считывания (get-индекса). Индекс записи определяет, где будет храниться следующий элемент данных, а индекс считывания — с какой позиции будет считан следующий элемент данных. Следует помнить о невозможности считывания одного и того же элемента дважды. Хотя очередь, которую мы создаем в проекте, предназначена для хранения символов, аналогичную логику можно применить для хранения объектов любого типа. Итак, начнем с создания класса Queue.

```
const int maxQsize = 100; // Максимально возможное число
                           // элементов, которое может
                           // храниться в очереди.
```

```
class Queue {
    char q[maxQsize]; // Массив, на котором строится
                       // очередь.
    int size;          // Реальный (максимальный) размер
                       // очереди.
```

int putloc, getloc; // put- и get-индексы

Константная (const) переменная maxQsize определяет размер самой большой очереди, которую можно создать. Реальный размер очереди хранится в поле size.

3. Конструктор для класса Queue создает очередь заданного размера. Вот его код.

## 384 Модуль 8. Классы и объекты

```
// Конструктор очереди заданного размера.  
Queue(int len) {  
    // Длина очереди должна быть положительной и быть  
    // меньше заданного максимума.  
    if(len > maxQsize) len = maxQsize;  
    else if(len <= 0) len = 1;  
  
    size = len;  
    putloc = getloc = 0;  
}
```

Если указанный размер очереди превышает значение `maxQsize`, то создается очередь максимального размера. Если указанный размер очереди равен нулю или выражен отрицательным числом, то создается очередь длиной в 1 элемент. Размер очереди хранится в поле `size`. Индексы считывания и записи первоначально устанавливаются равными нулю.

4. Создаем функцию записи элементов в очередь, `put()`.

```
// Функция внесения элемента в очередь.  
void put(char ch) {  
    if(putloc == size) {  
        cout << " -- Очередь заполнена.\n";  
        return;  
    }  
  
    putloc++;  
    q[putloc] = ch;  
}
```

Функция сначала проверяет возможность очереди принять новый элемент. Если значение `putloc` равно максимальному размеру очереди, значит, в ней больше нет места для приема новых элементов. В противном случае переменная `putloc` инкрементируется, и ее новое значение определит позицию для хранения нового элемента. Таким образом, переменная `putloc` всегда содержит индекс элемента, запомненного в очереди последним.

5. Чтобы прочитать элемент из очереди, необходимо создать функцию считывания, `get()`.

```
// Функция считывания символа из очереди.  
char get() {  
    if(getloc == putloc) {  
        cout << " -- Очередь пуста.\n";  
        return -1;  
    }  
    return q[getloc];  
}
```

```

    return 0;
}

getloc++;
return q[getloc];
}

```

Обратите внимание на то, что в этой функции сначала проверяется, не пуста ли очередь. Очередь считается пустой, если индексы getloc и putloc указывают на один и тот же элемент. (Именно поэтому в конструкторе класса Queue оба индекса getloc и putloc инициализируются нулевым значением.) Затем индекс getloc инкрементируется, и функция возвращает следующий элемент. Таким образом, переменная getloc всегда содержит индекс элемента, считанного из очереди последним.

## 6. Вот полный текст программы Queue.cpp.

```

/*
Проект 8.2.

Класс очереди для хранения символов.

*/
#include <iostream>
using namespace std;

const int maxQsize = 100; // Максимально возможное число
                         // элементов, которое может
                         // храниться в очереди.

class Queue {
    char q[maxQsize]; // Массив, на котором строится
                      // очередь.
    int size;          // Реальный (максимальный) размер
                      // очереди.
    int putloc, getloc; // put- и get-индексы
public:

    // Конструктор очереди заданного размера.
    Queue(int len) {
        // Длина очереди должна быть положительной и быть
        // меньше заданного максимума.
        if(len > maxQsize) len = maxQsize;
    }
}

```

## 386 Модуль 8. Классы и объекты

```
else if(len <= 0) len = 1;

size = len;
putloc = getloc = 0;
}

// Функция внесения элемента в очередь.
void put(char ch) {
    if(putloc == size) {
        cout << " -- Очередь заполнена.\n";
        return;
    }

    putloc++;
    q[putloc] = ch;
}

// Функция считывания символа из очереди.
char get() {
    if(getloc == putloc) {
        cout << " -- Очередь пуста.\n";
        return 0;
    }

    getloc++;
    return q[getloc];
};

// Демонстрация использования класса Queue.
int main() {
    Queue bigQ(100);
    Queue smallQ(4);
    char ch;
    int i;

    cout << "Использование очереди bigQ для хранения
            алфавита.\n";
    // Помещаем ряд чисел в очередь bigQ.
```

```

for(i=0; i < 26; i++)
    bigQ.put('A' + i);

// Считываем и отображаем элементы из очереди bigQ.
cout << "Содержимое очереди bigQ: ";
for(i=0; i < 26; i++) {
    ch = bigQ.get();
    if(ch != 0) cout << ch;
}

cout << "\n\n";

cout << "Использование очереди smallQ для
        генерирования ошибок.\n";

// Теперь используем очередь smallQ для генерирования
// ошибок.
for(i=0; i < 5; i++) {
    cout << "Попытка сохранить " <<
        (char) ('Z' - i);

    smallQ.put('Z' - i);

    cout << "\n";
}
cout << "\n";

// Генерирование ошибок при считывании элементов из
// очереди smallQ.
cout << "Содержимое очереди smallQ: ";
for(i=0; i < 5; i++) {
    ch = smallQ.get();

    if(ch != 0) cout << ch;
}

cout << "\n";
}

```

7. Вот как выглядят результаты, сгенерированные этой программой.

Использование очереди `bigQ` для хранения алфавита.

Содержимое очереди `bigQ`: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Использование очереди `smallQ` для генерирования ошибок.

Попытка сохранить Z

Попытка сохранить Y

Попытка сохранить X

Попытка сохранить W

Попытка сохранить V -- Очередь заполнена.

Содержимое очереди `smallQ`: ZYXW -- Очередь пуста.

8. Самостоятельно попробуйтесь модифицировать класс `Queue` так, чтобы он мог хранить объекты других типов, например, `int` или `double`.

**ВАЖНО!**

## 8.7 Массивы объектов

Массивы объектов можно создавать точно так же, как создаются массивы значений других типов. Например, в следующей программе создается массив для хранения объектов типа `MyClass`, а доступ к объектам, которые являются элементами этого массива, осуществляется с помощью обычной процедуры индексирования массива.

```
// Создание массива объектов.
```

```
#include <iostream>
using namespace std;

class MyClass {
    int x;
public:
    void set_x(int i) { x = i; }
    int get_x() { return x; }
};

int main()
{
```

```

 MyClass obs[4]; // ← Создание массива объектов.
int i;

for(i=0; i < 4; i++)
    obs[i].set_x(i);

for(i=0; i < 4; i++)
    cout << "obs[" << i << "].get_x(): " <<
        obs[i].get_x() << "\n";

return 0;
}

```

Эта программа генерирует такие результаты.

```

obs[0].get_x(): 0
obs[1].get_x(): 1
obs[2].get_x(): 2
obs[3].get_x(): 3

```

**ВАЖНО!**

## 8.8. Инициализация массивов объектов

Если класс включает параметризованный конструктор, то массив объектов такого класса можно инициализировать. Например, в следующей программе используется параметризованный класс `MyClass` и инициализируемый массив `obs` объектов этого класса.

```
// Инициализация массива объектов класса.
```

```

#include <iostream>
using namespace std;

class MyClass {
    int x;
public:
    MyClass(int i) { x = i; }
    int get_x() { return x; }
};

```

## 390 Модуль 8. Классы и объекты

```
int main()
{
    MyClass obs[4] = { -1, -2, -3, -4 };
    int i;

    for(i=0; i < 4; i++)
        cout << "obs[" << i << "].get_x(): " <<
            obs[i].get_x() << "\n";

    return 0;
}
```

В этом примере конструктору класса `MyClass` передаются значения от  $-1$  до  $-4$ . При выполнении эта программа отображает следующие результаты.

```
obs[0].get_x(): -1
obs[1].get_x(): -2
obs[2].get_x(): -3
obs[3].get_x(): -4
```

В действительности синтаксис инициализации массива, выраженный строкой

```
MyClass obs[4] = { -1, -2, -3, -4 },
```

представляет собой сокращенный вариант следующего (более длинного) формата:

```
MyClass obs[4] = { MyClass(-1), MyClass(-2), // Еще один
                    MyClass(-3), MyClass(-4) }; // способ
   // инициализации массива.
```

Как разъяснялось выше, если конструктор принимает только один аргумент, выполняется неявное преобразование из типа аргумента в тип класса. При использовании более длинного формата инициализации массива конструктор вызывается напрямую.

При инициализации массива объектов, конструкторы которых принимают несколько аргументов, необходимо использовать более длинный формат инициализации. Рассмотрим пример.

```
#include <iostream>
using namespace std;

class MyClass {
    int x, y;
public:
    MyClass(int i, int j) { x = i; y = j; }
    int get_x() { return x; }
```

```

int get_y() { return y; }
};

int main()
{
    MyClass obs[4][2] = {
        MyClass(1, 2), MyClass(3, 4), // "Длинный" формат
        MyClass(5, 6), MyClass(7, 8), // инициализации.
        MyClass(9, 10), MyClass(11, 12),
        MyClass(13, 14), MyClass(15, 16)
    };

    int i;

    for(i=0; i < 4; i++) {
        cout << obs[i][0].get_x() << ' ';
        cout << obs[i][0].get_y() << "\n";
        cout << obs[i][1].get_x() << ' ';
        cout << obs[i][1].get_y() << "\n";
    }

    return 0;
}

```

В этом примере конструктор класса `MyClass` принимает два аргумента. В функции `main()` объявляется и инициализируется массив `obs` путем непосредственных вызовов конструктора `MyClass()`. Инициализируя массивы, можно всегда использовать длинный формат инициализации, даже если объект принимает только один аргумент (короткая форма просто более удобна для применения). Нетрудно проверить, что при выполнении эта программа отображает такие результаты.

```

1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16

```

ВАЖНО!

## 8.9. Указатели на объекты

Доступ к объекту можно получить напрямую (как во всех предыдущих примерах) или через указатель на этот объект. Чтобы получить доступ к отдельному элементу (члену) объекта с помощью указателя на этот объект, необходимо использовать оператор “стрелка” (он состоит из знака “минус” и знака “больше”: “->”).

Чтобы объявить указатель на объект, используется тот же синтаксис, как и в случае объявления указателей на значения других типов. В следующей программе создается простой класс `P_example`, определяется объект этого класса `ob` и объявляется указатель на объект типа `P_example` с именем `p`. В этом примере показано, как можно напрямую получить доступ к объекту `ob` и как использовать для этого указатель (в этом случае мы имеем дело с косвенным доступом).

```
// Простой пример использования указателя на объект.
```

```
#include <iostream>
using namespace std;

class P_example {
    int num;
public:
    void set_num(int val) {num = val;}
    void show_num() { cout << num << "\n"; }
};

int main()
{
    P_example ob, *p; // Объявляем объект и
                      // указатель на него.

    ob.set_num(1); // Получаем прямой доступ к объекту ob.
    ob.show_num();

    p = &ob; // Присваиваем указателю p адрес объекта ob.
    p->set_num(20); // Вызываем функции с использованием
    p->show_num(); // указателя на объект ob.

    return 0;
}
```

Обратите внимание на то, что адрес объекта `ob` получается путем использования оператора “`&`”, что соответствует получению адреса для переменных любого другого типа.

Как вы знаете, при инкрементации или декрементации указателя он инкрементируется или декрементируется так, чтобы всегда указывать на следующий или предыдущий элемент базового типа. То же самое происходит и при инкрементации или декрементации указателя на объект: он будет указывать на следующий или предыдущий объект. Чтобы проиллюстрировать этот механизм, модифицируем предыдущую программу. Теперь вместо одного объекта `ob` объявим двухэлементный массив `ob` типа `P_example`. Обратите внимание на то, как инкрементируется и декрементируется указатель `p` для доступа к двум элементам этого массива.

```
// Инкрементация и декрементация указателя
// на объект.

#include <iostream>
using namespace std;

class P_example {
    int num;
public:
    void set_num(int val) {num = val;}
    void show_num() { cout << num << "\n"; }
};

int main()
{
    P_example ob[2], *p;

    ob[0].set_num(10); // прямой доступ к объектам
    ob[1].set_num(20);

    p = &ob[0];      // Получаем указатель на первый элемент.
    p->show_num(); // Отображаем значение элемента ob[0]
                    // с помощью указателя.

    p++;            // Переходим к следующему объекту.
    p->show_num(); // Отображаем значение элемента ob[1]
                    // с помощью указателя.
```

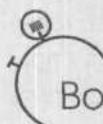
## 394 Модуль 8. Классы и объекты

```
    p--;           // Возвращаемся к предыдущему объекту.  
    p->show_num(); // Снова отображаем значение  
                   // элемента ob[0].  
  
    return 0;  
}
```

Вот как выглядят результаты выполнения этой программы.

```
10  
20  
10
```

Как будет показано ниже в этой книге, указатели на объекты играют главную роль в реализации одного из важнейших принципов C++: полиморфизма.



### Вопросы для текущего контроля

1. Можно ли массиву объектов присвоить начальные значения?
2. Какой оператор используется для доступа к члену объекта, если определен указатель на этот объект?\*



### Тест для самоконтроля по модулю 8

1. Каково различие между классом и объектом?
2. Какое ключевое слово используется для объявления класса?
3. Собственную копию чего имеет каждый объект?
4. Покажите, как объявить класс `Test`, который содержит две закрытые `int`-переменные `count` и `max`.
5. Какое имя носит конструктор? Как называется деструктор?
6. Дано следующее объявление класса.

```
class Sample {  
    int i;  
public:
```

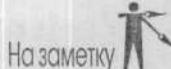
1. Да, массиву объектов можно присвоить начальные значения.
2. При доступе к члену объекта через указатель используется оператор "стрелка".

```
Sample(int x) { i = x; }
// ...
};
```

Покажите, как объявить объект класса Sample, который инициализирует переменную i значением 10.

7. Какая оптимизация происходит автоматически при определении функции-члена в объявлении класса?
8. Создайте класс Triangle, в котором для хранения длины основания и высоты прямоугольного треугольника используются две переменные экземпляра. Для установки этих значений включите в класс конструктор. Определите две функции. Первая, `hypot()`, должна возвращать длину гипotenузы, а вторая, `area()`, — его площадь.
9. Расширьте класс Help так, чтобы он хранил целочисленный идентификационный номер (ID) для идентификации каждого пользователя этого класса. Обеспечьте отображение этого номера при разрушении объекта справочной системы. Создайте функцию `getID()`, которая будет возвращать ID-номер.

Ответы на эти вопросы можно найти на Web-странице данной книги по адресу: <http://www.osborne.com>.





# Модуль 9

## О классах подробнее

- 9.1.** Перегрузка конструкторов
- 9.2.** Присваивание объектов
- 9.3.** Передача объектов функциям
- 9.4.** Возвращение объектов функциями
- 9.5.** Создание и использование конструктора копии
- 9.6.** Функции-“друзья”
- 9.7.** Структуры и объединения
- 9.8.** Ключевое слово `this`
- 9.9.** Перегрузка операторов
- 9.10.** Перегрузка операторов с использованием функций-членов
- 9.11.** Перегрузка операторов с использованием функций-не членов класса

## 398 Модуль 9. О классах подробнее

В этом модуле мы продолжим рассмотрение классов, начатое в модуле 8. Здесь вы узнаете о перегрузке конструкторов, а также о возможности передачи и возвращения объектов функциями. Вы познакомитесь с конструктором специального типа, именуемого *конструктором копии*, который используется, когда возникает необходимость в создании копии объекта. Кроме того, здесь рассматривается применение “дружественных” функций, структур, объединений и ключевого слова `this`. Завершает этот модуль описание одного из самых интересных средств C++ – перегрузки операторов.

**ВАЖНО!**

### 9.1. Перегрузка конструкторов

Несмотря на выполнение конструкторами уникальных действий, они не сильно отличаются от функций других типов и также могут подвергаться перегрузке. Чтобы перегрузить конструктор класса, достаточно объявить его во всех нужных форматах и определить каждое действие, связанное с соответствующим форматом. Например, в следующей программе определяются три конструктора.

```
// Перегрузка конструктора.

#include <iostream>
using namespace std;

class Sample {
public:
    int x;
    int y;

    // Перегрузка конструктора:
    // конструктор по умолчанию
    Sample() { x = y = 0; }

    // конструктор с одним параметром
    Sample(int i) { x = y = i; }

    // конструктор с двумя параметрами
    Sample(int i, int j) { x = i; y = j; }
};
```

```

int main() {
    Sample t;           // вызываем конструктор по умолчанию
    Sample t1(5);      // используем вариант Sample(int)
    Sample t2(9, 10);   // используем вариант Sample(int, int)

    cout << "t.x: " << t.x << ", t.y: " << t.y << "\n";
    cout << "t1.x: " << t1.x << ", t1.y: " << t1.y << "\n";
    cout << "t2.x: " << t2.x << ", t2.y: " << t2.y << "\n";

    return 0;
}

```

Результаты выполнения этой программы таковы.

```

t.x: 0, t.y: 0
t1.x: 5, t1.y: 5
t2.x: 9, t2.y: 10

```

В этой программе создается три конструктора. Первый – конструктор без параметров – инициализирует члены данных *x* и *y* нулевыми значениями. Этот конструктор играет роль конструктора по умолчанию, который (в случае его отсутствия) был бы предоставлен средствами C++ автоматически. Второй конструктор принимает один параметр, значение которого присваивается обоим членам данных *x* и *y*. Третий конструктор принимает два параметра, которые используются для индивидуальной инициализации переменных *x* и *y*.

Перегрузка конструкторов полезна по нескольким причинам. Во-первых, она делает классы, определяемые программистом, более гибкими, позволяя создавать объекты различными способами. Во-вторых, перегрузка конструкторов предоставляет удобства для пользователя класса, поскольку он может выбрать самый подходящий способ построения объектов для каждой конкретной задачи. В-третьих, определение конструктора как без параметров, так и параметризованного, позволяет создавать как инициализированные объекты, так и неинициализированные.

#### ВАЖНО!

## 9.2. Присваивание объектов

Если два объекта имеют одинаковый тип (т.е. оба они – объекты одного класса), то один объект можно присвоить другому. Для присваивания недостаточно, чтобы два класса были физически подобны; должны совпадать и имена классов, объекты которых участвуют в операции присваивания. Если один объект присва-

## 400 Модуль 9. О классах подробнее

ивается другому, то по умолчанию данные первого объекта поразрядно копируются во второй (т.е. второму объекту присваивается поразрядная копия данных первого объекта). Таким образом, после выполнения присваивания по умолчанию объекты будут идентичными, но отдельными. Присваивание объектов демонстрируется в следующей программе.

```
// Демонстрация присваивания объектов.

#include <iostream>
using namespace std;

class Test {
    int a, b;
public:
    void setab(int i, int j) { a = i, b = j; }
    void showab() {
        cout << "a равно " << a << '\n';
        cout << "b равно " << b << '\n';
    }
};

int main()
{
    Test ob1, ob2;

    ob1.setab(10, 20);
    ob2.setab(0, 0);
    cout << "Объект ob1 до присваивания: \n";
    ob1.showab();
    cout << "Объект ob2 до присваивания: \n";
    ob2.showab();
    cout << '\n';

    ob2 = ob1; // Присваиваем объект ob1 объекту ob2.

    cout << "Объект ob1 после присваивания: \n";
    ob1.showab();
    cout << "Объект ob2 после присваивания: \n";
    ob2.showab();
    cout << '\n';
```

```

ob1.setab(-1, -1); // Изменяем объект ob1.

cout << "Объект ob1 после изменения объекта ob1: \n";
ob1.showab();
cout << "Объект ob2 после изменения объекта ob1: \n";
ob2.showab();

return 0;
}

```

При выполнении эта программа генерирует такие результаты.

Объект ob1 до присваивания:

a равно 10  
b равно 20

Объект ob2 до присваивания:

a равно 0  
b равно 0

Объект ob1 после присваивания:

a равно 10  
b равно 20

Объект ob2 после присваивания:

a равно 10  
b равно 20

Объект ob1 после изменения объекта ob1:

a равно -1  
b равно -1

Объект ob2 после изменения объекта ob1:

a равно 10  
b равно 20

Как подтверждают результаты выполнения этой программы, после присваивания одного объекта другому эти два объекта будут содержать одинаковые значения. Тем не менее эти объекты совершенно независимы. Поэтому последующая модификация данных одного объекта не оказывает никакого влияния на другой. Однако при присваивании объектов возможны побочные эффекты. Например, если объект А содержит указатель на некоторый другой объект В, то после копирования объекта А его копия также будет содержать поле, которое указывает на

объект В. В этом случае изменение объекта В повлияет на оба объекта. В такой ситуации, как будет показано ниже в этом модуле, необходимо заблокировать создание поразрядной копии путем определения "собственного" оператора присваивания для данного класса.

**ВАЖНО!**

### 9.3. Передача объектов функциям

Объект можно передать функции точно так же, как значение любого другого типа данных. Объекты передаются функциям путем использования обычного C++-соглашения о передаче параметров по значению. Таким образом, функции передается не сам объект, а его копия. Следовательно, изменения, внесенные в объект при выполнении функции, не оказывают никакого влияния на объект, используемый в качестве аргумента для функции. Этот механизм демонстрируется в следующей программе.

```
// Передача объекта функции.

#include <iostream>
using namespace std;

class MyClass {
    int val;
public:
    MyClass(int i) {
        val = i;
    }

    int getval() { return val; }
    void setval(int i) { val = i; }
};

void display(MyClass ob) // Функция display() принимает
                        // в качестве параметра объект
                        // класса MyClass.
{
    cout << ob.getval() << '\n';
}

void change(MyClass ob) // Функция change() также принимает
```

```

    // в качестве параметра объект
    // класса MyClass.

{
    ob.setval(100); // Это никак не влияет на аргумент.

    cout << "Значение объекта ob в функции change(): ";
    display(ob);
}

int main()
{
    MyClass a(10);

    cout << "Значение объекта a до вызова функции change(): ";
    display(a);

    change(a);
    cout << "Значение объекта a после вызова функции change(): ";
};

    display(a);

    return 0;
}

```

При выполнении эта программа генерирует следующие результаты.

Значение объекта a до вызова функции change(): 10

Значение объекта ob в функции change(): 100

Значение объекта a после вызова функции change(): 10

Судя по результатам, изменение значения ob в функции change() не оказывает влияния на объект a в функции main().

## Конструкторы, деструкторы и передача объектов

Несмотря на то что передача функциям простых объектов в качестве аргументов — довольно несложная процедура, при этом могут происходить непредвиденные события, имеющие отношение к конструкторам и деструкторам. Чтобы разобраться в этом, рассмотрим следующую программу.

// Конструкторы, деструкторы и передача объектов.

```
#include <iostream>
```

## 404 Модуль 9. О классах подробнее

```
using namespace std;

class MyClass {
    int val;
public:
    MyClass(int i) {
        val = i;
        cout << "Внутри конструктора.\n";
    }

    ~MyClass()
    { cout << "Разрушение объекта.\n"; } // Обратите внимание
  // на это сообщение.
    int getval() { return val; }
};

void display(MyClass ob)
{
    cout << ob.getval() << '\n';
}

int main()
{
    MyClass a(10);

    cout << "До вызова функции display().\n";
    display(a);
    cout << "После выполнения функции display().\n";

    return 0;
}
```

Эта программа генерирует довольно неожиданные результаты.  
Внутри конструктора.  
До вызова функции display().  
10  
Разрушение объекта.  
После выполнения функции display().  
Разрушение объекта.

Как видите, здесь выполняется одно обращение к функции конструктора (при создании объекта `a`), но почему-то *два* обращения к функции деструктора. Давайте разбираться, в чем тут дело.

При передаче объекта функции создается его копия (и эта копия становится параметром в функции). Создание копии означает "рождение" нового объекта. Когда выполнение функции завершается, копия аргумента (т.е. параметр) разрушается. Здесь возникает сразу два вопроса. Во-первых, вызывается ли конструктор объекта при создании копии? Во-вторых, вызывается ли деструктор объекта при разрушении копии? Ответы могут удивить вас.

Когда при вызове функции создается копия аргумента, обычный конструктор не вызывается. Вместо этого вызывается *конструктор копии* объекта. Конструктор копии определяет, как должна быть создана копия объекта. (Как создать конструктор копии, будет показано ниже в этой главе.) Но если в классе явно не определен конструктор копии, C++ предоставляет его по умолчанию. Конструктор копии по умолчанию создает побитовую (т.е. идентичную) копию объекта. Поскольку для инициализации некоторых аспектов объекта используется обычный конструктор, он не должен вызываться для создания копии уже существующего объекта. Такой вызов изменил бы его содержимое. При передаче объекта функции имеет смысл использовать текущее состояние объекта, а не начальное.

Но когда функция завершается и разрушается копия объекта, используемая в качестве аргумента, вызывается деструктор этого объекта. Необходимость вызова деструктора связана с выходом объекта из области видимости. Именно поэтому предыдущая программа имела два обращения к деструктору. Первое произошло при выходе из области видимости параметра функции `display()`, а второе — при разрушении объекта `a` в функции `main()` по завершении программы.

Итак, когда объект передается функции в качестве аргумента, обычный конструктор не вызывается. Вместо него вызывается конструктор копии, который по умолчанию создает побитовую (идентичную) копию этого объекта. Но когда эта копия разрушается (обычно при выходе за пределы области видимости по завершении функции), обязательно вызывается деструктор.

## Передача объектов по ссылке

Объект можно передать функции не только по значению, но и по ссылке. В этом случае функции передается ссылка, и функция будет непосредственно обрабатывать объект, используемый в качестве аргумента. Это значит, что изменения, вносимые в параметр, автоматически повлияют на аргумент. Однако передача объекта по ссылке, применима не во всех ситуациях. Если же это возможно, мы выигрываем в двух аспектах. Во-первых, поскольку функции передается не весь объект, а только его адрес, то можно говорить о более быстрой и эффектив-

ной передаче объекта по ссылке по сравнению с передачей объекта по значению. Во-вторых, при передаче объекта по ссылке никакие новые объекты не создаются, а, значит, не тратится время и ресурсы системы на построение и разрушение временного объекта.

Теперь рассмотрим пример передачи объекта по ссылке.

// Конструкторы, деструкторы и передача объектов по ссылке.

```
#include <iostream>
using namespace std;

class MyClass {
    int val;
public:
    MyClass(int i) {
        val = i;
        cout << "Внутри конструктора.\n";
    }

    ~MyClass() { cout << "Разрушение объекта.\n"; }
    int getval() { return val; }
    void setval(int i) { val = i; }
};

void display(MyClass &ob) // Здесь объект типа MyClass
                           // передается по ссылке.
{
    cout << ob.getval() << '\n';
}

void change(MyClass &ob) // Здесь объект типа MyClass
                           // передается по ссылке.
{
    ob.setval(100);
}

int main()
{
    MyClass a(10);
```

```

cout << "До вызова функции display().\n";
display(a);
cout << "После возвращения из функции display().\n";

change(a);
cout << "После вызова функции change().\n";
display(a);

return 0;
}

```

Результаты выполнения этой программы таковы.

Внутри конструктора.

До вызова функции display().

10

После возвращения из функции display().

После вызова функции change().

100

Разрушение объекта.

В этой программе обе функции `display()` и `change()` используют ссылочные параметры. Таким образом, каждой из этих функций передается не копия аргумента, а ее адрес, и функция обрабатывает сам аргумент. Например, когда выполняется функция `change()`, все изменения, вносимые ею в параметр `ob`, непосредственно отражаются на объекте `a`, существующем в функции `main()`. Обратите также внимание на то, что здесь создается и разрушается только один объект `a`. В этой программе обошлось без создания временных объектов.

## Потенциальные проблемы при передаче параметров

Несмотря на то что в случае, когда объекты передаются функциям “по значению” (т.е. посредством обычного C++-механизма передачи параметров, который теоретически защищает аргумент и изолирует его от принимаемого параметра), здесь все-таки возможен побочный эффект или даже угроза для “жизни” объекта, используемого в качестве аргумента. Например, если объект, используемый как аргумент, требует динамического выделения памяти и освобождает эту память при разрушении, его локальная копия при вызове деструктора освободит ту же самую область памяти, которая была выделена оригинальному объекту. И этот факт становится уже целой проблемой, поскольку оригиналный объект все еще использует эту (уже освобожденную) область памяти. Описанная ситуация делает исходный объект “ущербным” и, по сути, непригодным для использования.

Одно из решений описанной проблемы — передача объекта по ссылке, которая была продемонстрирована в предыдущем разделе. В этом случае никакие копии объектов не создаются, и, следовательно, ничего не разрушается при выходе из функции. Как отмечалось выше, передача по ссылке ускоряет вызов функции, поскольку функции при этом передается только адрес объекта. Однако передача объектов по ссылке применима далеко не во всех случаях. К счастью, есть более общее решение: можно создать собственную версию конструктора копии. Это позволит точно определить, как именно следует создавать копию объекта, чтобы избежать описанных выше проблем. Но прежде чем мы займемся конструктором копии, имеет смысл рассмотреть еще одну ситуацию, в обработке которой мы также можем выиграть от создания конструктора копии.

**ВАЖНО!****9.4. Возвращение объектов функциями**

Если объекты можно передавать функциям, то “с таким же успехом” функции могут возвращать объекты. Чтобы функция могла вернуть объект, во-первых, необходимо объявить в качестве типа возвращаемого ею значения тип соответствующего класса. Во-вторых, нужно обеспечить возврат объекта этого типа с помощью обычной инструкции `return`. Рассмотрим пример. В следующей программе функция-член `mkBigger()` возвращает объект, в котором его член данных `val` содержит вдвое большее значение по сравнению с вызывающим объектом.

```
// Возвращение объектов функциями.
```

```
#include <iostream>
using namespace std;

class MyClass {
    int val;
public:
    // Обычный конструктор.
    MyClass(int i) {
        val = i;
        cout << "Внутри конструктора.\n";
    }

    ~MyClass() {
        cout << "Разрушение объекта.\n";
    }
}
```

```

}

int getval() { return val; }

// Возвращение объекта.
MyClass mkBigger() { // Функция mkBigger() возвращает
                     // объект типа MyClass.
    MyClass o(val * 2);

    return o;
};

void display(MyClass ob)
{
    cout << ob.getval() << '\n';
}

int main()
{
    cout << "Перед созданием объекта а.\n";
    MyClass a(10);
    cout << "После создания объекта а.\n\n";

    cout << "Перед вызовом функции display().\n";
    display(a);
    cout << "После возвращения из функции display().\n\n";

    cout << "Перед вызовом функции mkBigger().\n";
    a = a.mkBigger();
    cout << "После возвращения из функции mkBigger().\n\n";

    cout << "Перед вторым вызовом функции display().\n";
    display(a);
    cout << "После возвращения из функции display().\n\n";

    return 0;
}

```

Вот какие результаты генерирует эта программа.

## 410 Модуль 9. О классах подробнее

Перед созданием объекта а.

Внутри конструктора.

После создания объекта а.

Перед вызовом функции display().

10

**Разрушение объекта.**

После возвращения из функции display().

Перед вызовом функции mkBigger().

Внутри конструктора.

Разрушение объекта.

**Разрушение объекта.**

После возвращения из функции mkBigger().

Перед вторым вызовом функции display().

20

Разрушение объекта.

После возвращения из функции display().

**Разрушение объекта.**

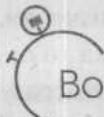
В этом примере функция mkBigger() создает локальный объект о класса MyClass, в котором его член данных val содержит вдвое большее значение, чем соответствующий член данных вызывающего объекта. Этот объект затем возвращается функцией и присваивается объекту а в функции main(). Затем объект о разрушается, о чем свидетельствует первое сообщение “Разрушение объекта.”. Но чем можно объяснить второй вызов деструктора?

Если функция возвращает объект, автоматически создается временный объект, который хранит возвращаемое значение. Именно этот объект реально и возвращается функцией. После возврата значения объект разрушается. Вот откуда взялось второе сообщение “Разрушение объекта.” как раз перед сообщением “После возвращения из функции mkBigger().”. Это – вещественное доказательство разрушения временного объекта.

Потенциальные проблемы, подобные тем, что возникают при передаче параметров, возможны и при возвращении объектов из функций. Разрушение временного объекта в некоторых ситуациях может вызвать непредвиденные побочные эффекты. Например, если объект, возвращаемый функцией, имеет деструктор, который освобождает некоторый ресурс (например, динамически выделяемую память или файловый дескриптор), этот ресурс будет освобожден даже в том

случае, если объект, получающий возвращаемое функцией значение, все еще ее использует. Решение проблемы такого рода включает применение конструктора копии, которому посвящен следующий раздел.

И еще. Можно организовать функцию так, чтобы она возвращала объект по ссылке. Но при этом необходимо позаботиться о том, чтобы адресуемый ссылкой объект не выходил из области видимости по завершении функции.



## Вопросы для текущего контроля

1. Конструкторы перегружать нельзя. Верно ли это утверждение?
2. Когда объект передается функции по значению, создается его копия. Разрушается ли эта копия, когда выполнение функции завершается?
3. Когда объект возвращается функцией, создается временный объект, который содержит возвращаемое значение. Верно ли это утверждение?\*

**ВАЖНО!**

## 9.5. Создание и использование конструктора копии

Как было показано в предыдущих примерах, при передаче объекта функции или возврате объекта из функции создается копия объекта. По умолчанию копия представляет собой побитовый клон оригинального объекта. Во многих случаях такое поведение “по умолчанию” вполне приемлемо, но в остальных ситуациях желательно бы уточнить, как должна быть создана копия объекта. Это можно сделать, определив явным образом конструктор копии для класса. *Конструктор копии* представляет собой специальный тип перегруженного конструктора, который автоматически вызывается, когда в копии объекта возникает необходимость.

Для начала еще раз сформулируем проблемы, для решения которых мы хотим определить конструктор копии. При передаче объекта функции создается побитовая (т.е. точная) копия этого объекта, которая передается параметру этой функции. Однако возможны ситуации, когда такая идентичная копия нежелательна. Например, если оригинальный объект использует такой ресурс, как от-

1. Это неверно. Конструкторы можно перегружать.
2. Да, когда выполнение функции завершается, эта копия разрушается.
3. Верно. Значение, возвращаемое функцией, обеспечивается за счет создания временного объекта.

## 412 Модуль 9. О классах подробнее

крытый файл, то копия объекта будет использовать *тот же самый* ресурс. Следовательно, если копия внесет изменения в этот ресурс, то изменения коснутся также оригинального объекта! Более того, при завершении функции копия будет разрушена (с вызовом деструктора). Это может освободить ресурс, который еще нужен оригинальному объекту.

Аналогичная ситуация возникает при возврате объекта из функции. Компилятор генерирует временный объект, который будет хранить копию значения, возвращаемого функцией. (Это делается автоматически, и без нашего на то согласия.) Этот временный объект выходит за пределы области видимости сразу же, как только инициатору вызова этой функции будет возвращено "обещанное" значение, после чего незамедлительно вызывается деструктор временного объекта. Но если этот деструктор разрушит что-либо нужное для выполняемого далее кода, последствия будут печальны.

В центре рассматриваемых проблем лежит создание побитовой копии объекта. Чтобы предотвратить их возникновение, необходимо точно определить, что должно происходить, когда создается копия объекта, и тем самым избежать нежелательных побочных эффектов. Этого можно добиться путем создания конструктора копии.

Прежде чем подробнее знакомиться с использованием конструктора копии, важно понимать, что в C++ определено два отдельных вида ситуаций, в которых значение одного объекта передается другому. Первой такой ситуацией является присваивание, а второй — инициализация. Инициализация может выполняться тремя способами, т.е. в случаях, когда:

- один объект явно инициализирует другой объект, как, например, в объявлении;
- копия объекта передается параметру функции;
- генерируется временный объект (чаще всего в качестве значения, возвращаемого функцией).

Конструктор копии применяется только к инициализациям. Он не применяется к присваиванию.

Вот как выглядит самый распространенный формат конструктора копии.

```
имя_класса(const имя_класса &obj) {  
    // тело конструктора  
}
```

Здесь элемент *obj* означает ссылку на объект, которая используется для инициализации другого объекта. Например, предположим, у нас есть класс *MyClass* и объект у типа *MyClass*, тогда при выполнении следующих инструкций будет вызван конструктор копии класса *MyClass*.

```
MyClass x = y; // Объект у явно инициализирует объект x.
func1(y);      // Объект у передается в качестве параметра.
y = func2();    // Объект у принимает объект,
                // возвращаемый функцией.
```

В первых двух случаях конструктору копии будет передана ссылка на объект *y*, а в третьем – ссылка на объект, возвращаемый функцией *func2()*. Таким образом, когда объект передается в качестве параметра, возвращается функцией или используется в инициализации, для дублирования объекта и вызывается конструктор копии.

Помните: конструктор копии не вызывается в случае, когда один объект присваивается другому. Например, при выполнении следующего кода конструктор копии вызван не будет.

```
MyClass x;
MyClass y;

x = y; // Конструктор копии здесь не используется.
```

Итак, присваивания обрабатываются оператором присваивания, а не конструктором копии.

Использование конструктора копии демонстрируется в следующей программе.

```
/* Конструктор копии вызывается при передаче
объекта функции. */
```

```
#include <iostream>
using namespace std;

class MyClass {
    int val;
    int copynumber;
public:
    // Обычный конструктор.
    MyClass(int i) {
        val = i;
        copynumber = 0;
        cout << "Внутри обычного конструктора.\n";
    }

    // Конструктор копии.
    MyClass(const MyClass &o) {
```

## 414 Модуль 9. О классах подробнее

```
val = o.val;
copynumber = o.copynumber + 1;
cout << "Внутри конструктора копии.\n";
}

~MyClass() {
    if(copynumber == 0)
        cout << "Разрушение оригинального объекта.\n";
    else
        cout << "Разрушение копии " <<
            copynumber << "\n";
}

int getval() { return val; }
};

void display(MyClass ob)
{
    cout << ob.getval() << '\n';
}

int main()
{
    MyClass a(10);

    display(a); // Конструктор копии вызывается, когда
                // функции display() передается объект a.

    return 0;
}
```

При выполнении программы генерирует такие результаты.

Внутри обычного конструктора.

Внутри конструктора копии.

10

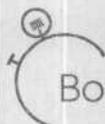
Разрушение копии 1

Разрушение оригинального объекта..

При выполнении этой программы здесь происходит следующее: когда в функции main() создается объект a, “стараниями” обычного конструктора значение

переменной `copynumber` устанавливается равным нулю. Затем объект `a` передается функции `display()`, а именно — ее параметру `obj`. В этом случае вызывается конструктор копии, который создает копию объекта `a`. Конструктор копии инкрементирует значение переменной `copynumber`. По завершении функции `display()` объект `obj` выходит из области видимости. Этот выход сопровождается вызовом его деструктора. Наконец, по завершении функции `main()` выходит из области видимости объект `a`, что также сопровождается вызовом его деструктора.

Вам было бы полезно немного поэкспериментировать с предыдущей программой. Например, создайте функцию, которая бы возвращала объект класса `MyClass`, и понаблюдайте, когда именно вызывается конструктор копии.



## Вопросы для текущего контроля

1. Какая копия объекта создается при использовании конструктора копии по умолчанию?
2. Конструктор копии вызывается в случае, когда один объект присваивается другому. Верно ли это?
3. В каких случаях может понадобиться явное определение конструктора копии для класса?\*

ВАЖНО!

### 9.6. ФУНКЦИИ-“ДРУЗЬЯ”

В общем случае только члены класса имеют доступ к закрытым членам этого класса. Однако в C++ существует возможность разрешить доступ к закрытым членам класса функциям, которые не являются членами этого класса. Для этого достаточно объявить эти функции “дружественными” (или “друзьями”) по отношению к рассматриваемому классу. Чтобы сделать функцию “другом” класса, включите ее прототип в `public`-раздел объявления класса и предварите его ключевым словом `friend`. Например, в этом фрагменте кода функция `frnd()` объявляется “другом” класса `MyClass`.

1. Конструктор копии по умолчанию создает поразрядную (т.е. идентичную) копию.
2. Не верно. Конструктор копии не вызывается, когда один объект присваивается другому.
3. Явное определение конструктора копии может понадобиться, когда копия объекта не должна быть идентичной оригиналу (чтобы не повредить его).

```
class MyClass {  
    // ...  
public:  
    friend void frnd(MyClass ob);  
    // ...  
};
```

Как видите, ключевое слово `friend` предваряет остальную часть прототипа функции. Функция может быть “другом” нескольких классов.

Рассмотрим короткий пример, в котором функция-“друг” используется для доступа к закрытым членам класса `MyClass`, чтобы определить, имеют ли они общий множитель.

// Демонстрация использования функции-“друга”.

```
#include <iostream>  
using namespace std;  
  
class MyClass {  
    int a, b;  
public:  
    MyClass(int i, int j) { a=i; b=j; }  
    friend int comDenom(MyClass x); // Функция comDenom() -  
                                    // “друг” класса MyClass.  
};  
  
// Обратите внимание на то, что функция comDenom()  
// не является членом ни одного класса.  
  
int comDenom(MyClass x)  
{  
    /* Поскольку функция comDenom() -- “друг” класса MyClass,  
       она имеет право на прямой доступ к его членам  
       данных a и b. */  
    int max = x.a < x.b ? x.a : x.b;  
  
    for(int i=2; i <= max; i++)  
        if((x.a%i)==0 && (x.b%i)==0) return i;  
  
    return 0;
```

```

} // Вызывает функцию comDenom() в классе MyClass, которая возвращает значение общего
// множителя для всех членов класса MyClass. Возвращаемое значение определяется
// как произведение значений всех членов класса MyClass.

int main()
{
    MyClass n(18, 111);

    if(comDenom(n)) // Функция comDenom() вызывается обычным
                     // способом, без использования имени
                     // объекта и оператора "точка".
        cout << "Общий множитель равен " <<
            comDenom(n) << "\n";
    else
        cout << "Общего множителя нет.\n";

    return 0;
}

```

В этом примере функция `comDenom()` не является членом класса `MyClass`. Тем не менее она имеет полный доступ к `private`-членам класса `MyClass`. В частности, она может непосредственно использовать значения `x.a` и `x.b`. Обратите также внимание на то, что функция `comDenom()` вызывается обычным способом, т.е. без привязки к объекту (и без использования оператора “точка”). Поскольку она не функция-член, то при вызове ее не нужно квалифицировать с указанием имени объекта. (Точнее, при ее вызове нельзя задавать имя объекта.) Обычно функции-“друг” в качестве параметра передается один или несколько объектов класса, для которого она является “другом”, как в случае функции `comDenom()`.

Несмотря на то что в данном примере мы не извлекаем никакой пользы из объявления функции `comDenom()` “другом”, а не членом класса `MyClass`, существуют определенные обстоятельства, при которых статус функции-“друга” имеет большое значение. Во-первых, как вы узнаете ниже из этого модуля, функции-“друзья” могут быть полезны для перегрузки операторов определенных типов. Во-вторых, функции-“друзья” упрощают создание некоторых функций ввода-вывода (об этом речь пойдет в модуле 11).

Третья причина использования функций-“друзей” заключается в том, что в некоторых случаях два (или больше) класса могут содержать члены, которые находятся во взаимной связи с другими частями программы. Например, у нас есть два различных класса, `Cube` и `Cylinder`, которые определяют характеристики куба и цилиндра (одной из этих характеристик является цвет объекта). Чтобы можно было легко сравнивать цвет куба и цилиндра, определим функцию-“друга”, которая, получив доступ к “цвету” каждого объекта, возвратит значение ИСТИНА,

если сравниваемые цвета совпадают, и значение ЛОЖЬ в противном случае. Эта идея иллюстрируется на примере следующей программы.

```
// Функциями-“друзьями” могут пользоваться сразу несколько
// классов.

#include <iostream>
using namespace std;

class Cylinder; // опережающее объявление

enum colors { red, green, yellow };

class Cube {
    colors color;
public:
    Cube(colors c) { color = c; }
    friend bool sameColor(Cube x, Cylinder y); // Функция-“друг”
  // класса Cube.
    // ...
};

class Cylinder {
    colors color;
public:
    Cylinder(colors c) { color= c; }
    friend bool sameColor(Cube x, Cylinder y); // Функция-
  // “друг” и
  // класса
  // Cylinder.
    // ...
};

bool sameColor(Cube x, Cylinder y)
{
    if(x.color == y.color) return true;
    else return false;
}
```

```

int main()
{
    Cube cube1(red);
    Cube cube2(green);
    Cylinder cyl(green);

    if (sameColor(cube1, cyl))
        cout << "Объекты cube1 и cyl одинакового цвета.\n";
    else
        cout << "Объекты cube1 и cyl разного цвета.\n";

    if (sameColor(cube2, cyl))
        cout << "Объекты cube2 и cyl одинакового цвета.\n";
    else
        cout << "Объекты cube2 и cyl разного цвета.\n";

    return 0;
}

```

При выполнении эта программа генерирует такие результаты.

Объекты cube1 и cyl разного цвета.

Объекты cube2 и cyl одинакового цвета.

Обратите внимание на то, что в этой программе используется *опережающее объявление* (также именуемое *опережающей ссылкой*) для класса Cylinder. Его необходимость обусловлена тем, что объявление функции sameColor() в классе Cube использует ссылку на класс Cylinder до его объявления. Чтобы создать опережающее объявление для класса, достаточно использовать формат, представленный в этой программе.

"Друг" одного класса может быть членом другого класса. Перепишем предыдущую программу так, чтобы функция sameColor() стала членом класса Cube. Обратите внимание на использование оператора разрешения области видимости (или оператора разрешения контекста) при объявлении функции sameColor() в качестве "друга" класса Cylinder.

```
/* Функция может быть членом одного класса и одновременно
   "другом" другого. */
```

```
#include <iostream>
using namespace std;
```

## 420 Модуль 9. О классах подробнее

```
class Cylinder; // опережающее объявление

enum colors { red, green, yellow };

class Cube {
    colors color;
public:
    Cube(colors c) { color= c; }
    bool sameColor(Cylinder y); // функция sameColor() теперь
                                // является членом класса Cube.
    // ...
};

class Cylinder {
    colors color;
public:
    Cylinder(colors c) { color = c; }
    friend bool Cube::sameColor(Cylinder y); // функция
  // Cube::sameColor() -
  // "друг" класса Cylinder.
    // ...
};

bool Cube::sameColor(Cylinder y) {
    if(color == y.color) return true;
    else return false;
}

int main()
{
    Cube cubel(red);
    Cube cube2(green);
    Cylinder cyl(green);

    if(cubel.sameColor(cyl))
        cout << "Объекты cubel и cyl одного цвета.\n";
    else
        cout << "Объекты cubel и cyl разного цвета.\n";
}
```

```

if(cube2.sameColor(cyl))
    cout << "Объекты cube2 и cyl одного цвета.\n";
else
    cout << "Объекты cube2 и cyl разного цвета.\n";

return 0;
}

```

Поскольку функция `sameColor()` является членом класса `Cube`, она должна вызываться для объектов класса `Cube`, т.е. она имеет прямой доступ к переменной `color` объектов типа `Cube`. Следовательно, в качестве параметра необходимо передавать функции `sameColor()` только объекты типа `Cylinder`.



### Вопросы для текущего контроля

- Что такое функция-“друг”? Какое ключевое слово используется для ее объявления?
- Нужно ли использовать при вызове функции-“друга” (для объекта) оператор “точка”?
- Может ли функция-“друг” одного класса быть членом другого класса?\*

**ВАЖНО!**

## 9.7 Структуры и объединения

В дополнение к ключевому слову `class` в C++ предусмотрена возможность создания еще двух “классовых” типов: *структур* и *объединений*.

### Структуры

Структуры “достались в наследство” от языка С и объявляются с помощью ключевого слова `struct` (`struct`-обявление синтаксически подобно `class`-об объявлению). Оба объявления создают “классовый” тип. В языке С структура

- Функция-“друг” – это функция, которая не является членом класса, но имеет доступ к закрытым членам этого класса. Функция-“друг” объявляется с использованием ключевого слова `friend`.
- Нет, функция-“друг” вызывается как обычная функция, не имеющая “членства”.
- Да, функция-“друг” одного класса может быть членом другого класса.

может содержать только члены данных, но это ограничение в C++ не действует. В C++ структура, по сути, представляет собой лишь альтернативный способ определения класса. Единственное различие в C++ между `struct`- и `class`-объявлениями состоит в том, что по умолчанию все члены открыты в структуре (`struct`) и закрыты в классе (`class`). Во всех остальных аспектах C++-структур и классы эквивалентны.

Рассмотрим пример структуры.

```
#include <iostream>
using namespace std;

struct Test {
    int get_i() { return i; }      // Эти члены открыты
    void put_i(int j) { i = j; }   // (public) по умолчанию.
private:
    int i;
};

int main()
{
    Test s;

    s.put_i(10);
    cout << s.get_i();

    return 0;
}
```

Эта простая программа определяет тип структуры `Test`, в которой функции `get_i()` и `put_i()` открыты, а член данных `i` закрыт. Обратите внимание на использование ключевого слова `private` для объявления закрытых элементов структуры.

В следующем примере показана эквивалентная программа, в которой вместо `struct`-объявления используется `class`-объявление.

```
#include <iostream>
using namespace std;

class Test {
    int i; // закрытый член по умолчанию
public:
```

```

int get_i() { return i; }
void put_i(int j) { i = j; }

};

int main()
{
    Test s;

    s.put_i(10);
    cout << s.get_i();

    return 0;
}

```

9

О классах подробнее

## Спросим у опытного программиста

**Вопрос.** Если структура и класс так похожи, то почему в C++ существуют оба варианта объявления "классового" типа?

**Ответ.** С одной стороны, действительно, в существовании структур и классов присутствует определенная избыточность, и многие начинающие программисты удивляются этому. Нередко можно даже услышать предложения ликвидировать одно из этих средств.

Однако не стоит забывать о том, что создатель C++ стремился обеспечить совместимость C++ и C. Согласно современному определению C++ стандартная C-структура является абсолютно допустимой C++-структурой. Но в C все члены структур открыты по умолчанию (в C вообще нет понятия открытых и закрытых членов). Вот почему члены C++-структур открытые (а не закрыты) по умолчанию. Поскольку ключевое слово `class` введено для поддержки инкапсуляции, в том, что его члены закрыты по умолчанию, есть определенный резон. Таким образом, чтобы избежать несовместимости с C в этом аспекте, C++-объявление структуры по умолчанию не должно отличаться от ее C-объявления. Поэтому-то и было добавлено новое ключевое слово. Но в перспективе можно говорить о более веской причине для отделения структур от классов. Поскольку тип `class` синтаксически отделен от типа `struct`, определение класса вполне открыто для эволюционных изменений, которые синтаксически могут оказаться несовместимыми с C-структурами. Так как мы имеем дело с двумя отдельными типами, будущее направление развития языка C++ не обременяется "моральными обязательствами", связанными с совместимостью с C-структурами.

C++-программисты тип `class` используют главным образом для определения формы объекта, который содержит функции-члены, а тип `struct` — для создания объектов, которые содержат только члены данных. Иногда для описания структуры, которая не содержит функции-члены, используется аббревиатура POD (Plain Old Data).

## Объединения

*Объединение* — это область памяти, которую разделяют несколько различных переменных. Объединение создается с помощью ключевого слова `union`. Его объявление, как нетрудно убедиться на следующем примере, подобно объявлению структуры.

```
union utype {
    short int i;
    char ch;
};
```

Здесь объявляется объединение, в котором значение типа `short int` и значение типа `char` разделяют одну и ту же область памяти. Необходимо сразу же прояснить один момент: невозможно сделать так, чтобы это объединение хранило и целочисленное значение, и символ одновременно, поскольку переменные `i` и `ch` накладываются (в памяти) друг на друга. Но программа в любой момент может обрабатывать информацию, содержащуюся в этом объединении, как целочисленное значение или как символ. Следовательно, объединение обеспечивает два (или больше) способа представления одной и той же порции данных.

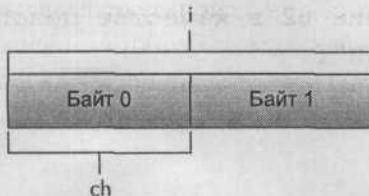
Переменную объединения можно объявить, разместив ее имя в конце его объявления либо воспользовавшись отдельной инструкцией объявления. Чтобы объявить переменную объединения именем `u_var` типа `utype`, достаточно записать следующее.

```
utype u_var;
```

В переменной объединения `u_var` как переменная `i` типа `short int`, так и символьная переменная `ch` занимают одну и ту же область памяти. (Безусловно, переменная `i` занимает два байта, а символьная переменная `ch` использует только один.) Как переменные `i` и `ch` разделяют одну область памяти, показано на рис. 9.1.

Согласно “идеологии” C++ объединение, по сути, является классом, в котором все элементы хранятся в одной области памяти. Поэтому объединение может включать конструкторы и деструкторы, а также функции-члены. Поскольку объединение унаследовано от языка C, его члены открыты (а не закрыты) по умолчанию.

Рассмотрим программу, в которой объединение используется для отображения символов, составляющих младший и старший байты короткого целочисленного значения (в предположении, что “размер” типа `short int` составляет два байта).



*Рис. 9.1. Переменные i и ch вместе используют объединение u\_var*

// Демонстрация использования объединения.

```
#include <iostream>
using namespace std;

union u_type {
    u_type(short int a) { i = a; };
    u_type(char x, char y) { ch[0] = x; ch[1] = y; }

    void showchars() {
        cout << ch[0] << " ";
        cout << ch[1] << "\n";
    }

    short int i; // Эти члены данных объединения
    char ch[2]; // разделяют одну и ту же область памяти.
};

int main()
{
    u_type u(1000);
    u_type u2('X', 'Y');

    // Данные в объекте типа u_type можно использовать в
    // качестве целочисленного значения или двух символов.
    cout << "Объединение u в качестве целого числа: ";
    cout << u.i << "\n";
    cout << "Объединение u в качестве двух символов: ";
    u.showchars();
}
```

```

cout << "Объединение u2 в качестве целого числа: ";
cout << u2.i << "\n";
cout << "Объединение u2 в качестве двух символов: ";
u2.showchars();

return 0;
}

```

Результаты выполнения этой программы таковы.

Объединение u в качестве целого числа: 1000

Объединение u в качестве двух символов: щ !

Объединение u2 в качестве целого числа: 22872

Объединение u2 в качестве двух символов: X Y

Как подтверждают эти результаты, используя объединение типа `u_type`, на одни и те же данные можно “смотреть” по-разному.

Подобно структуре, C++-объединение также произошло от своего С-предшественника. Но в C++ оно имеет более широкие “классовые” возможности (ведь в С объединения могут включать только члены данных, а функции и конструкторы не разрешены). Однако лишь то, что C++ наделяет “свои” объединения более мощными средствами и большей степенью гибкости, не означает, что вы непременно должны их использовать. Если вас вполне устраивает объединение с традиционным стилем поведения, вы вольны именно таким его и использовать. Но в случаях, когда можно инкапсулировать данные объединения вместе с функциями, которые их обрабатывают, все же стоит воспользоваться C++-возможностями, что придаст вашей программе дополнительные преимущества.

При использовании C++-объединений необходимо помнить о некоторых ограничениях, связанных с их применением. Большая их часть связана со средствами языка, которые рассматриваются ниже в этой книге, но все же здесь стоит их упомянуть. Прежде всего, объединение не может наследовать класс и не может быть базовым классом. Объединение не может иметь виртуальные функции-члены. Статические переменные не могут быть членами объединения. В объединении нельзя использовать ссылки. Членом объединения не может быть объект, который перегружает оператор “=” . Наконец, членом объединения не может быть объект, в котором явно определен конструктор или деструктор.

## Анонимные объединения

В C++ предусмотрен специальный тип объединения, который называется *анонимным*. Анонимное объединение не имеет наименования типа, и поэтому объект

такого объединения объявить невозможно. Но анонимное объединение сообщает компилятору о том, что его члены разделяют одну и ту же область памяти. При этом обращение к самим переменным объединения происходит непосредственно, без использования оператора "точка".

Рассмотрим такой пример.

// Демонстрация использования анонимного объединения.

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    // Определяем анонимное объединение.
    union {
        long l;
        double d;
        char s[4];
    } ;

    // К членам анонимного объединения выполняется
    // обращение напрямую.
    l = 100000;
    cout << l << " ";
    d = 123.2342;
    cout << d << " ";
    strcpy(s, "hi");
    cout << s;

    return 0;
}
```

Как видите, к элементам объединения можно получить доступ так же, как к обычным переменным. Несмотря на то что они объявлены как часть анонимного объединения, их имена находятся на том же уровне области видимости, что и другие локальные переменные того же блока. Таким образом, чтобы избежать конфликта имен, член анонимного объединения не должен иметь имя, совпадающее с именем любой другой переменной, объявленной в той же области видимости.

Все описанные выше ограничения, налагаемые на использование объединений, применимы и к анонимным объединениям. К ним необходимо добавить

следующие. Анонимные объединения должны содержать только члены данных (функции-члены недопустимы). Анонимные объединения не могут включать `private`- или `protected`-элементы. (Спецификатор `protected` описан в модуле 10.) Наконец, глобальные анонимные объединения должны быть объявлены с использованием спецификатора `static`.

**ВАЖНО!**

## 9.8. Ключевое слово `this`

Прежде чем переходить к теме перегрузки операторов, необходимо рассмотреть ключевое слово `this`. При каждом вызове функции-члена ей автоматически передается указатель, именуемый `this`, на объект, для которого вызывается эта функция. Указатель `this` — это *неявный* параметр, принимаемый всеми функциями-членами. Следовательно, в любой функции-члене указатель `this` можно использовать для ссылки на вызывающий объект.

Как вы знаете, функция-член может иметь прямой доступ к закрытым (`private`) членам данных своего класса.

Например, у нас определен такой класс.

```
class Test {
    int i;
    void f() { ... };
    // ...
};
```

В функции `f()` для присваивания члену `i` значения 10 можно использовать следующую инструкцию.

```
i = 10;
```

В действительности предыдущая инструкция представляет собой сокращенную форму следующей.

```
this->i = 10;
```

Чтобы понять, как работает указатель `this`, рассмотрим следующую короткую программу.

```
// Использование указателя this.
```

```
#include <iostream>
using namespace std;

class Test {
```

```

int i;
public:
    void load_i(int val) {
        this->i = val; // то же самое, что i = val
    }
    int get_i() {
        return this->i; // то же самое, что return i
    }
};

int main()
{
    Test o;

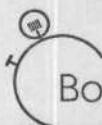
    o.load_i(100);
    cout << o.get_i();

    return 0;
}

```

При выполнении эта программа отображает число 100. Безусловно, предыдущий пример тривиален, но в нем показано, как можно использовать указатель `this`. Скорее вы поймете, почему указатель `this` так важен для программирования на C++.

И еще. Функции-“друзья” не имеют указателя `this`, поскольку они не являются членами класса. Только функции-члены имеют указатель `this`.



### Вопросы для текущего контроля

---

1. Может ли структура (`struct`) содержать функции-члены?
  2. Какова основная характеристика объединения (`union`)?
  3. На что указывает слово `this`?\*
- 

1. Да, структура (`struct`) может содержать функции-члены.
2. Члены данных объединения разделяют одну и ту же область памяти.
3. Слово `this` указывает на объект, для которого была вызвана функция-член.

ВАЖНО!

## 9.9. Перегрузка операторов

Остальная часть модуля посвящена одному из самых интересных средств — *перегрузке операторов*. В C++ операторы можно перегружать для “классовых” типов, определяемых программистом. Принципиальный выигрыш от перегрузки операторов состоит в том, что она позволяет органично интегрировать новые типы данных в среду программирования.

Перегружая оператор, программист определяет его назначение для конкретного класса. Например, класс, который определяет связный список, может использовать оператор “+” для добавления объекта к списку. Класс, которые реализует стек, может использовать оператор “+” для записи объекта в стек. В какомнибудь другом классе тот же оператор “+” мог бы служить для совершенно иной цели. При перегрузке оператора ни одно из оригинальных его значений не теряется. Перегруженный оператор (в своем новом качестве) работает как совершенновый новый оператор. Поэтому перегрузка оператора “+” для обработки, например, связного списка не приведет к изменению его функции (т.е. операции сложения) по отношению к целочисленным значениям.

Перегрузка операторов тесно связана с перегрузкой функций. Чтобы перегрузить оператор, необходимо определить значение новой операции для класса, к которому она будет применяться. Для этого создается функция `operator` (операторная функция), которая определяет действие этого оператора. Общий формат функции `operator` таков.

```
тип имя_класса::operator#(список_аргументов)
{
    операция_над_классом
}
```

Здесь перегружаемый оператор обозначается символом “#”, а элемент `тип` представляет собой тип значения, возвращаемого заданной операцией. И хотя он в принципе может быть любым, тип значения, возвращаемого функцией `operator`, часто совпадает с именем класса, для которого перегружается данный оператор. Такая корреляция облегчает использование перегруженного оператора в составных выражениях. Как будет показано ниже, конкретное значение элемента `список_аргументов` определяется несколькими факторами.

Операторная функция может быть членом класса или не быть им. Операторные функции, не являющиеся членами класса, часто определяются как его “друзья”. Операторные функции-члены и функции-не члены класса различаются по форме перегрузки. Каждый из вариантов мы рассмотрим в отдельности.



На заметку

Поскольку в C++ определено довольно много операторов, в этой книге невозможно описать все аспекты обширной темы перегрузки операторов. Для получения более подробной информации обратитесь к моей книге *Полный справочник по C++*, Издательский дом "Вильямс".

ВАЖНО!

## 9.10. Перегрузка операторов с использованием функций-членов

Начнем с простого примера. В следующей программе создается класс ThreeD, который обеспечивает поддержку координат объекта в трехмерном пространстве. Для класса ThreeD здесь реализована перегрузка операторов "+" и "=" . Итак, рассмотрим внимательно код этой программы.

// Перегрузка операторов "+" и "=" для класса ThreeD.

```
#include <iostream>
using namespace std;

class ThreeD {
    int x, y, z; // 3-мерные координаты
public:
    ThreeD() { x = y = z = 0; }
    ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

    ThreeD operator+(ThreeD op2); // Операнд op1
                                    // передается неявно.
    ThreeD operator=(ThreeD op2); // Операнд op1
                                    // передается неявно.

    void show();
};

// Перегрузка оператора "+" для класса ThreeD.
ThreeD ThreeD::operator+(ThreeD op2)
{
    ThreeD temp;
```

## 432 Модуль 9. О классах подробнее

```
temp.x = x + op2.x; // При выполнении операции сложения
temp.y = y + op2.y; // целочисленных значений сохраняется
temp.z = z + op2.z; // оригинальный смысл оператора "+".  
  
return temp; // Возвращается новый объект. Аргументы
// функции не изменяются.  
}  
  
// Перегрузка оператора присваивания (=) для класса ThreeD.  
ThreeD ThreeD::operator=(ThreeD op2)
{
    x = op2.x; // При выполнении операции присваивания
    y = op2.y; // целочисленных значений сохраняется
    z = op2.z; // оригинальный смысл оператора "=".  
  
return *this; // Возвращается модифицированный объект.
}  
  
// Отображение координат X, Y, Z.  
void ThreeD::show()
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}  
  
int main()
{
    ThreeD a(1, 2, 3), b(10, 10, 10), c;  
  
cout << "Исходные значения координат объекта a: ";
    a.show();
    cout << "Исходные значения координат объекта b: ";
    b.show();
    cout << "\n";  
  
c = a + b; // сложение объектов a и b
    cout << "Значения координат объекта c после c = a + b: ";
```

```

c.show();
cout << "\n";
c = a + b + c; // сложение объектов a, b и с
cout << "Значения координат объекта с после c = a + b + c: ";
c.show();
cout << "\n";
c = b = a; // демонстрация множественного присваивания
cout << "Значения координат объекта с после c = b = a: ";
c.show();
cout << "Значения координат объекта b после c = b = a: ";
b.show();

return 0;
}

```

Эта программа генерирует такие результаты.

Исходные значения координат объекта a: 1, 2, 3

Исходные значения координат объекта b: 10, 10, 10

Значения координат объекта с после c = a + b: 11, 12, 13

Значения координат объекта с после c = a + b + c: 22, 24, 26

Значения координат объекта с после c = b = a: 1, 2, 3

Значения координат объекта b после c = b = a: 1, 2, 3

Исследуя код этой программы, вы, вероятно, удивились, увидев, что обе операторные функции имеют только по одному параметру, несмотря на то, что они перегружают бинарные операции. Это, на первый взгляд, "вопиющее" противоречие можно легко объяснить. Дело в том, что при перегрузке бинарного оператора с использованием функции-члена ей передается явным образом только один аргумент. Второй же неявно передается через указатель `this`. Таким образом, в строке

`temp.x = x + op2.x;`

под членом `x` подразумевается член `this->x`, т.е. член `x` связывается с объектом, который вызывает данную операторную функцию. Во всех случаях неявно передается объект, указываемый слева от символа операции, который стал причиной

вызыва операторной функции. Объект, располагаемый с правой стороны от символа операции, передается этой функции в качестве аргумента. В общем случае при использовании функции-члена для перегрузки унарного оператора параметры не используются вообще, а для перегрузки бинарного – только один параметр. (Тернарный оператор “?” перегружать нельзя.) В любом случае объект, который вызывает операторную функцию, неявно передается через указатель `this`.

Чтобы понять, как работает механизм перегрузки операторов, рассмотрим внимательно предыдущую программу, начиная с перегруженного оператора “+”. При обработке двух объектов типа `ThreeD` оператором “+” выполняется сложение значений соответствующих координат, как показано в функции `operator+` (). Но заметьте, что эта функция не модифицирует значение ни одного операнда. В качестве результата операции эта функция возвращает объект типа `ThreeD`, который содержит результаты попарного сложения координат двух объектов. Чтобы понять, почему операция “+” не изменяет содержимое ни одного из объектов-участников, рассмотрим стандартную арифметическую операцию сложения, примененную, например, к числам 10 и 12. Результат операции  $10 + 12$  равен 22, но при его получении ни 10, ни 12 не были изменены. Хотя не существует правила, которое бы не позволяло перегруженному оператору изменять значение одного из его operandов, все же лучше, чтобы он не противоречил общепринятым нормам и оставался в согласии со своим оригинальным назначением.

Обратите внимание на то, что функция `operator+` () возвращает объект типа `ThreeD`. Несмотря на то что она могла бы возвращать значение любого допустимого в C++ типа, тот факт, что она возвращает объект типа `ThreeD`, позволяет использовать оператор “+” в таких составных выражениях, как `a+b+c`. Часть этого выражения, `a+b`, генерирует результат типа `ThreeD`, который затем суммируется с объектом `c`. И если бы эта часть выражения генерировала значение иного типа (а не типа `ThreeD`), такое составное выражение попросту не работало бы.

В отличие от оператора “+”, оператор присваивания приводит к модификации одного из своих аргументов. (Прежде всего, это составляет саму суть присваивания.) Поскольку функция `operator=` () вызывается объектом, который расположен слева от символа присваивания (`=`), именно этот объект и модифицируется в результате операции присваивания. После выполнения этой операции значение, возвращаемое перегруженным оператором, содержит объект, указанный слева от символа присваивания. (Такое положение вещей вполне согласуется с традиционным действием оператора “=”.) Например, чтобы можно было выполнять инструкции, подобные следующей

`a = b = c = d;`,

необходимо, чтобы операторная функция `operator=()` возвращала объект, адресуемый указателем `this`, и чтобы этот объект располагался слева от опера-

тора “=” . Это позволит выполнить любую цепочку присваиваний. Операция присваивания — это одно из самых важных применений указателя `this`.

В предыдущей программе не было насущной необходимости перегружать оператор “=” , поскольку оператор присваивания, по умолчанию предоставляемый языком C++, вполне соответствует требованиям класса `ThreeD`. (Как разъяснялось выше в этом модуле, при выполнении операции присваивания по умолчанию создается побитовая копия объекта.) Здесь оператор “=” был перегружен исключительно в демонстрационных целях. В общем случае оператор “=” следует перегружать только тогда, когда побитовую копию использовать нельзя. Поскольку применения оператора “=” по умолчанию вполне достаточно для класса `ThreeD`, в последующих примерах этого модуля мы перегружать его не будем.

## О значении порядка operandов

Перегружая бинарные операторы, помните, что во многих случаях порядок следования operandов имеет значение. Например, выражение `A + B` коммутативно, а выражение `A - B` — нет. (Другими словами, `A - B` не то же самое, что `B - A`!) Следовательно, реализуя перегруженные версии некоммутативных операторов, необходимо помнить, какой operand стоит слева от символа операции, а какой — справа от него. Например, в следующем фрагменте кода демонстрируется перегрузка оператора вычитания для класса `ThreeD`.

```
// Перегрузка оператора вычитания.
ThreeD ThreeD::operator-(ThreeD op2)
{
    ThreeD temp;

    temp.x = x - op2.x;
    temp.y = y - op2.y;
    temp.z = z - op2.z;
    return temp;
}
```

Помните, что именно левый operand вызывает операторную функцию. Правый operand передается в явном виде.

## Использование функций-членов для перегрузки унарных операторов

Можно также перегружать такие унарные операторы, как “`++`”, “`--`”, или унарные “`-`” и “`+`”. Как упоминалось выше, при перегрузке унарного оператора с помо-

## 436 Модуль 9. О классах подробнее

шью функции-члена операторной функции ни один объект не передается явным образом. Операция же выполняется над объектом, который генерирует вызов этой функции через неявно переданный указатель `this`. Например, рассмотрим программу, в которой для объектов типа `ThreeD` определяется операция инкремента.

```
// Перегрузка унарного оператора “++”.  
  
#include <iostream>  
using namespace std;  
  
class ThreeD {  
    int x, y, z; // 3-мерные координаты  
public:  
    ThreeD() { x = y = z = 0; }  
    ThreeD(int i, int j, int k) {x = i; y = j; z = k; }  
  
    ThreeD operator++(); // префиксная версия оператора “++”  
  
    void show();  
};  
  
// Перегрузка префиксной версии оператора “++” для  
// класса ThreeD.  
ThreeD ThreeD::operator++()  
{  
    x++; // инкремент координат x, y и z  
    y++;  
    z++;  
    return *this; // Возвращаем инкрементированный объект.  
}  
  
// Отображение координат X, Y, Z.  
void ThreeD::show()  
{  
    cout << x << ", ";  
    cout << y << ", ";  
    cout << z << "\n";  
}  
  
int main()
```

```

    ThreeD a(1, 2, 3);

    cout << "Исходные значения координат объекта а: ";
    a.show();

    ++a; // инкремент объекта а
    cout << "Значения координат после ++a: ";
    a.show();

    return 0;
}

```

Результаты выполнения этой программы таковы.

Исходное значение координат объекта а: 1, 2, 3

Значение координат после ++a: 2, 3, 4

Как видно по последней строке результатов программы, операторная функция `operator++()` инкрементирует каждую координату объекта и возвращает модифицированный объект, что вполне согласуется с традиционным действием оператора “`++`”.

Как вы знаете, операторы “`++`” и “`--`” имеют префиксную и постфиксную формы. Например, оператор инкремента можно использовать в форме

`++x;`

или

`x++;`

Как отмечено в комментариях к предыдущей программе, функция `operator++()` определяет префиксную форму оператора “`++`” для класса `ThreeD`. Но нам ничего не мешает перегрузить и постфиксную форму. Прототип постфиксной формы оператора “`++`” для класса `three_d` имеет следующий вид.

`ThreeD operator++(int notused);`

Параметр `notused` не используется самой функцией (игнорируется). Он служит индикатором для компилятора, позволяющим отличить префиксную форму оператора инкремента от постфиксной. (Этот параметр также используется в качестве признака постфиксной формы и для оператора декремента.) Ниже приводится один из возможных способов реализации постфиксной версии оператора “`++`” для класса `ThreeD`.

// Перегрузка постфиксной версии оператора “`++`”.

`ThreeD ThreeD::operator++(int notused) // Обратите внимание`

## 438 Модуль 9. О классах подробнее

```
// на использование
// параметра notused.
{
    ThreeD temp = *this; // сохранение исходного значения

    x++; // инкремент координат x, y и z
    y++;
    z++;
    return temp; // возврат исходного значения
}
```

Обратите внимание на то, что эта функция сохраняет текущее значение операнда путем выполнения такой инструкции.

```
ThreeD temp = *this;
```

Затем сохраненное значение операнда (в объекте `temp`) возвращается с помощью инструкции `return`. Следует иметь в виду, что традиционный постфиксный оператор инкремента сначала получает значение операнда, а затем его инкрементирует. Следовательно, прежде чем инкрементировать текущее значение операнда, его нужно сохранить, а затем и возвратить (не забывайте, что постфиксный оператор инкремента не должен возвращать модифицированное значение своего операнда).

В следующей версии исходной программы реализованы обе формы оператора `++`.

```
// Демонстрация перегрузки оператора “++” с
// использованием его префиксной и постфиксной форм.
```

```
#include <iostream>
using namespace std;

class ThreeD {
    int x, y, z; // 3-мерные координаты
public:
    ThreeD() { x = y = z = 0; }
    ThreeD(int i, int j, int k) {x = i; y = j; z = k; }

    ThreeD operator++(); // префиксная версия оператора “++”
    ThreeD operator++(int notused); // постфиксная версия
                                    // оператора “++”

    void show() ;
};
```

```

// Перегрузка префиксной версии оператора "++".
ThreeD ThreeD::operator++()
{
    x++; // инкрементируем координаты x, у и z
    y++;
    z++;
    return *this; // возвращаем измененное значение объекта
}

// Перегрузка постфиксной версии оператора "++".
ThreeD ThreeD::operator++(int notused)
{
    ThreeD temp = *this; // сохраняем исходное значение

    x++; // инкрементируем координаты x, у и z
    y++;
    z++;
    return temp; // возвращаем исходное значение объекта
}

// Отображение координат X, Y, Z.
void ThreeD::show()
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}

int main()
{
    ThreeD a(1, 2, 3);
    ThreeD b;

    cout << "Исходные значения координат объекта а: ";
    a.show();

    cout << "\n";
}

```

## 440 Модуль 9. О классах подробнее

```
++a; // Вызывается функция префиксной формы инкремента.  
cout << "Значения координат после ++a: ";  
a.show();  
  
a++; // Вызывается функция постфиксной формы инкремента.  
cout << "Значения координат после a++: ";  
a.show();  
  
cout << "\n";  
  
b = ++a; // Объект b получает значение объекта a  
// после инкремента.  
cout << "Значения координат объекта a после b = ++a: ";  
a.show();  
cout << "Значения координат объекта b после b = ++a: ";  
b.show();  
  
cout << "\n";  
  
b = a++; // Объект a получает значение объекта с  
// до инкремента.  
cout << "Значения координат объекта a после b = a++: ";  
a.show();  
cout << "Значения координат объекта b после b = a++: ";  
b.show();  
  
return 0;  
}
```

При выполнении программа генерирует такие результаты.  
Исходные значения координат объекта a: 1, 2, 3

Значения координат после ++a: 2, 3, 4

Значения координат после a++: 3, 4, 5

Значения координат объекта a после b = ++a: 4, 5, 6

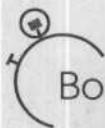
Значения координат объекта b после b = ++a: 4, 5, 6

Значения координат объекта a после b = a++: 5, 6, 7

Значения координат объекта b после b = a++: 4, 5, 6

Помните, что если символ “`++`” стоит перед операндом, вызывается операторная функция `operator++()`, а если после операнда – операторная функция `operator++(int notused)`. Тот же подход используется и для перегрузки префиксной и постфиксной форм оператора декремента для любого класса. В качестве упражнения самостоятельно определите оператор декремента для класса `ThreeD`.

Интересно отметить, что ранние версии языка C++ не содержали различий между префиксной и постфиксной формами операторов инкремента и декремента. Тогда в обоих случаях вызывалась префиксная форма операторной функции. Это следует иметь в виду, если вам придется работать со старыми C++-программами.



## Вопросы для текущего контроля

- Перегрузка операторов должна осуществляться относительно класса. Верно ли это?
- Сколько параметров имеет операторная функция-член для реализации бинарного оператора?
- Левый операнд для операторной функции-члена передается через \_\_\_\_\_.\*

**ВАЖНО!**

## 9.11. Перегрузка операторов с использованием функций-не членов класса

Перегрузку оператора для класса можно реализовать и с использованием функции, не являющейся членом этого класса. Такие функции часто определяются “друзьями” класса. Как упоминалось выше, функции-не члены (в том числе и функции-“друзья”) не имеют указателя `this`. Следовательно, если для перегрузки бинарного оператора используется функция-“друг”, явным образом передаются оба операнда. Если же с помощью функции-“друга” перегружается унарный оператор, операторной функции передается один оператор. С использованием функций-не членов класса нельзя перегружать такие операторы: `=`, `( )`, `[ ]` и `->`.

- Верно, оператор перегружается относительно класса.
- Бинарная операторная функция-член имеет один параметр, который получает значение правого операнда.
- Левый операнд для операторной функции-члена передается через указатель `this`.

## 442 Модуль 9. О классах подробнее

Например, в следующей программе для перегрузки оператора "+" для класса ThreeD вместо функции-члена используется функция-“друг”.

```
// Перегрузка оператора "+" с помощью функции-“друга”.
```

```
#include <iostream>
using namespace std;

class ThreeD {
    int x, y, z; // 3-мерные координаты
public:
    ThreeD() { x = y = z = 0; }
    ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

    // Здесь функция operator+() объявлена "другом" класса
    // ThreeD. Обратите внимание на необходимость
    // использования в этом случае двух параметров.
    friend ThreeD operator+(ThreeD op1, ThreeD op2);

    void show();
};

// Реализация операторной функции-“друга” operator+().
ThreeD operator+(ThreeD op1, ThreeD op2)
{
    ThreeD temp;

    temp.x = op1.x + op2.x;
    temp.y = op1.y + op2.y;
    temp.z = op1.z + op2.z;
    return temp;
}

// Отображение координат X, Y, Z.
void ThreeD::show()
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}
```

```

int main()
{
    ThreeD a(1, 2, 3), b(10, 10, 10), c;

    cout << "Исходные значения координат объекта a: ";
    a.show();
    cout << "Исходные значения координат объекта b: ";
    b.show();

    cout << "\n";

    c = a + b; // сложение объектов a и b
    cout << "Значения координат объекта с после c = a + b: ";
    c.show();

    cout << "\n";

    c = a + b + c; // сложение объектов a, b и с
    cout << "Значения координат объекта с после c = a + b + c: ";
    c.show();

    cout << "\n";

    c = b = a; // демонстрация множественного присваивания
    cout << "Значения координат объекта с после c = b = a: ";
    c.show();
    cout << "Значения координат объекта b после c = b = a: ";
    b.show();

    return 0;
}

```

Результаты выполнения этой программы таковы.

Исходные значения координат объекта a: 1, 2, 3

Исходные значения координат объекта b: 10, 10, 10

Значения координат объекта с после c = a + b: 11, 12, 13

Значения координат объекта с после c = a + b + c: 22, 24, 26

## 444 Модуль 9. О классах подробнее

Значения координат объекта с после `c = b = a: 1, 2, 3`

Значения координат объекта b после `c = b = a: 1, 2, 3`

Как видите, операторной функции `operator+()` теперь передаются два операнда. Левый операнд передается параметру `op1`, а правый — параметру `op2`.

Во многих случаях при перегрузке операторов с помощью функций-“друзей” нет никакого преимущества по сравнению с использованием функций-членов класса. Однако возможна ситуация (когда нужно, чтобы слева от бинарного оператора стоял объект встроенного типа), в которой функция-“друг” оказывается чрезвычайно полезной. Чтобы понять это, рассмотрим следующее. Как вы знаете, указатель на объект, который вызывает операторную функцию-член, передается с помощью ключевого слова `this`. При использовании бинарного оператора функцию вызывает объект, расположенный слева от него. И это замечательно при условии, что левый объект определяет заданную операцию. Например, предположим, что у нас есть некоторый объект `T`, для которого определена операция сложения с целочисленным значением, тогда следующая запись представляет собой вполне допустимое выражение.

`T = T + 10; // будет работать`

Поскольку объект `T` стоит слева от оператора “`+`”, он вызывает перегруженную операторную функцию, которая (предположительно) способна выполнить операцию сложения целочисленного значения с некоторым элементом объекта `T`. Но эта инструкция работать не будет.

`T = 10 + T; // не будет работать`

Дело в том, что в этой инструкции объект, расположенный слева от оператора “`+`”, представляет собой целое число, т.е. значение встроенного типа, для которого не определена ни одна операция, включающая целое число и объект классового типа.

Решение описанной проблемы состоит в перегрузке оператора “`+`” с использованием двух функций-“друзей”. В этом случае операторной функции явным образом передаются оба аргумента, и она вызывается подобно любой другой перегруженной функции, т.е. на основе типов ее аргументов. Одна версия операторной функции `operator+()` будет обрабатывать аргументы `объект + int-значение`, а другая — аргументы `int-значение + объект`. Перегрузка оператора “`+`” (или любого другого бинарного оператора) с использованием функций-“друзей” позволяет ставить значение встроенного типа как справа, так и слева от оператора. Реализация этого решения показана в следующей программе. Здесь определяются две версии операторной функции `operator+()` для объектов типа `ThreeD`. Обе они выполняют сложение целочисленного значения с каждой из координат

ThreeD-объекта. При этом целочисленное значение может находиться как слева, так и справа от оператора.

```
// Перегрузка операторной функции operator+() для таких
// наборов операндов: (int + объект) и (объект + int).
```

```
#include <iostream>
using namespace std;

class ThreeD {
    int x, y, z; // 3-мерные координаты
public:
    ThreeD() { x = y = z = 0; }
    ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

    friend ThreeD operator+(ThreeD op1, int op2); // объект +
  // int
    friend ThreeD operator+(int op1, ThreeD op2); // int +
  // объект

    void show() ;
};

// Эта версия обрабатывает аргументы
// ThreeD-объект + int-значение.
ThreeD operator+(ThreeD op1, int op2)
{
    ThreeD temp;

    temp.x = op1.x + op2;
    temp.y = op1.y + op2;
    temp.z = op1.z + op2;
    return temp;
}

// Эта версия обрабатывает аргументы
// int-значение + ThreeD-объект
ThreeD operator+(int op1, ThreeD op2)
{
    ThreeD temp;
```

## 446 Модуль 9. О классах подробнее

```
temp.x = op2.x + op1;
temp.y = op2.y + op1;
temp.z = op2.z + op1;
return temp;
}

// Отображение координат X, Y, Z.
void ThreeD::show()
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}

int main()
{
    ThreeD a(1, 2, 3), b;

    cout << "Исходные значения координат объекта a: ";
    a.show();

    cout << "\n";

    b = a + 10; // объект + int-значение
    cout << "Значения координат объекта b после b = a + 10: ";
    b.show();

    cout << "\n";

    // Здесь слева от оператора сложения находится значение
    // встроенного типа.
    b = 10 + a; // int-значение + объект
    cout << "Значения координат объекта b после b = 10 + a: ";
    b.show();

    return 0;
}
```

Вот как выглядят результаты выполнения этой программы.  
Исходные значения координат объекта a: 1, 2, 3

Значения координат объекта `b` после `b = a + 10: 11, 12, 13`

Значения координат объекта `b` после `b = 10 + a: 11, 12, 13`

Поскольку операторная функция `operator+()` перегружается дважды, мы можем предусмотреть два способа участия целого числа и объекта типа `ThreeD` в операции сложения.

## Использование функций-“друзей” для перегрузки унарных операторов

С помощью функций-“друзей” можно перегружать и унарные операторы. Но это потребует от программиста дополнительных усилий. Если вы хотите перегрузить операторы инкремента (`++`) или декремента (`--`), то такой операторной функции необходимо передать объект по ссылке. Поскольку ссылочный параметр представляет собой неявный указатель на аргумент, то изменения, внесенные в параметр, повлияют и на аргумент. Таким образом, применение ссылочного параметра позволяет функции успешно инкрементировать или декрементировать объект, используемый в качестве опаранда.

Если для перегрузки операторов инкремента или декремента используется функция-“друг”, ее префиксная форма принимает один параметр (который и является операндом), а постфиксная форма — два параметра (вторым является целочисленное значение, которое не используется).

Ниже приведен код перегрузки двух форм инкремента объекта трехмерных координат, в котором используется операторная функция-“друг” `operator++()` для класса `ThreeD`.

```
/* Перегрузка префиксной формы оператора “++”
   с помощью функции-“друга”. Для этого
   необходимо использование ссылочного параметра. */

ThreeD operator++(ThreeD &op1)
{
    op1.x++;
    op1.y++;
    op1.z++;
    return op1;
}

/* Перегрузка постфиксной формы оператора “++”
   с помощью функции-“друга”. Для этого также
   необходимо использование ссылочного параметра. */
```

```
ThreeD operator++(ThreeD &opl, int notused)
{
    ThreeD temp = opl;

    opl.x++;
    opl.y++;
    opl.z++;

    return temp;
}
```

### Вопросы для текущего контроля



1. Сколько параметров имеет бинарная операторная функция-не член класса?
2. Как должен быть передан операнд для перегрузки оператора инкремента (++) при использовании операторной функции-не члена?
3. Важным достоинством применения операторных функций-“друзей” является тот факт, что они позволяют в качестве левого операнда использовать значение встроенного типа. Верно ли это?\*

### Советы по реализации перегрузки операторов

Действие перегруженного оператора применительно к классу, для которого он определяется, не обязательно должно иметь отношение к стандартному действию этого оператора применительно ко встроенным C++-типам. Например, операторы “<<” и “>>”, применяемые к объектам `cout` и `cin`, имеют мало общего с аналогичными операторами, применяемыми к значениям целочисленного типа. Но для улучшения структурированности и читабельности программного кода создаваемый перегруженный оператор должен по возможности отражать исходное назначение того или иного оператора. Например, оператор “+”, перегруженный для класса `ThreeD`, концептуально подобен оператору “+”, определенному для целочисленных типов. Ведь вряд ли есть логика в определении для класса,

1. Бинарная операторная функция-не член имеет два параметра.
2. Для перегрузки оператора инкремента (++) при использовании операторной функции-не члена операнд должен быть передан по ссылке.
3. Верно. Применение операторных функций-“друзей” позволяет в качестве левого операнда использовать значение встроенного типа (например, `int`).

например, оператора “+”, который по своему действию больше напоминает оператор деления (/). Таким образом, основная идея создания перегруженного оператора — наделить его новыми (нужными для вас) возможностями, которые, тем не менее, связаны с его первоначальным назначением.

## Спросим у опытного программиста

**Вопрос.** Существуют ли особенности перегрузки операторов отношения?

**Ответ.** Операторы отношений (например, “==” или “<”) также можно перегружать, причем делать это совсем нетрудно. Как правило, перегруженная операторная функция возвращает объект класса, для которого она перегружается. Однако перегруженный оператор отношения возвращает одно из двух возможных значений: `true` или `false`. Это соответствует обычному применению этих операторов и позволяет использовать их в условных выражениях. Вышеизложенное справедливо и в отношении логических операторов.

Чтобы показать, как можно реализовать перегруженный оператор отношения, рассмотрим пример перегрузки оператора “==” для уже знакомого нам класса `ThreeD`.

```
// Перегрузка оператора "==".
bool ThreeD::operator==( ThreeD op2)
{
    if((x == op2.x) && (y == op2.y) && (z == op2.z))
        return true;
    else
        return false;
}
```

Если считать, что операторная функция `operator==()` уже реализована, следующий фрагмент кода совершенно корректен.

```
ThreeD a(1, 1, 1), b(2, 2, 2);
// ...
if(a == b) cout << "a равно b\n";
else cout << "a не равно b\n";
```

На перегрузку операторов налагается ряд ограничений. Во-первых, нельзя изменять приоритет оператора. Во-вторых, нельзя изменять количество операндов, принимаемых оператором, хотя операторная функция могла бы игнорировать любой operand. Наконец, за исключением оператора вызова функции (о нем речь впереди), операторные функции не могут иметь аргументов по умолчанию.

Почти все C++-операторы можно перегружать (в том числе и оператор индексирования массива “[ ]”, вызова функции “()”, а также оператор “->”). Операторы, перегрузка которых запрещена, перечислены ниже.

:: . \* ?

Оператор “.\*” — это оператор специального назначения, рассмотрение которого выходит за рамки этой книги.

## Проект 9.1. Создание класса множества

Используя перегрузку операторов, можно создавать классы, которые полностью интегрируются в среду программирования C++. Определив необходимый набор операций, мы разрешаем использовать в программе некоторый классовый тип подобно тому, как мы используем любые встроенные типы. Объекты такого класса мы сможем тогда обрабатывать с помощью известных операторов и включать их в выражения. Чтобы продемонстрировать создание и интеграцию нового класса в среду C++, в этом проекте мы создадим класс Set, который определяет тип множества.

Но сначала уточним, что мы понимаем под множеством. Для данного проекта ограничимся таким определением: множество — это коллекция уникальных элементов. Другими словами, множество не может содержать двух одинаковых элементов. Порядок следования элементов в множестве несущественен. Это значит, что множество

{ A, B, C }

совпадает со множеством

{ A, C, B }

Множество может быть пустым.

Множества поддерживают некоторый набор операций. В нашем проекте мы реализуем такие:

- включение элемента в множество;
- удаление элемента из множества;
- объединение множеств;
- получение разности множеств.

Такие операции, как внесение элемента в множество и удаление элемента из него него, не требуют дополнительных разъяснений. О других же необходимо сказать несколько слов.

*Объединение* двух множеств представляет собой множество, которое содержит все элементы из объединяемых множеств. (При этом дублирование элементов не

разрешено.) Для выполнения операции объединения множеств будем использовать оператор “+”.

Разность множеств — это множество, содержащее те элементы из первого множества, которые не являются частью второго. Для выполнения операции вычитания множеств будем использовать оператор “-”. Например, если даны множества S1 и S2, то при выполнении следующей инструкции элементы множества S2 будут удалены из множества S1, а результат помещен в S3.

$$S3 = S1 - S2$$

Если множества S1 и S2 одинаковы, то S3 будет пустым множеством.

Класс Set также будет включать функцию `isMember()`, которая определяет, присутствует ли заданный элемент в данном множестве.

Над множествами можно выполнять и другие операции. Некоторые из них предложены для разработки в разделе “Тест для самоконтроля по модулю 9”. Кроме того, вы можете самостоятельно расширить набор операций над множествами.

Для простоты разрабатываемый здесь класс Set предназначен для хранения символов, но те же самые принципы можно использовать для создания класса множества, способного хранить элементы и других типов.

## Последовательность действий

- Создайте новый файл с именем `Set.cpp`.
- Начните создание типа множества с объявления класса Set.

```
const int MaxSize = 100;

class Set {
    int len; // количество членов
    char members[MaxSize]; // На этом массиве и будет
                           // построено множество.

    /* Функция find() закрыта, поскольку она не используется
       вне класса Set. */
    int find(char ch); // выполняет поиск элемента

public:

    // Построение пустого множества.
    Set() { len = 0; }

    // Получение количества элементов в множестве.
```

```
int getLength() { return len; }

void showset(); // отображение множества
bool isMember(char ch); // проверка членства

Set operator +(char ch); // добавление элемента
Set operator -(char ch); // удаление элемента

Set operator +(Set ob2); // объединение множеств
Set operator -(Set ob2); // получение разности множеств
};
```

Каждое множество хранится в char-массиве `members`. Реальное количество членов в множестве содержится в переменной `len`. Максимальный размер множества определяется константой `MaxSize`, которая установлена равной числу 100. (При необходимости это значение можно увеличить.)

Конструктор `Set()` создает пустое множество, которое не содержит членов. Создавать другие конструкторы или определять конструктор копии для класса `Set` нет необходимости, поскольку для целей данного проекта вполне достаточно побитовой копии. Функция `getLength()` возвращает значение переменной `len`, в которой хранится текущее количество элементов в множестве.

**3.** Определение функций-членов начните с закрытой функции `find()`.

```
/* Функция возвращает индекс элемента, заданного
   параметром ch, или -1, если таковой не найден. */
int Set::find(char ch) {
    int i;

    for(i=0; i < len; i++)
        if(members[i] == ch) return i;

    return -1;
}
```

Эта функция определяет, является ли элемент, переданный параметру `ch`, членом данного множества. Если заданный элемент найден, функция возвращает его индекс, в противном случае – число `-1`. Функция `find()` определена закрытой, поскольку она не используется вне класса `Set`. Закрытая функция-член может быть вызвана только другими функциями-членами того же класса.

4. Включите в состав членов класса функцию `showset()`, которая отображает содержимое множества.

```
// Отображение содержимого множества.
void Set::showset() {
    cout << "{ ";
    for(int i=0; i<len; i++)
        cout << members[i] << " ";
    cout << "}\n";
}
```

5. Добавьте функцию `isMember()`, которая определяет, является ли заданный символ членом данного множества.

```
/* Функция возвращает значение true, если символ ch
является членом множества.

В противном случае возвращается значение false. */
bool Set::isMember(char ch) {
    if(find(ch) != -1) return true;
    return false;
}
```

Эта функция, чтобы определить, является ли символ `ch` членом данного множества, вызывает функцию `find()`. Если такой символ был найден в множестве, функция `isMember()` возвращает значение `true`, в противном случае — значение `false`.

6. Теперь перейдите к определению операторов множества. Начните с оператора `“+”`. Перегрузите оператор `“+”` для объектов типа `Set` так, чтобы он позволял включать в множество новый элемент.

```
// Добавление нового элемента в множество.
Set Set::operator +(char ch) {
    Set newset;

    if(len == MaxSize) {
        cout << "Множество заполнено.\n";
        return *this; // возвращает существующее множество
    }

    newset = *this; // дублирует существующее множество
```

## 454 Модуль 9. О классах подробнее

```
// Проверяем, существует ли уже в множестве такой
// элемент.
if(find(ch) == -1) { // Если элемент не найден, его
                      // можно добавлять в множество.
    // Включаем новый элемент в новое множество.
    newset.members[newset.len] = ch;
    newset.len++;
}
return newset; // возвращает модифицированное множество
}
```

Эта операторная функция требует некоторых пояснений. Прежде всего, она создает новое множество, в котором будет храниться содержимое исходного, а также символ, заданный параметром `ch`. Но перед добавлением нового символа проверяется, достаточно ли свободного места в множестве для принятия нового члена. Если достаточно, то содержимое исходного множества дублируется в объект `newset` типа `Set`. Затем вызывается функция `find()`, чтобы определить, не является ли кандидат в члены уже частью данного множества. Если нет, символ `ch` добавляется в множество `newset` и модифицируется переменная `len`. В любом случае в результате выполнения операции сложения возвращается новое множество `newset`, а исходное множество не изменяется.

7. Перегружаем оператор “–” так, чтобы с его помощью можно было удалить элемент из множества.

```
// Удаляем элемент из множества.
Set Set::operator -(char ch) {
    Set newset;
    int i = find(ch); // Переменная i получит значение -1,
                      // если заданный элемент не будет
                      // найден в множестве.

    // Копируем и уплотняем оставшиеся элементы.
    for(int j=0; j < len; j++)
        if(j != i) newset = newset + members[j];

    return newset;
}
```

Выполнение этой операторной функции начинается с создания нового пустого множества. Затем вызывается функция `find()`, чтобы определить индекс

заданного символа ch в исходном множестве. Вспомните, что функция find () возвращает значение -1, если символ ch не является членом множества. Затем в новое множество помещаются элементы исходного множества за исключением элемента, индекс которого совпадает с полученным в результате выполнения функции find (). Таким образом, новое множество будет содержать все элементы исходного, но без символа ch. Если символ ch не является частью исходного множества, новое множество будет эквивалентно исходному.

8. Снова перегружаем операторы “+” и “-”, но теперь для реализации объединения и разности множеств.

```
// Объединение множеств.
Set Set::operator +(Set ob2) {
    Set newset = *this; // Копируем первое множество.

    // Добавляем в первое множество уникальные элементы
    // из второго.
    for(int i=0; i < ob2.len; i++)
        newset = newset + ob2.members[i];

    return newset; // Возвращаем модифицированное множество.
}

// Разность множеств.
Set Set::operator -(Set ob2) {
    Set newset = *this; // Копируем первое множество.

    // "Вычитаем" элементы из второго множества.
    for(int i=0; i < ob2.len; i++)
        newset = newset - ob2.members[i];

    return newset; // Возвращаем модифицированное множество.
}
```

Как видите, эти операторные функции используют ранее определенные версии операторов “+” и “-”. При выполнении операции объединения сначала создается новое множество, которое содержит элементы первого множества. Затем в него добавляются элементы второго множества. Поскольку оператор “+” добавит элемент только в том случае, если он еще не является частью множества, то в результате получим объединение (без элементов-дубликатов) двух множеств. При получении разности множеств с помощью

## 456 Модуль 9. О классах подробнее

оператора “-” из первого множества удаляются совпадающие элементы.

9. Ниже приводится полный код программы, включающей определение класса Set и функции main(), которая демонстрирует его использование.

```
/*
Проект 9.1.

Класс множества для символов.

*/
#include <iostream>
using namespace std;

const int MaxSize = 100;

class Set {
    int len; // количество членов
    char members[MaxSize]; // На этом массиве и будет
                           // построено множество.

    /* Функция find() закрыта, поскольку она не используется
       вне класса Set. */
    int find(char ch); // выполняет поиск элемента

public:

    // Построение пустого множества.
    Set() { len = 0; }

    // Получение количества элементов в множестве.
    int getLength() { return len; }

    void showset(); // отображение множества
    bool isMember(char ch); // проверка членства

    Set operator +(char ch); // добавление элемента
    Set operator -(char ch); // удаление элемента

    Set operator +(Set ob2); // объединение множеств
```

```

Set operator -(Set ob2); // получение разности множеств
};

/* /* Функция возвращает индекс элемента, заданного
   параметром ch, или -1, если таковой не найден. */
int Set::find(char ch) {
    int i;

    for(i=0; i < len; i++)
        if(members[i] == ch) return i;

    return -1;
}

// Отображение содержимого множества.
void Set::showset() {
    cout << "{ ";
    for(int i=0; i<len; i++)
        cout << members[i] << " ";

    cout << "}\n";
}

/* Функция возвращает значение true, если символ ch
является членом множества.

В противном случае возвращается значение false. */
bool Set::isMember(char ch) {
    if(find(ch) != -1) return true;
    return false;
}

// Добавление нового элемента в множество.
Set Set::operator +(char ch) {
    Set newset;

    if(len == MaxSize) {
        cout << "Множество заполнено.\n";
        return *this; // возвращает существующее множество
    }
}

```

## 458 Модуль 9. О классах подробнее

```
newset = *this; // дублирует существующее множество

// Проверяем, существует ли уже в множестве такой
// элемент.
if(find(ch) == -1) { // Если элемент не найден, его
                      // можно добавлять в множество.
    // Включаем новый элемент в новое множество.
    newset.members[newset.len] = ch;
    newset.len++;
}
return newset; // возвращает модифицированное множество
}

// Удаляем элемент из множества.
Set Set::operator -(char ch) {
    Set newset;
    int i = find(ch); // Переменная i получит значение -1,
                      // если заданный элемент не будет
                      // найден в множестве.

    // Копируем и уплотняем оставшиеся элементы.
    for(int j=0; j < len; j++)
        if(j != i) newset = newset + members[j];

    return newset;
}

// Объединение множеств.
Set Set::operator +(Set ob2) {
    Set newset = *this; // Копируем первое множество.

    // Добавляем в первое множество уникальные элементы
    // из второго.
    for(int i=0; i < ob2.len; i++)
        newset = newset + ob2.members[i];

    return newset; // Возвращаем модифицированное множество.
}
```

```

// Разность множеств.
Set Set::operator -(Set ob2) {
    Set newset = *this; // Копируем первое множество.

    // "Вычитаем" элементы из второго множества.
    for(int i=0; i < ob2.len; i++)
        newset = newset - ob2.members[i];

    return newset; // Возвращаем модифицированное множество.
}

// Демонстрация использования класса Set.
int main() {
    // Создаем пустое множество.
    Set s1;
    Set s2;
    Set s3;

    s1 = s1 + 'A';
    s1 = s1 + 'B';
    s1 = s1 + 'C';

    cout << "Множество s1 после добавления
            символов А В С: ";
    s1.showset();

    cout << "\n";

    cout << "Тестирование членства с помощью функции
            isMember().\n";
    if(s1.isMember('B'))
        cout << "В - член множества s1.\n";
    else
        cout << "В - не член множества s1.\n";

    if(s1.isMember('T'))
        cout << "Т - член множества s1.\n";
    else

```

## 460 Модуль 9. О классах подробнее

```
cout << "T - не член множества s1.\n";  
  
cout << "\n";  
  
s1 = s1 - 'B';  
cout << "Множество s1 после s1 = s1 - 'B': ";  
s1.showset();  
  
s1 = s1 - 'A';  
cout << "Множество s1 после s1 = s1 - 'A': ";  
s1.showset();  
  
s1 = s1 - 'C';  
cout << "Множество s1 после s1 = s1 - 'C': ";  
s1.showset();  
  
cout << "\n";  
  
s1 = s1 + 'A';  
s1 = s1 + 'B';  
s1 = s1 + 'C';  
cout << "Множество s1 после добавления А В С: ";  
s1.showset();  
  
cout << "\n";  
  
s2 = s2 + 'A';  
s2 = s2 + 'X';  
s2 = s2 + 'W';  
  
cout << "Множество s2 после добавления А Х В: ";  
s2.showset();  
  
cout << "\n";  
  
s3 = s1 + s2;  
cout << "Множество s3 после s3 = s1 + s2: ";  
s3.showset();
```

```

s3 = s3 - s1;
cout << "Множество s3 после s3 = s3 - s1: ";
s3.showset();

cout << "\n";

cout << "Множество s2 после s2 = s2 - s2: ";
s2 = s2 - s2; // очищаем s2
s2.showset();

cout << "\n";

s2 = s2 + 'C'; // Помещаем символы ABC в обратном
// порядке.
s2 = s2 + 'B';
s2 = s2 + 'A';

cout << "Множество s2 после добавления С В А: ";
s2.showset();

return 0;
}

```

Результаты выполнения этой программы таковы.

Множество s1 после добавления символов А В С: { А В С }

Тестирование членства с помощью функции isMember().

В - член множества s1.

Т - не член множества s1.

Множество s1 после s1 = s1 - 'B': { А С }

Множество s1 после s1 = s1 - 'A': { С }

Множество s1 после s1 = s1 - 'C': { }

Множество s1 после добавления А В С: { А В С }

Множество s2 после добавления А Х В: { А Х В }

Множество s3 после s3 = s1 + s2: { А В С Х В }

Множество s3 после s3 = s3 - s1: { Х В }

Множество `s2` после `s2 = s2 - s2`: { }

Множество `s2` после добавления С В А: { С В А }



## Тест для самоконтроля по модулю 9

- Что представляет собой конструктор копии и в каких случаях он вызывается? Представьте общий формат конструктора копии.
- Поясните, что происходит при возвращении объекта функцией? В частности, в каких случаях вызывается его деструктор?
- Дано следующее определение класса.

```
class T {  
    int i, j;  
public:  
    int sum() {  
        return i + j;  
    }  
};
```

Покажите, как следует переписать функцию `sum()`, чтобы она использовала указатель `this`.

- Что такое структура? Что такое объединение?
- На что ссылается `*this` в коде функции-члена?
- Что представляет собой `friend`-функция?
- Представьте общий формат перегруженной бинарной операторной функции-члена.
- Что необходимо сделать, чтобы разрешить выполнение операций над объектом класса и значением встроенного типа?
- Можно ли перегрузить оператор "?"? Можно ли менять приоритет оператора?
- Для класса `Set`, определенного в проекте 9.1, определите операторы "<" и ">", чтобы с их помощью можно было узнать, является ли одно множество подмножеством или супермножеством другого. Пусть оператор "<" возвращает значение `true`, если левое множество является подмножеством правого, и значение `false` в противном случае. Пусть оператор ">" возвращает значение `true`, если левое множество является супермножеством правого, и значение `false` в противном случае.

11. Для класса `Set` определите оператор “`&`”, чтобы с его помощью можно было получить пересечение двух множеств.
12. Самостоятельно попытайтесь добавить в класс `Set` другие операторы. Например, попробуйте определить оператор “`!`” для получения *строгой дизъюнкции* двух множеств. Результат применения строгой дизъюнкции должен включать элементы, которых нет ни в одном из рассматриваемых множеств.



Ответы на эти вопросы можно найти на Web-странице данной книги по адресу: <http://www.osborne.com>.

# Модуль 10

## Наследование

- 10.1.** Понятие о наследовании
- 10.2.** Управление доступом к членам базового класса
- 10.3.** Использование защищенных членов
- 10.4.** Вызов конструкторов базового класса
- 10.5.** Создание многоуровневой иерархии
- 10.6.** Наследование нескольких базовых классов
- 10.7.** Когда выполняются функции конструкторов и деструкторов
- 10.8.** Указатели на производные типы
- 10.9.** Виртуальные функции и полиморфизм
- 10.10.** Чисто виртуальные функции и абстрактные классы

В этом модуле рассматриваются три средства C++, которые непосредственно связаны с объектно-ориентированным программированием: наследование, виртуальные функции и полиморфизм. Наследование – это свойство, которое позволяет одному классу наследовать характеристики другого. Используя наследование, можно создать общий класс, который определяет черты, присущие множеству связанных элементов. Этот класс затем может быть унаследован другими, узкоспециализированными классами с добавлением в каждый из них своих, уникальных особенностей. На основе наследования построено понятие виртуальной функции, причем виртуальная функция является одним из факторов поддержки полиморфизма, который реализует принцип объектно-ориентированного программирования “один интерфейс – множество методов”.

**ВАЖНО!****10.1. Понятие о наследовании**

В стандартной терминологии языка C++ класс, который наследуется, называется *базовым*. Класс, который наследует базовый класс, называется *производным*. Следовательно, производный класс представляет собой специализированную версию базового класса. Производный класс наследует все члены, определенные в базовом, и добавляет к ним собственные уникальные элементы.

Язык C++ поддерживает механизм наследования, который предусматривает возможность в объявление одного класса встраивать другой класс. Для этого достаточно при объявлении производного класса указать базовый. Лучше всего это пояснить на примере. В следующей программе создается базовый класс TwoDShape, в котором хранятся основные параметры (ширина и высота) двумерного объекта, и производный класс Triangle, создаваемый на основе класса TwoDShape. Обратите особое внимание на то, как объявляется класс Triangle.

```
// Простая иерархия классов.
```

```
#include <iostream>
#include <cstring>
using namespace std;

// Класс определения двумерных объектов.
class TwoDShape {
public:
    double width;
    double height;
```

```

void showDim() {
    cout << "Ширина и высота равны " <<
        width << " и " << height << "\n";
}

};

// Класс Triangle выводится из класса TwoDShape.
class Triangle : public TwoDShape { // Обратите внимание на
                                    // синтаксис объявления.
public:
    char style[20];

    double area() {
        return width * height / 2; // Класс Triangle может
                                    // обращаться к членам класса
                                    // TwoDShape так, как если бы они
                                    // были частью класса Triangle.
    }

    void showStyle() {
        cout << "Этот треугольник " << style << "\n";
    }
};

int main() {
    Triangle t1;
    Triangle t2;

    t1.width = 4.0; // Все члены класса Triangle доступны
    t1.height = 4.0; // объектам класса Triangle, даже те,
                     // которые унаследованы от класса TwoD-
Shape.
    strcpy(t1.style, "равнобедренный");

    t2.width = 8.0;
    t2.height = 12.0;
    strcpy(t2.style, "прямоугольный");
}

```

## 468 Модуль 10. Наследование

```
cout << "Информация о треугольнике t1:\n";
t1.showStyle();
t1.showDim();
cout << "Площадь равна " << t1.area() << "\n";

cout << "\n";
cout << "Информация о треугольнике t2:\n";
t2.showStyle();
t2.showDim();
cout << "Площадь равна " << t2.area() << "\n";

return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

Информация о треугольнике t1:

Этот треугольник равнобедренный

Ширина и высота равны 4 и 4

Площадь равна 8

Информация о треугольнике t2:

Этот треугольник прямоугольный

Ширина и высота равны 8 и 12

Площадь равна 48

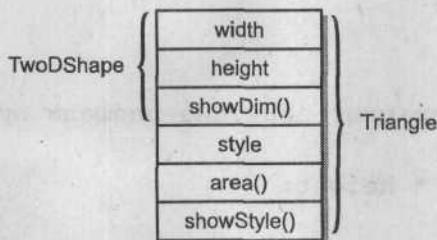
Здесь класс TwoDShape определяет атрибуты “обобщенной” двумерной геометрической фигуры (например, квадрата, прямоугольника, треугольника и т.д.). В классе Triangle создается специфический тип объекта класса TwoDShape, в данном случае треугольник. Класс Triangle содержит все элементы класса TwoDShape и, кроме того, поле style, функцию area() и функцию showStyle(). В переменной style хранится описание типа треугольника, функция area() вычисляет и возвращает его площадь, а функция showStyle() отображает данные о типе треугольника.

Ниже приведен синтаксис, который используется в объявлении класса Triangle, чтобы сделать его производным от класса TwoDShape.

```
class Triangle : public TwoDShape {
```

Этот синтаксис довольно прост для запоминания и использования. Если один класс наследует другой, то имя базового класса (в данном случае это класс TwoDShape) указывается после имени производного (в данном случае это класс Triangle), причем имена классов разделяются двоеточием.

Поскольку класс `Triangle` включает все члены базового класса, `TwoDShape`, он может обращаться к членам `width` и `height` внутри метода `area()`. Кроме того, внутри метода `main()` объекты `t1` и `t2` могут непосредственно ссылаться на члены `width` и `height`, как если бы они были частью класса `Triangle`. Включение класса `TwoDShape` в класс `Triangle` схематически показано на рис. 10.1.



*Рис. 10.1. Схематическое представление класса `Triangle`*

Несмотря на то что класс `TwoDShape` является базовым для класса `Triangle`, он совершенно независим и автономен. То, что его в качестве базового использует производный класс, не означает невозможность применения его самого.

Общий формат объявления класса, который наследует базовый класс, имеет такой вид:

```

class имя_производного_класса : доступ имя_базового_класса {
    // тело производного класса
}
  
```

Здесь элемент `доступ` необязателен. При необходимости его можно выразить одним из спецификаторов доступа: `public`, `private` или `protected` (подробнее о них вы узнаете ниже в этом модуле). Пока (для простоты изложения) при объявлении производного класса будем использовать спецификатор `public`. Если базовый класс наследуется как `public`-класс, все его `public`-члены становятся `public`-членами производного класса.

Основное достоинство наследования состоит в том, что, создав базовый класс, который определяет общие атрибуты для множества объектов, его можно использовать для создания любого числа более специализированных производных классов. В каждом производном классе можно затем точно “настроить” собственную классификацию. Вот, например, как из базового класса `TwoDShape` можно вывести производный класс, который инкапсулирует прямоугольники.

```
// Класс прямоугольников Rectangle, производный
// от класса TwoDShape.
class Rectangle : public TwoDShape {
public:
    // Функция возвращает значение true, если
    // прямоугольник является квадратом.
    bool isSquare() {
        if(width == height) return true;
        return false;
    }

    // Функция возвращает значение площади прямоугольника.
    double area() {
        return width * height;
    }
}
```

Класс Rectangle включает класс TwoDShape и добавляет к нему функцию isSquare(), которая определяет, является ли прямоугольник квадратом, и функцию area(), вычисляющую площадь прямоугольника.

## Доступ к членам класса и наследование

Как разъяснялось в модуле 8, члены класса часто объявляются закрытыми, чтобы предотвратить их несанкционированное использование и внесение изменений. Наследование класса *не* отменяет ограничения, связанные с закрытым доступом. Таким образом, несмотря на то, что производный класс включает все члены базового класса, он не может получить доступ к тем из них, которые объявлены закрытыми. Например, как показано в следующем коде, если переменные width и height являются private-членами в классе TwoDShape, то класс Triangle не сможет получить к ним доступ.

```
// Доступ к private-членам не передается по наследству
// производным классам.
```

```
class TwoDShape {
    // Теперь это private-члены.
    double width;
    double height;
public:
    void showDim() {
```

```

cout << "Ширина и высота равны " <<
      width << " и " << height << "\n";
}

};

// Класс Triangle выводится из класса TwoDShape.
class Triangle : public TwoDShape {
public:
    char style[20];

    double area() {
        return width * height / 2; // Ошибка: нельзя получить
                                   // прямой доступ к закрытым
                                   // членам базового класса.
    }

    void showStyle() {
        cout << "Этот треугольник " << style << "\n";
    }
};

```

Класс `Triangle` не скомпилируется, поскольку ссылка на члены `width` и `height` внутри функции `area()` приведет к нарушению прав доступа. Поскольку `width` и `height` — закрытые члены, они доступны только для членов их собственного класса. На производные классы эта доступность не распространяется.

На первый взгляд может показаться, что невозможность доступа к закрытым членам базового класса со стороны производного — серьезное ограничение. Однако это не так, поскольку в C++ предусмотрены возможности решения этой проблемы. Одна из них — `protected`-члены, о которых пойдет речь в следующем разделе. Вторая возможность — использование открытых свойств и функций, позволяющих получить доступ к закрытым данным. Как было показано в предыдущих модулях, C++-программисты обычно предоставляют доступ к закрытым членам класса посредством открытых функций. Функции, которые обеспечивают доступ к закрытым данным, называются *функциями доступа* (иногда *аксессорными функциями*). Перед вами — новая версия класса `TwoDShape`, в которую как раз и добавлены функции доступа к членам `width` и `height`.

```
// Доступ к закрытым данным с помощью функций доступа.
```

```
#include <iostream>
```

## 472 Модуль 10. Наследование

```
#include <cstring>
using namespace std;

// Класс двумерных объектов.
class TwoDShape {
    // Закрытые члены класса.
    double width;
    double height;
public:
    void showDim() {
        cout << "Ширина и высота равны " <<
            width << " и " << height << "\n";
    }

    // Функции доступа к членам width и height.
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
};

// Класс Triangle выводится из класса TwoDShape.
class Triangle : public TwoDShape {
public:
    char style[20];

    double area() { // Для получения значений ширины и высоты
                    // здесь используются функции доступа.
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        cout << "Этот треугольник " << style << "\n";
    }
};

int main() {
    Triangle t1;
```

```

Triangle t2;

t1.setWidth(4.0);
t1.setHeight(4.0);
strcpy(t1.style, "равнобедренный");

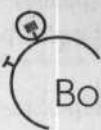
t2.setWidth(8.0);
t2.setHeight(12.0);
strcpy(t2.style, "прямоугольный");

cout << "Информация о треугольнике t1:\n";
t1.showStyle();
t1.showDim();
cout << "Площадь равна " << t1.area() << "\n";

cout << "\n";
cout << "Информация о треугольнике t2:\n";
t2.showStyle();
t2.showDim();
cout << "Площадь равна " << t2.area() << "\n";

return 0;
}

```



### Вопросы для текущего контроля

1. Как в программе указать, что базовый класс наследуется производным классом?
2. Включает ли производный класс члены своего базового класса?
3. Имеет ли производный класс доступ к закрытым членам базового класса?\*

- 
1. Имя базового класса указывается после имени производного. Имена этих классов должны разделяться двоеточием.
  2. Да, производный класс включает члены своего базового класса.
  3. Нет, производный класс не имеет доступа к закрытым членам базового класса.

### Спросим у опытного программиста

**Вопрос.** Я слышал, что программисты на языке Java используют термины "суперкласс" и "подкласс". Как эти термины соотносятся с C++-терминологией?

**Ответ.** То, что Java-программист называет суперклассом, C++-программист называет базовым классом. Надо сказать, что оба набора терминов применяются во многих языках. Однако в этой книге мы будем придерживаться стандартной терминологии C++. (Замечу, что в C# применительно к классам также используют термины "базовый" и "производный").

ВАЖНО!

### 10.2. Управление доступом к членам базового класса

Как разъяснялось выше, если один класс наследует другой, члены базового класса становятся членами производного. Однако статус доступа членов базового класса в производном классе определяется спецификатором доступа, используемым для наследования базового класса. Спецификатор доступа базового класса выражается одним из ключевых слов: `public`, `private` или `protected`. Если спецификатор доступа не указан, то по умолчанию используется спецификатор `private`, если речь идет о наследовании типа `class`. Если же наследуется тип `struct`, то при отсутствии явно заданного спецификатора доступа по умолчанию используется спецификатор `public`. Рассмотрим развикиацию (разветвление) использования спецификаторов `public` или `private`. (Спецификатор `protected` описан в следующем разделе.)

Если базовый класс наследуется как `public`-класс, все его `public`-члены становятся `public`-членами производного класса. Во всех случаях `private`-члены базового класса остаются закрытыми в рамках этого класса и не доступны для членов производного. Например, в следующей программе `public`-члены класса `B` становятся `public`-членами класса `D`. Следовательно, они будут доступны и для других частей программы.

```
// Демонстрация public-наследования.
```

```
#include <iostream>
using namespace std;
```

```
class B {
    int i, j;
```

```

public:
    void set(int a, int b) { i = a; j = b; }
    void show() { cout << i << " " << j << "\n"; }
};

class D : public B { // ← Здесь класс B наследуется
                    //      открытым способом.
    int k;
public:
    D(int x) { k = x; }
    void showk() { cout << k << "\n"; }

    // i = 10; // Ошибка! Член i закрыт в рамках класса B и
              // доступ к нему не разрешен.
};

int main()
{
    D ob(3);

    ob.set(1, 2); // Получаем доступ к членам базового класса.
    ob.show();     // Получаем доступ к членам базового класса.

    ob.showk();   // Здесь используется член производного
                  // класса.

    return 0;
}

```

Поскольку функции `set()` и `show()` являются открытыми (`public`) членами класса `B`, их можно вызывать для объекта типа `D` в функции `main()`. Но поскольку переменные `i` и `j` определены как `private`-члены, они остаются закрытыми в рамках своего класса `B`. Поэтому следующая строка кода превращена в комментарий.

```
// i = 10; // Ошибка! Член i закрыт в рамках класса B и
          // доступ к нему не разрешен.
```

Класс `D` не может получить доступ к закрытому члену класса `B`.

Противоположностью открытому (`public`) наследованию является закрытое (`private`). Если базовый класс наследуется как `private`-класс, все его

## 476 Модуль 10. Наследование

public-члены становятся private-членами производного класса. Например, следующая программа не скомпилируется, поскольку обе функции set() и show() теперь стали private-членами класса derived, и поэтому их нельзя вызывать из функции main().

```
// Использование закрытого способа наследования.  
// (Эта программа не скомпилируется.)
```

```
#include <iostream>  
using namespace std;  
  
class B {  
    int i, j;  
public:  
    void set(int a, int b) { i = a; j = b; }  
    void show() { cout << i << " " << j << "\n"; }  
};  
  
// Открытые элементы класса B становятся закрытыми в классе D.  
class D : private B { // ← Теперь класс B наследуется  
                      // закрытым способом.  
    int k;  
public:  
    D(int x) { k = x; }  
    void showk() { cout << k << "\n"; }  
};  
  
int main()  
{  
    D ob(3);  
  
    // Теперь функции set() и show() не доступны для  
    // частей программы вне класса D.  
    ob.set(1, 2); // Ошибка, доступ к функции set() невозможен.  
    ob.show();    // Ошибка, доступ к функции show() невозможен.  
  
    return 0;  
}
```

Итак, если базовый класс наследуется как private-класс, его открытые члены становятся закрытыми (private) членами производного класса. Это означа-

ет, что они доступны для членов производного класса, но не доступны для других частей программы.

**ВАЖНО!**

### 10.3. Использование защищенных членов

Как упоминалось выше, закрытый член базового класса недоступен для производного класса. Казалось бы, это означает, что, если производному классу нужен доступ к члену базового класса, его следует сделать открытым. При этом придется смириться с тем, что открытый член будет доступным для любого другого кода, что иногда нежелательно. К счастью, таких ситуаций можно избежать, поскольку C++ позволяет создавать *зашитенные члены*. Защищенным является член, который открыт для своей иерархии классов, но закрыт вне этой иерархии.

Зашитенный член создается с помощью модификатора доступа `protected`. При объявлении `protected`-члена он, по сути, является закрытым, но с одним исключением. Это исключение вступает в силу, когда защищенный член наследуется. В этом случае защищенный член базового класса становится защищенным членом производного класса, а следовательно, и доступным для производного класса. Таким образом, используя модификатор доступа `protected`, можно создавать закрытые (для “внешнего мира”) члены класса, но вместе с тем они будут наследоваться с возможностью доступа со стороны производных классов. Спецификатор `protected` можно также использовать со структурами.

Рассмотрим простой пример использования защищенных членов класса.

```
// Демонстрация защищенных членов класса.

#include <iostream>
using namespace std;

class B {
protected: // Здесь переменные i и j - защищенные члены.
    int i, j; // Хотя эти члены закрыты в рамках класса B,
              // тем не менее они доступны для класса D.
public:
    void set(int a, int b) { i = a; j = b; }
    void show() { cout << i << " " << j << "\n"; }
};
```

## 478 Модуль 10. Наследование

```
class D : public B {
    int k;
public:
    // Класс D может обращаться к членам i и j класса B,
    // поскольку они не private-, а protected-члены.
    void setk() { k = i*j; }

    void showk() { cout << k << "\n"; }
};

int main()
{
    D ob;

    ob.set(2, 3); // OK, функция set() открыта в классе B.
    ob.show();     // OK, функция show() открыта в классе B.

    ob.setk();
    ob.showk();

    return 0;
}
```

Поскольку класс B унаследован классом D открытым способом (т.е. как `public`-класс), и поскольку члены `i` и `j` объявлены защищенными в классе B, функция `setk()` (член класса D) имеет полное право на доступ к ним. Если бы члены `i` и `j` были объявлены в классе B закрытыми, то класс D не мог бы обращаться к ним, и эта программа не скомпилировалась бы.

Если базовый класс наследуется как `public`-класс, защищенные (`protected`) члены базового класса становятся защищенными членами производного класса, т.е. доступными для производного класса. Если же базовый класс наследуется закрытым способом (т.е. с использованием спецификатора `private`), защищенные члены этого базового класса становятся закрытыми (`private`) членами производного класса.

Спецификатор защищенного доступа может стоять в любом месте объявления класса, но, как правило, `protected`-члены объявляются после (объявляемых по умолчанию) `private`-членов и перед `public`-членами. Таким образом, самый общий формат объявления класса обычно выглядит так.

```
class имя_класса {
    // private-члены по умолчанию
```

```

protected:
    // protected-члены
public:
    // public-члены
};

```

Напомню, что раздел защищенных членов необязателен.

Ключевое слово `protected` можно использовать не только для придания членам класса статуса “зашитенности”, но и в качестве спецификатора доступа при наследовании базового класса. Если базовый класс наследуется как защищенный, все его открытые и защищенные члены становятся защищенными членами производного класса. Например, если класс D наследует класс B таким способом

```
class D : protected B {,
```

то все незакрытые члены класса B станут защищенными членами класса D.



## Вопросы для текущего контроля

- Если базовый класс наследуется закрытым способом, то его открытые члены становятся закрытыми членами производного класса. Верно ли это?
- Может ли закрытый член базового класса стать открытым благодаря механизму наследования?
- Какой спецификатор доступа нужно использовать, чтобы сделать член класса доступным в рамках иерархии классов, но закрытым для остального “мира”?\*

- 
- Верно: если базовый класс наследуется закрытым способом, его открытые члены становятся закрытыми членами производного класса.
  - Нет, закрытый член всегда остается закрытым в рамках своего класса.
  - Чтобы сделать член класса доступным в рамках иерархии классов, но закрытым для остального “мира”, нужно использовать спецификатор `protected`.

## Спросим у опытного программиста

**Вопрос.** Хотелось бы обобщить сведения об использовании спецификаторов `public`, `protected` и `private`.

**Ответ.** При объявлении члена класса открытым (с использованием ключевого слова `public`) к нему можно получить доступ из любой другой части программы. Если член класса объявляется закрытым (с помощью спецификатора `private`), к нему могут получать доступ только члены того же класса. Более того, к закрытым членам базового класса не имеют доступа даже производные классы. Если же член класса объявляется защищенным (`protected`-членом), к нему могут получать доступ только члены того же или производных классов. Таким образом, спецификатор `protected` позволяет наследовать члены, но оставляет их закрытыми в рамках иерархии классов.

Если базовый класс наследуется с использованием ключевого слова `public`, его `public`-члены становятся `public`-членами производного класса, а его `protected`-члены – `protected`-членами производного класса. Если базовый класс наследуется с использованием спецификатора `protected`, его `public`- и `protected`-члены становятся `protected`-членами производного класса. Если же базовый класс наследуется с использованием ключевого слова `private`, его `public`- и `protected`-члены становятся `private`-членами производного класса. Во всех случаях `private`-члены базового класса остаются закрытыми в рамках этого класса.

## Конструкторы и наследование

В иерархии классов как базовые, так и производные классы могут иметь собственные конструкторы. При этом возникает важный вопрос: какой конструктор отвечает за создание объекта производного класса? Конструктор базового, производного класса, или оба одновременно? Ответ таков: конструктор базового класса создает часть объекта, соответствующую базовому классу, а конструктор производного класса – часть объекта, соответствующую производному классу. И это вполне логично, потому что базовый класс “не видит” элементы производного класса или не имеет доступа к ним. Поэтому их “конструкции” должны быть раздельными. В предыдущих примерах классы опирались на конструкторы по умолчанию, создаваемые автоматически средствами C++, и поэтому мы не сталкивались с подобной проблемой. Но на практике большинство классов имеет конструкторы, и вы должны знать, как выходить из этой ситуации.

Если конструктор определяется только в производном классе, процесс создания объекта несложен: просто создается объект производного класса. Часть объекта, соответствующая базовому классу, создается автоматически с помощью конструктора по умолчанию. Например, рассмотрим переработанную версию класса `Triangle`, в которой явно определяется конструктор. Здесь член `style` объявлен `private`-членом, поскольку теперь он устанавливается конструктором.

```
// Добавление конструктора в класс Triangle.

#include <iostream>
#include <cstring>
using namespace std;

// Класс двумерных объектов.
class TwoDShape {
    // Теперь эти члены закрыты.
    double width;
    double height;
public:
    void showDim() {
        cout << "Ширина и высота равны " <<
            width << " и " << height << "\n";
    }

    // Функции доступа к закрытым членам класса.
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
};

// Класс Triangle - производный от класса TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // Теперь это private-член.
public:

    // Конструктор класса Triangle.
    Triangle(char *str, double w, double h) {
        // Инициализируем "базовую" часть класса, т.е. часть,
```

## 482 Модуль 10. Наследование

```
// определяемую в Triangle-объекте классом TwoDShape.
setWidth(w);
setHeight(h);

// Инициализируем "производную" часть класса, т.е. часть,
// определяемую в Triangle-объекте конкретным типом
// треугольника.
strcpy(style, str);
}

double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    cout << "Этот треугольник " << style << "\n";
}

};

int main() {
    Triangle t1("равнобедренный", 4.0, 4.0);
    Triangle t2("прямоугольный", 8.0, 12.0);

    cout << "Информация о треугольнике t1:\n";
    t1.showStyle();
    t1.showDim();
    cout << "Площадь равна " << t1.area() << "\n";

    cout << "\n";
    cout << "Информация о треугольнике t2:\n";
    t2.showStyle();
    t2.showDim();
    cout << "Площадь равна " << t2.area() << "\n";

    return 0;
}
```

Здесь конструктор класса Triangle инициализирует наследуемые им члены класса TwoDShape, а также собственное поле style.

Если конструкторы определены и в базовом, и в производном классе, процесс создания объектов несколько усложняется, поскольку в этом случае должны выполниться конструкторы обоих классов.

**ВАЖНО!**

## 10.4. ВЫЗОВ КОНСТРУКТОРОВ БАЗОВОГО КЛАССА

Если базовый класс имеет конструктор, то производный класс должен явно вызвать его для инициализации той части "своего" объекта, которая соответствует базовому классу. Производный класс может вызывать конструктор, определенный в его базовом классе, используя расширенную форму объявления конструктора производного класса. Формат такого расширенного объявления имеет следующий вид.

```
конструктор_производного_класса(список_аргументов) :
    конструктор_базового_класса(список_аргументов) {
    // тело конструктора производного класса
}
```

Здесь элемент *конструктор\_базового\_класса* представляет собой имя базового класса, наследуемого производным классом. Обратите внимание на использование двоеточия, которое разделяет объявление конструктора производного класса от конструктора базового. (Если класс наследует несколько базовых классов, то имена их конструкторов необходимо перечислить через запятую в виде списка.) С помощью элемента *список\_аргументов* задаются аргументы, необходимые конструктору в соответствующем классе.

В следующей программе показано, как передаются аргументы конструктору базового класса. Здесь определяется конструктор класса *TwoDShape*, который инициализирует члены данных *width* и *height*.

```
// Добавление конструктора в класс TwoDShape.
```

```
#include <iostream>
#include <cstring>
using namespace std;

// Класс двумерных объектов.
class TwoDShape {
    // Закрытые члены класса.
    double width;
```

## 484 Модуль 10. Наследование

```
double height;
public:

// Конструктор класса TwoDShape.
TwoDShape(double w, double h) {
    width = w;
    height = h;
}

void showDim() {
    cout << "Ширина и высота равны " <<
        width << " и " << height << "\n";
}

// Функции доступа.
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
};

// Класс Triangle выведен из класса TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // Это private-член.
public:

// Конструктор класса Triangle.
Triangle(char *str, double w,
         double h) : TwoDShape(w, h) { // Здесь вызывается
                                // конструктор класса TwoDShape.
    strcpy(style, str);
}

double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    cout << "Этот треугольник " << style << "\n";
```

```

}

};

int main() {
    Triangle t1("равнобедренный", 4.0, 4.0);
    Triangle t2("прямоугольный", 8.0, 12.0);

    cout << "Информация о треугольнике t1:\n";
    t1.showStyle();
    t1.showDim();
    cout << "Площадь равна " << t1.area() << "\n";

    cout << "\n";
    cout << "Информация о треугольнике t2:\n";
    t2.showStyle();
    t2.showDim();
    cout << "Площадь равна " << t2.area() << "\n";

    return 0;
}

```

Здесь конструктор `Triangle()` вызывает конструктор `TwoDShape()` с параметрами `w` и `h`, который инициализирует свойства `width` и `height` соответствующими значениями. Класс `Triangle` больше не инициализирует эти значения сам. Ему остается установить только одно значение, уникальное для класса треугольников, а именно член `style` (тип треугольника). Такой подход дает классу `TwoDShape` свободу выбора среди возможных способов построения подобъектов. Более того, со временем мы можем расширить функции класса `TwoDShape`, но об этом расширении ранее созданные производные классы не будут “знать”, что спасет существующий код от разрушения.

Конструктор производного класса может вызвать конструктор любого формата, определенный в базовом классе. Реально же выполнится тот конструктор, параметры которого будут соответствовать переданным при вызове аргументам. Например, вот как выглядят расширенные версии классов `TwoDShape` и `Triangle`, которые включают дополнительные конструкторы.

```
// Класс TwoDShape с дополнительным конструктором.
```

```
#include <iostream>
#include <cstring>
```

```
using namespace std;

// Класс двумерных объектов.
class TwoDShape {
    // Это private-члены.
    double width;
    double height;
public:

    // Конструктор класса TwoDShape по умолчанию.
    TwoDShape() {
        width = height = 0.0;
    }

    // Конструктор класса TwoDShape с параметрами.
    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }

    // Конструктор объекта, у которого ширина равна высоте.
    TwoDShape(double x) {
        width = height = x;
    }

    void showDim() {
        cout << "Ширина и высота равны " <<
            width << " и " << height << "\n";
    }

    // Функции доступа к закрытым членам класса.
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
};

// Класс треугольников, производный от класса TwoDShape.
class Triangle : public TwoDShape {
```

```

char style[20]; // Теперь это private-член.
public:

/* Конструктор по умолчанию. Он автоматически вызывает
конструктор по умолчанию класса TwoDShape. */
Triangle() {
    strcpy(style, "неизвестный");
}

// Конструктор с тремя параметрами.
Triangle(char *str, double w,
         double h) : TwoDShape(w, h) {
    strcpy(style, str);
}

// Конструктор равнобедренного треугольника.
Triangle(double x) : TwoDShape(x) {
    strcpy(style, "равнобедренный");
}

double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    cout << "Этот треугольник " << style << "\n";
}
};

int main() {
    Triangle t1;
    Triangle t2("прямоугольный", 8.0, 12.0);
    Triangle t3(4.0);

    t1 = t2;

    cout << "Информация о треугольнике t1: \n";
    t1.showStyle();
    t1.showDim();
}

```

## 488 Модуль 10. Наследование

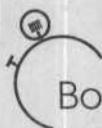
```
cout << "Площадь равна " << t1.area() << "\n";  
  
cout << "\n";  
  
cout << "Информация о треугольнике t2: \n";  
t2.showStyle();  
t2.showDim();  
cout << "Площадь равна " << t2.area() << "\n";  
  
cout << "\n";  
  
cout << "Информация о треугольнике t3: \n";  
t3.showStyle();  
t3.showDim();  
cout << "Площадь равна " << t3.area() << "\n";  
  
cout << "\n";  
  
return 0;  
}
```

Вот как выглядят результаты выполнения этой версии программы.

Информация о треугольнике t1:  
Этот треугольник прямоугольный  
Ширина и высота равны 8 и 12  
Площадь равна 48

Информация о треугольнике t2:  
Этот треугольник прямоугольный  
Ширина и высота равны 8 и 12  
Площадь равна 48

Информация о треугольнике t3:  
Этот треугольник равнобедренный  
Ширина и высота равны 4 и 4  
Площадь равна 8



## Вопросы для текущего контроля

- Каким образом производный класс выполняет конструктор базового класса?
- Можно ли передать параметры конструктору базового класса?
- Какой конструктор отвечает за инициализацию “базовой” части объекта производного класса: определенный производным или базовым классом?\*

### Проект 10.1. Расширение возможностей класса `Vehicle`

`TruckDemo.cpp`

Цель этого проекта — создать подкласс класса `Vehicle`, к разработке которого мы приступили в модуле 8. Напомню, что класс `Vehicle` инкапсулирует основные данные о транспортных средствах (возможное количество перевозимых пассажиров, общая вместимость топливных резервуаров и расход топлива). Теперь мы можем использовать класс `Vehicle` в качестве отправной точки для разработки более специализированных классов. Например, возьмем такой тип транспортного средства, как грузовик. Одной из важнейших его технических характеристик является грузовместимость. Итак, для создания класса `Truck` используем в качестве базового класс `Vehicle` и добавим к его содержимому переменную для хранения грузовместимости. Таким образом, в этом проекте мы создадим класс `Truck` на основе класса `Vehicle`, сделав переменные базового класса закрытыми, но позаботившись о доступе к ним посредством аксессорных функций.

#### Последовательность действий

- Создайте файл `TruckDemo.cpp` и скопируйте в него последнюю реализацию класса `Vehicle` из модуля 8.
- При определении конструктора производного класса указывается имя конструктора базового класса.
- Да, конструктору базового класса можно передать параметры.
- За инициализацию “базовой” части объекта производного класса отвечает конструктор, определенный базовым классом.

## 2. Создайте класс Truck.

```
// Используем класс Vehicle для создания
// специализированного класса Truck.
class Truck : public Vehicle {
    int cargocap; // грузовместимость в фунтах
public:

    // Конструктор класса Truck.
    Truck(int p, int f, int m, int c) : Vehicle(p, f, m)
    {
        cargocap = c;
    }

    // Функция доступа к члену данных cargocap.
    int get_cargocap() { return cargocap; }
};
```

Здесь класс `Truck` наследует класс `Vehicle` с добавлением члена `cargocap`. Таким образом, класс `Truck` включает все общие атрибуты транспортных средств, определенные в классе `Vehicle`, и нам остается лишь добавить элементы, которые уникальны для данного класса.

3. Приведем полный текст программы демонстрации класса `Truck`.

```
// Создание подкласса Truck, выведенного
// из класса Vehicle.

#include <iostream>
using namespace std;

// Объявление класса Vehicle.
class Vehicle {
    // Это private-члены.
    int passengers; // количество пассажиров
    int fuelcap; // вместимость топливных резервуаров
                  // в литрах
    int mpg;      // расход горючего в километрах на литр
public:
    // Конструктор класса Vehicle.
    Vehicle(int p, int f, int m) {
        passengers = p;
```

```

fuelcap = f;
mpg = m;
}

// Функция вычисления максимального пробега.
int range() { return mpg * fuelcap; }

// Функции доступа к членам класса.
int get_passengers() { return passengers; }
int get_fuelcap() { return fuelcap; }
int get_mpg() { return mpg; }
};

// Использование класса Vehicle
// специализированного класса Truck.
class Truck : public Vehicle {
    int cargocap; // грузовместимость в фунтах
public:

    // Конструктор класса Truck.
    Truck(int p, int f, int m, int c) : Vehicle(p, f, m)
    {
        cargocap = c;
    }

    // Функция доступа к члену данных cargocap.
    int get_cargocap() { return cargocap; }
};

int main() {

    // Создаем несколько "грузовых" объектов.
    Truck semi(2, 200, 7, 44000);
    Truck pickup(3, 28, 15, 2000);
    int dist = 252;

    cout << "Полупикап может перевезти "
        << semi.get_cargocap() << " фунтов груза.\n";
    cout << "После заправки она может проехать максимум "
}

```

## 492 Модуль 10. Наследование

```
<<
    semi.range() << " километров.\n";
cout << "Чтобы проехать " << dist
<< " километра, полуторке необходимо " <<
    dist / semi.get_mpg() <<
    " литров топлива.\n\n";

cout << "Пикап может перевезти " << pickup.
get_cargocap() <<
    " фунтов груза.\n";
cout << "После заправки он может проехать максимум "
<<
    pickup.range() << " километров.\n";
cout << "Чтобы проехать " << dist
<< " километра, пикапу необходимо " <<
    dist / pickup.get_mpg() <<
    " литров топлива.\n";

return 0;
}
```

### 4. Результаты выполнения этой программы таковы.

Полуторка может перевезти 44000 фунтов груза.

После заправки она может проехать максимум 1400 километров.

Чтобы проехать 252 километра, полуторке необходимо 36 литров топлива.

Пикап может перевезти 2000 фунтов груза.

После заправки он может проехать максимум 420 километров.

Чтобы проехать 252 километра, пикапу необходимо 16 литров топлива.

### 5. Из класса Vehicle можно вывести множество других типов классов. Например, при использовании следующей схемы можно создать класс внедорожных транспортных средств, в котором предусмотрено хранение такого атрибута, как высота дорожного просвета.

```
// Создание класса внедорожных транспортных средств.
class OffRoad : public Vehicle {
```

```

int groundClearance; // высота дорожного просвета
                     // в дюймах
public:
    ...
};

```

Главное — понимать, что, создав базовый класс, определяющий общие параметры объекта, можно сформировать множество специализированных классов путем наследования базового. В каждый производный класс достаточно добавить уникальные атрибуты. В этом и состоит суть наследования.

**ВАЖНО!**

## **10.5. Создание многоуровневой иерархии**

До сих пор мы использовали простые иерархии, состоящие только из базового и производного классов. Но можно построить иерархии, которые содержат любое количество уровней наследования. Как упоминалось выше, один производный класс вполне допустимо использовать в качестве базового для другого. Например, из трех классов (A, B и C) C может быть производным от B, который, в свою очередь, может быть производным от A. В подобной ситуации каждый производный класс наследует содержимое всех своих базовых классов. В данном случае класс C наследует все члены классов B и A.

Чтобы понять, какую пользу можно получить от многоуровневой иерархии, рассмотрим следующую программу. В ней производный (от `TwoDShape`) класс `Triangle` используется в качестве базового для создания еще одного производного класса с именем `ColorTriangle`. Класс `ColorTriangle` наследует все члены классов `Triangle` и `TwoDShape` и добавляет собственное поле `color`, которое содержит цвет треугольника.

```

// Многоуровневая иерархия.

#include <iostream>
#include <cstring>
using namespace std;

// Класс двумерных объектов.
class TwoDShape {
    // Это private-члены.

```

## 494 Модуль 10. Наследование

```
double width;
double height;
public:

// Конструктор по умолчанию.
TwoDShape() {
    width = height = 0.0;
}

// Конструктор класса TwoDShape с параметрами.
TwoDShape(double w, double h) {
    width = w;
    height = h;
}

// Конструктор объекта, у которого ширина равна высоте.
TwoDShape(double x) {
    width = height = x;
}

void showDim() {
    cout << "Ширина и высота равны " <<
        width << " и " << height << "\n";
}

// Функции доступа к членам класса.
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
};

// Класс Triangle - производный от класса TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // private-член
public:

/* Конструктор по умолчанию. Он автоматически вызывает
конструктор по умолчанию класса TwoDShape. */
Triangle() {
```

```

    strcpy(style, "неизвестный");
}

// Конструктор с тремя параметрами.
Triangle(char *str, double w,
         double h) : TwoDShape(w, h) {
    strcpy(style, str);
}

// Конструктор равнобедренных треугольников.
Triangle(double x) : TwoDShape(x) {
    strcpy(style, "равнобедренный");
}

double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    cout << "Этот треугольник " << style << "\n";
};

// Класс цветных треугольников.
class ColorTriangle : public Triangle { // Класс ColorTriangle
    // наследует класс Triangle, который в свою
    // очередь наследует класс TwoDShape.
    char color[20];
public:
    ColorTriangle(char *clr, char *style, double w,
                  double h) : Triangle(style, w, h) {
        strcpy(color, clr);
    }

    // Функция отображения цвета.
    void showColor() {
        cout << "Цвет " << color << "\n";
    }
};

```

## 496 Модуль 10. Наследование

```
int main() {
    ColorTriangle t1("синий", "прямоугольный", 8.0, 12.0);
    ColorTriangle t2("красный", "равнобедренный", 2.0, 2.0);

    cout << "Информация о треугольнике t1:\n";
    t1.showStyle();
    t1.showDim();
    t1.showColor();
    cout << "Площадь равна " << t1.area() << "\n";

    cout << "\n";

    cout << "Информация о треугольнике t2:\n";
    t2.showStyle(); // Объект класса ColorTriangle может
    t2.showDim(); // вызывать функции, определенные
    // в нем самом,
    t2.showColor(); // а также функции, определенные в базовых
    // классах.
    cout << "Площадь равна " << t2.area() << "\n";

    return 0;
}
```

Эта программа генерирует такие результаты.

Информация о треугольнике t1:  
Этот треугольник прямоугольный  
Ширина и высота равны 8 и 12  
Цвет синий  
Площадь равна 48

Информация о треугольнике t2:  
Этот треугольник равнобедренный  
Ширина и высота равны 2 и 2  
Цвет красный  
Площадь равна 2

Благодаря наследованию класс ColorTriangle может использовать ранее определенные классы Triangle и TwoDShape, добавляя только ту информацию, которая необходима для собственного (специального) применения. В этом и состоит ценность наследования: оно позволяет многократно использовать однажды созданный код.

Этот пример иллюстрирует еще один важный момент: если конструктору базового класса требуются параметры, все производные классы должны передавать эти параметры "по иерархии", независимо от того, нужны ли эти параметры самому производному классу.

**ВАЖНО!**

## 10.6 Наследование нескольких базовых классов

В C++ производный класс одновременно может наследовать два или больше базовых классов. Например, в этой короткой программе класс D наследует оба класса B1 и B2.

```
// Пример использования нескольких базовых классов.
```

```
#include <iostream>
using namespace std;

class B1 {
protected:
    int x;
public:
    void showx() { cout << x << "\n"; }
};

class B2 {
protected:
    int y;
public:
    void showy() { cout << y << "\n"; }
};

// Наследование двух базовых классов.
class D: public B1, public B2 { // Класс D наследует
                                // одновременно и класс B1 и класс B2.
public:
    /* Члены x и y доступны, поскольку они не закрыты,
       а защищены в классах B1 и B2. */
    void set(int i, int j) { x = i; y = j; }
};
```

```
int main()
{
    D ob;

    ob.set(10, 20); // член класса D
    ob.showx();      // член класса B1
    ob.showy();      // член класса B2

    return 0;
}
```

Как видно из этого примера, чтобы обеспечить наследование нескольких базовых классов, необходимо через запятую перечислить их имена в виде списка. При этом нужно указать спецификатор доступа для каждого наследуемого базового класса.

**ВАЖНО!**

## **10.7. Когда выполняются функции конструкторов и деструкторов**

Базовый или производный класс (или оба одновременно) могут содержать конструктор и/или деструктор. Важно понимать порядок, в котором выполняются эти функции. При использовании механизма наследования обычно возникает два важных вопроса, связанных с конструкторами и деструкторами. Первый: в каком порядке вызываются конструкторы базового и производного классов? Второй: в каком порядке вызываются деструкторы базового и производного классов? Чтобы ответить на эти вопросы, рассмотрим простую программу.

```
#include <iostream>
using namespace std;

class B {
public:
    B() { cout << "Построение базовой части объекта.\n"; }
    ~B() { cout << "Разрушение базовой части объекта.\n"; }
};

class D: public B {
public:
    D() { cout << "Построение производной части объекта.\n"; }
```

```

~D() { cout << "Разрушение производной части объекта.\n";
}

int main()
{
    D ob;

    // Никаких действий, кроме создания и
    // разрушения объекта ob.

    return 0;
}

```

Как отмечено в комментариях функции `main()`, эта программа лишь создает и тут же разрушает объект `ob`, который имеет тип `D`. При выполнении программа отображает такие результаты.

Построение базовой части объекта.

Построение производной части объекта.

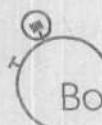
Разрушение производной части объекта.

Разрушение базовой части объекта.

Судя по результатам, сначала выполняется конструктор класса `B`, а за ним — конструктор класса `D`. Затем (по причине немедленного разрушения объекта `ob` в этой программе) вызывается деструктор класса `D`, а за ним — деструктор класса `B`.

Результаты вышеописанного эксперимента можно обобщить следующим образом. При создании объекта производного класса сначала вызывается конструктор базового класса, а за ним — конструктор производного класса. При разрушении объекта производного класса сначала вызывается его “родной” деструктор, а за ним — деструктор базового класса. Другими словами, конструкторы вызываются в порядке происхождения классов, а деструкторы — в обратном порядке.

При многоуровневой иерархии классов (т.е. в ситуации, когда производный класс становится базовым классом для еще одного производного) применяется то же общее правило: конструкторы вызываются в порядке происхождения классов, а деструкторы — в обратном порядке. Если некоторый класс наследует несколько базовых классов одновременно, вызов конструкторов, заданных в списке наследования этого производного класса, происходит в направлении слева направо, а вызов деструкторов — справа налево.



### Вопросы для текущего контроля

1. Можно ли производный класс использовать в качестве базового для другого производного класса?
2. В каком порядке (с точки зрения иерархии классов) вызываются конструкторы при создании объекта?
3. В каком порядке (с точки зрения иерархии классов) вызываются деструкторы при разрушении объекта?\*

### Спросим у опытного программиста

**Вопрос.** *Почему конструкторы вызываются в порядке происхождения классов, а деструкторы — в обратном порядке?*

**Ответ.** Вполне логично, что функции конструкторов выполняются в порядке происхождения их классов. Поскольку базовый класс “ничего не знает” ни о каком производном классе, операции по инициализации, которые ему нужно выполнить, не зависят от операций инициализации, выполняемых производным классом, но, возможно, создают предварительные условия для последующей работы. Поэтому конструктор базового класса должен выполняться первым.

Аналогичная логика присутствует и в том, что деструкторы выполняются в порядке, обратном порядку происхождения классов. Поскольку базовый класс лежит в основе производного класса, разрушение базового класса подразумевает разрушение производного. Следовательно, деструктор производного класса имеет смысл вызвать до того, как объект будет полностью разрушен.

1. Да, производный класс можно использовать в качестве базового для другого производного класса.
2. Конструкторы вызываются в порядке происхождения классов.
3. Деструкторы вызываются в порядке, обратном порядку происхождения классов.

ВАЖНО!

10

## 10.8. Указатели на производные типы

Прежде чем переходить к виртуальным функциям и понятию полиморфизма, необходимо рассмотреть один важный аспект использования указателей. Указатели на базовые и производные классы связаны такими отношениями, которые не свойственны указателям других типов. Как правило, указатель одного типа не может указывать на объект другого типа. Однако указатели на базовые классы и объекты производных классов — исключения из этого правила. В C++ указатель на базовый класс также можно использовать для ссылки на объект любого класса, выведенного из базового. Например, предположим, что у нас есть базовый класс `B` и класс `D`, который выведен из класса `B`. Любой указатель, объявленный как указатель на класс `B`, можно использовать также для ссылки на объект типа `D`. Следовательно, после этих объявлений

```
B *p; // указатель на объект типа B
B B_obj; // объект типа B
D D_obj; // объект типа D
```

обе следующие инструкции абсолютно законны:

```
p = &B_obj; // p указывает на объект типа B
p = &D_obj; /* p указывает на объект типа D,
который является объектом,
выведенным из класса B. */
```

Любой указатель на базовый класс (в данном примере это указатель `p`) можно использовать для доступа только к тем частям объекта производного класса, которые были унаследованы от базового класса. Так, в этом примере указатель `p` можно использовать для доступа ко всем элементам объекта `D_obj`, выведенным из объекта `B_obj`. Однако к элементам, которые составляют специфическую "надстройку" (над базой, т.е. над базовым классом `B`) объекта `D_obj`, доступ с помощью указателя `p` получить нельзя.

Кроме того, необходимо понимать, что хотя "базовый" указатель можно использовать для доступа к объектам любого производного типа, обратное утверждение неверно. Другими словами, используя указатель на производный класс, нельзя получить доступ к объекту базового типа.

Как вы уже знаете, указатель инкрементируется и декрементируется относительно своего базового типа. Следовательно, если указатель на базовый класс используется для доступа к объекту производного типа, инкрементирование или декрементирование не заставит его ссылаться на следующий объект производного класса. Вместо этого он будет указывать ("по его мнению") на

следующий объект базового класса. Таким образом, инкрементирование или декрементирование указателя на базовый класс следует расценивать как некорректную операцию, если этот указатель используется для ссылки на объект производного класса.

Тот факт, что указатель на базовый тип можно использовать для ссылки на любой объект, выведенный из базового, чрезвычайно важен и принципиален для C++. Как будет показано ниже, эта гибкость является ключевым моментом для способа реализации динамического полиморфизма в C++.

## Ссылки на производные типы

Подобно указателям, ссылку на базовый класс также можно использовать для доступа к объекту производного типа. Эта возможность особенно часто применяется при передаче аргументов функциям. Параметр, который имеет тип ссылки на базовый класс, может принимать объекты базового класса, а также объекты любого другого типа, выведенного из него.

**ВАЖНО!**

## 10.9. Виртуальные функции и полиморфизм

Фундамент, на котором в C++ строится поддержка полиморфизма, состоит из механизма наследования и указателей на базовый класс. Конкретным средством, которое в действительности позволяет реализовать полиморфизм, является виртуальная функция. (Оставшаяся часть этого модуля как раз и посвящена рассмотрению этого важного средства.)

### Понятие о виртуальной функции

*Виртуальная функция* — это функция, которая объявляется в базовом классе с использованием ключевого слова `virtual` и переопределяется в одном или нескольких производных классах. Таким образом, каждый производный класс может иметь собственную версию виртуальной функции. Интересно рассмотреть ситуацию, когда виртуальная функция вызывается через указатель (или ссылку) на базовый класс. В этом случае C++ определяет, какую именно версию виртуальной функции необходимо вызвать, по *типу* объекта, адресуемого этим указателем. Причем следует иметь в виду, что это решение принимается во время выполнения программы. Следовательно, при указании на различные объекты будут вызываться и различные версии виртуальной функции. Другими словами,

именно по *типу адресуемого объекта* (а не по типу самого указателя) определяется, какая версия виртуальной функции будет выполнена. Таким образом, если базовый класс содержит виртуальную функцию и если из этого базового класса выведено два (или больше) других класса, то при адресации различных типов объектов через указатель на базовый класс будут выполняться и различные версии этой виртуальной функции. Аналогичный механизм работает и при использовании ссылки на базовый класс.

Функция объявляется виртуальной в базовом классе с помощью ключевого слова `virtual`. При переопределении виртуальной функции в производном классе ключевое слово `virtual` повторять не нужно (хотя это не будет ошибкой).

Класс, который включает виртуальную функцию, называется *полиморфным* классом. Этот термин также применяется к классу, который наследует базовый класс, содержащий виртуальную функцию.

Рассмотрим следующую короткую программу, в которой демонстрируется использование виртуальной функции.

```
// Пример использования виртуальной функции.

#include <iostream>
using namespace std;

class B {
public:
    virtual void who() { // ← Объявление виртуальной функции.
        cout << "Базовый класс.\n";
    }
};

class D1 : public B {
public:
    void who() { // Переопределение функции who() для
                  // класса D1.
        cout << "Первый производный класс.\n";
    }
};

class D2 : public B {
public:
    void who() { // Еще одно переопределение функции who()
                  // для класса D2.
    }
};
```

## 504 Модуль 10. Наследование

```
cout << "Второй производный класс.\n";
}

};

int main()
{
    B base_obj;
    B *p;
    D1 D1_obj;
    D2 D2_obj;

    p = &base_obj;
    // Вызываем виртуальную функцию who() через указатель на
    // базовый класс.
    p->who(); // доступ к функции who() класса B

    p = &D1_obj;
    p->who(); // доступ к функции who() класса D1

    p = &D2_obj;
    p->who(); // доступ к функции who() класса D2

    return 0;
}
```

Эта программа генерирует такие результаты.

Базовый класс.

Первый производный класс.

Второй производный класс.

Теперь рассмотрим код этой программы подробно, чтобы понять, как она работает.

В классе B функция who() объявлена виртуальной. Это означает, что ее можно переопределить в производном классе (в классе, выведенном из B). И она действительно переопределяется в обоих производных классах D1 и D2. В функции main() объявляются четыре переменные: base\_obj (объект типа B), p (указатель на объект класса B), а также два объекта D1\_obj и D2\_obj двух производных классов D1 и D2 соответственно. Затем указателю p присваивается адрес объекта base\_obj, и вызывается функция who(). Поскольку функция who() объявлена виртуальной, C++ во время выполнения программы опреде-

ляет, к какой именно версии функции `who()` здесь нужно обратиться, причем решение принимается путем анализа типа объекта, адресуемого указателем `p`. В данном случае `p` указывает на объект типа `B`, поэтому сначала выполняется та версия функции `who()`, которая объявлена в классе `B`. Затем указателю `p` присваивается адрес объекта `D1_obj`. Вспомните, что с помощью указателя на базовый класс можно обращаться к объекту любого его производного класса. Поэтому, когда функция `who()` вызывается во второй раз, C++ снова выясняет тип объекта, адресуемого указателем `p`, и, исходя из этого типа, определяет, какую версию функции `who()` нужно вызвать. Поскольку `p` здесь указывает на объект типа `D1`, то выполняется версия функции `who()`, определенная в классе `D1`. Аналогично после присвоения `p` адреса объекта `D2_obj` вызывается версия функции `who()`, объявленная в классе `D2`.

Итак, запомните: то, какая версия виртуальной функции действительно будет вызвана, определяется во время выполнения программы. Решение основывается исключительно на типе объекта, адресуемого указателем (используемым при вызове рассматриваемой функции) на базовый класс.

Виртуальную функцию можно вызывать обычным способом (не через указатель), используя оператор “точка” и задавая имя вызывающего объекта. Это означает, что в предыдущем примере было бы синтаксически корректно обратиться к функции `who()` с помощью следующей инструкции:

```
D1_obj.who();
```

Однако при вызове виртуальной функции таким способом игнорируются ее полиморфные атрибуты. И только при обращении к виртуальной функции через указатель на базовый класс достигается динамический полиморфизм.

Поначалу может показаться, что переопределение виртуальной функции в производном классе представляет собой специальную форму перегрузки функций. Но это не так. В действительности мы имеем дело с двумя принципиально разными процессами. Прежде всего, версии перегруженной функции должны отличаться друг от друга типом и/или количеством параметров, в то время как тип и количество параметров у версий переопределенной виртуальной функции должны в точности совпадать. И в самом деле, прототипы виртуальной функции и ее переопределений должны быть абсолютно одинаковыми. Если прототипы будут различными, то такая функция будет попросту считаться перегруженной, и ее “виртуальная сущность” утратится. Кроме того, виртуальная функция должна быть членом класса, для которого она определяется, а не его “другом”. Но в то же время виртуальная функция может быть “другом” иностранного класса. И еще: функциям деструкторов разрешается быть виртуальными, а функциям конструкторов — нет.

## Наследование виртуальных функций

Если функция объявляется как виртуальная, она остается такой независимо от того, через сколько уровней производных классов она может пройти. Например, если бы класс D2 был выведен из класса D1, а не из класса B, как показано в следующем примере, то функция who() по-прежнему оставалась бы виртуальной, и механизм выбора соответствующей версии по-прежнему работал бы корректно.

```
// Этот класс выведен из класса D1, а не из B.  
class D2 : public D1 {  
public:  
    void who() { // Переопределение функции who() для класса D2.  
        cout << "Второй производный класс.\n";  
    }  
};
```

Если производный класс не переопределяет виртуальную функцию, то используется функция, определенная в базовом классе. Например, проверим, как поведет себя версия предыдущей программы, если в классе D2 не будет переопределена функция who().

```
#include <iostream>  
using namespace std;  
  
class B {  
public:  
    virtual void who() {  
        cout << "Базовый класс.\n";  
    }  
};  
  
class D1 : public B {  
public:  
    void who() {  
        cout << "Первый производный класс.\n";  
    }  
};  
  
class D2 : public B { // Класс D2 не переопределяет  
// функцию who().  
// Функция who() не определена.  
};
```

```

int main()
{
    B base_obj;
    B *p;
    D1 D1_obj;
    D2 D2_obj;

    p = &base_obj;
    p->who(); // Обращаемся к функции who() класса B.

    p = &D1_obj;
    p->who(); // Обращаемся к функции who() класса D1.

    p = &D2_obj;
    p->who(); /* Обращаемся к функции who() класса B, поскольку
    в классе D2 она не переопределена. */
}

return 0;
}

```

При выполнении эта программа генерирует такие результаты.

Базовый класс.

Первый производный класс.

Базовый класс.

Как подтверждают результаты выполнения этой программы, поскольку функция `who()` не переопределена классом `D2`, то при ее вызове с помощью инструкции `p->who()` (когда член `p` указывает на объект `D2_obj`) выполняется та версия функции `who()`, которая определена в классе `B`.

Следует иметь в виду, что наследуемые свойства спецификатора `virtual` являются иерархическими. Поэтому, если предыдущий пример изменить так, чтобы класс `D2` был выведен из класса `D1`, а не из класса `B`, то при обращении к функции `who()` через объект типа `D2` будет вызвана та ее версия, которая объявлена в классе `D1`, поскольку этот класс является "ближайшим" (по иерархическим "меркам") к классу `D2`, а не функция `who()` из класса `B`.

## Зачем нужны виртуальные функции

Как отмечалось выше, виртуальные функции в сочетании с производными типами позволяют C++ поддерживать динамический полиморфизм. Полимор-

физм существен для объектно-ориентированного программирования по одной важной причине: он обеспечивает возможность некоторому обобщенному классу определять функции, которые будут использовать все производные от него классы, причем производный класс может определить собственную реализацию некоторых или всех этих функций. Иногда эта идея выражается следующим образом: базовый класс диктует общий *интерфейс*, который будет иметь любой объект, выведенный из этого класса, но позволяет при этом производному классу определить *метод*, используемый для реализации этого интерфейса. Вот почему для описания полиморфизма часто используется фраза “один интерфейс, множество методов”.

Для успешного применения полиморфизма необходимо понимать, что базовый и производный классы образуют иерархию, развитие которой направлено от большей степени обобщения к меньшей (т.е. от базового класса к производному). При корректной разработке базовый класс обеспечивает все элементы, которые производный класс может использовать напрямую. Он также определяет функции, которые производный класс должен реализовать самостоятельно. Это дает производному классу гибкость в определении собственных методов, но в то же время обязывает использовать общий интерфейс. Другими словами, поскольку формат интерфейса определяется базовым классом, любой производный класс должен разделять этот общий интерфейс. Таким образом, использование виртуальных функций позволяет базовому классу определять обобщенный интерфейс, который будет использован всеми производными классами.

Теперь у вас может возникнуть вопрос: почему же так важен общий интерфейс со множеством реализаций? Ответ снова возвращает нас к основной побудительной причине возникновения объектно-ориентированного программирования: такой интерфейс позволяет программисту справляться со все возрастающей сложностью программ. Например, если корректно разработать программу, то можно быть уверенным в том, что ко всем объектам, выведенным из базового класса, можно будет получить доступ единым (общим для всех) способом, несмотря на то, что конкретные действия у одного производного класса могут отличаться от действий у другого. Это означает, что программисту придется иметь дело только с одним интерфейсом, а не с великим их множеством. Кроме того, производный класс волен использовать любые или все функции, предоставленные базовым классом. Другими словами, разработчику производного класса не нужно заново изобретать элементы, уже имеющиеся в базовом классе.

Отделение интерфейса от реализации позволяет создавать *библиотеки классов*, написанием которых могут заниматься сторонние организации. Корректно реализованные библиотеки должны предоставлять общий интерфейс, который программист может использовать для выведения классов в соответствии со своими конкретными

потребностями. Например, как библиотека базовых классов Microsoft (Microsoft Foundation Classes – MFC), так и более новая библиотека классов .NET Framework Windows Forms поддерживают разработку приложений для Microsoft Windows. Использование этих классов позволяет писать программы, которые могут унаследовать множество функций, нужных любой Windows-программе. Вам понадобится лишь добавить в нее средства, уникальные для вашего приложения. Это – большое подспорье при программировании сложных систем.

## Применение виртуальных функций

Чтобы лучше почувствовать силу виртуальных функций, применим их к классу TwoDShape. В предыдущих примерах каждый класс, выведенный из класса TwoDShape, определяет функцию с именем area(). Это наводит нас на мысль о том, не лучше ли сделать функцию вычисления площади фигуры area() виртуальной в классе TwoDShape. Это даст нам возможность переопределить ее в производных классах таким образом, чтобы она вычисляла площадь согласно типу конкретной геометрической фигуры, которую инкапсулирует данный класс. Эта мысль и реализована в следующей программе. Для удобства в класс TwoDShape вводится поле name, которое упрощает демонстрацию этих классов.

```
// Использование виртуальных функций и механизма полиморфизма.
```

```
#include <iostream>
#include <cstring>
using namespace std;

// Класс двумерных объектов.
class TwoDShape {
    // Это private-члены.
    double width;
    double height;

    // Добавляем поле name.
    char name[20];

public:

    // Конструктор по умолчанию (без параметров).
    TwoDShape() {
        width = height = 0.0;
        strcpy(name, "неизвестный");
    }
```

## 510 Модуль 10. Наследование

```
}

// Конструктор объектов с тремя параметрами.
TwoDShape(double w, double h, char *n) {
    width = w;
    height = h;
    strcpy(name, n);
}

// Конструктор объектов, у которых ширина равна высоте.
TwoDShape(double x, char *n) {
    width = height = x;
    strcpy(name, n);
}

void showDim() {
    cout << "Ширина и высота равны " <<
        width << " и " << height << "\n";
}

// Функции доступа к членам класса.
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
char *getName() { return name; }

// Добавляем в класс TwoDShape функцию area()
// и делаем ее виртуальной.
virtual double area() { // Теперь это виртуальная функция.
    cout << "\nОшибка: функцию area() нужно переопределить.\n";
    return 0.0;
}

};

// Класс Triangle - производный от класса TwoDShape
// (класс треугольников).
class Triangle : public TwoDShape {
```

```
char style[20]; // private-член
public:
    /* Конструктор по умолчанию. Он автоматически вызывает
     * действующий по умолчанию конструктор класса TwoDShape.
 */
Triangle() {
    strcpy(style, "неизвестный");
}

// Конструктор с тремя параметрами.
Triangle(char *str, double w,
         double h) : TwoDShape(w, h, "треугольник") {
    strcpy(style, str);
}

// Конструктор равнобедренных треугольников.
Triangle(double x) : TwoDShape(x, "треугольник") {
    strcpy(style, "равнобедренный");
}

// Переопределение функции area(), объявленной
// в классе TwoDShape, в классе Triangle.
double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    cout << "Этот треугольник " << style << "\n";
}
};

// Класс Rectangle - производный от класса TwoDShape
// (класс прямоугольников).
class Rectangle : public TwoDShape {
public:
    // Конструктор прямоугольника.
    Rectangle(double w, double h) :
```

## 512 Модуль 10. Наследование

```
TwoDShape(w, h, "прямоугольник") { }

// Конструктор квадратов.
Rectangle(double x) :
    TwoDShape(x, "прямоугольник") { }

bool isSquare() {
    if(getWidth() == getHeight()) return true;
    return false;
}

// Еще одно переопределение функции area(), объявленной
// в классе TwoDShape, в классе Rectangle.
double area() {
    return getWidth() * getHeight();
}
};

int main() {
    // Объявляем массив указателей на объекты типа TwoDShape.
    TwoDShape *shapes[5];

    shapes[0] = &Triangle("прямоугольный", 8.0, 12.0);
    shapes[1] = &Rectangle(10);
    shapes[2] = &Rectangle(10, 4);
    shapes[3] = &Triangle(7.0);
    shapes[4] = &TwoDShape(10, 20, "общий");

    for(int i=0; i < 5; i++) {
        cout << "Этот объект " <<
            shapes[i]->getName() << "\n";

        cout << "Площадь равна " <<
            shapes[i]->area() << "\n"; // Для каждого объекта
   // теперь вызывается
   // нужная версия
   // функции area().

        cout << "\n";
    }
}
```

```
    return 0;  
}
```

Результаты выполнения этой программы таковы.

Этот объект треугольник

Площадь равна 48

Этот объект прямоугольник

Площадь равна 100

Этот объект прямоугольник

Площадь равна 40

Этот объект треугольник

Площадь равна 24.5

Этот объект общий

Площадь равна

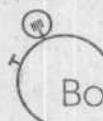
Ошибка: функцию area() нужно переопределить.

0

Рассмотрим эту программу подробнее. Во-первых, функция `area()` объявляется в классе `TwoDShape` с использованием ключевого слова `virtual` и переопределяется в классах `Triangle` и `Rectangle`. В классе `TwoDShape` функция `area()` представляет собой своего рода "заглушку", которая просто информирует пользователя о том, что в производном классе эту функцию необходимо переопределить. Каждое переопределение функции `area()` реализует вариант вычисления площади, соответствующий типу объекта, инкапсулируемому производным классом. Таким образом, если бы вы реализовали класс эллипсов, то функция `area()` в этом классе вычисляла бы площадь эллипса.

В предыдущей программе проиллюстрирован еще один важный момент. Обратите внимание на то, что в функции `main()` переменная `shapes` объявляется как массив указателей на объекты типа `TwoDShape`. Однако элементам этого массива присваиваются указатели на объекты классов `Triangle`, `Rectangle` и `TwoDShape`. Это вполне допустимо, поскольку указатель на базовый класс может ссылаться на объект производного класса. Затем программа в цикле опрашивает массив `shapes`, отображая информацию о каждом объекте. Несмотря на простоту, этот цикл иллюстрирует силу как наследования, так и виртуальных функций. Конкретный тип объекта, адресуемый указателем на базовый класс, определя-

ется во время выполнения программы, что позволяет принять соответствующие меры, т.е. выполнить действия, соответствующие объекту данного типа. Если объект (в конечном счете) выведен из класса `TwoDShape`, его площадь можно узнать посредством вызова функции `area()`. Интерфейс для выполнения этой операции одинаков для всех производных (от `TwoDShape`) классов, независимо от конкретного типа используемой фигуры.



### Вопросы для текущего контроля

1. Что представляет собой виртуальная функция?
2. Почему виртуальные функции столь важны для C++-программирования?
3. Какая версия переопределенной виртуальной функции выполняется при ее вызове через указатель на базовый класс?\*

ВАЖНО!

## 10.10 Чисто виртуальные функции и абстрактные классы

Иногда полезно создать базовый класс, определяющий только своего рода “пустой бланк”, который унаследуют все производные классы, причем каждый из них заполнит этот “бланк” собственной информацией. Такой класс определяет “суть” функций, которые производные классы должны реализовать, но сам при этом не обеспечивает реализации одной или нескольких функций. Подобная ситуация может возникнуть, когда базовый класс попросту не в состоянии реализовать функцию. Этот случай был проиллюстрирован версией класса `TwoDShape` (из предыдущей программы), в которой определение функции `area()` представляло собой “заглушку”, поскольку в ней площадь фигуры не вычислялась и, естественно, не отображалась.

В будущем, создавая собственные библиотеки классов, вы убедитесь, что отсутствие у функции четкого определения в контексте своего (базового) класса,

1. Виртуальная функция — это функция, которая объявляется в базовом классе с помощью ключевого слова `virtual` и переопределяется в производном классе.
2. Виртуальные функции представляют собой один из способов поддержки полиморфизма в C++.
3. Конкретная версия выполняемой виртуальной функции определяется типом объекта, используемого в момент ее вызова. Таким образом, определение нужной версии происходит во время выполнения программы.

не является чем-то необычным. Описанную ситуацию можно обработать двумя способами. Один из них, который продемонстрирован в предыдущем примере, состоит в выводе предупреждающего сообщения. И хотя такой подход может быть полезным в определенных обстоятельствах (например, при отладке программы), все же он не соответствует уровню профессионального программирования. Существует и другой способ. Наша цель — заставить производные классы переопределить функции, которые в базовом классе не имеют никакого смысла. Рассмотрим класс `Triangle`. Им нельзя пользоваться, если не определена функция `area()`. В этом случае нам не обойтись без “сильно действующего” средства, благодаря которому производный класс обязательно переопределит все необходимые функции. Этим средством в C++ является *чисто виртуальная функция*.

Чисто виртуальная функция — это функция, объявленная в базовом классе, но не имеющая в нем никакого определения. Поэтому любой производный класс должен определить собственную версию этой функции, ведь у него просто нет никакой возможности использовать версию из базового класса (по причине ее отсутствия). Чтобы объявить чисто виртуальную функцию, используйте следующий общий формат:

```
virtual тип имя_функции(список_параметров) = 0;
```

Здесь под элементом `тип` подразумевается тип значения, возвращаемого функцией, а под элементом `имя_функции` — ее имя. Обозначение `= 0` является признаком того, что функция здесь объявлена как *чисто виртуальная*.

Используя возможность создания чисто виртуальных функций, можно усовершенствовать класс `TwoDShape`. Например, в следующей версии определения этого класса (см. предыдущую программу) функция `area()` уже представлена как чисто виртуальная. Это означает, что все классы, выведенные из класса `TwoDShape`, обязательно переопределят функцию `area()`.

```
// Использование чисто виртуальной функции.
```

```
#include <iostream>
#include <cstring>
using namespace std;

// Класс двумерных объектов.
class TwoDShape {
    // Это private-члены.
    double width;
    double height;
```

## 516 Модуль 10. Наследование

```
char name[20];
public:

// Конструктор по умолчанию.
TwoDShape() {
    width = height = 0.0;
    strcpy(name, "неизвестный");
}

// Конструктор объектов с тремя параметрами.
TwoDShape(double w, double h, char *n) {
    width = w;
    height = h;
    strcpy(name, n);
}

// Конструктор объектов, у которых ширина равна высоте.
TwoDShape(double x, char *n) {
    width = height = x;
    strcpy(name, n);
}

void showDim() {
    cout << "Ширина и высота равны " <<
        width << " и " << height << "\n";
}

// Функции доступа к членам класса.
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
char *getName() { return name; }

// Функция area() теперь чисто виртуальная.
virtual double area() = 0;

};
```

```
// Класс треугольников Triangle (производный от
// класса TwoDShape).
class Triangle : public TwoDShape {
    char style[20]; // now private
public:

/* Конструктор по умолчанию. Он автоматически вызывает
конструктор класса TwoDShape, действующий по умолчанию.
*/
Triangle() {
    strcpy(style, "неизвестный");
}

// Конструктор треугольников с тремя параметрами.
Triangle(char *str, double w,
         double h) : TwoDShape(w, h, "треугольник") {
    strcpy(style, str);
}

// Конструктор равнобедренных треугольников.
Triangle(double x) : TwoDShape(x, "треугольник") {
    strcpy(style, "равнобедренный");
}

// Здесь переопределяется функция area(), объявленная
// в классе TwoDShape.
double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    cout << "Этот треугольник " << style << "\n";
};

// Класс прямоугольников Rectangle (производный от
// класса TwoDShape).
class Rectangle : public TwoDShape {
public:
```

## 518 Модуль 10. Наследование

```
// Конструктор прямоугольников.  
Rectangle(double w, double h) :  
    TwoDShape(w, h, "прямоугольник") {}  
  
// Конструктор квадратов.  
Rectangle(double x) :  
    TwoDShape(x, "прямоугольник") {}  
  
bool isSquare() {  
    if(getWidth() == getHeight()) return true;  
    return false;  
}  
  
// Это еще одно переопределение функции area().  
double area() {  
    return getWidth() * getHeight();  
}  
};  
  
int main() {  
    // Объявляем массив указателей на объекты типа TwoDShape.  
    TwoDShape *shapes[4];  
  
    shapes[0] = &Triangle("прямоугольный", 8.0, 12.0);  
    shapes[1] = &Rectangle(10);  
    shapes[2] = &Rectangle(10, 4);  
    shapes[3] = &Triangle(7.0);  
  
    for(int i=0; i < 4; i++) {  
        cout << "Этот объект " <<  
            shapes[i]->getName() << "\n";  
  
        cout << "Площадь равна " <<  
            shapes[i]->area() << "\n";  
  
        cout << "\n";  
    }  
}
```

```
return 0;
}
```

Если класс имеет хотя бы одну чисто виртуальную функцию, его называют *абстрактным*. Абстрактный класс характеризуется одной важной особенностью: у такого класса не может быть объектов. Чтобы убедиться в этом, попробуйте создать производный класс, который не переопределяет функцию `area()` (для этого достаточно удалить переопределение этой функции из класса `Triangle`). Вы тут же (т.е. во время компиляции) получите сообщение об ошибке. Абстрактный класс можно использовать только в качестве базового, из которого будут выводиться другие классы. Причина того, что абстрактный класс нельзя использовать для создания объектов, лежит, безусловно, в отсутствии определения для его одной или нескольких функций. Поэтому в функции `main()` предыдущей программы размер массива `shapes` сокращен до 4 элементов, и больше не создается "заготовка для фигуры" в виде "обобщенного" объекта класса `TwoDShape`. Но, как доказывает эта программа, даже если базовый класс является абстрактным, мы все равно можем объявлять указатели на его тип, которые затем будем использовать для указания на объекты производных классов.



## Тест для самоконтроля по модулю 10

- Класс, который наследуется, называется \_\_\_\_\_ классом. А класс, который является наследником (потомком), называется \_\_\_\_\_ классом.
- Имеет ли базовый класс доступ к членам производного? Имеет ли производный класс доступ к членам базового?
- Создайте класс `Circle`, производный от класса `TwoDShape`. Включите в него функцию `area()` для вычисления площади круга.
- Каким образом производному классу можно не разрешить доступ к членам базового класса?
- Представьте общий формат конструктора, который вызывает конструктор базового класса.
- Дана следующая иерархия классов.

```
class Alpha { ...  
class Beta : public Alpha { ...  
class Gamma: public Beta { ...
```

В каком порядке вызываются конструкторы этих классов при создании объекта класса `Gamma`?

- Как можно получить доступ к `protected`-членам?

8. С помощью указателя на базовый класс можно ссылаться на объект производного класса. Поясните, почему это важно для переопределения функций.
9. Что представляет собой чисто виртуальная функция? Что такое абстрактный класс?
10. Можно ли создать объект абстрактного класса?
11. Поясните, каким образом чисто виртуальная функция способствует реализации такого аспекта полиморфизма, как "один интерфейс, множество методов".



Ответы на эти вопросы можно найти на Web-странице данной книги по адресу: <http://www.osborne.com>.

# Модуль 11

## C++-система ввода-вывода

- 11.1.** Потоки C++
- 11.2.** Классы потоков
- 11.3.** Перегрузка операторов ввода-вывода
- 11.4.** Форматирование данных с использованием функций-членов класса ios
- 11.5.** Использование манипуляторов ввода-вывода
- 11.6.** Создание собственных манипуляторных функций
- 11.7.** Как открыть и закрыть файл
- 11.8.** Чтение и запись текстовых файлов
- 11.9.** Неформатированный ввод-вывод данных в двоичном режиме
- 11.10.** Использование других функций ввода-вывода
- 11.11.** Произвольный доступ
- 11.12.** Получение информации об операциях ввода-вывода

С самого начала книги мы использовали C++-систему ввода-вывода, но практически не давали никаких пояснений по этому поводу. Поскольку C++-система ввода-вывода построена на иерархии классов, ее теорию и детали невозможно освоить, не рассмотрев сначала классы и механизм наследования. Теперь настало время для подробного изучения C++-средств ввода-вывода.

Необходимо сразу отметить, что C++-система ввода-вывода — довольно обширная тема, и здесь описаны лишь самые важные и часто применяемые средства. В частности, вы узнаете, как перегрузить операторы “<<” и “>>” для ввода и вывода объектов создаваемых вами классов, а также как отформатировать вводимые или выводимые данные и использовать манипуляторы ввода-вывода. Завершает этот модуль рассмотрение средств файлового ввода-вывода.

## Сравнение старой и новой C++-систем ввода-вывода

В настоящее время существуют две версии библиотеки объектно-ориентированного ввода-вывода, причем обе широко используются программистами: более старая, основанная на оригинальных спецификациях языка C++, и новая, определенная стандартом языка C++. Старая библиотека ввода-вывода поддерживается за счет заголовочного файла `<iostream.h>`, а новая — посредством заголовка `<iostream>`. Новая библиотека ввода-вывода, по сути, представляет собой обновленную и усовершенствованную версию старой. Основное различие между ними состоит в реализации, а не в том, как их нужно использовать.

С точки зрения программиста, есть два существенных различия между старой и новой C++-библиотеками ввода-вывода. Во-первых, новая библиотека содержит ряд дополнительных средств и определяет несколько новых типов данных. Таким образом, новую библиотеку ввода-вывода можно считать супермножеством старой. Практически все программы, написанные для старой библиотеки, успешно компилируются при использовании новой, не требуя внесения каких-либо значительных изменений. Во-вторых, старая библиотека ввода-вывода была определена в глобальном пространстве имен, а новая использует пространство имен `std`. (Вспомните, что пространство имен `std` используется всеми библиотеками стандарта C++.) Поскольку старая библиотека ввода-вывода уже устарела, в этой книге описывается только новая, но большая часть информации применима и к старой.

ВАЖНО!

## 11.1 Потоки C++

Принципиальным для понимания C++-системы ввода-вывода является то, что она опирается на понятие потока. *Поток* (stream) — это абстракция, которая либо синтезирует информацию, либо потребляет ее и связывается с любым физическим устройством с помощью C++-системы ввода-вывода. Характер поведения всех потоков одинаков, несмотря на различные физические устройства, с которыми они связываются. Поскольку потоки действуют одинаково, то практически ко всем типам устройств можно применить одни и те же функции и операторы ввода-вывода. Например, методы, используемые для записи данных на экран, также можно использовать для вывода их на принтер или для записи в дисковый файл.

В самой общей форме поток можно назвать логическим интерфейсом с файлом. C++-определение термина “файл” можно отнести к дисковому файлу, экрану, клавиатуре, порту, файлу на магнитной ленте и пр. Хотя файлы отличаются по форме и возможностям, все потоки одинаковы. Достоинство этого подхода (с точки зрения программиста) состоит в том, что одно устройство компьютера может “выглядеть” подобно любому другому. Это значит, что поток обеспечивает интерфейс, согласующийся со всеми устройствами.

Поток связывается с файлом при выполнении операции открытия файла, а отсоединяется от него с помощью операции закрытия.

Существует два типа потоков: *текстовый* и *двоичный*. Текстовый поток используется для ввода-вывода символов. При этом могут происходить некоторые преобразования символов. Например, при выводе символ новой строки может быть преобразован в последовательность символов: возврата каретки и перехода на новую строку. Поэтому может не быть взаимно-однозначного соответствия между тем, что посыпается в поток, и тем, что в действительности записывается в файл. Двоичный поток можно использовать с данными любого типа, причем в этом случае никакого преобразования символов не выполняется, и между тем, что посыпается в поток, и тем, что потом реально содержится в файле, существует взаимно-однозначное соответствие.

Говоря о потоках, необходимо понимать, что вкладывается в понятие “текущей позиции”. *Текущая позиция* — это место в файле, с которого будет выполняться следующая операция доступа к файлу. Например, если длина файла равна 100 байт, и известно, что уже прочитана половина этого файла, то следующая операция чтения произойдет на байте 50, который в данном случае и является текущей позицией.

Итак, в языке C++ механизм ввода-вывода функционирует с использованием логического интерфейса, именуемого *потоком*. Все потоки имеют аналогичные свойства, которые позволяют выполнять одинаковые функции ввода-вывода, независимо от того, с файлом какого типа установлена связь. Под *файлом* понимается реальное физическое устройство, которое содержит данные. Если файлы различаются между собой, то потоки – нет. (Конечно, некоторые устройства могут не поддерживать все операции ввода-вывода, например операции с произвольной выборкой, поэтому и связанные с ними потоки тоже не будут поддерживать эти операции.)

## Встроенные C++-потоки

В C++ содержится ряд встроенных потоков (`cin`, `cout`, `cerr` и `clog`), которые автоматически открываются, как только программа начинает выполняться. Как вы знаете, `cin` – это стандартный входной, а `cout` – стандартный выходной поток. Потоки `cerr` и `clog` (они предназначены для вывода информации об ошибках) также связаны со стандартным выводом данных. Разница между ними состоит в том, что поток `clog` буферизован, а поток `cerr` – нет. Это означает, что любые выходные данные, посланные в поток `cerr`, будут немедленно выведены, а при использовании потока `clog` данные сначала записываются в буфер, и реальный их вывод происходит только тогда, когда буфер полностью заполняется. Обычно потоки `cerr` и `clog` используются для записи информации об отладке или ошибках.

В C++ также предусмотрены двухбайтовые (16-битовые) символьные версии стандартных потоков, именуемые `wcin`, `wcout`, `wcerr` и `wclog`. Они предназначены для поддержки таких языков, как китайский, для представления которых требуются большие символьные наборы. В этой книге двухбайтовые стандартные потоки не используются.

По умолчанию стандартные C++-потоки связываются с консолью, но программным способом их можно перенаправить на другие устройства или файлы. Перенаправление может также выполнить операционная система.

**ВАЖНО!**

### 11.2. Классы потоков

Как вы узнали в модуле 1, C++-система ввода-вывода использует заголовок `<iostream>`, в котором для поддержки операций ввода-вывода определена довольно сложная иерархия классов. Эта иерархия начинается с системы шаблонных классов. Как будет отмечено в модуле 12, шаблонный класс определяет фор-

му, не задавая в полном объеме данные, которые он должен обрабатывать. Имея шаблонный класс, можно создавать его конкретные экземпляры. Для библиотеки ввода-вывода стандарт C++ создает две специальные версии шаблонных классов: одну для 8-, а другую для 16-битовых ("широких") символов. Эти версии действуют подобно другим классам, и для использования C++-системы ввода-вывода никакого предварительного знакомства с шаблонами не требуется.

C++-система ввода-вывода построена на двух связанных, но различных иерархиях шаблонных классов. Первая выведена из класса низкоуровневого ввода-вывода `basic_streambuf`. Этот класс поддерживает базовые низкоуровневые операции ввода и вывода и обеспечивает поддержку для всей C++-системы ввода-вывода. Если вы не собираетесь заниматься программированием специализированных операций ввода-вывода, то вам вряд ли придется использовать напрямую класс `basic_streambuf`. Иерархия классов, с которой C++-программистам наверняка предстоит работать вплотную, выведена из класса `basic_ios`. Это — класс высокоуровневого ввода-вывода, который обеспечивает форматирование, контроль ошибок и предоставляет статусную информацию, связанную с потоками ввода-вывода. (Класс `basic_ios` выведен из класса `ios_base`, который определяет ряд свойств, используемых классом `basic_ios`.) Класс `basic_ios` используется в качестве базового для нескольких производных классов, включая классы `basic_istream`, `basic_ostream` и `basic_iostream`. Эти классы применяются для создания потоков, предназначенных для ввода данных, вывода и ввода-вывода соответственно.

Как упоминалось выше, библиотека ввода-вывода создает две специализированные иерархии шаблонных классов: одну для 8-, а другую для 16-битовых символов. В этой книге описываются классы только для 8-битовых символов, поскольку они используются гораздо чаще. Ниже приводится список имен шаблонных классов и соответствующих им "символьных" версий.

#### Шаблонные классы

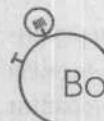
|                              |
|------------------------------|
| <code>basic_streambuf</code> |
| <code>basic_ios</code>       |
| <code>basic_istream</code>   |
| <code>basic_ostream</code>   |
| <code>basic_iostream</code>  |
| <code>basic_fstream</code>   |
| <code>basic_ifstream</code>  |
| <code>basic_ofstream</code>  |

#### Символьные классы

|                        |
|------------------------|
| <code>streambuf</code> |
| <code>ios</code>       |
| <code>istream</code>   |
| <code>ostream</code>   |
| <code>iostream</code>  |
| <code>fstream</code>   |
| <code>ifstream</code>  |
| <code>ofstream</code>  |

В остальной части этой книги используются имена символьных классов, поскольку именно они применяются в программах. Те же имена используются и старой библиотекой ввода-вывода. Вот поэтому старая и новая библиотеки совместимы на уровне исходного кода.

И еще: класс `ios` содержит множество функций-членов и переменных, которые управляют основными операциями над потоками или отслеживают результаты их выполнения. Поэтому имя класса `ios` будет употребляться в этой книге довольно часто. И помните: если включить в программу заголовок `<iostream>`, она будет иметь доступ к этому важному классу.



### Вопросы для текущего контроля

1. Что такое поток? Что такое файл?
2. Какой объект связан со стандартным выходным потоком?
3. C++-система ввода-вывода поддерживается сложной иерархией классов. Верно ли это?\*

**ВАЖНО!**

## 11.3. Перегрузка операторов ввода-вывода

В примерах из предыдущих модулей при необходимости выполнить операцию ввода или вывода данных, связанных с некоторым классом, создавались функции-члены, назначение которых состояло лишь в том, чтобы ввести или вывести эти данные. Несмотря на то что в самом этом решении нет ничего неправильного, в C++ предусмотрен более удачный способ выполнения операций ввода-вывода “классовых” данных: путем перегрузки операторов ввода-вывода “`<<`” и “`>>`”.

В языке C++ оператор “`<<`” называется *оператором вывода или вставки*, поскольку он вставляет символы в поток. Аналогично оператор “`>>`” называется *оператором ввода или извлечения*, поскольку он извлекает данные из потока.

Операторы ввода-вывода уже прегружены (в заголовке `<iostream>`), чтобы они могли выполнять операции потокового ввода или вывода данных любых встроенных C++-типов. Здесь вы узнаете, как определить эти операторы для собственных классов.

1. Поток — это логическая абстракция, которая либо синтезирует, либо потребляет информацию. Файл — это физический объект, который содержит данные.
2. Со стандартным выходным потоком связан объект `cout`.
3. Верно, C++-система ввода-вывода поддерживается сложной иерархией классов.

## Создание перегруженных операторов вывода

В качестве простого примера рассмотрим создание оператора вывода для следующей версии класса ThreeD.

```
class ThreeD {
public:
    int x, y, z; // 3-мерные координаты
    ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
};
```

Чтобы создать операторную функцию вывода для объектов типа ThreeD, необходимо перегрузить оператор “`<<`” для класса ThreeD. Вот один из возможных способов.

```
// Отображение координат X, Y, Z (оператор вывода
// для класса ThreeD).
ostream &operator<<(ostream &stream, ThreeD obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // возвращает параметр stream
}
```

Рассмотрим внимательно эту функцию, поскольку ее содержимое характерно для многих функций вывода данных. Во-первых, отметьте, что согласно объявлению она возвращает ссылку на объект типа `ostream`. Это позволяет несколько операторов вывода объединить в одном составном выражении. Затем обратите внимание на то, что эта функция имеет два параметра. Первый представляет собой ссылку на поток, который используется в левой части оператора “`<<`”. Вторым является объект, который стоит в правой части этого оператора. (При необходимости второй параметр также может иметь тип ссылки на объект.) Само тело функции состоит из инструкций вывода трех значений координат, содержащихся в объекте типа `ThreeD`, и инструкции возврата потока `stream`.

Перед вами короткая программа, в которой демонстрируется использование оператора вывода.

```
// Демонстрация использования перегруженного оператора вывода.
```

```
#include <iostream>
using namespace std;
```

```

class ThreeD {
public:
    int x, y, z; // 3-D coordinates
    ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
};

// Отображение координат X, Y, Z (оператор вывода
// для класса ThreeD).
ostream &operator<<(ostream &stream, ThreeD obj) // Оператор
  // вывода для класса ThreeD.
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // возвращает параметр stream
}

int main()
{
    ThreeD a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);

    cout << a << b << c; // Использование оператора вывода,
                           // перегруженного для класса ThreeD,
                           // для вывода координат.

    return 0;
}

```

Эта программа генерирует такие результаты.

```

1, 2, 3
3, 4, 5
5, 6, 7

```

Если удалить код, относящийся конкретно к классу ThreeD, останется “скелет”, подходящий для любой функции вывода данных.

```

ostream &operator<<(ostream &stream, class_type obj)
{
    // код, относящийся к конкретному классу
    return stream; // возвращает параметр stream
}

```

Как уже отмечалось, для параметра `obj` разрешается использовать передачу по ссылке.

В широком смысле конкретные действия функции вывода определяются программистом. Но если вы хотите следовать профессиональному стилю программирования, то ваша функция вывода должна все-таки выводить информацию. И потом, всегда нeliшне убедиться в том, что она возвращает параметр `stream`.

## Использование функций-“друзей” для перегрузки операторов вывода

В предыдущей программе перегруженная функция вывода не была определена как член класса `ThreeD`. В действительности ни функция вывода, ни функция ввода не могут быть членами класса. Дело здесь вот в чем. Если `operator`-функция является членом класса, левый operand (невидимо передаваемый с помощью указателя `this`) должен быть объектом класса, который сгенерировал обращение к этой операторной функции. И это изменить нельзя. Однако при перегрузке операторов вывода левый operand должен быть потоком, а правый — объектом класса, данные которого подлежат выводу. Следовательно, перегруженные операторы вывода не могут быть функциями-членами.

В связи с тем, что операторные функции вывода не должны быть членами класса, для которого они определяются, возникает серьезный вопрос: как перегруженный оператор вывода может получить доступ к закрытым элементам класса? В предыдущей программе переменные `x`, `y` и `z` были определены как открытые, и поэтому оператор вывода без проблем мог получить к ним доступ. Но ведь скрытие данных — важная часть объектно-ориентированного программирования, и требовать, чтобы все данные были открытыми, попросту нелогично. Однако существует решение и для этой проблемы: оператор вывода можно сделать “другом” класса. Если функция является “другом” некоторого класса, то она получает легальный доступ к его `private`-данным. Как можно объявить “другом” класса перегруженную функцию вывода, покажем на примере класса `ThreeD`.

```
// Использование friend-функции для перегрузки
// оператора вывода (<<).
```

```
#include <iostream>
using namespace std;

class ThreeD {
    int x, y, z; // 3-мерные координаты (теперь это
                  // private-члены)
```

## 530 Модуль 11. C++-система ввода-вывода

```
public:  
    ThreeD(int a, int b, int c) { x = a; y = b; z = c; }  
    // Оператор вывода, определяемый для класса ThreeD, теперь  
    // является функцией-“другом” и поэтому имеет доступ к его  
    // закрытым данным.  
    friend ostream &operator<<(ostream &stream, ThreeD obj);  
};  
  
// Отображение координат X, Y, Z (определение оператора  
// вывода данных для класса ThreeD).  
ostream &operator<<(ostream &stream, ThreeD obj)  
{  
    stream << obj.x << ", ";  
    stream << obj.y << ", ";  
    stream << obj.z << "\n";  
    return stream; // Функция возвращает поток.  
}  
  
int main()  
{  
    ThreeD a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);  
  
    cout << a << b << c;  
  
    return 0;  
}
```

Обратите внимание на то, что переменные *x*, *y* и *z* в этой версии программы являются закрытыми в классе *ThreeD*, тем не менее, операторная функция вывода обращается к ним напрямую. Вот где проявляется великая сила “дружбы”: объявляя операторные функции ввода и вывода “друзьями” класса, для которого они определяются, мы тем самым поддерживаем принцип инкапсуляции объектно-ориентированного программирования.

### Перегрузка операторов ввода

Для перегрузки операторов ввода используйте тот же метод, который мы применяли при перегрузке оператора вывода. Например, следующий оператор ввода обеспечивает ввод трехмерных координат. Обратите внимание на то, что он также выводит соответствующее сообщение для пользователя.

```
// Прием трехмерных координат (определение оператора ввода
// данных для класса ThreeD).
istream &operator>>(istream &stream, ThreeD &obj)
{
    cout << "Введите значения координат X, Y, Z: ";
    stream >> obj.x >> obj.y >> obj.z;
    return stream;
}
```

Оператор ввода должен возвращать ссылку на объект типа `istream`. Кроме того, первый параметр должен представлять собой ссылку на объект типа `istream`. Этим типом определяется поток, указанный слева от оператора “`>>`”. Второй параметр является ссылкой на переменную, которая принимает вводимое значение. Поскольку второй параметр – ссылка, он может быть модифицирован при вводе информации.

Общий формат оператора ввода имеет следующий вид.

```
istream &operator>>(istream &stream, object_type &obj)
{
    // код операторной функции ввода данных
    return stream;
}
```

Использование функции ввода данных для объектов типа `ThreeD` демонстрируется в следующей программе.

```
// Использование перегруженного оператора ввода.

#include <iostream>
using namespace std;

class ThreeD {
    int x, y, z; // 3-мерные координаты
public:
    ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
    friend ostream &operator<<(ostream &stream, ThreeD obj);
    friend istream &operator>>(istream &stream, ThreeD &obj);
};

// Отображение координат X, Y, Z (определение оператора
// вывода для класса ThreeD).
ostream &operator<<(ostream &stream, ThreeD obj)
```

## 532 Модуль 11. C++-система ввода-вывода

```
{  
    stream << obj.x << ", ";  
    stream << obj.y << ", ";  
    stream << obj.z << "\n";  
    return stream; // Функция возвращает поток.  
}  
  
// Прием трехмерных координат (определение оператора ввода  
// данных для класса ThreeD).  
istream &operator>>(istream &stream, ThreeD &obj)  
{  
    cout << "Введите значения координат X,Y,Z: ";  
    stream >> obj.x >> obj.y >> obj.z;  
    return stream;  
}  
  
int main()  
{  
    ThreeD a(1, 2, 3);  
  
    cout << a;  
  
    cin >> a;  
    cout << a;  
  
    return 0;  
}
```

Один из возможных вариантов выполнения этой программы выглядит так.

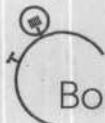
1, 2, 3

Введите значения координат X,Y,Z: 5 6 7

5, 6, 7

Подобно функциям вывода, функции ввода не могут быть членами класса, для обработки данных которого они предназначены. Их можно объявить "друзьями" этого класса или просто независимыми функциями.

За исключением того, что функция ввода должна возвращать ссылку на объект типа `istream`, тело этой функции может содержать все, что вы считаете нужным в нее включить. Но логичнее использовать операторы ввода все же по прямому назначению, т.е. для выполнения операций ввода.



## Вопросы для текущего контроля

1. Какие действия выполняет оператор вывода для класса?
2. Какие действия выполняет оператор ввода для класса?
3. Почему в качестве операторных функций ввода и вывода данных некоторого класса часто используются функции-“друзья”?\*

## Форматированный ввод-вывод данных

До сих пор при вводе или выводе информации в наших примерах программ действовали параметры форматирования, которые C++-система ввода-вывода использует по умолчанию. Но программист может сам управлять форматом представления данных, причем двумя способами. Первый способ предполагает использование функций-членов класса `ios`, а второй — функций специального типа, именуемых *манипуляторами* (*manipulator*). Мы же начнем освоение возможностей форматирования с функций-членов класса `ios`.

**ВАЖНО!**

### 11.4. Форматирование данных с использованием функций-членов класса `ios`

В системе ввода-вывода C++ каждый поток связан с набором флагов форматирования, управляющих процессом форматирования информации. В классе `ios` объявляется перечисление `fmtflags`, в котором определены следующие значения. (Точнее, эти значения определены в классе `ios_base`, который, как упоминалось выше, является базовым для класса `ios`.)

1. Оператор вывода помещает данные в поток.
2. Оператор ввода извлекает данные из потока.
3. Функции-“друзья” часто используются в качестве операторных функций ввода и вывода данных некоторого класса, поскольку они получают доступ к закрытым данным этого класса.

|             |            |            |           |
|-------------|------------|------------|-----------|
| adjustfield | floatfield | right      | skipws    |
| basefield   | hex        | scientific | unitbuf   |
| boolalpha   | internal   | showbase   | uppercase |
| dec         | left       | showpoint  |           |
| fixed       | oct        | showpos    |           |

Эти значения используются для установки или очистки флагов форматирования. При использовании старого компилятора может оказаться, что он не определяет тип перечисления `fmtflags`. В этом случае флаги форматирования будут кодироваться как целочисленные `long`-значения.

Если флаг `skipws` установлен, то при потоковом вводе данных ведущие “пробельные” символы, или символы пропуска (т.е. пробелы, символы табуляции и новой строки), отбрасываются. Если же флаг `skipws` не установлен, пробельные символы не отбрасываются.

Если установлен флаг `left`, выводимые данные выравниваются по левому краю, а если установлен флаг `right` – по правому. Если установлен флаг `internal`, числовое значение дополняется пробелами, которыми заполняется поле между ним и знаком числа или символом основания системы счисления. Если ни один из этих флагов не установлен, результат выравнивается по правому краю по умолчанию.

По умолчанию числовые значения выводятся в десятичной системе счисления. Однако основание системы счисления можно изменить. Установка флага `oct` приведет к выводу результата в восьмеричном представлении, а установка флага `hex` – в шестнадцатеричном. Чтобы при отображении результата вернуться к десятичной системе счисления, достаточно установить флаг `dec`.

Установка флага `showbase` приводит к отображению обозначения основания системы счисления, в которой представляются числовые значения. Например, если используется шестнадцатеричное представление, то значение `1F` будет отображено как `0x1F`.

По умолчанию при использовании экспоненциального представления чисел отображается строчный вариант буквы “`e`”. Кроме того, при отображении шестнадцатеричного значения используется также строчная буква “`x`”. После установки флага `uppercase` отображается прописной вариант этих символов.

Установка флага `showpos` вызывает отображение ведущего знака “плюс” перед положительными значениями.

Установка флага `showpoint` приводит к отображению десятичной точки и хвостовых нулей для всех чисел с плавающей точкой – нужны они или нет.

После установки флага `scientific` числовые значения с плавающей точкой отображаются в экспоненциальном представлении. Если установлен флаг `fixed`, вещественные значения отображаются в обычном представлении. Если не уста-

новлен ни один из этих флагов, компилятор сам выбирает соответствующий метод представления.

При установленном флаге `unitbuf` содержимое буфера сбрасывается на диск после каждой операции вывода данных.

Если установлен флаг `boolalpha`, значения булева типа можно вводить или выводить, используя ключевые слова `true` и `false`.

Поскольку часто приходится обращаться к полям `oct`, `dec` и `hex`, на них допускается коллективная ссылка `basefield`. Аналогично поля `left`, `right` и `internal` можно собирательно назвать `adjustfield`. Наконец, поля `scientific` и `fixed` можно назвать `floatfield`.

## Установка и сброс флагов форматирования

Для установки любого флага используется функция `setf()`, которая является членом класса `ios`. В самом общем виде ее формат выглядит так.

```
fmtflags setf(fmtflags flags);
```

Эта функция возвращает значение предыдущих установок флагов форматирования и устанавливает их в соответствии со значением, заданным параметром `flags`. Например, чтобы установить флаг `showbase`, можно использовать такую инструкцию.

```
stream.setf(ios::showbase);
```

Здесь элемент `stream` означает поток, параметры форматирования которого вы хотите изменить. Обратите внимание на использование префикса `ios::` для уточнения "классовой" принадлежности параметра `showbase`. Поскольку параметр `showbase` представляет собой перечислимую константу, определенную в классе `ios`, то при обращении к ней необходимо указывать имя класса `ios`. Этот принцип относится ко всем флагам форматирования.

В следующей программе функция `setf()` используется для установки флагов `showpos` и `scientific`.

```
// Использование функции setf().

#include <iostream>
using namespace std;

int main()
{
    // Установка флагов форматирования showpos и scientific
    // с помощью функции setf().
    cout.setf(ios::showpos);
    cout.setf(ios::scientific);
```

## 536 Модуль 11. C++-система ввода-вывода

```
cout << 123 << " " << 123.23 << " " ;  
  
return 0;  
}
```

При выполнении эта программа генерирует такие результаты.

```
+123 +1.232300e+002
```

С помощью операции ИЛИ можно установить сразу несколько нужных флагов форматирования в одном вызове функции `setf()`. Например, предыдущую программу можно сократить, объединив по ИЛИ флаги `scientific` и `showpos`, поскольку в этом случае выполняется только одно обращение к функции `setf()`.

```
cout.setf(ios::scientific | ios::showpos);
```

Чтобы сбросить флаг, используйте функцию `unsetf()`, прототип которой выглядит так.

```
void unsetf(fmtflags flags);
```

В этом случае будут обнулены флаги, заданные параметром `flags`. (При этом все другие флаги остаются в прежнем состоянии.)

Для того чтобы узнать текущие установки флагов форматирования, воспользуйтесь функцией `flags()`, прототип которой имеет следующий вид.

```
fmtflags flags();
```

Эта функция возвращает текущее значение флагов форматирования для вызывающего потока.

При использовании следующего формата вызова функции `flags()` устанавливаются значения флагов форматирования в соответствии с содержимым параметра `flags` и возвращаются их предыдущие значения.

```
fmtflags flags(fmtflags flags);
```

Чтобы понять, как работают функции `flags()` и `unsetf()`, рассмотрим следующую программу.

```
// Демонстрация использования функций flags() и unsetf().
```

```
#include <iostream>  
using namespace std;  
  
int main()  
{
```

```

ios::fmtflags f;

f = cout.flags(); // Считывание флагов форматирования.

if(f & ios::showpos)
    cout << "Флаг showpos установлен для потока cout.\n";
else
    cout << "Флаг showpos сброшен для потока cout.\n";

cout << "\nУстановка флага showpos для потока cout.\n";
cout.setf(ios::showpos); // Установка флагов форматирования.

f = cout.flags();

if(f & ios::showpos)
    cout << "Флаг showpos установлен для потока cout.\n";
else
    cout << "Флаг showpos сброшен для потока cout.\n";

cout << "\nСброс флага showpos для потока cout.\n";
cout.unsetf(ios::showpos); // Сброс флагов форматирования.

f = cout.flags();

if(f & ios::showpos)
    cout << "Флаг showpos установлен для потока cout.\n";
else
    cout << "Флаг showpos сброшен для потока cout.\n";

return 0;
}

```

Результаты выполнения этой программы таковы.

Флаг showpos сброшен для потока cout.

Установка флага showpos для потока cout.

Флаг showpos установлен для потока cout.

Сброс флага `showpos` для потока `cout`.

Флаг `showpos` сброшен для потока `cout`.

В этой программе обратите внимание на то, что тип `fmtflags` указан с уточняющим префиксом `ios::`. Необходимость этого уточнения вызвана тем, что тип `fmtflags` определен в классе `ios`. В общем случае при использовании имени типа или перечислимой константы, определенной в некотором классе, соответствующее имя обязательно нужно указывать вместе с именем класса.

## Установка ширины поля, точности и символов заполнения

Помимо флагов форматирования можно также устанавливать ширину поля, символ заполнения и количество цифр после десятичной точки (точность). Для этого достаточно использовать следующие три функции, определенные в классе `ios::width()`, `fill()` и `precision()`.

По умолчанию выводимое значение имеет такую ширину поля (т.е. количество занимаемых позиций), которая определяется количеством символов, необходимых для его отображения. Но с помощью функции `width()` можно установить минимальную ширину этого поля. Вот как выглядит прототип этой функции.

```
streamsize width(streamsize w);
```

Функция `width()` возвращает текущую ширину поля и устанавливает новую равной значению параметра `w`. Некоторые реализации C++-компилятора требуют, чтобы ширина поля устанавливалась перед каждой операцией вывода данных. В противном случае ширина поля выводимого значения будет установлена по умолчанию. Если количество символов, из которых состоит выводимое значение, превышает заданную ширину поля, границы этого поля будут “нарушены”. Значения при выводе не усекаются. Тип `streamsize` определен как разновидность целочисленного типа.

После установки минимального размера ширины поля оставшиеся (после вывода значения) позиции (если такие имеются) заполняются текущим символом заполнения (по умолчанию используется пробел). Функция `fill()` возвращает текущий символ заполнения и устанавливает новый, заданный параметром `ch`. Этот символ используется для дополнения результата символами, недостающими для достижения заданной ширины поля. Прототип этой функции выглядит так.

```
char fill(char ch);
```

Функция `precision()` возвращает текущее количество цифр, отображаемых после десятичной точки, и устанавливает новое текущее значение точности равным содержимому параметра `p`. (По умолчанию после десятичной точки отобра-

жается шесть цифр.) Некоторые реализации C++-компилятора требуют, чтобы значение точности устанавливалось перед каждой операцией вывода значений с плавающей точкой. В противном случае значение точности будет использовано по умолчанию.

```
streamsize precision(streamsize p);
```

Рассмотрим программу, которая демонстрирует использование этих трех функций.

```
// Демонстрация использования функций width(), precision()
// и fill().

#include <iostream>
using namespace std;

int main()
{
    cout.setf(ios::showpos);
    cout.setf(ios::scientific);
    cout << 123 << " " << 123.23 << "\n";

    // Установка точности.
    cout.precision(2); // Две цифры после десятичной точки.
    // Установка ширины поля.
    cout.width(10); // Все поле состоит из 10 символов.
    cout << 123 << " ";
    cout.width(10); // Установка ширины поля равной 10.
    cout << 123.23 << "\n";

    // Установка символа заполнения.
    cout.fill('#'); // Для заполнителя возьмем символ "#"
    cout.width(10); // Установка ширины поля равной 10.
    cout << 123 << " ";
    cout.width(10); // Установка ширины поля равной 10.
    cout << 123.23;

    return 0;
}
```

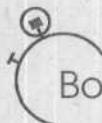
Программа генерирует такие результаты.

```
+123. +1.232300e+002  
+123 +1.23e+002  
#####+123 +1.23e+002
```

Как упоминалось выше, в некоторых реализациях необходимо переустанавливать значение ширины поля перед выполнением каждой операции вывода. Поэтому функция `width()` в предыдущей программе вызывалась несколько раз.

В системе ввода-вывода C++ определены и перегруженные версии функций `width()`, `precision()` и `fill()`, которые не изменяют текущие значения соответствующих параметров форматирования и используются только для их получения. Вот как выглядят их прототипы.

```
char fill();  
streamsize width();  
streamsize precision();
```



### Вопросы для текущего контроля

1. Каково назначение флага `boolalpha`?
2. Какие действия выполняет функция `setf()`?
3. Какая функция используется для установки символа заполнения поля?\*

**ВАЖНО!**

## 11.5. Использование манипуляторов ввода-вывода

В C++-системе ввода-вывода предусмотрен и второй способ изменения параметров форматирования, связанных с потоком. Он реализуется с помощью специальных функций, называемых *манипуляторами*, которые можно включать в выражение ввода-вывода. Стандартные манипуляторы описаны в табл. 11.1.

1. Если флаг `boolalpha` установлен, булевые значения вводятся и выводятся с использованием слов `true` и `false`.
2. Функция `setf()` устанавливает один или несколько флагов форматирования.
3. Для установки символа заполнения поля используется функция `fill()`.

Таблица 11.1. Стандартные C++-манипуляторы ввода-вывода

| Манипулятор          | Назначение                                                                                                                                 | Функция    |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------|------------|
| boolalpha            | Устанавливает флаг <code>boolalpha</code>                                                                                                  | Ввод-вывод |
| dec                  | Устанавливает флаг <code>dec</code>                                                                                                        | Ввод-вывод |
| endl                 | Выывает символ новой строки и “сбрасывает” поток, т.е. переписывает содержимое буфера, связанного с потоком, на соответствующее устройство | Вывод      |
| ends                 | Вставляет в поток нулевой символ ('\\0')                                                                                                   | Вывод      |
| fixed                | Устанавливает флаг <code>fixed</code>                                                                                                      | Вывод      |
| flush                | “Сбрасывает” поток                                                                                                                         | Вывод      |
| hex                  | Устанавливает флаг <code>hex</code>                                                                                                        | Ввод-вывод |
| internal             | Устанавливает флаг <code>internal</code>                                                                                                   | Вывод      |
| left                 | Устанавливает флаг <code>left</code>                                                                                                       | Вывод      |
| noboolalpha          | Обнуляет флаг <code>boolalpha</code>                                                                                                       | Ввод-вывод |
| noshowbase           | Обнуляет флаг <code>showbase</code>                                                                                                        | Вывод      |
| noshowpoint          | Обнуляет флаг <code>showpoint</code>                                                                                                       | Вывод      |
| noshowpos            | Обнуляет флаг <code>showpos</code>                                                                                                         | Вывод      |
| noskipws             | Обнуляет флаг <code>skipws</code>                                                                                                          | Ввод       |
| nounitbuf            | Обнуляет флаг <code>unitbuf</code>                                                                                                         | Вывод      |
| nouppercase          | Обнуляет флаг <code>uppercase</code>                                                                                                       | Вывод      |
| oct                  | Устанавливает флаг <code>oct</code>                                                                                                        | Ввод-вывод |
| resetiosflags (      | Обнуляет флаги, заданные в параметре <i>f</i>                                                                                              | Ввод-вывод |
| fmtflags <i>f</i> )  |                                                                                                                                            |            |
| right                | Устанавливает флаг <code>right</code>                                                                                                      | Вывод      |
| scientific           | Устанавливает флаг <code>scientific</code>                                                                                                 | Вывод      |
| setbase (            | Устанавливает основание системы счисления                                                                                                  | Вывод      |
| int <i>base</i> )    | равной значению <i>base</i>                                                                                                                |            |
| setfill (            | Устанавливает символ-заполнитель равным                                                                                                    | Вывод      |
| int <i>ch</i> )      | значению параметра <i>ch</i>                                                                                                               |            |
| setiosflags (        | Устанавливает флаги, заданные в параметре <i>f</i>                                                                                         | Ввод-вывод |
| fmtflags <i>f</i> )  |                                                                                                                                            |            |
| setprecision (       | Устанавливает количество цифр точности                                                                                                     | Вывод      |
| int <i>p</i> )       | (после десятичной точки)                                                                                                                   |            |
| setw (int <i>w</i> ) | Устанавливает ширину поля равной значению параметра <i>w</i>                                                                               | Вывод      |
| showbase             | Устанавливает флаг <code>showbase</code>                                                                                                   | Вывод      |
| showpoint            | Устанавливает флаг <code>showpoint</code>                                                                                                  | Вывод      |
| showpos              | Устанавливает флаг <code>showpos</code>                                                                                                    | Вывод      |
| skipws               | Устанавливает флаг <code>skipws</code>                                                                                                     | Ввод       |
| unitbuf              | Устанавливает флаг <code>unitbuf</code>                                                                                                    | Вывод      |
| uppercase            | Устанавливает флаг <code>uppercase</code>                                                                                                  | Вывод      |
| ws                   | Пропускает ведущие “пробельные” символы                                                                                                    | Ввод       |

Манипулятор используется как часть выражения ввода-вывода. Вот пример программы, в которой показано, как с помощью манипуляторов можно управлять форматированием выводимых данных.

```
// Демонстрация использования манипуляторов ввода-вывода.
```

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << setprecision(2) << 1000.243 << endl;
    cout << setw(20) << "Привет всем!";

    return 0;
}
```

При выполнении программа генерирует такие результаты.

```
1e+003
Привет всем!
```

Обратите внимание на то, как используются манипуляторы в цепочке операций ввода-вывода. Кроме того, отметьте, что, если манипулятор вызывается без аргументов (как, например, манипулятор `endl` в нашей программе), то его имя указывается без пары круглых скобок.

В следующей программе используется манипулятор `setiosflags()` для установки флагов `scientific` и `showpos`.

```
// Использование манипулятора setiosflags().
```

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << setiosflags(ios::showpos) <<
        setiosflags(ios::scientific) <<
        123 << " " << 123.23;

    return 0;
}
```

А в этой программе демонстрируется использование манипулятора ws, который пропускает ведущие "пробельные" символы при вводе строки в массив s:

```
// Пропуск начальных "пробельных" символов.

#include <iostream>
using namespace std;

int main()
{
    char s[80];

    cin >> ws >> s; // Использование манипулятора ws.
    cout << s;

    return 0;
}
```

**ВАЖНО!**

## 11.6 Создание собственных манипуляторных функций

Программист может создавать собственные манипуляторные функции. Существует два типа манипуляторных функций: принимающие и не принимающие аргументы. Для создания параметризованных манипуляторов используются методы, рассмотрение которых выходит за рамки этой книги. Однако создание манипуляторов, которые не имеют параметров, не вызывает особых трудностей.

Все манипуляторные функции вывода данных без параметров имеют следующую структуру.

```
ostream &manip_name(ostream &stream)
{
    // код манипуляторной функции
    return stream;
}
```

Здесь элемент *manip\_name* означает имя манипулятора. Важно понимать, что, несмотря на то, что манипулятор принимает в качестве единственного аргумента указатель на поток, который он обрабатывает, при использовании манипулятора в результирующем выражении ввода-вывода аргументы не указываются вообще.

В следующей программе создается манипулятор `setup()`, который устанавливает флаг выравнивания по левому краю, ширину поля равной 10 и задает в качестве заполняющего символа знак доллара.

```
// Создание манипулятора для форматирования выводимой
// информации.

#include <iostream>
#include <iomanip>
using namespace std;

ostream &setup(ostream &stream) // Определение манипуляторной
{
    // функции.
    stream.setf(ios::left);
    stream << setw(10) << setfill('$');
    return stream;
}

int main()
{
    cout << 10 << " " << setup << 10;

    return 0;
}
```

Собственные манипуляторы полезны по двум причинам. Во-первых, иногда возникает необходимость выполнять операции ввода-вывода с использованием устройства, к которому ни один из встроенных манипуляторов не применяется (например, плоттер). В этом случае создание собственных манипуляторов сделает вывод данных на это устройство более удобным. Во-вторых, может оказаться, что у вас в программе некоторая последовательность инструкций повторяется несколько раз. И тогда, как показано в предыдущей программе, вы можете объединить эти операции в один манипулятор.

Все манипуляторные функции ввода данных без параметров имеют следующий формат.

```
istream &manip_name(istream &stream)
{
    // код манипуляторной функции
    return stream;
}
```

Например, в следующей программе создается манипулятор `prompt()`. Он настраивает входной поток на прием данных в шестнадцатеричном представлении и отображает для пользователя наводящее сообщение.

```
// Создание манипулятора для форматирования вводимой
// информации.

#include <iostream>
#include <iomanip>
using namespace std;

istream &prompt(istream &stream)
{
    cin >> hex;
    cout << "Введите число в шестнадцатеричном формате: ";

    return stream;
}

int main()
{
    int i;

    cin >> prompt >> i;
    cout << i;

    return 0;
}
```

Помните: очень важно, чтобы ваш манипулятор возвращал потоковый объект (элемент `stream`). В противном случае этот манипулятор нельзя будет использовать в цепочке операций ввода или вывода.

## Файловый ввод-вывод данных

В C++-системе ввода-вывода также предусмотрены средства для выполнения операций с файлами. Файловые операции ввода-вывода можно реализовать после включения в программу заголовка `<fstream>`, в котором определены все необходимые для этого классы и значения.



### Вопросы для текущего контроля

1. Какие действия выполняет манипулятор endl?
2. Какие действия выполняет манипулятор ws?
3. Можно ли использовать манипулятор ввода-вывода как часть выражения ввода-вывода?\*

**ВАЖНО!**

## 11.7 Как открыть и закрыть файл

В C++ файл открывается путем связывания его с потоком. Как вы знаете, существуют потоки трех типов: ввода, вывода и ввода-вывода. Чтобы открыть входной поток, необходимо объявить потоковый объект типа ifstream. Для открытия выходного потока нужно объявить поток класса ofstream. Поток, который предполагается использовать для операций как ввода, так и вывода, должен быть объявлен как объект класса fstream. Например, при выполнении следующего фрагмента кода будет создан входной поток, выходной и поток, позволяющий выполнять операции в обоих направлениях.

```
ifstream in; // входной поток
ofstream out; // выходной поток
fstream both; // поток ввода-вывода
```

Создав поток, его нужно связать с файлом. Это можно сделать с помощью функции open(), причем в каждом из трех потоковых классов есть своя функция-член open(). Представим их прототипы.

```
void ifstream::open(const char *filename,
                     ios::openmode mode = ios::in);

void ofstream::open(
    const char *filename,
    ios::openmode mode = ios::out | ios::trunc);
```

1. Манипулятор endl обеспечивает переход на новую строку.
2. Манипулятор ws обеспечивает пропуск ведущих пробельных символов при вводе данных.
3. Да, манипулятор ввода-вывода можно использовать как часть выражения ввода-вывода.

```
void fstream::open(
    const char *filename,
    ios::openmode mode = ios::in | ios::out);
```

Здесь элемент *filename* означает имя файла, которое может включать спецификатор пути. Элемент *mode* определяет способ открытия файла. Он должен принимать одно или несколько значений перечисления *openmode*, которое определено в классе *ios*. Ниже приведены значения этого перечисления.

```
ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc
```

Несколько значений перечисления *openmode* можно объединять посредством логического сложения (ИЛИ).

Включение значения *ios::app* в параметр *mode* обеспечит присоединение к концу файла всех выводимых данных. Это значение можно применять только к файлам, открытym для вывода данных. При открытии файла с использованием значения *ios::ate* поиск будет начинаться с конца файла. Несмотря на это, операции ввода-вывода могут по-прежнему выполняться по всему файлу.

Значение *ios::in* говорит о том, что данный файл открывается для ввода данных, а значение *ios::out* обеспечивает открытие файла для их вывода.

Значение *ios::binary* позволяет открыть файл в двоичном режиме. По умолчанию все файлы открываются в текстовом режиме. Как упоминалось выше, в текстовом режиме могут происходить некоторые преобразования символов (например, последовательность, состоящая из символов возврата каретки и перехода на новую строку, может быть преобразована в символ новой строки). Но при открытии файла в двоичном режиме никакого преобразования символов не выполняется. Следует иметь в виду, что любой файл, содержащий форматированный текст или еще необработанные данные, можно открыть как в двоичном, так и в текстовом режиме. Единственное различие между этими режимами состоит в преобразовании (или нет) символов.

Использование значения *ios::trunc* приводит к разрушению содержимого файла, имя которого совпадает с параметром *filename*, а сам этот файл усекается до нулевой длины. При создании выходного потока типа *ofstream* любой существующий файл с именем *filename* автоматически усекается до нулевой длины.

При выполнении следующего фрагмента кода открывается обычный выходной файл.

```
ofstream out;
out.open("тест");
```

Поскольку параметр *mode* функции `open()` по умолчанию устанавливается равным значению, соответствующему типу открываемого потока, в предыдущем примере вообще нет необходимости задавать его значение.

Не открытый в результате неудачного выполнения функции `open()` поток при использовании в булевом выражении устанавливается равным значению ЛОЖЬ. Этот факт может служить для подтверждения успешного открытия файла, например, с помощью такой `if`-инструкции.

```
if(!mystream) {
    cout << "Не удается открыть файл.\n";
    // обработка ошибки
}
```

Прежде чем делать попытку получить доступ к файлу, следует всегда проверять результат вызова функции `open()`.

Можно также проверить факт успешного открытия файла с помощью функции `is_open()`, которая является членом классов `fstream`, `ifstream` и `ofstream`. Вот ее прототип.

```
bool is_open();
```

Эта функция возвращает значение ИСТИНА, если поток связан с открытым файлом, и ЛОЖЬ – в противном случае. Например, используя следующий код, можно узнать, открыт ли в данный момент потоковый объект `mystream`.

```
if(!mystream.is_open()) {
    cout << "Файл не открыт.\n";
    // ...
```

Хотя вполне корректно использовать функцию `open()` для открытия файла, в большинстве случаев это делается по-другому, поскольку классы `ifstream`, `ofstream` и `fstream` включают конструкторы, которые автоматически открывают заданный файл. Параметры у этих конструкторов и их значения (действующие по умолчанию) совпадают с параметрами и соответствующими значениями функции `open()`. Поэтому чаще всего файл открывается так, как показано в следующем примере.

```
ifstream mystream("myfile"); // файл открывается для ввода
```

Если по какой-то причине файл открыть невозможно, потоковая переменная, связываемая с этим файлом, устанавливается равной значению ЛОЖЬ.

Чтобы закрыть файл, используйте функцию-член `close()`. Например, чтобы закрыть файл, связанный с потоковым объектом `mystream`, используйте такую инструкцию.

```
mystream.close();
```

Функция `close()` не имеет параметров и не возвращает никакого значения.

**ВАЖНО!**

## 11.8. Чтение и запись текстовых файлов

Проще всего считывать данные из текстового файла или записывать их в него с помощью операторов “`<<`” и “`>>`”. Например, в следующей программе выполняется запись в файл `test` целого числа, значения с плавающей точкой и строки.

```
// Запись данных в файл.

#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream out("test"); // Создаем файл с именем test и
                          // открываем его для вывода данных.
    if(!out) {
        cout << "Не удается открыть файл.\n";
        return 1;
    }

    out << 10 << " " << 123.23 << "\n"; // Выводим данные
  // в файл.

    out << "Это короткий текстовый файл.";

    out.close(); // Закрываем файл.

    return 0;
}
```

Следующая программа считывает целое число, `float`-значение, символ и строку из файла, созданного при выполнении предыдущей программы.

## 550 Модуль 11. C++-система ввода-вывода

```
// Считывание данных из файла.

#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    char ch;
    int i;
    float f;
    char str[80];

    ifstream in("test"); // Открываем файл для ввода данных.
    if(!in) {
        cout << "Не удается открыть файл.\n";
        return 1;
    }

    in >> i; // Считываем данные из файла.
    in >> f;
    in >> ch;
    in >> str;

    cout << i << " " << f << " " << ch << "\n";
    cout << str;

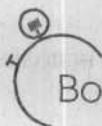
    in.close(); // Закрываем файл.
    return 0;
}
```

Следует иметь в виду, что при использовании оператора “>>” для считывания данных из текстовых файлов происходит преобразование некоторых символов. Например, “пробельные” символы опускаются. Если необходимо предотвратить какие бы то ни было преобразования символов, откройте файл в двоичном режиме доступа. Кроме того, помните, что при использовании оператора “>>” для считывания строки ввод прекращается при обнаружении первого “пробельного” символа.

## Спросим у опытного программиста

**Вопрос.** Как разъяснялось в модуле 1, C++ является супермножеством языка С. Мне известно, что в С определена собственная система ввода-вывода. Тогда возникает вопрос, может ли C+-программист использовать С-систему ввода-вывода? Если да, то имеет ли смысл ее применение в C+-программах?

**Ответ.** На первый вопрос ответ положительный: С-система ввода-вывода доступна для C+-программистов. Но на второй вопрос отвечу: "Нет". С-система ввода-вывода не является объектно-ориентированной. Поэтому для применения в C+-программах больше подходят C+-средства ввода-вывода. Однако С-система ввода-вывода по-прежнему широко используется благодаря простоте и небольшим расходам системных ресурсов. Таким образом, для ряда узкоспециализированных программ она может оказаться вполне удачным выбором. Подробную информацию о С-системе ввода-вывода можно найти в моей книге *Полный справочник по C++* (М.: Издательский дом "Вильямс").



## Вопросы для текущего контроля

1. Какой класс используется для создания входного файла?
2. С помощью какой функции можно открыть файл?
3. Можно ли считывать информацию из файла и записывать в него, используя операторы “<<” и “>>”?\*

ВАЖНО!

## 11.2 Неформатированный ввод-вывод данных в двоичном режиме

Несмотря на простоту чтения (и записи) форматированных текстовых файлов (подобных тем, которые использовались в предыдущих примерах), они не

1. Чтобы создать входной файл, используйте класс `ifstream`.
2. Чтобы открыть файл, используйте конструктор соответствующего класса или функцию `open()`.
3. Да, операторы “<<” и “>>” можно использовать для считывания информации из файла и записи в него.

являются наиболее эффективным способом обработки файлов. Иногда просто необходимо сохранять неформатированные двоичные данные, а не текст. Поэтому C++ поддерживает ряд функций файлового ввода-вывода в двоичном режиме, которые могут выполнять операции без форматирования данных.

Для выполнения двоичных операций файлового ввода-вывода необходимо открыть файл с использованием спецификатора режима `ios::binary`. Необходимо отметить, что функции обработки неформатированных файлов могут работать с файлами, открытыми в текстовом режиме доступа, но при этом могут иметь место преобразования символов, которые сводят на нет основную цель выполнения двоичных файловых операций.

В общем случае существует два способа записи неформатированных двоичных данных в файл и считывания их из файла. Первый состоит в использовании функции-члена `put()` (для записи байта в файл) и функции-члена `get()` (для считывания байта из файла). Второй способ предполагает применение "блочных" C++-функций ввода-вывода `read()` и `write()`. Рассмотрим каждый способ в отдельности.

### Использование функций `get()` и `put()`

Функции `get()` и `put()` имеют множество форматов, но чаще всего используются следующие их версии:

```
istream &get(char &ch);  
ostream &put(char ch);
```

Функция `get()` считывает один символ из соответствующего потока и помещает его значение в переменную `ch`. Она возвращает ссылку на поток, связанный с предварительно открытым файлом. При достижении конца этого файла значение ссылки станет равным нулю. Функция `put()` записывает символ `ch` в поток и возвращает ссылку на этот поток.

При выполнении следующей программы на экран будет выведено содержимое любого заданного файла. Здесь используется функция `get()`.

```
// Отображение содержимого файла с помощью функции get().  
  
#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main(int argc, char *argv[])  
{  
    char ch;
```

```

if(argc!=2) {
    cout << "Применение: имя_программы <имя_файла>\n";
    return 1;
}

ifstream in(argv[1], ios::in | ios::binary); // Открываем
  // файл в двоичном режиме.

if(!in) {
    cout << "Не удается открыть файл.\n";
    return 1;
}

// Считываем данные до тех пор, пока не будет достигнут
// конец файла.

while(in) { // При достижении конца файла потоковый
            // объект in примет значение false.

    in.get(ch);
    if(in) cout << ch;
}

in.close();

return 0;
}

```

Рассмотрим цикл `while`. При достижении конца файла потоковый объект `in` примет значение ЛОЖЬ, которое и остановит выполнение этого цикла.

Существует более короткий вариант цикла, предназначенного для считывания и отображения содержимого файла.

```
while(in.get(ch))
    cout << ch;
```

Этот вариант также имеет право на существование, поскольку функция `get()` возвращает потоковый объект `in`, который при достижении конца файла примет значение `false`.

В следующей программе для записи строки в файл используется функция `put()`.

```
// Использование функции put() для записи строки в файл.

#include <iostream>
```

## 554 Модуль 11. C++-система ввода-вывода

```
#include <fstream>
using namespace std;

int main()
{
    char *p = "Всем привет!\n";

    ofstream out("test", ios::out | ios::binary);
    if(!out) {
        cout << "Не удается открыть файл.\n";
        return 1;
    }

    // Записываем символы до тех пор, пока не встретится
    // символ завершения строки.
    while(*p) out.put(*p++);
    // Записываем строку в файл с
    // помощью функции put() без
    // каких-либо преобразований
    // символов.

    out.close();

    return 0;
}
```

После выполнения этой программы файл `test` будет содержать строку `Всем привет!`, за которой будет следовать символ новой строки. Никакого преобразования символов здесь не происходит.

### Считывание и запись в файл блоков данных

Чтобы считывать и записывать в файл блоки двоичных данных, используйте функции-члены `read()` и `write()`. Их прототипы имеют следующий вид.

```
istream &read(char *buf, streamsize num);
ostream &write(const char *buf, streamsize num);
```

Функция `read()` считывает `num` байт данных из связанного с файлом потока и помещает их в буфер, адресуемый параметром `buf`. Функция `write()` записывает `num` байт данных в связанный с файлом поток из буфера, адресуемого параметром `buf`. Как упоминалось выше, тип `streamsize` определен библиотекой C++ как некоторая разновидность целочисленного типа. Он позволяет хранить

самое большое количество байтов, которое может быть передано в процессе любой операции ввода-вывода.

При выполнении следующей программы сначала в файл записывается массив целых чисел, а затем он же считывается из файла.

```
// Использование функций read() и write().  
  
#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main()  
{  
    int n[5] = {1, 2, 3, 4, 5};  
    register int i;  
  
    ofstream out("test", ios::out | ios::binary);  
    if(!out) {  
        cout << "Не удается открыть файл.\n";  
        return 1;  
    }  
  
    out.write((char *) &n, sizeof n); // Записываем блок данных.  
  
    out.close();  
  
    for(i=0; i<5; i++) // очищаем массив  
        n[i] = 0;  
  
    ifstream in("test", ios::in | ios::binary);  
    if(!in) {  
        cout << "Не удается открыть файл.\n";  
        return 1;  
    }  
  
    in.read((char *) &n, sizeof n); // Считываем блок данных.  
  
    for(i=0; i<5; i++) // Отображаем значения, считанные  
                        // из файла.  
        cout << n[i] << " ";
```

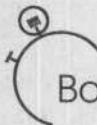
```
    in.close();  
  
    return 0;  
}
```

Обратите внимание на то, что в инструкциях обращения к функциям `read()` и `write()` выполняются операции приведения типа, которые обязательны при использовании буфера, определенного не в виде символьного массива.

Если конец файла будет достигнут до того, как будет считано *n* символов, функция `read()` просто прекратит выполнение, а буфер будет содержать столько символов, сколько удалось считать до этого момента. Точное количество считанных символов можно узнать с помощью еще одной функции-члена `gcount()`, которая имеет такой прототип.

```
streamsize gcount();
```

Функция `gcount()` возвращает количество символов, считанных в процессе выполнения последней операции ввода данных.



### Вопросы для текущего контроля

1. Какой спецификатор режима используется при открытии файла для считывания или записи двоичных данных?
2. Каково назначение функции `get()`? Каково назначение функции `put()`?
3. С помощью какой функции можно считать блок данных?\*

ВАЖНО!

## 11.10 Использование других функций ввода-вывода

C++-система ввода-вывода содержит множество других полезных функций. Рассмотрим некоторые из них.

- 
1. Чтобы считать или записать двоичные данные, используйте спецификатор режима `ios::binary`.
  2. Функция `get()` считывает символ, а функция `put()` записывает символ.
  3. Чтобы считать блок данных, используйте функцию `read()`.

## Версии функции `get()`

Помимо приведенного выше формата использования функции `get()` существуют и другие ее перегруженные версии. Приведем прототипы для трех из них, используемых чаще всего.

```
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
int get();
```

Первая версия позволяет считывать символы в массив, заданный параметром `buf`, до тех пор, пока либо не будет считано `num-1` символов, либо не встретится символ новой строки, либо не будет достигнут конец файла. После выполнения функции `get()` массив, адресуемый параметром `buf`, будет иметь завершающий нуль-символ. Символ новой строки, если таковой обнаружится во входном потоке, не извлекается. Он остается там до тех пор, пока не выполнится следующая операция ввода-вывода.

Вторая версия предназначена для считывания символов в массив, адресуемый параметром `buf`, до тех пор, пока либо не будет считано `num-1` символов, либо не обнаружится символ, заданный параметром `delim`, либо не будет достигнут конец файла. После выполнения функции `get()` массив, адресуемый параметром `buf`, будет иметь завершающий нуль-символ. Символ-разделитель (заданный параметром `delim`), если таковой обнаружится во входном потоке, не извлекается. Он остается там до тех пор, пока не выполнится следующая операция ввода-вывода.

Третья перегруженная версия функции `get()` возвращает из потока следующий символ. Он содержится в младшем байте значения, возвращаемого функцией. Следовательно, значение, возвращаемое функцией `get()`, можно присвоить переменной типа `char`. При достижении конца файла эта функция возвращает значение `EOF`, которое определено в заголовке `<iostream>`.

Функцию `get()` полезно использовать для считывания строк, содержащих пробелы. Как вы знаете, если для считывания строки используется оператор `">>"`, процесс ввода останавливается при обнаружении первого же пробельного символа. Это делает оператор `">>"` бесполезным для считывания строк, содержащих пробелы. Но эту проблему, как показано в следующей программе, можно обойти с помощью функции `get(buf, num)`.

```
// Использование функции get() для считывания строк,
// содержащих пробелы.
```

```
#include <iostream>
#include <fstream>
```

```

using namespace std;

int main()
{
    char str[80];

    cout << "Введите имя: ";
    cin.get(str, 79); // Используем функцию get() для считывания
                      // строки, содержащей пробелы.

    cout << str << '\n';

    return 0;
}

```

Здесь в качестве символа-разделителя при считывании строки с помощью функции `get()` используется символ новой строки. Это делает функцию `get()` гораздо безопаснее функции `gets()`.

## Функция `getline()`

Рассмотрим еще одну функцию, которая позволяет вводить данные. Речь идет о функции `getline()`, которая является членом каждого потокового класса, предназначенного для ввода информации. Вот как выглядят прототипы версий этой функции.

```

istream &getline(char *buf, streamsize num);
istream &getline(char *buf, streamsize num, char delim);

```

При использовании первой версии символычитываются в массив, адресуемый указателем `buf`, до тех пор, пока либо не будет считано `num-1` символов, либо не встретится символ новой строки, либо не будет достигнут конец файла. После выполнения функции `getline()` массив, адресуемый параметром `buf`, будет иметь завершающий нуль-символ. Символ новой строки, если таковой обнаружится во входном потоке, при этом извлекается, но не помещается в массив `buf`.

Вторая версия предназначена для считывания символов в массив, адресуемый параметром `buf`, до тех пор, пока либо не будет считано `num-1` символов, либо не обнаружится символ, заданный параметром `delim`, либо не будет достигнут конец файла. После выполнения функции `getline()` массив, адресуемый параметром `buf`, будет иметь завершающий нуль-символ. Символ-разделитель (за-

данный параметром *delim*), если таковой обнаружится во входном потоке, извлекается, но не помещается в массив *buf*.

Как видите, эти две версии функции `getline()` практически идентичны версиям `get(buf, num)` и `get(buf, num, delim)` функции `get()`. Обе считывают символы из входного потока и помещают их в массив, адресуемый параметром *buf*, до тех пор, пока либо не будет считано *num*-1 символов, либо не обнаружится символ, заданный параметром *delim*. Различие между функциями `get()` и `getline()` состоит в том, что функция `getline()` считывает и удаляет символ-разделитель из входного потока, а функция `get()` этого не делает.

## Функция обнаружения конца файла

Обнаружить конец файла можно с помощью функции-члена `eof()`, которая имеет такой прототип.

```
bool eof();
```

Эта функция возвращает значение `true` при достижении конца файла; в противном случае она возвращает значение `false`.

## Функции `peek()` и `putback()`

Следующий символ из входного потока можно получить и не удалять его оттуда с помощью функции `peek()`. Вот как выглядит ее прототип.

```
int peek();
```

Функция `peek()` возвращает следующий символ потока, или значение EOF, если достигнут конец файла. Считанный символ возвращается в младшем байте значения, возвращаемого функцией.

Последний символ, считанный из потока, можно вернуть в поток, используя функцию `putback()`. Ее прототип выглядит так.

```
istream &putback(char c);
```

Здесь параметр *c* содержит символ, считанный из потока последним.

## Функция `flush()`

При выводе данных немедленной их записи на физическое устройство, связанное с потоком, не происходит. Подлежащая выводу информация накапливается во внутреннем буфере до тех пор, пока этот буфер не заполнится целиком. И только тогда его содержимое переписывается на диск. Однако существует возможность немедленной перезаписать на диск хранимую в буфере информацию,

не дожидаясь его заполнения. Для этого следует вызвать функцию `flush()`. Ее прототип имеет такой вид.

```
ostream &flush();
```

К вызовам функции `flush()` следует прибегать в случае, если программа предназначена для выполнения в неблагоприятных средах (для которых характерны, например, частые отключения электричества).

 Содержимое всех буферов переписывается на диск также при закрытии файла или завершении программы.

## Проект 11.1. Программа сравнения файлов

`CompFiles.cpp`

В этом проекте разрабатывается довольно простое, но полезное средство сравнения файлов. Механизм сравнения состоит в следующем. После открытия обоих файлов из каждого из нихчитываются порции байтов и сравниваются между собой. При обнаружении несовпадения делается вывод о том, что сравниваемые файлы различны. Если же конец обоих файлов достигается одновременно и при этом никакого совпадения не обнаруживается, эти файлы считаются одинаковыми.

### Последовательность действий

1. Создайте файл с именем `CompFiles.cpp`.
2. Этот файл должен начинаться такими строками кода.

```
/*
    Проект 11.1
```

```
    Создание средства сравнения файлов.
```

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    register int i;
    int numread;
```

```

unsigned char buf1[1024], buf2[1024];

if(argc!=3) {
    cout << "Применение: compfiles <file1> <file2>\n";
    return 1;
}

```

Обратите внимание на то, что имена сравниваемых файлов указываются в командной строке.

- Добавьте код, который обеспечивает открытие файлов для выполнения операций ввода данных в двоичном режиме.

```

ifstream f1(argv[1], ios::in | ios::binary);
if(!f1) {
    cout << "Не удается открыть первый файл.\n";
    return 1;
}
ifstream f2(argv[2], ios::in | ios::binary);
if(!f2) {
    cout << "Не удается открыть второй файл.\n";
    return 1;
}

```

Файлы здесь открываются для выполнения операций ввода в двоичном режиме, чтобы не допустить преобразований символов, которые характерны для текстового режима.

- Добавьте код, который, собственно, и реализует сравнение файлов.

```

cout << "Сравнение файлов...\n";

do {
    f1.read((char *) buf1, sizeof buf1);
    f2.read((char *) buf2, sizeof buf2);

    if(f1.gcount() != f2.gcount()) {
        cout << "Файлы имеют различный размер.\n";
        f1.close();
        f2.close();
        return 0;
    }
}

```

```

// Сравниваем содержимое буферов.
for(i=0; i<f1.gcount(); i++)
    if(buf1[i] != buf2[i]) {
        cout << "Файлы различны.\n";
        f1.close();
        f2.close();
        return 0;
    }

} while(!f1.eof() && !f2.eof());

cout << "Файлы одинаковы.\n";

```

При выполнении этого кода с помощью функции `read()` считывается по одному буферу из каждого файла. Затем сравнивается содержимое этих буферов. Если окажется, что буферы имеют различное содержимое, файлы будут закрыты, на экране появится сообщение "Файлы различны.", и программа завершится. В противном случае в буферы `buf1` и `buf2` для сравнения будут считываться новые порции байтов до тех пор, пока не обнаружится конец одного (или обоих) файлов. Поскольку в конце любого файла возможно считывание неполного буфера, в программе используется функция `gcount()`, которая позволяет точно определить количество символов в каждом буфере. Если окажется, что один из файлов короче другого, значения, возвращаемые функцией `gcount()` для файлов, будут различными при достижении конца одного из файлов. В этом случае на экран выводится сообщение "Файлы имеют различный размер.". Наконец, если файлы одинаковы, то при обнаружении конца одного из файлов также должен быть обнаружен конец и другого файла. Это подтверждается вызовом функции `eof()` для каждого потока. И только в том случае, если равенство файлов будет безоговорочным во всех отношениях, на экран будет выведено соответствующее сообщение.

5. Завершая программу, не забудьте закрыть файлы.

```

f1.close();
f2.close();

return 0;

```

6. Вот как выглядит полный текст программы `CompFiles.cpp`.

```
/*
```

Проект 11.1

```

Создание средства сравнения файлов.

*/
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    register int i;
    int numread;

    unsigned char buf1[1024], buf2[1024];

    if(argc!=3) {
        cout << "Применение: compfiles <file1> <file2>\n";
        return 1;
    }

    ifstream f1(argv[1], ios::in | ios::binary);
    if(!f1) {
        cout << "Не удается открыть первый файл.\n";
        return 1;
    }
    ifstream f2(argv[2], ios::in | ios::binary);
    if(!f2) {
        cout << "Не удается открыть второй файл.\n";
        return 1;
    }

    cout << "Сравнение файлов...\n";

    do {
        f1.read((char *) buf1, sizeof buf1);
        f2.read((char *) buf2, sizeof buf2);

        if(f1.gcount() != f2.gcount()) {
            cout << "Файлы имеют различный размер.\n";
        }
    }
}
```

```

        f1.close();
        f2.close();
        return 0;
    }

    // Сравниваем содержимое буферов.
    for(i=0; i<f1.gcount(); i++)
        if(buf1[i] != buf2[i]) {
            cout << "Файлы различны.\n";
            f1.close();
            f2.close();
            return 0;
        }

    } while(!f1.eof() && !f2.eof());

    cout << "Файлы одинаковы.\n";

    f1.close();
    f2.close();

    return 0;
}

```

7. Чтобы испытать программу CompFiles.cpp, сначала скопируйте файл CompFiles.cpp в другой файл, который назовите, скажем, temp.txt. Затем (после компиляции программы CompFiles.cpp) введите следующую командную строку.

```
CompFiles CompFiles.cpp temp.txt
```

При выполнении этой программы вы должны получить сообщение о том, что файлы одинаковы. Затем сравните файл CompFiles.cpp с каким-нибудь другим файлом (например, с одним из программных файлов этого модуля). В этом случае вы должны получить сообщение о том, что файлы различны.

8. Самостоятельно попробуйте усовершенствовать программу CompFiles.cpp. Например, добавьте средство, которое позволяет игнорировать регистр букв, в результате чего строчные и прописные буквы не должны различаться программой сравнения файлов. Можно также позаботиться о том, чтобы новая версия программы CompFiles.cpp сообщала позицию, в которой обнаружено несовпадение в содержимом файлов.

ВАЖНО!

## 11.11. Произвольный доступ

До сих пор мы использовали файлы, доступ к содержимому которых был организован строго последовательно, байт за байтом. Но в C++ также можно получать доступ к файлу в произвольном порядке. В этом случае необходимо использовать функции `seekg()` и `seekp()`. Вот как выглядят прототипы их самых распространенных форматов.

```
istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);
```

Используемый здесь целочисленный тип `off_type` (он определен в классе `ios`) позволяет хранить самое большое допустимое значение, которое может иметь параметр `offset`. Тип `seekdir` определен как перечисление, которое имеет следующие значения.

| Значение              | Описание        |
|-----------------------|-----------------|
| <code>ios::beg</code> | Начало файла    |
| <code>ios::cur</code> | Текущая позиция |
| <code>ios::end</code> | Конец файла     |

В C++-системе ввода-вывода предусмотрена возможность управления двумя указателями, связанными с файлом. Эти так называемые *get-* и *put-*указатели определяют, в каком месте файла должна выполниться следующая операция ввода и вывода соответственно. При каждом выполнении операции ввода или вывода соответствующий указатель автоматически изменяется. Используя функции `seekg()` и `seekp()`, можно “перемещать” нужный указатель и получать тем самым доступ к файлу в произвольном порядке.

Функция `seekg()` перемещает текущий *get-*указатель соответствующего файла на `offset` байт относительно позиции, заданной параметром `origin`. Функция `seekp()` перемещает текущий *put-*указатель соответствующего файла на `offset` байт относительно позиции, заданной параметром `origin`.

В общем случае произвольный доступ для операций ввода-вывода должен выполняться только для файлов, открытых в двоичном режиме. Преобразования символов, происходящие в текстовых файлах, могут привести к тому, что запрашиваемая позиция файла не будет соответствовать его реальному содержимому.

В следующей программе демонстрируется использование функции `seekp()`. Она позволяет задать имя файла в командной строке, а за ним — конкретный байт, который нужно в нем изменить. Программа затем записывает в указанную позицию символ “X”. Обратите внимание на то, что обрабатываемый файл должен быть открыт для выполнения операций чтения-записи.

## 566 Модуль 11. C++-система ввода-вывода

```
// Демонстрация произвольного доступа к файлу.

#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Применение: ИМЯ_ПРОГРАММЫ <имя_файла> <байт>\n";
        return 1;
    }

    fstream out(argv[1], ios::in | ios::out | ios::binary);
    if(!out) {
        cout << "Не удается открыть файл.\n";
        return 1;
    }

    // Функция seekp() устанавливает put-указатель
    // в соответствии
    // с заданным байтом:
    out.seekp(atoi(argv[2]), ios::beg);

    out.put('X');
    out.close();

    return 0;
}
```

Теперь продемонстрируем использование функции `seekg()`. При выполнении следующей программы отображается содержимое файла, начиная с заданной позиции (имя файла и позиция указываются в командной строке).

```
// Отображение содержимого файла с заданной
// стартовой позиции.
```

```
#include <iostream>
```

```
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=3) {
        cout << "Применение: ИМЯ_ПРОГРАММЫ "
            << "<имя_файла> <стартовая_позиция>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);
    if(!in) {
        cout << "Не удается открыть файл.\n";
        return 1;
    }

    in.seekg(atoi(argv[2]), ios::beg);

    while(in.get(ch))
        cout << ch;

    return 0;
}
```

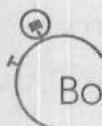
Текущую позицию каждого файлового указателя можно определить с помощью этих двух функций.

```
pos_type tellg();
pos_type tellp();
```

Здесь используется тип `pos_type` (он определен в классе `ios`), позволяющий хранить самое большое значение, которое может возвратить любая из этих функций.

Существуют перегруженные версии функций `seekg()` и `seekp()`, которые перемещают файловые указатели в позиции, заданные значениями, возвращаемыми функциями `tellg()` и `tellp()` соответственно. Вот как выглядят их прототипы.

```
istream &seekg(pos_type position);
ostream &seekp(pos_type position);
```



## Вопросы для текущего контроля

1. С помощью какой функции можно обнаружить конец файла?
2. Какие действия выполняет функция `getline()`?
3. Какие функции позволяют получить доступ к произвольно заданной позиции в файле?\*

**ВАЖНО!**

## 11.12. Получение информации об операциях ввода-вывода

C++-система ввода-вывода поддерживает информацию о результатах выполнения каждой операции ввода-вывода. Текущее состояние потока ввода-вывода описывается в объекте типа `iostate`, который представляет собой перечисление (оно определено в классе `ios`), включающее следующие члены.

| Имя                       | Значение                                                                   |
|---------------------------|----------------------------------------------------------------------------|
| <code>ios::goodbit</code> | Ошибки отсутствуют                                                         |
| <code>ios::eofbit</code>  | 1 при обнаружении конца файла; 0 в противном случае                        |
| <code>ios::failbit</code> | 1 при возникновении исправимой ошибки ввода-вывода; 0 в противном случае   |
| <code>ios::badbit</code>  | 1 при возникновении неисправимой ошибки ввода-вывода; 0 в противном случае |

Статусную информацию о результатах выполнения операций ввода-вывода можно получать двумя способами. Во-первых, можно вызвать функцию `rdstate()`, которая является членом класса `ios`. Она имеет такой прототип.

```
iosstate rdstate();
```

Функция `rdstate()` возвращает текущий статус флагов ошибок. Нетрудно догадаться, что, судя по приведенному выше списку флагов, функция `rdstate()` возвратит значение `goodbit` при отсутствии каких бы то ни было ошибок. В противном случае она возвращает соответствующий флаг ошибки.

1. Конец файла можно обнаружить с помощью функции `eof()`.
2. Функция `getline()` считывает строку текста.
3. Для обработки запросов, предполагающих доступ к произвольно заданной позиции в файле, используются функции `seekg()` и `seekp()`.

Во-вторых, о наличии ошибки можно узнать с помощью одной или нескольких следующих функций-членов класса `ios`.

```
bool bad();
bool eof();
bool fail();
bool good();
```

Функция `eof()` рассматривалась выше. Функция `bad()` возвращает значение ИСТИНА, если в результате выполнения операции ввода-вывода был установлен флаг `badbit`. Функция `fail()` возвращает значение ИСТИНА, если в результате выполнения операции ввода-вывода был установлен флаг `failbit`. Функция `good()` возвращает значение ИСТИНА, если при выполнении операции ввода-вывода ошибок не произошло. В противном случае эти функции возвращают значение ЛОЖЬ.

Если при выполнении операции ввода-вывода произошла ошибка, то, возможно, прежде чем продолжать выполнение программы, имеет смысл сбросить флаги ошибок. Для этого используйте функцию `clear()` (член класса `ios`), прототип которой выглядит так.

```
void clear(iostate flags = ios::goodbit);
```

Если параметр `flags` равен значению `goodbit` (оно устанавливается по умолчанию), все флаги ошибок очищаются. В противном случае флаги устанавливаются в соответствии с заданным вами значением.

Прежде чем переходить к следующему модулю, стоит опробовать функции, которые сообщают данные о состоянии флагов ошибок, внеся в предыдущие примеры программ код проверки ошибок.



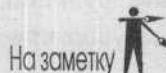
## Тест для самоконтроля по модулю 11

- Назовите четыре встроенных потока.
- Предусмотрены ли в C++ (помимо 8-разрядных) двухбайтовые (16-разрядные) версии стандартных потоков?
- Представьте общий формат перегрузки оператора вывода (вставки) данных в поток.
- Какие действия выполняет манипулятор `ios::scientific`?
- Каково назначение функции `width()`?
- В выражениях ввода-вывода можно использовать любой манипулятор ввода-вывода. Верно ли это?

7. Покажите, как открыть файл для считывания данных в текстовом режиме.
8. Покажите, как открыть файл для записи данных в текстовом режиме.
9. Каково назначение манипулятора `ios::binary`?
10. При достижении конца файла переменная потока принимает значение **ЛОЖЬ**. Верно ли это?
11. Предположим, что файл связан с входным потоком `strm`. Покажите, как считать все содержимое файла? -
12. Напишите программу копирования файла. Обеспечьте возможность для пользователя задавать входной и выходной файлы в командной строке. Убедитесь, что ваша программа может копировать как текстовые, так и двоичные файлы.
13. Напишите программу объединения двух текстовых файлов. Обеспечьте возможность для пользователя задавать имена объединяемых файлов и файл-результата в командной строке. Например, если ваша программа будет называться `merge`, то для объединения файлов `MyFile1.txt` и `MyFile2.txt` в файл `Target.txt` необходимо использовать следующую командную строку.

`Merge MyFile1.txt MyFile2.txt Target.txt`

14. Покажите, как с помощью функции `seekg()` перейти к 300-му байту в потоке `MyStrm`.



Ответы на эти вопросы можно найти на Web-странице данной книги по адресу: <http://www.osborne.com>.

# Модуль 12

## Исключения, шаблоны и кое-что еще

- 12.1. Обработка исключительных ситуаций**
- 12.2. Обобщенные функции**
- 12.3. Обобщенные классы**
- 12.4. Динамическое распределение памяти**
- 12.5. Пространства имён**
- 12.6. Статические члены класса**
- 12.7. Динамическая идентификация типов (RTTI)**
- 12.8. Операторы приведения типов**

С начала этой книги мы с вами прошли огромный путь. И в этом последнем модуле мы рассмотрим ряд таких важных тем, как обработка исключительных ситуаций, использование шаблонов, динамическое распределение памяти и пространства имен. Очевидно, что в руководстве для начинающих невозможно охватить все средства и нюансы программирования, ведь C++ — довольно сложный и объемный язык, предназначенный для создания высокоеффективных программ, которые требуют профессионального подхода. Тём не менее, освоив материал этого модуля, вы овладеете основными элементами языка C++ и сможете приступить к написанию прикладных программ.

**ВАЖНО!**

## 12.1. Обработка исключительных ситуаций

*Исключительная ситуация* (или *исключение*) — это ошибка, которая возникает во время выполнения программы. Используя C++-подсистему обработки исключительных ситуаций, с такими ошибками вполне можно справляться. При их возникновении во время работы программы автоматически вызывается так называемый *обработчик исключений*. В этом-то и состоит принципиальное преимущество системы обработки исключений, поскольку именно она “отвечает” за код обработки ошибок, который прежде приходилось “вручную” вводить в и без того объемные программы.

### Основы обработки исключительных ситуаций

Управление C++-механизмом обработки исключений зиждется на трех ключевых словах: `try`, `catch` и `throw`. В самых общих чертах их работа состоит в следующем. Программные инструкции, которые вы считаете нужным проконтролировать на предмет исключений, помещаются в `try`-блок. Если исключение (т.е. ошибка) таки возникает в этом блоке, оно дает знать о себе *выбросом* определенного рода информации (с помощью ключевого слова `throw`). Это *выброшенное исключение* может быть перехвачено программным путем с помощью `catch`-блока и обработано соответствующим образом. А теперь подробнее.

Код, в котором возможно возникновение исключительных ситуаций, должен выполняться в рамках `try`-блока. (Любая функция, вызываемая из этого `try`-блока, также подвергается контролю.) Исключения, которые могут быть “выброшены” контролируемым кодом, перехватываются `catch`-инструкцией, непосредственно следующей за `try`-блоком, в котором фиксируются эти “выбросы” исключений. Общий формат `try`- и `catch`-блоков выглядит так.

```

try {
    // try-блок (блок кода, подлежащий проверке
    // на наличие ошибок)
}
catch (type1 arg) {
    // catch-блок (обработчик исключения типа type1)
}
catch (type2 arg) {
    // catch-блок (обработчик исключения типа type2)
}
catch (type3 arg) {
    // catch-блок (обработчик исключения типа type3)
}
// ...
catch (typeN arg) {
    // catch-блок (обработчик исключения типа typeN)
}

```

Блок `try` должен содержать код, который, по вашему мнению, следует проверять на предмет возникновения ошибок. Этот блок может включать лишь несколько инструкций некоторой функции либо охватывать весь код функции `main()` (в этом случае, по сути, “под колпаком” системы обработки исключений будет находиться вся программа).

После “выброса” исключение перехватывается соответствующей инструкцией `catch`, которая выполняет его обработку. С одним `try`-блоком может быть связана не одна, а несколько `catch`-инструкций. Какая именно из них будет выполнена, определяется типом исключения. Другими словами, будет выполнена та `catch`-инструкция, тип исключения которой (т.е. тип данных, заданный в `catch`-инструкции) совпадает с типом сгенерированного исключения (а все остальные будут проигнорированы). После перехвата исключения параметр `arg` примет его значение. Таким путем могут перехватываться данные любого типа, включая объекты классов, созданных программистом.

Общий формат инструкции `throw` выглядит так:

```
throw exception;
```

Здесь с помощью элемента `exception` задается исключение, сгенерированное инструкцией `throw`. Если это исключение подлежит перехвату, то инструкция `throw` должна быть выполнена либо в самом блоке `try`, либо в любой вызываемой из него функции (т.е. прямо или косвенно).

Если в программе обеспечивается “выброс” исключения, для которого не предусмотрена соответствующая `catch`-инструкция, произойдет аварийное завершение программы. Поэтому программист должен предусмотреть перехват всех возможных исключений в программе.

Рассмотрим простой пример обработки исключений средствами языка C++.

```
// Простой пример обработки исключений.
```

```
#include <iostream>
using namespace std;

int main()
{
    cout << "НАЧАЛО\n";

    try { // начало try-блока
        cout << "В try-блоке\n";
        throw 99; // генерирование ошибки ("выброс" исключения)
        cout << "Эта инструкция не будет выполнена.";
    }
    catch (int i) { // перехват ошибки
        cout << "Перехват исключения. Его значение равно: ";
        cout << i << "\n";
    }

    cout << "КОНЕЦ";

    return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
НАЧАЛО
В try-блоке
Перехват исключения. Его значение равно: 99
КОНЕЦ
```

Рассмотрим внимательно код этой программы. Как видите, здесь `try`-блок содержит три инструкции, а инструкция `catch (int i)` предназначена для обработки исключения целочисленного типа. В этом `try`-блоке выполняются только две из трех инструкций: `cout` и `throw`. После генерирования исключения управление передается `catch`-выражению, при этом выполнение `try`-блока прекра-

щается. Необходимо понимать, что `catch`-инструкция *не* вызывается, а просто с нее продолжается выполнение программы после "выброса" исключения. (Стек программы автоматически настраивается в соответствии с создавшейся ситуацией.) Поэтому `cout`-инструкция, следующая после `throw`-инструкции, никогда не выполнится.

Обычно при выполнении `catch`-блока делается попытка исправить ошибку путем выполнения соответствующих действий. Если ошибку можно исправить, то после выполнения `catch`-блока управление программой передается инструкции, следующей за этим блоком. В противном случае программа должна быть прекращена.

Как упоминалось выше, тип исключения должен совпадать с типом, заданным в `catch`-инструкции. Например, если в предыдущей программе тип `int`, указанный в `catch`-выражении, заменить типом `double`, то исключение перехвачено не будет, и произойдет аварийное завершение программы. Вот как выглядят последствия внесения такого изменения.

```
// Этот пример работать не будет.

#include <iostream>
using namespace std;

int main()
{
    cout << "НАЧАЛО\n";

    try { // начало try-блока
        cout << "В try-блоке\n";
        throw 99; // генерирование ошибки
        cout << "Эта инструкция не будет выполнена.";
    }
    catch (double i) { // Перехват исключения типа int
        // не состоится.
        cout << "Перехват исключения. Его значение равно: ";
        cout << i << "\n";
    }
    cout << "КОНЕЦ";

    return 0;
}
```

Такие результаты выполнения этой программы объясняются тем, что исключение целочисленного типа не перехватывается инструкцией `catch (double i)`.

НАЧАЛО

В try-блоке

Abnormal program termination

Последнее сообщение об аварийном завершении программы (`Abnormal program termination`) может отличаться от приведенного в результатах выполнения предыдущего примера. Это зависит от используемого вами компилятора.

Исключение, сгенерированное функцией, вызванной из try-блока, может быть обработано этим же try-блоком. Рассмотрим, например, следующую вполне корректную программу.

```
/* Генерирование исключения из функции, вызываемой
   из try-блока.
*/
#include <iostream>
using namespace std;

void Xtest(int test)
{
    cout << "В функции Xtest(), значение test равно: "
        << test << "\n";
    if(test) throw test; // Это исключение перехватывается
                          // catch-инструкцией в функции
main().
}

int main()
{
    cout << "НАЧАЛО\n";

    try { // начало try-блока
        cout << "В try-блоке\n";
        Xtest(0); // Поскольку Xtest() вызывается из try-блока,
        Xtest(1); // ее код также подвергается контролю на
        Xtest(2); // ошибки.
    }
    catch (int i) { // перехват ошибки
        cout << "Перехват исключения. Его значение равно: ";
    }
}
```

```

    cout << i << "\n";
}

cout << "КОНЕЦ";

return 0;
}

```

Эта программа генерирует такие результаты.

НАЧАЛО

В try-блоке

В функции Xtest(), значение test равно: 0

В функции Xtest(), значение test равно: 1

Перехват исключения. Его значение равно: 1

КОНЕЦ

Как подтверждают результаты выполнения этой программы, исключение, сгенерированное в функции Xtest(), было перехвачено обработчиком в функции main().

Блок try может быть локализован в рамках функции. В этом случае при каждом ее выполнении запускается и обработка исключений, связанная с этой функцией. Рассмотрим следующую простую программу.

```
// Блок try может быть локализован в рамках функции.
```

```

#include <iostream>
using namespace std;

// Функционирование блоков try/catch возобновляется
// при каждом входе в функцию.
void Xhandler(int test)
{
    try{ // Этот try-блок локален по отношению
        // к функции Xhandler().
        if(test) throw test;
    }
    catch(int i) {
        cout << "Перехват! Исключение №: " << i << '\n';
    }
}

int main()

```

## 578 Модуль 12. Исключения, шаблоны и кое-что еще

```
{  
    cout << "НАЧАЛО\n";  
  
    Xhandler(1);  
    Xhandler(2);  
    Xhandler(0);  
    Xhandler(3);  
  
    cout << "КОНЕЦ";  
  
    return 0;  
}
```

При выполнении этой программы отображаются такие результаты.

НАЧАЛО

Перехват! Исключение №: 1

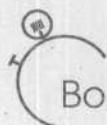
Перехват! Исключение №: 2

Перехват! Исключение №: 3

КОНЕЦ

Как видите, программа сгенерировала три исключения. После каждого исключения функция `Xhandler()` передавала управление в функцию `main()`. Когда она снова вызывалась, возобновлялась и обработка исключения.

В общем случае `try`-блок возобновляет свое функционирование при каждом входе в него. Поэтому если `try`-блок является частью цикла, то он будет запускаться при каждом повторении этого цикла.



### Вопросы для текущего контроля

1. Что понимается под исключением в языке C++?
2. На каких трех ключевых словах основана обработка исключений?
3. При перехвате исключения важен его тип. Верно ли это?\*

- 
1. Исключение — это ошибка, которая возникает во время выполнения программы.
  2. Обработка исключений основана на трех ключевых словах: `try`, `catch` и `throw`.
  3. Верно. Исключение будет перехвачено только в том случае, если его тип совпадет с типом, заданным в `catch`-инструкции.

## Использование нескольких catch-инструкций

Как упоминалось выше, с try-блоком можно связывать не одну, а несколько catch-инструкций. В действительности именно такая практика и является обычной. Но при этом все catch-инструкции должны перехватывать исключения различных типов. Например, в приведенной ниже программе обеспечивается перехват как целых чисел, так и указателей на символы.

```
// Использование нескольких catch-инструкций.

#include <iostream>
using namespace std;

// Здесь возможен перехват исключений различных типов.
void Xhandler(int test)
{
    try {
        if(test) throw test; // "Выброс" исключений типа int.
        else throw "Значение равно нулю."; // "Выброс"
  // исключений типа char *.
    }
    catch(int i) { // Перехват исключений типа int.
        cout << "Перехват! Исключение №: " << i << '\n';
    }
    catch(char *str) { // Перехват исключений типа char *.
        cout << "Перехват строки: ";
        cout << str << '\n';
    }
}

int main()
{
    cout << "НАЧАЛО\n";

    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);

    cout << "КОНЕЦ";
```

```
    return 0;  
}
```

Эта программа генерирует такие результаты.

НАЧАЛО

Перехват! Исключение №: 1

Перехват! Исключение №: 2

Перехват строки: Значение равно нулю.

Перехват! Исключение №: 3

КОНЕЦ

Как видите, каждая `catch`-инструкция отвечает только за исключение "своего" типа.

В общем случае `catch`-выражения проверяются в порядке следования, и выполняется только тот `catch`-блок, в котором тип заданного исключения совпадает с типом сгенерированного исключения. Все остальные `catch`-блоки игнорируются.

### Перехват исключений базового класса

Важно понимать, как выполняются `catch`-инструкции, связанные с производными классами. Дело в том, что `catch`-выражение для базового класса "отреагирует совпадением" на исключение любого производного типа (т.е. типа, выведенного из этого базового класса). Следовательно, если нужно отдельно перехватывать исключения как базового, так и производного типов, в `catch`-последовательности `catch`-инструкцию для производного типа необходимо поместить перед `catch`-инструкцией для базового типа. В противном случае `catch`-выражение для базового класса будет перехватывать (помимо "своих") и исключения всех производных классов. Рассмотрим, например, следующую программу:

```
// Перехват исключений производных типов.  
// Эта программа не корректна!
```

```
#include <iostream>  
using namespace std;  
  
class B {  
};  
  
class D: public B {
```

```

};

int main()
{
    D derived;

    try {
        throw derived;
    }
    catch(B b) { // Эти catch-инструкции расположены не в
                  // корректном порядке! Исключения
                  // производных типов нужно перехватывать
                  // до исключений базовых типов.
        cout << "Перехват исключения базового класса.\n";
    }
    catch(D d) {
        cout << "Этот перехват никогда не произойдет.\n";
    }
}

return 0;
}

```

Поскольку здесь объект `derived` – это объект класса `D`, который выведен из базового класса `B`, то исключение типа `derived` будет всегда перехватываться первым `catch`-выражением; вторая же `catch`-инструкция при этом никогда не выполнится. Одни компиляторы отреагируют на такое положение вещей предупреждающим сообщением. Другие могут выдать сообщение об ошибке. В любом случае, чтобы исправить ситуацию, достаточно поменять порядок следования этих `catch`-инструкций на противоположный.

## Перехват всех исключений

Иногда имеет смысл создать обработчик для перехвата всех исключений, а не исключений только определенного типа. Для этого достаточно использовать такой формат `catch`-блока.

```

catch(...) {
    // Обработка всех исключений.
}

```

Здесь заключенное в круглые скобки многоточие обеспечивает совпадение с любым типом данных.

Использование формата `catch(...)` иллюстрируется в следующей программе.

// В этой программе перехватываются исключения всех типов.

```
#include <iostream>
using namespace std;

void Xhandler(int test)
{
    try{
        if(test==0) throw test; // генерирует int-исключение
        if(test==1) throw 'a'; // генерирует char-исключение
        if(test==2) throw 123.23; // генерирует
                                // double-исключение
    }
    catch(...) { // перехват всех исключений
        cout << "Перехват!\n";
    }
}

int main()
{
    cout << "НАЧАЛО\n";

    Xhandler(0);
    Xhandler(1);
    Xhandler(2);

    cout << "КОНЕЦ";

    return 0;
}
```

Эта программа генерирует такие результаты.

НАЧАЛО

Перехват!

Перехват!

Перехват!

КОНЕЦ

Как видите, функция `Xhandler()` генерирует исключения трех типов: `int`, `char` и `double`. И все они перехватываются с помощью одной-единственной инструкции `catch(...)`.

Зачастую имеет смысл использовать инструкцию `catch(...)` в качестве последнего “рубежа” `catch`-последовательности. В этом случае она обеспечивает перехват исключений “всех остальных” типов (т.е. не предусмотренных предыдущими `catch`-выражениями). Это — удобный способ перехватить все исключения, которые вам не хочется обрабатывать в явном виде. Кроме того, перехватывая абсолютно все исключения, вы предотвращаете возможность аварийного завершения программы, которое может быть вызвано каким-то непредусмотренным (а значит, необработанным) исключением.

## Определение исключений, генерируемых функциями

Программист может определить тип “внешних” исключений, генерируемых функцией. Можно также оградить функцию от генерирования каких бы то ни было исключений вообще. Для формирования этих ограничений необходимо внести в определение функции `throw`-выражение. Общий формат определения функции с использованием `throw`-выражения выглядит так.

```
тип имя_функции(список_аргументов) throw(список_имен_типов)
{
    // ...
}
```

Здесь элемент `список_имен_типов` должен включать только те имена типов данных, которые разрешается генерировать функции (элементы списка разделяются запятыми). Генерирование исключения любого другого типа приведет к аварийному окончанию программы. Если нужно, чтобы функция вообще не могла генерировать исключения, используйте в качестве этого элемента пустой список.



На момент написания этой книги среда Visual C++ не обеспечивала для функции запрет генерировать исключения, тип которых не задан в `throw`-выражении. Это говорит о нестандартном поведении данной среды. Тем не менее вы все равно можете задавать “ограничивающее” `throw`-выражение, но оно в этом случае будет играть лишь “уведомительную” роль.

На примере следующей программы показано, как можно указать типы исключений, которые способна генерировать функция.

```
// Ограничение типов исключений, генерируемых функцией.

#include <iostream>
using namespace std;

// Эта функция может генерировать
// исключения только типа int, char и double.
void Xhandler(int test) throw(int, char, double) // Здесь
    // указаны типы исключений, которые
    // могут быть сгенерированы функцией
    // Xhandler().
{
    if(test==0) throw test; // генерирует int-исключение
    if(test==1) throw 'a'; // генерирует char-исключение
    if(test==2) throw 123.23; // генерирует
        // double-исключение
}

int main()
{
    cout << "НАЧАЛО\n";

    try{
        Xhandler(0); // Попробуйте также передать
            // функции Xhandler() аргументы 1 и 2.
    }
    catch(int i) {
        cout << "Перехват int-исключения.\n";
    }
    catch(char c) {
        cout << "Перехват char-исключения.\n";
    }
    catch(double d) {
        cout << "Перехват double-исключения.\n";
    }

    cout << "КОНЕЦ";

    return 0;
}
```

В этой программе функция `Xhandler()` может генерировать исключения только типа `int`, `char` и `double`. При попытке сгенерировать исключение любого другого типа произойдет аварийное завершение программы. Чтобы убедиться в этом, удалите из `throw`-списка, например, тип `int` и перезапустите программу. (Как упоминалось выше, среда Visual C++ пока не ограничивает исключения, которые может генерировать функция.)

Важно понимать, что диапазон исключений, разрешенных для генерирования функции, можно ограничивать только типами, генерируемыми ею в `try`-блоке, из которого она была вызвана. Другими словами, любой `try`-блок, расположенный в теле самой функции, может генерировать исключения любого типа, если они перехватываются в теле той же функции. Ограничение применяется только для ситуаций, когда “выброс” исключений происходит за пределами функции.

## Повторное генерирование исключений

Используя `throw`-инструкцию без указания типа исключения, можно повторно сгенерировать исключение в его обработчике. В этом случае текущее исключение будет передано во внешнюю `try/catch`-последовательность. Чаще всего причиной для такого выполнения инструкции `throw` служит стремление позволить доступ к одному исключению нескольким обработчикам. Например, первый обработчик будет сообщать об одном аспекте исключения, а второй — о другом. Исключение можно повторно сгенерировать только в `catch`-блоке (или в любой функции, вызываемой из этого блока). При повторном генерировании исключение не будет перехватываться той же `catch`-инструкцией. Оно распространится на следующий `catch`-блок. Повторное генерирование исключения демонстрируется в следующей программе (в данном случае повторно генерируется исключение типа `char *`).

```
// Пример “повторного генерирования” исключения.

#include <iostream>
using namespace std;

void Xhandler()
{
    try {
        throw "Привет"; // генерирует исключение типа char *
    }
    catch(char *) { // перехватывает исключение типа char *
        cout << "Перехват исключения в функции Xhandler.\n";
        throw ; // Повторное генерирование исключения
    }
}
```

## 586 Модуль 12. Исключения, шаблоны и кое-что еще

```
// типа char *, которое будет перехвачено
// вне функции Xhandler.
}

}

int main()
{
    cout << "НАЧАЛО\n";
    try{
        Xhandler();
    }
    catch(char *) {
        cout << "Перехват исключения в функции main().\n";
    }
    cout << "КОНЕЦ";

    return 0;
}
```

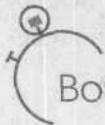
При выполнении эта программа генерирует такие результаты.

НАЧАЛО

Перехват исключения в функции Xhandler.

Перехват исключения в функции main().

КОНЕЦ



### Вопросы для текущего контроля

1. Покажите, как обеспечить перехват всех исключений.
2. Как указать тип исключений, которые могут быть сгенерированы за пределами тела функции?
3. Как можно повторно сгенерировать исключение?\*

1. Чтобы обеспечить перехват всех исключений, достаточно использовать формат `catch(...)`.
2. Чтобы указать тип исключений, которые могут быть сгенерированы за пределами тела функции, необходимо включить в ее определение `throw`-выражение.
3. Чтобы повторно сгенерировать исключение, достаточно использовать ключевое слово `throw` без значения.

## Спросим у опытного программиста

**Вопрос.** *Из всего вышесказанного следует, что функция может сообщить об ошибке двумя способами: путем генерирования исключения или возврата кода ошибки. В каких случаях следует использовать эти способы?*

**Ответ.** В настоящее время профессионалы рекомендуют использовать исключения, а не коды ошибок. В языках Java и C# о распространенных ошибках (например, возникающих при открытии файла или переполнении) сообщается только с помощью исключений. Поскольку C++ произошел от С, для отчета об ошибках в нем предусмотрена возможность использования как кодов ошибок, так и исключений. Поэтому индикация многих ошибочных ситуаций, связанных с библиотечными функциями, обеспечивается за счет кодов ошибок. Но в новых программах согласно современным тенденциям в программировании вам следует для этих целей использовать исключения.

## Шаблоны

Шаблон — это одно из самых сложных и мощных средств в C++. Он не вошел в исходную спецификацию C++, и лишь несколько лет назад стал неотъемлемой частью программирования на C++. Шаблоны позволяют достичь одну из самых трудных целей в программировании — создать многократно используемый код.

Используя шаблоны, можно создавать обобщенные функции и классы. В обобщенной функции (или классе) обрабатываемый ею (им) тип данных задается как параметр. Таким образом, одну функцию или класс можно использовать для разных типов данных, не предоставляя явным образом конкретные версии для каждого типа данных.

**ВАЖНО!**

### 12.2. Обобщенные функции

Обобщенная функция определяет общий набор операций, которые предназначены для применения к данным различных типов. Тип данных, обрабатываемых функцией, передается ей как параметр. Используя обобщенную функцию, к широкому диапазону данных можно применить единую общую процедуру. Возможно, вам известно, что многие алгоритмы имеют одинаковую логику для разных типов данных. Например, один и тот же алгоритм сортировки Quicksort применяется и к массиву целых чисел, и к массиву значений с плавающей точкой. Различие здесь состоит только в типе сортируемых данных. Создавая обобщенную

функцию, можно определить природу алгоритма независимо от типа данных. После этого компилятор автоматически сгенерирует корректный код для типа данных, который в действительности используется при выполнении этой функции. По сути, создавая обобщенную функцию, вы создаете функцию, которая автоматически перегружает себя саму.

Обобщенная функция создается с помощью ключевого слова `template`. Обычное значение слова “`template`” точно отражает цель его применения в C++. Это ключевое слово используется для создания шаблона, который описывает действия, выполняемые функцией. Компилятору же остается “дополнить недостающие детали” в соответствии с заданным значением параметра. Общий формат определения шаблонной функции имеет следующий вид.

```
template <class Ttype> тип имя_функции(список_параметров)
{
    // тело функции
}
```

Здесь элемент `Ttype` представляет собой “заполнитель” для типа данных, обрабатываемых функцией. Это имя может быть использовано в теле функции. Но оно означает всего лишь “заглушку”, вместо которой компилятор автоматически подставит реальный тип данных при создании конкретной версии функции. И хотя для задания обобщенного типа в `template`-объявлении по традиции применяется ключевое слово `class`, можно также использовать ключевое слово `typename`.

В следующем примере создается обобщенная функция, которая меняет местами значения двух переменных, используемых при ее вызове. Поскольку общий процесс обмена значениями переменных не зависит от их типа, он является прекрасным кандидатом для создания обобщенной функции.

```
// Пример шаблонной функции.
```

```
#include <iostream>
using namespace std;

// Определение шаблонной функции, которая меняет местами
// значения своих аргументов.
template <class X> void swapargs(X &a, X &b) // X - это
  // обобщенный тип данных.
{
    X temp;
```

```

temp = a;
a = b;
b = temp;
}

int main()
{
    int i=10, j=20;
    double x=10.1, y=23.3;
    char a='x', b='z';

    cout << "Исходные значения i, j: " << i << ' '
        << j << '\n';
    cout << "Исходные значения x, y: " << x << ' '
        << y << '\n';
    cout << "Исходные значения a, b: " << a << ' '
        << b << '\n';

    // Компилятор автоматически создает версии функции
    // swapargs(), которые используют различные типы данных,
    // заданные ее аргументами.
    swapargs(i, j); // перестановка целых чисел
    swapargs(x, y); // перестановка значений
                     // с плавающей точкой
    swapargs(a, b); // перестановка символов

    cout << "После перестановки i, j: " << i << ' '
        << j << '\n';
    cout << "После перестановки x, y: " << x << ' '
        << y << '\n';
    cout << "После перестановки a, b: " << a << ' '
        << b << '\n';

    return 0;
}

```

Вот как выглядят результаты выполнения этой программы.

Исходные значения i, j: 10 20

Исходные значения x, y: 10.1 23.3

Исходные значения a, b: x z

После перестановки i, j: 20 10

После перестановки x, y: 23.3 10.1

После перестановки a, b: z x

Итак, рассмотрим внимательно код программы. Стока

```
template <class X> void swapargs(X &a, X &b)
```

сообщает компилятору, во-первых, о том, что создается шаблон, и, во-вторых, что здесь начинается обобщенное определение. Обозначение X представляет собой обобщенный тип, который используется в качестве "заполнителя". За template-заголовком следует объявление функции swapargs(), в котором символ X означает тип данных для значений, которые будут меняться местами. В функции main() демонстрируется вызов функции swapargs() с использованием трех различных типов данных: int, float и char. Поскольку функция swapargs() является обобщенной, компилятор автоматически создает три версии функции swapargs(): одну для обмена целых чисел, вторую для обмена значений с плавающей точкой и третью для обмена символов. Таким образом, одну и ту же обобщенную функцию swapargs() можно использовать для обмена аргументов любого типа данных.

Здесь необходимо уточнить некоторые важные термины, связанные с шаблонами. Во-первых, обобщенная функция (т.е. функция, объявление которой предваряется template-инструкцией) также называется *шаблонной функцией*. Оба термина используются в этой книге взаимозаменяюще. Когда компилятор создает конкретную версию этой функции, то говорят, что создается ее *специализация* (или *конкретизация*). Специализация также называется *порождаемой функцией* (generated function). Действие порождения функции определяют как ее *реализацию* (instantiating). Другими словами, порождаемая функция является конкретным экземпляром шаблонной функции.

## ФУНКЦИЯ С ДВУМЯ ОБОБЩЕННЫМИ ТИПАМИ

В template-инструкции можно определить несколько обобщенных типов данных, используя список элементов, разделенных запятой. Например, в следующей программе создается шаблонная функция с двумя обобщенными типами.

```
#include <iostream>
using namespace std;
```

```
template <class Type1, class Type2> // Два обобщенных типа.
void myfunc(Type1 x, Type2 y)
{
```

```

cout << x << ' ' << y << '\n';
}

int main()
{
    myfunc(10, "Привет");

    myfunc(0.23, 10L);

    return 0;
}

```

В этом примере при выполнении функции `main()`, когда компилятор генерирует конкретные экземпляры функции `myfunc()`, заполнители типов `type1` и `type2` заменяются сначала парой типов данных `int` и `char *`, а затем парой `double` и `long` соответственно.

## Явно заданная перегрузка обобщенной функции

Несмотря на то что обобщенная функция сама перегружается по мере необходимости, это можно делать и явным образом. Формально этот процесс называется *явной специализацией*. При перегрузке обобщенная функция переопределяется “в пользу” этой конкретной версии. Рассмотрим, например, следующую программу, которая представляет собой переработанную версию приведенного выше примера программы обмена значений аргументов.

```

// Специализация шаблонной функции.
#include <iostream>
using namespace std;

template <class X> void swapargs(X &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
    cout << "Выполняется шаблонная функция swapargs().\n";
}

// Эта функция явно переопределяет обобщенную версию функции

```

```

// swapargs() для int-параметров.
void swapargs(int &a, int &b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
    cout << "Это int-специализация функции swapargs().\n";
}

int main()
{
    int i=10, j=20;
    double x=10.1, y=23.3;
    char a='x', b='z';

    cout << "Исходные значения i, j: " << i << ' '
                                  << j << '\n';
    cout << "Исходные значения x, y: " << x << ' '
                                  << y << '\n';
    cout << "Исходные значения a, b: " << a << ' '
                                  << b << '\n';

    swapargs(i, j); // Вызывается явно перегруженная
                     // функция swapargs().
    swapargs(x, y); // Вызывается обобщенная функция
                     // swapargs().
    swapargs(a, b); // Вызывается обобщенная функция
                     // swapargs().

    cout << "После перестановки i, j: " << i << ' '
                                  << j << '\n';
    cout << "После перестановки x, y: " << x << ' '
                                  << y << '\n';
    cout << "После перестановки a, b: " << a << ' '
                                  << b << '\n';

    return 0;
}

```

При выполнении эта программа генерирует такие результаты.

Исходные значения i, j: 10 20

Исходные значения x, y: 10.1 23.3

Исходные значения a, b: x z

Это int-специализация функции swapargs().

Выполняется шаблонная функция swapargs().

Выполняется шаблонная функция swapargs().

После перестановки i, j: 20 10

После перестановки x, y: 23.3 10.1

После перестановки a, b: z x

Как отмечено в комментариях, при вызове функции swapargs(i, j) выполняется явно перегруженная версия функции swapargs(), определенная в программе. Компилятор в этом случае не генерирует эту версию обобщенной функции swapargs(), поскольку обобщенная функция переопределяется явно заданным вариантом перегруженной функции.

Для обозначения явной специализации функции можно использовать новый альтернативный синтаксис, содержащий ключевое слово `template`. Например, если задать специализацию с использованием этого альтернативного синтаксиса, перегруженная версия функции swapargs() из предыдущей программы будет выглядеть так.

```
// Использование нового синтаксиса задания специализации.
template<> void swapargs<int>(int &a, int &b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
    cout << "Это int-специализация функции swapargs.\n";
}
```

Как видите, в новом синтаксисе для обозначения специализации используется конструкция `template<>`. Тип данных, для которых создается эта специализация, указывается в угловых скобках после имени функции. Для задания любого типа обобщенной функции используется один и тот же синтаксис. На данный момент ни один из синтаксических способов задания специализации не имеет никаких преимуществ перед другим, но с точки зрения перспективы развития языка, возможно, все же лучше использовать новый стиль.

Явная специализация шаблона позволяет спроектировать версию обобщенной функции в расчете на некоторую уникальную ситуацию, чтобы, возможно, воспользоваться преимуществами повышенного быстродействия программы только для одного типа данных. Но, как правило, если вам нужно иметь различные версии функции для разных типов данных, лучше использовать перегруженные функции, а не шаблоны.

ВАЖНО!

## 12.3. Обобщенные классы

Помимо обобщенных функций, можно также определить обобщенный класс. Для этого создается класс, в котором определяются все используемые им алгоритмы; при этом реальный тип обрабатываемых в нем данных будет задан как параметр при создании объектов этого класса.

Обобщенные классы особенно полезны в случае, когда в них используется логика, которую можно обобщить. Например, алгоритмы, которые поддерживают функционирование очереди целочисленных значений, также подходят и для очереди символов. Механизм, который обеспечивает поддержку связного списка почтовых адресов, также годится для поддержки связного списка, предназначенного для хранения данных о запчастях к автомобилям. После создания обобщенный класс сможет выполнять определенную программистом операцию (например, поддержку очереди или связного списка) для любого типа данных. Компилятор автоматически генерирует корректный тип объекта на основе типа, заданного при создании объекта.

Общий формат объявления обобщенного класса имеет следующий вид:

```
template <class Ttype> class имя_класса {
    // тело класса
}
```

Здесь элемент *Ttype* представляет собой “заполнитель” для имени типа, который будет задан при реализации класса. При необходимости можно определить несколько обобщенных типов данных, используя список элементов, разделенных запятыми.

Сформировав обобщенный класс, можно создать его конкретный экземпляр, используя следующий общий формат.

```
имя_класса <тип> имя_объекта;
```

Здесь элемент *тип* означает имя типа данных, которые будут обрабатываться экземпляром обобщенного класса. Функции-члены обобщенного класса автомати-

чески являются обобщенными. Поэтому вам не нужно использовать ключевое слово `template` для явного определения их таковыми.

Рассмотрим простой пример обобщенного класса.

// Пример простого обобщенного класса.

```
#include <iostream>
using namespace std;

template <class T> class MyClass { // Объявление обобщенного
                                    // класса. Здесь
                                    // T - обобщенный тип.

    T x, y;
public:
    MyClass(T a, T b) {
        x = a;
        y = b;
    }
    T div() { return x/y; }
};

int main()
{
    // Создаем версию класса MyClass для double-значений.
    MyClass<double> d_ob(10.0, 3.0); // Создание конкретного
                                       // экземпляра обобщенного класса.
    cout << "Результат деления double-значений: "
        << d_ob.div() << "\n";

    // Создаем версию класса MyClass для int-значений.
    MyClass<int> i_ob(10, 3);
    cout << "Результат деления int-значений: "
        << i_ob.div() << "\n";

    return 0;
}
```

Результаты выполнения этой программы таковы.

Результат деления double-значений: 3.33333

Результат деления int-значений: 3

Как видно по результатам, объект типа `double` выполнил деление чисел с плавающей точкой, а объект типа `int` — целочисленное деление.

После объявления конкретного экземпляра класса `MyClass` компилятор автоматически генерирует все версии функции `div()` и переменные `x` и `y`, необходимые для обработки реальных данных. В данном примере объявляются два объекта разного типа. Первый, `d_obj`, обрабатывает данные типа `double`. Это означает, что переменные `x` и `y`, а также результат деления (как и значение, возвращаемое функцией `div()`) имеют тип `double`. Второй объект, `i_obj`, обрабатывает целочисленные данные. Таким образом, переменные `x` и `y`, а также значение, возвращаемое функцией `div()`, имеют тип `int`. Обратите особое внимание на эти объявления:

```
MyClass<double> d_obj(10.0, 3.0);
MyClass<int> i_obj(10, 3);
```

Заметьте, как указывается нужный тип данных: он заключается в угловые скобки. Изменяя тип данных при создании объектов класса `MyClass`, можно изменить тип данных, обрабатываемых этим классом.

Шаблонный класс может иметь несколько обобщенных типов данных. Для этого достаточно объявить все нужные типы данных в `template`-спецификации в виде элементов списка, разделяемых запятыми. Например, в следующей программе создается класс, который использует два обобщенных типа данных.

```
/* Здесь используется два обобщенных типа данных
   в определении класса.
*/
#include <iostream>
using namespace std;

template <class T1, class T2> class MyClass
{
    T1 i;
    T2 j;
public:
    myclass(T1 a, T2 b) { i = a; j = b; }
    void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
    MyClass<int, double> ob1(10, 0.23);
```

```

 MyClass <char, char *> ob2('X', "Это тест.");
ob1.show(); // отображение int- и double-значений
ob2.show(); // отображение значений типа char и char *
return 0;
}

```

Эта программа генерирует такие результаты.

10 0.23

X Это тест.

В данной программе объявляется два вида объектов. Объект `ob1` использует данные типа `int` и `double`, а объект `ob2` — символ и указатель на символ. Для этих ситуаций компилятор автоматически генерирует данные и функции, соответствующие конкретному способу создания объектов.

## Явно задаваемые специализации классов

Подобно шаблонным функциям можно создавать и специализации обобщенных классов. Для этого используется конструкция `template<>`, которая работает по аналогии с явно задаваемыми специализациями функций. Рассмотрим пример.

// Демонстрация специализации класса.

```

#include <iostream>
using namespace std;

template <class T> class MyClass {
    T x;
public:
    MyClass(T a) {
        cout << "В теле обобщенного класса MyClass.\n";
        x = a;
    }
    T getx() { return x; }
};

// Явная специализация класса MyClass для типа int.
template <> class MyClass <int> {

```

```

int x;
public:
    MyClass(int a) {
        cout << "В теле специализации MyClass<int>.\n";
        x = a * a;
    }
    int getx() { return x; }
};

int main()
{
    MyClass<double> d(10.1);
    cout << "double: " << d.getx() << "\n\n";

    MyClass <int> i(5); // Здесь используется явная
                        // специализация класса MyClass.
    cout << "int: " << i.getx() << "\n";

    return 0;
}

```

При выполнении данной программы отображает такие результаты.

В теле обобщенного класса MyClass.

double: 10.1

В теле специализации MyClass<int>.

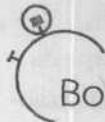
int: 25

В этой программе обратите особое внимание на следующую строку.

```
template <> class MyClass<int> {
```

Она уведомляет компилятор о том, что создается явная int-специализация класса MyClass. Тот же синтаксис используется и для любого другого типа специализации класса.

Явная специализация классов расширяет диапазон применения обобщенных классов, поскольку она позволяет легко обрабатывать одну или две специальные ситуации, оставляя все остальные варианты для автоматической обработки компилятором. Но если вы заметите, что у вас создается слишком много специализаций, то тогда, возможно, лучше вообще отказаться от создания шаблонного класса.



## Вопросы для текущего контроля

- Какое ключевое слово используется для объявления обобщенной функции или класса?
- Можно ли явным образом перегружать обобщенную функцию?
- Будут ли в обобщенном классе все его функции-члены обобщенными автоматически?\*

Исключения, шаблоны и кое-что еще

## Проект 12.1. Создание обобщенного класса очереди

GenericQ.cpp

В проекте 8.2 мы создали класс Queue, предназначенный для организации очереди символов. В этом проекте мы преобразуем его в обобщенный класс. Класс Queue — прекрасный кандидат для преобразования в обобщенный класс, поскольку его логика не зависит от типа данных, которыми он оперирует. Другими словами, один и тот же механизм в этом случае можно использовать для организации очереди переменных любого типа (например, целых чисел, значений с плавающей точкой или даже объектов классов, создаваемых программистом). Определив обобщенный класс Queue, мы сможем затем использовать его везде, где нам нужно организовать очередь объектов.

Проект  
12.1

### Последовательность действий

- Скопируйте класс Queue из проекта 8.2 в файл с именем GenericQ.cpp.
- Преобразуйте прежнее объявление класса Queue в шаблонное.

```
template <class QType> class Queue {
```

Здесь для обозначения обобщенного типа данных используется идентификатор QType.

Создание обобщенного класса очереди

- Для объявления обобщенной функции или класса используется ключевое слово template.
- Да, обобщенную функцию можно явным образом перегружать.
- Да, в обобщенном классе все его функции-члены автоматически будут обобщенными.

3. Замените прежний тип данных массива `q` идентификатором `QType`.

```
QType q[maxQsize]; // Этот массив будет содержать очередь.  
Поскольку массив q теперь имеет обобщенный тип, его можно использовать  
для хранения объектов любого типа, который будет объявлен при создании  
экземпляра класса Queue.
```

4. Замените прежний тип данных параметра функции `put()` идентификатором `QType`.

```
// Помещаем данные в очередь.  
void put(QType data) {  
    if(putloc == size) {  
        cout << " -- Очередь заполнена.\n";  
        return;  
    }  
  
    putloc++;  
    q[putloc] = data;  
}
```

5. Замените прежний тип данных параметра функции `get()` идентификатором `QType`.

```
// Извлекаем данные из очереди.  
QType get() {  
    if(getloc == putloc) {  
        cout << " -- Очередь пуста.\n";  
        return 0;  
    }  
  
    getloc++;  
    return q[getloc];  
}
```

6. Ниже приводится полный код определения класса `Queue` вместе с функцией `main()`, которая демонстрирует его использование.

```
/*
```

Проект 12.1

Шаблонный класс очереди.

```
*/
```

```
#include <iostream>
```

```

using namespace std;

const int maxQsize = 100;

// Создание обобщенного класса очереди.
template <class QType> class Queue {
    QType q[maxQsize]; // Этот массив будет содержать
                        // очередь.
    int size;          // Максимальное количество
                        // элементов, которое можно
                        // хранить в этой очереди.
    int putloc, getloc; // Индексы, используемые в
                        // функциях put() и get().

public:

    // Создание очереди конкретной длины.
    Queue(int len) {
        // Длина очереди должна не превышать значение maxQsize
        // и выражаться положительным числом.
        if(len > maxQsize) len = maxQsize;
        else if(len <= 0) len = 1;

        size = len;
        putloc = getloc = 0;
    }

    // Помещаем данные в очередь.
    void put(QType data) {
        if(putloc == size) {
            cout << " -- Очередь заполнена.\n";
            return;
        }

        putloc++;
        q[putloc] = data;
    }

    // Извлекаем данные из очереди.
    QType get() {

```

## 602 Модуль 12. Исключения, шаблоны и кое-что еще

```
if(getloc == putloc) {
    cout << " -- Очередь пуста.\n";
    return 0;
}

getloc++;
return q[getloc];
};

// Демонстрируем использование обобщенного класса Queue.
int main()
{
    Queue<int> iQa(10), iQb(10); // Создание двух
                                    // int-очередей.

    iQa.put(1);
    iQa.put(2);
    iQa.put(3);

    iQb.put(10);
    iQb.put(20);
    iQb.put(30);

    cout << "Содержимое int-очереди iQa: ";
    for(int i=0; i < 3; i++)
        cout << iQa.get() << " ";
    cout << endl;

    cout << "Содержимое int-очереди iQb: ";
    for(int i=0; i < 3; i++)
        cout << iQb.get() << " ";
    cout << endl;

    Queue<double> dQa(10), dQb(10); // Создание двух
                                    // double-очередей.

    dQa.put(1.01);
    dQa.put(2.02);
```

```

dQa.put(3.03);

dQb.put(10.01);
dQb.put(20.02);
dQb.put(30.03);

cout << "Содержимое double-очереди dQa: ";
for(int i=0; i < 3; i++)
    cout << dQa.get() << " ";
cout << endl;

cout << "Содержимое double-очереди dQb: ";
for(int i=0; i < 3; i++)
    cout << dQb.get() << " ";
cout << endl;

return 0;
}

```

Результаты выполнения этой программы таковы.

```

Содержимое int-очереди iQa: 1 2 3
Содержимое int-очереди iQb: 10 20 30
Содержимое double-очереди dQa: 1.01 2.02 3.03
Содержимое double-очереди dQb: 10.01 20.02 30.03

```

- На примере класса Queue нетрудно убедиться, что обобщенные функции и классы представляют собой мощные средства, которые помогут увеличить эффективность работы программиста. Они позволяют определить общий формат объекта, который можно затем использовать с любым типом данных. Обобщенные функции и классы избавляют вас от утомительного труда по созданию отдельных реализаций для каждого типа данных, подлежащих обработке единым алгоритмом. Эту работу сделает за вас компилятор: он автоматически создаст конкретные версии определенного вами класса.

**ВАЖНО!**

**12.4.**

## Динамическое распределение памяти

Для C++-программы существует два основных способа хранения информации в оперативной памяти компьютера. Первый состоит в использовании пере-

менных. Область памяти, предоставляемая переменным, закрепляется за ними во время компиляции и не может быть изменена при выполнении программы. Второй способ заключается в использовании C++-системы динамического распределения памяти. В этом случае память для данных выделяется по мере необходимости из раздела свободной памяти, который расположен между вашей программой (и ее постоянной областью хранения) и стеком. Этот раздел называется "кучей" (heap). (Расположение программы в памяти схематично показано на рис. 12.1.)

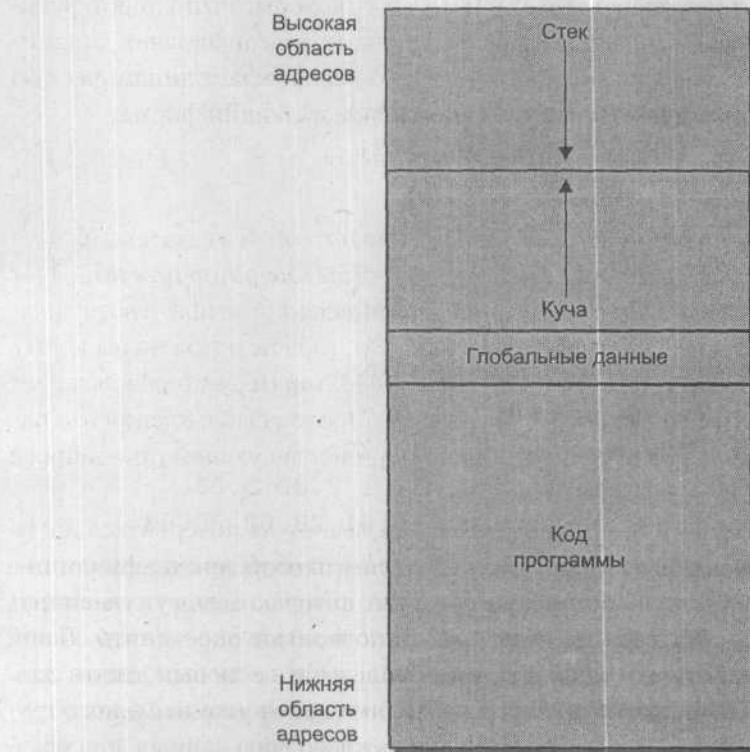


Рис. 12.1. Использование памяти в C++-программе

Динамическое выделение памяти — это получение программой памяти во время ее выполнения. Другими словами, благодаря этой системе программа может создавать переменные во время выполнения, причем в нужном (в зависимости от ситуации) количестве. Эта система динамического распределения памяти особенно ценна для поддержки таких структур данных, как связанные списки и двоичные деревья, которые изменяют свой размер по мере их использования. Динами-

ческое выделение памяти для тех или иных целей — важная составляющая почти всех реальных программ.

Чтобы удовлетворить запрос на динамическое выделение памяти, используется память из раздела, называемого “кучей”. Нетрудно предположить, что в некоторых чрезвычайных ситуациях свободная память “кучи” может исчерпаться. Следовательно, несмотря на то, что динамическое распределение памяти (по сравнению с фиксированным) обеспечивает большую гибкость, оно и в этом случае имеет свои пределы.

Язык C++ содержит два оператора, `new` и `delete`, которые выполняют функции по выделению и освобождению памяти. Оператор `new` позволяет динамически выделить область памяти, а оператор `delete` освобождает динамическую память, ранее выделенную оператором `new`. Приводим их общий формат.

```
переменная-указатель = new тип_переменной;
delete переменная-указатель;
```

Здесь элемент `переменная-указатель` представляет собой указатель на значение, тип которого задан элементом `типа_переменной`. Оператор `new` выделяет область памяти, достаточную для хранения значения заданного типа, и возвращает указатель на эту область памяти. С помощью оператора `new` можно выделить память для значений любого допустимого типа. Оператор `delete` освобождает область памяти, адресуемую заданным указателем. После освобождения эта память может быть снова выделена в других целях при последующем `new`-запросе на выделение памяти.

Поскольку объем “кучи” конечен, она может когда-нибудь исчерпаться. Если для удовлетворения очередного запроса на выделение памяти не существует достаточно свободной памяти, оператор `new` потерпит фиаско, и будет сгенерировано исключение типа `bad_alloc` (оно определено в заголовке `<new>`). Ваша программа должна обработать это исключение и по возможности выполнить действие, соответствующее конкретной ситуации. Если исключение не будет обработано вашей программой, ее выполнение будет прекращено.

Такое поведение оператора `new` в случае невозможности удовлетворить запрос на выделение памяти определено стандартом C++. На такую реализацию настроены также все современные компиляторы, включая последние версии Visual C++ и C++ Builder. Однако некоторые более ранние компиляторы обрабатывают `new`-инструкции по-другому. Сразу после изобретения языка C++ оператор `new` при неудачном выполнении возвращал нулевой указатель. Позже его реализация была изменена так, чтобы в случае неудачи генерировалось исключение, как было описано выше. Если вы используете более старый компилятор, обратитесь к прилагаемой к нему документации и уточните, как реализован оператор `new`.

Поскольку в этой книге мы придерживаемся стандарта C++, то во всех представленных здесь примерах предполагается именно генерирование исключений. Если ваш компилятор обрабатывает неудачи с выделением памяти по-другому, то вам придется внести в примеры соответствующие изменения.

Рассмотрим пример программы, которая иллюстрирует использование операторов `new` и `delete` (а именно выделяет память для хранения целочисленного значения).

```
// Демонстрация использования операторов new и delete.
```

```
#include <iostream>
#include <new>
using namespace std;

int main()
{
    int *p;

    try {
        p = new int; // Выделяем память для int-значения.
    } catch (bad_alloc xa) { // Проверка факта неудачного
                           // выполнения оператора new.
        cout << "Не удалось выделить память.\n";
        return 1;
    }

    *p = 100;

    cout << "По адресу " << p << " ";
    cout << "хранится значение " << *p << "\n";

    delete p; // Освобождаем память.

    return 0;
}
```

Эта программа присваивает указателю `p` адрес (заятой из “кучи”) области памяти, которая будет иметь размер, достаточный для хранения целочисленного значения. Затем в эту область памяти помещается число 100, после чего на экране отображается ее содержимое. Наконец, динамически выделенная память освобождается.

Оператор `delete` необходимо использовать только с тем указателем на память, который был возвращен в результате `new`-запроса на выделение памяти. Использование оператора `delete` с другим типом адреса может вызвать серьезные проблемы (даже полный отказ системы).

## Спросим у опытного программиста

**Вопрос.** Мне приходилось видеть C++-программы, в которых с целью динамического распределения памяти используются функции `malloc()` и `free()`. Что это за функции?

**Ответ.** Язык С не поддерживает операторы `new` и `delete`. Вместо них для выделения и освобождения памяти используются библиотечные функции `malloc()` и `free()`. Функция `malloc()` предназначена для выделения памяти, а функция `free()` — для ее освобождения. C++ по-прежнему поддерживает С-систему динамического распределения памяти. Поэтому эти функции могут встречаться в C++-программах, особенно в тех, которые были переведены с языка С. В новых C++-программах для выделения и освобождения памяти все же следует использовать операторы `new` и `delete` (не только потому, что они более удобны, но и потому, что они не позволят вам допустить некоторые ошибки, которые легко “проникают” в программу при работе с функциями `malloc()` и `free()`). И еще. Существует одно “неписанное” правило: не смешивайте функции `malloc()` и `free()` с операторами `new` и `delete` в одной программе. Нет никакой гарантии, что они взаимно совместимы.

## ИНИЦИАЛИЗАЦИЯ ДИНАМИЧЕСКИ ВЫДЕЛЕННОЙ ПАМЯТИ

Используя оператор `new`, динамически выделяемую память можно инициализировать. Для этого после имени типа задайте начальное значение (инициализатор), заключив его в круглые скобки. Вот как выглядит общий формат операции выделения памяти с использованием инициализатора:

`переменная-указатель = new тип(инициализатор);`

Безусловно, тип инициализатора должен быть совместим с типом объекта, для которого выделяется память.

Например, в следующей программе область памяти, адресуемая указателем `p`, инициализируется значением 87.

```
// Инициализация памяти.
```

```
#include <iostream>
```

```
#include <new>
using namespace std;

int main()
{
    int *p;

    try {
        p = new int (87); // Инициализируем память числом 87.
    } catch (bad_alloc xa) {
        cout << "Не удалось выделить память.\n";
        return 1;
    }

    cout << "По адресу " << p << " ";
    cout << "хранится значение " << *p << "\n";

    delete p;
}

return 0;
}
```

## Выделение памяти для массивов

С помощью оператора `new` можно выделять память и для массивов. Вот как выглядит общий формат операции выделения памяти для одномерного массива:

`переменная-указатель = new тип_массива [размер];`

Здесь элемент `размер` задает количество элементов в массиве.

Чтобы освободить память, выделенную для динамически созданного массива, используйте такой формат оператора `delete`:

`delete [] переменная-указатель;`

Здесь квадратные скобки означают, что динамически созданный массив удаляется, а вся область памяти, выделенная для него, автоматически освобождается.

Например, при выполнении следующей программы выделяется память для 10-элементного целочисленного массива.

```
// Выделение памяти для массива.
```

```
#include <iostream>
```

```
#include <new>
using namespace std;

int main()
{
    int *p, i;

    try {
        p = new int [10]; // Выделяем память для 10-элементного
                           // целочисленного массива.
    } catch (bad_alloc xa) {
        cout << "Не удалось выделить память.\n";
        return 1;
    }

    for(i=0; i<10; i++)
        p[i] = i;

    for(i=0; i<10; i++)
        cout << p[i] << " ";

    delete [] p; // Освобождаем память, занимаемую массивом.

    return 0;
}
```

Обратите внимание на инструкцию `delete`. Как упоминалось выше, при освобождении памяти, занимаемой массивом, с помощью оператора `delete` необходимо использовать квадратные скобки (`[]`). Как будет показано в следующем разделе, это особенно важно при выделении памяти для массивов объектов.

При динамическом выделении памяти для массива важно помнить, что его нельзя одновременно и инициализировать.

## Выделение памяти для объектов

Используя оператор `new`, можно выделять память и для объектов. При этом оператор `new` возвращает указатель на созданный объект. Объекты, созданные динамически, действуют подобно любым другим объектам. При создании объекта вызывается его конструктор, а при разрушении — его деструктор.

## 610 Модуль 12. Исключения, шаблоны и кое-что еще

Рассмотрим программу создания класса `Rectangle`, в котором инкапсулируются ширина и высота прямоугольника. В функции `main()` объект типа `Rectangle` создается динамически. Разрушение объекта происходит по завершении программы.

```
// Выделение памяти для объекта.
```

```
#include <iostream>
#include <new>
using namespace std;

class Rectangle {
    int width;
    int height;
public:
    Rectangle(int w, int h) {
        width = w;
        height = h;
        cout << "Создание прямоугольника шириной " << width <<
            " и высотой " << height << ".\n";
    }
    ~Rectangle() {
        cout << "Разрушение прямоугольника шириной " << width <<
            " и высотой " << height << ".\n";
    }
    int area() {
        return width * height;
    }
};

int main()
{
    Rectangle *p;

    try {
        p = new Rectangle(10, 8); // Выделяем память для объекта
                                // типа Rectangle. Это вызовет
                                // конструктор класса Rectangle.
    }
}
```

```

} catch (bad_alloc xa) {
    cout << "Не удалось выделить память.\n";
    return 1;
}

cout << "Площадь прямоугольника равна " << p->area()
    << ".\n";

cout << "\n";

delete p; // Освобождение памяти, занимаемой объектом.
           // Это вызовет деструктор класса Rectangle.
return 0;
}

```

Результаты выполнения этой программы таковы.

Создание прямоугольника шириной 10 и высотой 8.

Площадь прямоугольника равна 80.

Разрушение прямоугольника шириной 10 и высотой 8.

Обратите внимание на то, что аргументы для конструктора объекта задаются после имени типа. Кроме того, поскольку переменная p содержит указатель на объект, то при вызове функции area() используется оператор "стрелка", а не оператор "точка".

Можно также динамически создавать целый массив объектов, но с одной "оговоркой". Поскольку массив объектов, создаваемый с помощью оператора new, не может иметь инициализатора, то вы должны быть уверены в том, что если класс определяет конструкторы, то один из них должен быть без параметров. В противном случае C++-компилятор при попытке выделить память для этого массива не найдет соответствующий конструктор и не скомпилирует программу.

Новая версия предыдущей программы отличается от старой добавлением конструктора, не имеющего параметров, что позволит динамически создать массив объектов типа Rectangle. Кроме того, новое определение класса Rectangle содержит функцию set(), которая устанавливает размеры прямоугольника.

```
// Выделение памяти для массива объектов.
```

```
#include <iostream>
#include <new>
using namespace std;
```

## 612 Модуль 12. Исключения, шаблоны и кое-что еще

```
class Rectangle {  
    int width;  
    int height;  
public:  
    Rectangle() { // Добавляем конструктор без параметров.  
        width = height = 0;  
        cout << "Создание прямоугольника шириной " << width <<  
            " и высотой " << height << ".\n";  
    }  
  
    Rectangle(int w, int h) {  
        width = w;  
        height = h;  
        cout << "Создание прямоугольника шириной " << width <<  
            " и высотой " << height << ".\n";  
    }  
  
    ~Rectangle() {  
        cout << "Разрушение прямоугольника шириной " << width <<  
            " и высотой " << height << ".\n";  
    }  
  
    void set(int w, int h) { // Добавляем функцию set().  
        width = w;  
        height = h;  
    }  
  
    int area() {  
        return width * height;  
    }  
};  
  
int main()  
{  
    Rectangle *p;  
  
    try {  
        p = new Rectangle [3];
```

```

} catch (bad_alloc xa) {
    cout << "Не удалось выделить память.\n";
    return 1;
}

cout << "\n";

p[0].set(3, 4);
p[1].set(10, 8);
p[2].set(5, 6);

for(int i=0; i < 3; i++)
    cout << "Площадь равна " << p[i].area() << endl;

cout << "\n";

delete [] p; // Здесь вызывается деструктор для каждого
              // объекта массива.

return 0;
}

```

Вот как выглядят результаты выполнения этой программы.

Создание прямоугольника шириной 0 и высотой 0.

Создание прямоугольника шириной 0 и высотой 0.

Создание прямоугольника шириной 0 и высотой 0.

Площадь равна 12

Площадь равна 80

Площадь равна 30

Разрушение прямоугольника шириной 5 и высотой 6.

Разрушение прямоугольника шириной 10 и высотой 8.

Разрушение прямоугольника шириной 3 и высотой 4.

Поскольку указатель *p* освобождается с использованием оператора *delete* [], то, как подтверждают результаты выполнения этой программы, для каждого объекта в массиве выполняется деструктор. Обратите также внимание на то, что, поскольку указатель *p* индексируется подобно имени массива, для доступа к членам класса *Rectangle* используется оператор “точка”.



### Вопросы для текущего контроля

1. Какой оператор позволяет выделить память для переменных? Какой оператор освобождает выделенную память?
2. Что происходит в случае, если не может быть выполнен запрос на выделение памяти?
3. Можно ли инициализировать память при ее динамическом распределении?\*

**ВАЖНО!**

## 12.5. Пространства имен

Пространства имен были кратко описаны в модуле 1. Здесь мы рассмотрим их более детально. Они позволяют локализовать имена идентификаторов, чтобы избежать конфликтных ситуаций с ними. В C++-среде программирования используется огромное количество имен переменных, функций и имен классов. До введения пространств имен все эти имена конкурировали за память в глобальном пространстве имен, что и было причиной возникновения многих конфликтов. Например, если бы в вашей программе была определена функция `toUpper()`, она могла бы (в зависимости от списка параметров) переопределить стандартную библиотечную функцию `toUpper()`, поскольку оба имени должны были бы храниться в глобальном пространстве имен. Конфликты с именами возникали также при использовании в одной программе нескольких библиотек сторонних производителей. В этом случае имя, определенное в одной библиотеке, конфликовало с таким же именем из другой библиотеки. Подобная ситуация особенно неприятна при использовании одноименных классов. Например, если в вашей программе определен класс `VideoMode`, и в библиотеке, используемой вашей программой, определен класс с таким же именем, конфликта не избежать.

Для решения описанной проблемы было создано ключевое слово `namespace`. Поскольку оно локализует видимость объявленных в нем имен, это значит, что пространство имен позволяет использовать одно и то же имя в различных кон-

1. Выделить память для переменных можно с помощью оператора `new`, а освободить эту память — с помощью оператора `delete`.
2. Если запрос на выделение памяти не может быть выполнен, генерируется исключение типа `bad_alloc`.
3. Да, при динамическом распределении память можно инициализировать.

текстах, не вызывая при этом конфликта имен. Возможно, больше всего от нововведения “повезло” C++-библиотеке стандартных функций. До появления ключевого слова `namespace` вся C++-библиотека была определена в глобальном пространстве имен (которое было, конечно же, единственным). С наступлением `namespace`-“эры” C++-библиотека определяется в собственном пространстве имен, именуемом `std`, которое значительно понизило вероятность возникновения конфликтов имен. В своей программе программист волен создавать собственные пространства имен, чтобы локализовать видимость тех имен, которые, по его мнению, могут стать причиной конфликта. Это особенно важно, если вы занимаетесь созданием библиотек классов или функций.

## Понятие пространства имен

Ключевое слово `namespace` позволяет программисту “отделиться” от глобального пространства имен путем создания некоторой декларативной области. По сути, пространство имен определяет область видимости. Общий формат задания пространства имен таков.

```
namespace name {
    // объявления
}
```

Все, что определено в границах инструкции `namespace`, находится в области видимости этого пространства имен.

В следующей программе приведен пример использования `namespace`-инструкции. Она локализует имена, используемые для реализации простого класса счета в обратном направлении. В созданном здесь пространстве имен определяется класс `counter`, который реализует счетчик, и переменные `upperbound` и `lowerbound`, содержащие значения верхней и нижней границ, применяемых для всех счетчиков.

```
// Демонстрация использования пространства имен.
```

```
namespace CounterNameSpace { // Создание пространства
    // имен CounterNameSpace.

    int upperbound;
    int lowerbound;

    class counter {
        int count;
    public:
        counter(int n) {
```

## 616 Модуль 12. Исключения, шаблоны и кое-что еще

```
        if(n <= upperbound) count = n;
        else count = upperbound;
    }

    void reset(int n) {
        if(n <= upperbound) count = n;
    }

    int run() {
        if(count > lowerbound) return count--;
        else return lowerbound;
    }
};

}
```

Здесь переменные `upperbound` и `lowerbound`, а также класс `counter` являются частью области видимости, определенной пространством имен `CounterNameSpace`.

В любом пространстве имен к идентификаторам, которые в нем объявлены, можно обращаться напрямую, т.е. без указания этого пространства имен. Например, в функции `run()`, которая находится в пространстве имен `CounterNameSpace`, можно напрямую обращаться к переменной `lowerbound`:

```
if(count > lowerbound) return count--;
```

Но поскольку инструкция `namespace` определяет область видимости, то при обращении к объектам, объявленным в пространстве имен, извне этого пространства необходимо использовать оператор разрешения области видимости. Например, чтобы присвоить значение 10 переменной `upperbound` из кода, который является внешним по отношению к пространству имен `CounterNameSpace`, нужно использовать такую инструкцию.

```
CounterNameSpace::upperbound = 10;
```

Чтобы объявить объект типа `counter` вне пространства имен `CounterNameSpace`, используйте инструкцию, подобную следующей.

```
CounterNameSpace::counter ob;
```

В общем случае, чтобы получить доступ к некоторому члену пространства имен извне этого пространства, необходимо предварить имя этого члена именем пространства и отделить имена оператором разрешения области видимости (или оператора разрешения контекста, который также называется *оператором явного задания*).

Рассмотрим программу, в которой демонстрируется использование пространства имен CounterNamespace.

```
// Демонстрация использования пространства  
// имен CounterNameSpace.
```

```
#include <iostream>
using namespace std;

namespace CounterNameSpace {
    int upperbound;
    int lowerbound;

    class counter {
        int count;
    public:
        counter(int n) {
            if(n <= upperbound) count = n;
            else count = upperbound;
        }

        void reset(int n) {
            if(n <= upperbound) count = n;
        }

        int run() {
            if(count > lowerbound) return count--;
            else return lowerbound;
        }
    };
}

int main()
{
    CounterNameSpace::upperbound = 100; // Явное обращение к
    CounterNameSpace::lowerbound = 0; // членам пространства
                                    // имен
                                    // CounterNameSpace.
    CounterNameSpace::counter obj(10); // Обратите внимание на
                                    // использование
```

## 618 Модуль 12. Исключения, шаблоны и кое-что еще

```
// оператора
// разрешения контекста.

int i;

do {
    i = ob1.run();
    cout << i << " ";
} while(i > CounterNameSpace::lowerbound);
cout << endl;

CounterNameSpace::counter ob2(20);

do {
    i = ob2.run();
    cout << i << " ";
} while(i > CounterNameSpace::lowerbound);
cout << endl;

ob2.reset(100);
CounterNameSpace::lowerbound = 90;
do {
    i = ob2.run();
    cout << i << " ";
} while(i > CounterNameSpace::lowerbound);

return 0;
}
```

Обратите внимание на то, что при объявлении объекта класса `counter` и обращении к переменным `upperbound` и `lowerbound` используется имя пространства имен `CounterNameSpace`. Но после объявления объекта типа `counter` уже нет необходимости в полной квалификации его самого или его членов. Поскольку пространство имен однозначно определено, функцию `run()` объекта `ob1` можно вызывать напрямую, т.е. без указания (в качестве префикса) пространства имен (`ob1.run()`).

Программа может содержать несколько объявлений пространств имен с одинаковыми именами. Это означает, что пространство имен можно разбить на несколько файлов или на несколько частей в рамках одного файла. Вот пример.

```
namespace NS {
    int i;
```

```

}

// ...

namespace NS {
    int j;
}

```

Здесь пространство имен NS разделено на две части. Однако содержимое каждой части относится к одному и тому же пространству имен NS.

Любое пространство имен должно быть объявлено вне всех остальных областей видимости. Это означает, что нельзя объявлять пространства имен, которые локализованы, например, в рамках функции. При этом одно пространство имен может быть вложено в другое.

## Инструкция using

Если программа включает множество ссылок на члены некоторого пространства имен, то нетрудно представить, что необходимость указывать имя этого пространства имен при каждом к нему обращении, очень скоро утомит вас. Эту проблему позволяет решить инструкция `using`, которая применяется в двух форматах.

```
using namespace имя;
```

```
using name::член;
```

В первой форме элемент `имя` задает название пространства имен, к которому вы хотите получить доступ. Все члены, определенные внутри заданного пространства имен, попадают в "поле видимости", т.е. становятся частью текущего пространства имен, и их можно затем использовать без дополнительной квалификации (уточнения пространства имен). Во второй форме делается "видимым" только указанный член пространства имен. Например, если предположить, что пространство имен `CounterNameSpace` определено (как показано выше), следующие инструкции `using` и присваивания будут вполне законными.

```
using CounterNameSpace::lowerbound; // Видимым стал только
                                    // член lowerbound.

lowerbound = 10; // Все в порядке, поскольку член
                  // lowerbound находится в области
                  // видимости.
```

```
using namespace CounterNameSpace; // Все члены видимы.
```

**620** Модуль 12. Исключения, шаблоны и кое-что еще

```
upperbound = 100; // Все в порядке, поскольку все члены  
                  // видимы.
```

Использование инструкции `using` демонстрируется в следующей программе (которая представляет собой новый вариант счетчика из предыдущего раздела).

```
// Демонстрация использования инструкции using.
```

```

    // пространства имен.

// Теперь для установки значения переменной upperbound
// не нужно указывать пространство имен.
upperbound = 100;

// Но при обращении к переменной lowerbound и другим
// объектам по-прежнему необходимо указывать
// пространство имен.
CounterNameSpace::lowerbound = 0;
CounterNameSpace::counter ob1(10);
int i;

do {
    i = ob1.run();
    cout << i << " ";
} while(i > CounterNameSpace::lowerbound);
cout << endl;

// Теперь используем все пространство
// имен CounterNameSpace.
using namespace CounterNameSpace;

counter ob2(20);

do {
    i = ob2.run();
    cout << i << " ";
} while(i > lowerbound);
cout << endl;

ob2.reset(100);
lowerbound = 90;
do {
    i = ob2.run();
    cout << i << " ";
} while(i > lowerbound);

return 0;
}

```

Эта программа иллюстрирует еще один важный момент. Использование одного пространства имен не переопределяет другое. Если некоторое пространство имен становится “видимым”, это значит, что оно просто добавляет свои имена к именам других, уже действующих пространств. Поэтому к концу этой программы к глобальному пространству имен добавились и `std`, и `CounterNameSpace`.

## Неименованные пространства имен

Существует *неименованное* пространство имен специального типа, которое позволяет создавать идентификаторы, уникальные для данного файла. Общий формат его объявления выглядит так.

```
namespace {
    // объявления
}
```

Неименованные пространства имен позволяют устанавливать уникальные идентификаторы, которые известны только в области видимости одного файла. Другими словами, члены файла, который содержит неименованное пространство имен, можно использовать напрямую, без уточняющего префикса. Но вне файла эти идентификаторы неизвестны.

Как упоминалось выше в этой книге, использование модификатора типа `static` также позволяет ограничить область видимости глобального пространства имен файлом, в котором он объявлен. Несмотря на то что использование глобальных `static`-объявлений все еще разрешено в C++, для локализации идентификатора в рамках одного файла лучше использовать неименованное пространство имен.

## Пространство имен `std`

Стандарт C++ определяет всю свою библиотеку в собственном пространстве имен, именуемом `std`. Именно поэтому большинство программ в этой книге включает следующую инструкцию:

```
using namespace std;
```

При выполнении данной инструкции пространство имен `std` становится текущим, что открывает прямой доступ к именам функций и классов, определенных в этой библиотеке, т.е. при обращении к ним отпадает необходимость в использовании префикса `std::`.

Конечно, при желании можно явным образом квалифицировать каждое библиотечное имя префиксом `std::`. Например, можно было бы явно квалифицировать объект `cout` следующим образом.

```
std::cout << "Явно квалифицируем объект cout префиксом
std.";
```

Если ваша программа использует стандартную библиотеку только в ограниченных пределах, то, возможно, ее и не стоит вносить в глобальное пространство имен. Но если ваша программа содержит сотни ссылок на библиотечные имена, то гораздо проще сделать пространство имен `std` текущим, чем полностью квалифицировать каждое имя в отдельности.



### Вопросы для текущего контроля

- Что такое пространство имен? С помощью какого ключевого слова оно создается?
- Являются ли пространства имен аддитивными?
- Каково назначение инструкции `using`?\*

**ВАЖНО!**

## 12.6. Статические члены класса

Вы познакомились с ключевым словом `static` в модуле 7, где описывалось его использование для модификации объявлений локальных и глобальных переменных. Помимо этого, ключевое слово `static` можно применять и к членам класса: как переменным, так и функциям.

### Статические члены данных класса

Объявляя член данных (переменную) класса статическим, мы сообщаем компилятору, что независимо от того, сколько объектов этого класса будет создано, существует только одна копия данного `static`-члена. Другими словами, `static`-член разделяется всеми объектами класса. Все статические переменные

- Пространство имен — это декларативная область, которая определяет диапазон видимости объявленных в ней имен. Пространство имен создается с помощью ключевого слова `namespace`.
- Да, пространства имен являются аддитивными. Это значит, что имена одного пространства можно добавить к именам других, уже существующих пространств.
- Инструкция `using` позволяет сделать “видимыми” члены заданного пространства имен.

при первом создании объекта инициализируются нулевыми значениями, если не задано других значений инициализации.

При объявлении статического члена данных в классе программист его *не* определяет. Вместо этого он должен обеспечить его *глобальное определение* вне такого класса. Это реализуется путем повторного объявления этой статической переменной с помощью оператора разрешения области видимости, который позволяет идентифицировать, к какому классу она принадлежит. И только тогда для этой статической переменной будет выделена память.

Рассмотрим пример использования `static`-члена класса.

```
// Использование статических переменных реализации.

#include <iostream>
using namespace std;

class ShareVar {
    static int num; // ← Объявляем static-член данных. Его
                    // будут совместно использовать все
                    // экземпляры класса ShareVar.

public:
    void setnum(int i) { num = i; };
    void shownum() { cout << num << " "; }
};

int ShareVar::num; // ← Определяем static-член данных num.

int main()
{
    ShareVar a, b;

    a.shownum(); // Выводится значение 0.
    b.shownum(); // Выводится значение 0.

    a.setnum(10); // Устанавливаем static-член num равным 10.

    a.shownum(); // Выводится значение 10.
    b.shownum(); // Так же выводится значение 10.

    return 0;
}
```

Результаты выполнения этой программы таковы.

```
0 0 10 10
```

В этой программе обратите внимание на то, что статический целочисленный член `num` и объявлен в классе `ShareVar`, и определен в качестве глобальной переменной. Как было заявлено выше, необходимость такого двойного объявления вызвана тем, что при объявлении члена `num` в классе `ShareVar` память для него не выделяется. C++ инициализирует переменную `num` значением 0, поскольку никакой другой инициализации в программе нет. Поэтому в результате двух первых вызовов функции `shownum()` для объектов `a` и `b` отображается значение 0. Затем для объекта `a` член `num` устанавливается равным 10, после чего для обоих объектов `a` и `b` с помощью функции `shownum()` выводится на экран значение члена `num`. Но так как существует только одна копия переменной `num`, разделяемая объектами `a` и `b`, при вызове функции `shownum()` для обоих объектов будет выведено одно и то же значение 10.

Если `static`-переменная является открытой (т.е. `public`-переменной), к ней можно обращаться напрямую через имя ее класса, без ссылки на какой-либо конкретный объект. (Безусловно, обращаться можно также и через имя объекта.) Рассмотрим пример.

```
// Обращение к static-переменной через имя ее класса.
```

```
#include <iostream>
using namespace std;

class Test {
public:
    static int num;
    void shownum() { cout << num << endl; }
};

int Test::num; // Определяем static-переменную num.

int main()
{
    Test a, b;

    // Устанавливаем переменную num, используя имя класса.
    Test::num = 100; // ← Обращаемся к члену num через имя
                    // его класса Test.
```

```

    a.shownum(); // Выводится значение 100.
    b.shownum(); // Выводится значение 100.

    // Устанавливаем переменную num, используя имя объекта.
    a.num = 200; // ← Обращаемся к члену num через имя
                  // объекта a.
    a.shownum(); // Выводится значение 200.
    b.shownum(); // Выводится значение 200.

    return 0;
}

```

Обратите внимание на то, как устанавливается значение переменной num с использованием имени ее класса:

```
Test:::num = 100;
```

Вполне допустимо устанавливать значение открытой static-переменной и через объект:

```
a.num = 200;
```

Любой из этих способов установки переменной совершенно легален.

## Статические функции-члены класса

Можно также объявить статической и функцию-член, но это — нераспространенная практика. К статической функции-члену могут получить доступ только другие static-члены этого класса. (Конечно же, статическая функция-член может получать доступ к нестатическим глобальным данным и функциям.) Статическая функция-член не имеет указателя this. Создание виртуальных статических функций-членов не разрешено. Кроме того, их нельзя объявлять с модификаторами const или volatile.

Статическую функцию-член можно вызвать для объекта ее класса или независимо от какого бы то ни было объекта, а для обращения к ней достаточно использовать имя класса и оператор разрешения области видимости. Рассмотрим, например, следующую программу. В ней определяется static-переменная count, которая служит для хранения количества существующих в данный момент объектов.

```
// Демонстрация использования статических функций-членов.
```

```
#include <iostream>
using namespace std;
```

```

class Test {
    static int count;
public:

    Test() {
        count++;
        cout << "Создание объекта " <<
            count << endl;
    }

    ~Test() {
        cout << "Разрушение объекта " <<
            count << endl;
        count--;
    }

    static int numObjects() { return count; } // Статическая
  // функция-член.
};

int Test::count;

int main() {
    Test a, b, c;

    cout << "В данный момент существует " <<
        Test::numObjects() <<
        " объекта.\n\n";

    Test *p = new Test();

    cout << "После создания объекта класса Test " <<
        "насчитывается " <<
        Test::numObjects() << // Используется имя класса.
        " объекта.\n\n";

    delete p;
}

```

```
cout << "После удаления объекта " <<
    "насчитывается " <<
    a.numObjects() << // Используется имя объекта и
                      // оператор "точка".
    " объекта.\n\n";
return 0;
}
```

При выполнении эта программа генерирует такие результаты.

Создание объекта 1

Создание объекта 2

Создание объекта 3

В данный момент существует 3 объекта.

Создание объекта 4

После создания объекта класса Test насчитывается 4 объекта.

Разрушение объекта 4

После удаления объекта насчитывается 3 объекта.

Разрушение объекта 3

Разрушение объекта 2

Разрушение объекта 1

В этой программе обратите внимание на то, как вызывается статическая функция numObjects(). В первых двух ее вызовах используется имя класса, членом которого она является.

Test::numObjects()

В третьем вызове используется обычный синтаксис, т.е. имя объекта и оператор "точка".

a.numObjects()

**ВАЖНО!**

## **12.7. Динамическая идентификация типов (RTTI)**

С динамической идентификацией типов вы, возможно, незнакомы, поскольку это средство отсутствует в таких неполиморфных языках, как С или традицион-

ный BASIC. В неполиморфных языках попросту нет необходимости в получении информации о типе во время выполнения программы, так как тип каждого объекта известен во время компиляции (т.е. еще при написании программы). Но в таких полиморфных языках, как C++, возможны ситуации, в которых тип объекта неизвестен в период компиляции, поскольку точная природа этого объекта не будет определена до тех пор, пока программа не начнет выполняться. Как вы знаете, C++ реализует полиморфизм посредством использования иерархии классов, виртуальных функций и указателей на базовые классы. Указатель на базовый класс можно использовать для ссылки на объекты этого базового класса или на *объекты любых классов*, выведенных из него. Следовательно, не всегда заранее известно, на объект какого типа будет ссылаться указатель на базовый класс в произвольный момент времени. Это выяснится только при выполнении программы — при использовании одного из средств динамической идентификации типов.

Чтобы получить тип объекта во время выполнения программы, используйте оператор `typeid`. Для этого необходимо включить в программу заголовок `<typeinfo>`. Самый распространенный формат использования оператора `typeid` таков.

```
typeid(object)
```

Здесь элемент *object* означает объект, тип которого нужно получить. Можно запрашивать не только встроенный тип, но и тип класса, созданного программистом. Оператор `typeid` возвращает ссылку на объект типа `type_info`, который описывает тип объекта *object*.

В классе `type_info` определены следующие `public`-члены.

```
bool operator==(const type_info &ob);
bool operator!=(const type_info &ob);
bool before(const type_info &ob);
const char *name();
```

Перегруженные операторы “`==`” и “`!=`” служат для сравнения типов. Функция `before()` возвращает значение `true`, если вызывающий объект в порядке сопоставления стоит перед объектом (`объектом ob`), используемым в качестве параметра. (Эта функция предназначена в основном для внутреннего применения. Возвращаемый ею результат не имеет ничего общего с наследованием или иерархией классов.) Функция `name()` возвращает указатель на имя типа.

Рассмотрим простой пример использования оператора `typeid`.

```
// Пример использования оператора typeid.
```

```
#include <iostream>
#include <typeinfo>
```

## 630 Модуль 12. Исключения, шаблоны и кое-что еще

```
using namespace std;

class MyClass {
    // ...
};

int main()
{
    int i, j;
    float f;
    MyClass ob;

    // Использование оператора typeid для получения типа
    // объекта.
    cout << "Тип переменной i: " << typeid(i).name();
    cout << endl;
    cout << "Тип переменной f: " << typeid(f).name();
    cout << endl;
    cout << "Тип переменной ob: " << typeid(ob).name();
    cout << "\n\n";

    if(typeid(i) == typeid(j))
        cout << "Типы переменных i и j одинаковы.\n";

    if(typeid(i) != typeid(f))
        cout << "Типы переменных i и f неодинаковы.\n";

    return 0;
}
```

При выполнении этой программы получены такие результаты.

Тип переменной i: int

Тип переменной f: float

Тип переменной ob: class MyClass

Типы переменных i и j одинаковы.

Типы переменных i и f неодинаковы.

Если оператор typeid применяется к указателю на полиморфный базовый класс (вспомните: полиморфный класс — это класс, который содержит хотя бы

одну виртуальную функцию), он автоматически возвращает тип реального объекта, адресуемого этим указателем: будь то объект базового класса или объект класса, выведенного из базового. Следовательно, оператор typeid можно использовать для динамического определения типа объекта, адресуемого указателем на базовый класс. Применение этой возможности демонстрируется в следующей программе.

```
// Пример применения оператора typeid к иерархии
// полиморфных классов.

#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
    virtual void f() {} // делаем класс Base полиморфным
    // ...
};

class Derived1: public Base {
    // ...
};

class Derived2: public Base {
    // ...
};

int main()
{
    Base *p, baseob;
    Derived1 ob1;
    Derived2 ob2;

    p = &baseob;
    cout << "Переменная p указывает на объект типа ";
    cout << typeid(*p).name() << endl;

    p = &ob1;
    cout << "Переменная p указывает на объект типа ";
    cout << typeid(*p).name() << endl;
}
```

## 632 Модуль 12. Исключения, шаблоны и кое-что еще

```
p = &ob2;
cout << "Переменная p указывает на объект типа ";
cout << typeid(*p).name() << endl;

return 0;
}
```

Вот как выглядят результаты выполнения этой программы:

Переменная p указывает на объект типа Base

Переменная p указывает на объект типа Derived1

Переменная p указывает на объект типа Derived2

Если оператор typeid применяется к указателю на базовый класс полиморфного типа, тип реально адресуемого объекта, как подтверждают эти результаты, будет определен во время выполнения программы.

Во всех случаях применения оператора typeid к указателю на неполиморфную иерархию классов будет получен указатель на базовый тип, т.е. то, на что этот указатель реально указывает, определить нельзя. В качестве эксперимента превратите в комментарий виртуальную функцию f() в классе Base и посмотрите на результат. Вы увидите, что тип каждого объекта после внесения в программу этого изменения будет определен как Base, поскольку именно этот тип имеет указатель p.

Поскольку оператор typeid обычно применяется к разыменованному указателю (т.е. к указателю, к которому уже применен оператор "\*"), для обработки ситуации, когда этот разыменованный указатель оказывается нулевым, создано специальное исключение. В этом случае оператор typeid генерирует исключение типа bad\_typeid.

Ссылки на объекты иерархии полиморфных классов работают подобно указателям. Если оператор typeid применяется к ссылке на полиморфный класс, он возвращает тип объекта, на который она реально ссылается, и это может быть объект не базового, а производного типа. Описанное средство чаще всего используется при передаче объектов функциям по ссылке.

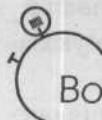
Существует еще один формат применения оператора typeid, который в качестве аргумента принимает имя типа. Этот формат выглядит так.

typeid(имя\_типа)

Например, следующая инструкция совершенно допустима.

```
cout << typeid(int).name();
```

Назначение этой версии оператора `typeid` — получить объект типа `type_info` (который описывает заданный тип данных), чтобы его можно было затем использовать в инструкции сравнения типов.



### Вопросы для текущего контроля

1. В чем состоит уникальность статических членов данных?
2. Каково назначение оператора `typeid`?
3. Объект какого типа возвращает оператор `typeid`?\*

**ВАЖНО!**

## 12.8. Операторы приведения типов

В C++ определено пять операторов приведения типов. Первый оператор (он описан выше в этой книге), применяемый в обычном (традиционном) стиле, был с самого начала встроен в C++. Остальные четыре (`dynamic_cast`, `const_cast`, `reinterpret_cast` и `static_cast`) были добавлены в язык всего несколько лет назад. Эти операторы служат дополнительными “рычагами управления” характером выполнения операций приведения типа. Рассмотрим каждый из них в отдельности.

### Оператор `dynamic_cast`

Возможно, самым важным из новых операторов является оператор динамического приведения типов `dynamic_cast`. Во время выполнения программы он проверяет обоснованность предлагаемой операции. Если в момент его вызова заданная операция оказывается недопустимой, приведение типов не реализуется. Общий формат применения оператора `dynamic_cast` таков.

`dynamic_cast<type>(expr)`

1. Пространство имен — это декларативная область, которая определяет диапазон видимости объявленных в ней имен. Пространство имен создается с помощью ключевого слова `namespace`.
2. Да, пространства имен являются аддитивными. Это значит, что имена одного пространства можно добавить к именам других, уже действующих пространств.
3. Инструкция `using` позволяет сделать “видимыми” члены заданного пространства имен.

Здесь элемент *type* означает новый тип, который является целью выполнения этой операции, а элемент *expr* – выражение, приводимое к этому новому типу. Тип *type* должен быть представлен указателем или ссылкой, а выражение *expr* должно приводиться к указателю или ссылке. Таким образом, оператор *dynamic\_cast* можно использовать для преобразования указателя одного типа в указатель другого или ссылку одного типа в ссылку другого.

Этот оператор в основном используется для динамического выполнения операций приведения типа среди полиморфных типов. Например, если даны полиморфные классы *B* и *D*, причем класс *D* выведен из класса *B*, то с помощью оператора *dynamic\_cast* всегда можно преобразовать указатель *D\** в указатель *B\**, поскольку указатель на базовый класс всегда можно использовать для указания на объект класса, выведенного из базового. Однако оператор *dynamic\_cast* может преобразовать указатель *B\** в указатель *D\** только в том случае, если адресуемым объектом действительно является объект класса *D*. И, вообще, оператор *dynamic\_cast* будет успешно выполнен только при условии, если разрешено полиморфное приведение типов, т.е. если указатель (или ссылка), приводимый к новому типу, может указывать (или ссылаться) на объект этого нового типа или объект, выведенный из него. В противном случае, т.е. если заданную операцию приведения типов выполнить нельзя, результат действия оператора *dynamic\_cast* оценивается как нулевой, если в этой операции участвуют указатели. (Если же попытка выполнить эту операцию оказалась неудачной при участии в ней ссылок, генерируется исключение типа *bad\_cast*.)

Рассмотрим простой пример. Предположим, что класс *Base* – полиморфный, а класс *Derived* выведен из класса *Base*.

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob; // Указатель на базовый класс ссылается
            // на объект класса Derived.
dp = dynamic_cast<Derived *>(bp); // Приведение к
            // указателю на
            // производный класс
            // разрешено.
```

```
if(dp) cout << "Приведение типа прошло успешно!";
```

Здесь приведение указателя *bp* (на базовый класс) к указателю *dp* (на производный класс) выполняется успешно, поскольку *bp* *действительно* указывает на объект класса *Derived*. Поэтому при выполнении этого фрагмента кода будет выведено сообщение Приведение типа прошло успешно!. Но в следующем

фрагменте кода попытка совершить операцию приведения типа будет неудачной, поскольку `bp` в действительности указывает на объект класса `Base`, а приводить указатель на базовый класс к типу указателя на производный неправомерно, если адресуемый им объект не является *на самом деле* объектом производного класса.

```
bp = &b_ob; // Указатель на базовый класс ссылается
           // на объект класса Base.
dp = dynamic_cast<Derived *> (bp); // ошибка!
if (!dp) cout << "Приведение типа выполнить не удалось";
```

Поскольку попытка выполнить операцию приведения типа оказалась неудачной, при выполнении этого фрагмента кода будет выведено сообщение Приведение типа выполнить не удалось.

## Оператор `const_cast`

Оператор `const_cast` используется для явного переопределения модификаторов `const` и/или `volatile`. Новый тип должен совпадать с исходным, за исключением его атрибутов `const` или `volatile`. Чаще всего оператор `const_cast` используется для удаления признака постоянства (атрибута `const`). Его общий формат имеет следующий вид,

```
const_cast<type> (expr)
```

Здесь элемент `type` задает новый тип операции приведения, а элемент `expr` означает выражение, которое приводится к новому типу. Необходимо подчеркнуть, что использование оператора `const_cast` для удаления `const`-атрибута является потенциально опасным средством. Поэтому обращайтесь с ним очень осторожно.

И еще. Удалять `const`-атрибут способен только оператор `const_cast`. Другими словами, ни `dynamic_cast`, ни `static_cast`, ни `reinterpret_cast` нельзя использовать для изменения `const`-атрибута объекта.

## Оператор `static_cast`

Оператор `static_cast` выполняет операцию неполиморфного приведения типов. Его можно использовать для любого стандартного преобразования. При этом во время работы программы никаких проверок на допустимость не выполняется. По сути, `static_cast` может заменить “традиционный” оператор приведения типов.

Оператор `static_cast` имеет следующий общий формат записи.

```
static_cast<type> (expr)
```

Здесь элемент *type* задает новый тип операции приведения, а элемент *expr* означает выражение, которое приводится к этому новому типу.

### Оператор `reinterpret_cast`

Оператор `reinterpret_cast` преобразует один тип в принципиально другой. Например, его можно использовать для преобразования указателя в целое значение и целого значения — в указатель. Его также можно использовать для приведения наследственно несовместимых типов указателей. Этот оператор имеет следующий общий формат записи.

`reinterpret_cast<type> (expr)`

Здесь элемент *type* задает новый тип операции приведения, а элемент *expr* означает выражение, которое приводится к этому новому типу.

## Что же дальше?

Цель этой книги — преподать начинающему программисту основные элементы языка C++, т.е. те средства и методы программирования, которые используются в обычной “ежедневной” практике. Освоив материал этой книги, вы сможете смело приступать к созданию реальных программ. Однако помните, что C++ — очень богатый язык, который содержит множество средств высокого уровня, без изучения которых невозможно стать профессиональным программистом. К ним относятся следующие:

- стандартная библиотека шаблонов (Standard Template Library — STL);
- “неконвертирующие”, или `explicit`-конструкторы;
- функции преобразования (типа класса в другой тип);
- константные (`const`) функции-члены и ключевое слово `mutable`;
- ключевое слово `asm`;
- перегруженные операторы индексации массивов (`[]`), вызова функции (`()`) и динамического распределения памяти `new` и `delete`.

Из перечисленных здесь средств самым важным для становления программиста я считаю стандартную библиотеку шаблонов (STL). Эта библиотека шаблонных классов предоставляет готовые решения для самых разных задач хранения и организации данных. Например, STL определяет такие обобщенные структуры данных, как очереди, стеки и списки, которые вы можете эффективно использовать в своих программах.

Полезно также изучить библиотеку функций C++. Она содержит широкий выбор готовых функций, применение которых позволит вам существенно упростить и ускорить создание собственных программ.

Для продолжения теоретического освоения C++ предлагаю обратиться к моей книге *Полный справочник по C++*, М.: Издательский дом “Вильямс”. Она содержит подробное описание элементов языка C++ и библиотек. Ваш нынешний багаж знаний вполне позволит вам осилить материал этого справочника.



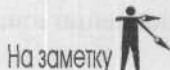
## Тест для самоконтроля по модулю 12

1. Поясните, как использовать инструкции `try`, `catch` и `throw` для поддержки обработки исключительных ситуаций.
2. Как должен быть организован список `catch`-инструкций при перехвате исключений как базового, так и производных классов?
3. Как указать, что исключение типа `MyExcept` может быть сгенерировано вне функции `func()`, которая возвращает значение типа `void`?
4. Определите исключение для обобщенного класса `Queue` (см. проект 12.1). Класс `Queue` должен генерировать это исключение при переполнении или потере значимости. Продемонстрируйте его использование.
5. Что представляет собой обобщенная функция и какое ключевое слово используется для ее создания?
6. Создайте обобщенные версии функций `quicksort()` и `qs()`, представленных в проекте 5.1. Продемонстрируйте их использование.
7. Используя приведенный ниже класс `Samp1`, создайте очередь, состоящую из трех `Samp1`-объектов. Для этого воспользуйтесь обобщенным классом `Queue` из проекта 12.1.

```
class Samp1 {
    int id;
public:
    Samp1() { id = 0; }
    Samp1(int x) { id = x; }
    void show() { cout << id << endl; }
};
```

8. Переделайте ответ на вопрос 7 так, чтобы хранимые в очереди объекты класса `Samp1` создавались динамически.
9. Покажите, как объявить пространство имен `RobotMotion`.
10. Какое пространство имен содержит стандартную библиотеку C++?

11. Может ли статическая (`static`) функция-член получить доступ к нестатическим членам данных класса?
12. С помощью какого оператора можно получить тип объекта при выполнении программы?
13. Какой оператор приведения типов следует использовать, чтобы определить допустимость операции приведения полиморфных типов при выполнении программы?
14. Какие действия выполняет оператор `const_cast`?
15. Попытайтесь самостоятельно внести класс `Queue` (из проекта 12.1) в собственное пространство имен `QueueCode`, а также в свой файл `Queue.cpp`. Затем переделайте функцию `main()`, включив в нее инструкцию `using`, чтобы пространство имен `QueueCode` стало “видимым”.
16. Продолжайте изучать средства C++. Это — самый мощный язык программирования на данный момент. Освоение его возможностей на практике позволит вам считать себя “принятым” в высшую лигу программистов. Успеха вам!



Ответы на эти вопросы можно найти на Web-странице данной книги по адресу: <http://www.osborne.com>.

# Приложение А

## Препроцессор

таблицы  
параметров

### запись реестром

именно в отдельном файле, при работе с которым не нужны диалоги. Такой алгоритмика очень удобна для работы с переносимой базой данных, состоящей из различных таблиц, и включает в себя различные функции обработки базы данных, а также возможность изменения структуры базы.

Файл реестра содержит информацию о всех параметрах, необходимых для работы с базой данных. Помимо этого, в нем хранятся сведения о типах данных, определенных в языке программирования, а также о способах обработки данных, которые используются для записи информации в базу.

Файл реестра может быть создан вручную или с помощью специальных инструментов. Для этого необходимо использовать специальные программы, например, MS Office Word или Microsoft Excel. Важно помнить, что в MS Word файлы реестра с типом файла должны иметь расширение .txt, а в Microsoft Excel - .xls.

Файл реестра может содержать различные данные, такие как конфигурации, настройки и т.д. Для этого в нем должны быть определены соответствующие параметры, которые будут использоваться для работы с базой данных. Для этого в реестре должны быть определены соответствующие параметры, такие как имя базы данных, расположение базы данных и т.д.

Препроцессор — это часть компилятора, которая подвергает вашу программу различным текстовым преобразованиям до реальной трансляции исходного кода в объектный. Программист может давать препроцессору команды, называемые *директивами препроцессора* (preprocessor directives), которые, не являясь формальной частью языка C++, способны расширить область действия его среди программирования.

Препроцессор C++ — прямой потомок препроцессора C, и некоторые его средства оказались избыточными после введения в C++ новых элементов. Однако он по-прежнему является важной частью C++-среды программирования, и многие программисты все еще активно его применяют.

Препроцессор C++ включает следующие директивы.

|         |        |          |
|---------|--------|----------|
| #define | #error | #include |
| #if     | #else  | #elif    |
| #endif  | #ifdef | #ifndef  |
| #undef  | #line  | #pragma  |

Как видите, все директивы препроцессора начинаются с символа “#”. Теперь рассмотрим каждую из них в отдельности.

## Директива #define

Директива `#define` используется для определения идентификатора и символьной последовательности, которая будет представлена вместо идентификатора везде, где он встречается в исходном коде программы. Этот идентификатор называется *макроименем*, а процесс замены — *макроподстановкой* (реализацией макрорасширения). Общий формат использования этой директивы имеет следующий вид.

`#define макроимя последовательность_символов`

Обратите внимание на то, что здесь нет точки с запятой. Заданная *последовательность\_символов* завершается только символом конца строки. Между элементами *имя\_макроса* и *последовательность\_символов* может быть любое количество пробелов.

Например, если вы хотите использовать слово UP в качестве значения 1 и слово DOWN в качестве значения 0, объявите такие две директивы `#define`.

```
#define UP 1
#define DOWN 0
```

Данные директивы вынудят компилятор подставлять 1 или 0 каждый раз, когда в файле исходного кода встретится слово UP или DOWN соответственно. Например, при выполнении инструкции

```
cout << UP << ' ' << DOWN << ' ' << UP + UP;
```

на экран будет выведено следующее:

```
1 0 2
```

Важно понимать, что макроподстановка — это просто замена идентификатора соответствующей строкой. Следовательно, если вам нужно определить стандартное сообщение, используйте код, подобный этому.

```
#define GETFILE "Введите имя файла"
// ...
```

Препроцессор заменит строкой "Введите имя файла" каждое вхождение идентификатора GETFILE. Для компилятора эта cout-инструкция

```
cout << GETFILE;
```

в действительности выглядит так.

```
cout << "Введите имя файла";
```

Никакой текстовой замены не произойдет, если идентификатор находится в строке, заключенной в кавычки. Например, при выполнении следующего кода

```
#define GETFILE "Введите имя файла"
// ...
cout << "GETFILE - это макроимя\n";
```

на экране будет отображена эта информация

GETFILE - это макроимя,

а не эта:

Введите имя файла - это макроимя

Если текстовая последовательность не помещается в строке, ее можно продолжить на следующей, поставив обратную косую черту в конце строки, как показано в этом примере.

```
#define LONG_STRING "Это очень длинная последовательность, \
которая используется в качестве примера."
```

Среди C++-программистов принято использовать для макроимен прописные буквы. Это соглашение позволяет с первого взгляда понять, что здесь используется макроподстановка. Кроме того, лучше всего поместить все директивы #define в начало файла или включить их в отдельный файл, чтобы не искать потом по всей программе.

Важно помнить, что в C++ предусмотрен еще один способ определения констант, который заключается в использовании спецификатора const. Однако многие программисты "пришли" в C++ из С-среды, где для этих целей обычно

использовалась директива `#define`. Поэтому вам еще часто придется с ней сталкиваться в C++-коде.

## Макроопределения, действующие как функции

Директиву `#define` можно использовать и по-другому: ее макроимя может иметь аргументы. В этом случае при каждом вхождении макроимени связанные с ним аргументы заменяются реальными аргументами, указанными в коде программы. Такие макроопределения действуют подобно функциям. Рассмотрим пример.

```
// Использование "функциональных" макроопределений.
```

```
#include <iostream>
using namespace std;

#define MIN(a,b)  ((a)<(b)) ? a : b

int main()
{
    int x, y;

    x = 10;
    y = 20;
    cout << "Минимум равен: " << MIN(x, y);

    return 0;
}
```

При компиляции этой программы выражение, определенное идентификатором `MIN(a,b)`, будет заменено, но `x` и `y` будут рассматриваться как операнды. Это значит, что `cout`-инструкция после компиляции будет выглядеть так.

```
cout << "Минимум равен: " << ((x)<(y)) ? x : y;
```

По сути, такое макроопределение представляет собой способ определить функцию, которая вместо вызова позволяет раскрыть свой код в строке.

Кажущиеся избыточными круглые скобки, в которые заключено макроопределение `MIN`, необходимы, чтобы гарантировать правильное восприятие компилятором заменяемого выражения. На самом деле дополнительные круглые скобки должны применяться практически ко всем макроопределениям, действующим подобно функциям. Нужно всегда очень внимательно относиться к определению таких макросов; в противном случае возможно получение неожиданных результатов. Рассмотрим, например, эту короткую программу, которая использует макрос для определения четности значения.

// Эта программа дает неверный ответ.

```
#include <iostream>
using namespace std;

#define EVEN(a) a%2==0 ? 1 : 0

int main()
{
    if(EVEN(9+1)) cout << "четное число";
    else cout << "нечетное число ";

    return 0;
}
```

Эта программа не будет работать корректно, поскольку в ней не обеспечена правильная подстановка значений. При компиляции выражение EVEN(9+1) будет заменено следующим образом.

$9+1\%2==0 ? 1 : 0$

Напомню, что оператор "%" имеет более высокий приоритет, чем оператор "+". Это значит, что сначала выполнится операция деления по модулю (%) для числа 1, а затем ее результат будет сложен с числом 9, что (конечно же) не равно 0. Чтобы исправить ошибку, достаточно заключить в круглые скобки аргумент a в макроопределении EVEN, как показано в следующей (исправленной) версии той же программы.

// Эта программа работает корректно.

```
#include <iostream>
using namespace std;

#define EVEN(a) (a)%2==0 ? 1 : 0

int main()
{
    if(EVEN(9+1)) cout << "четное число";
    else cout << "нечетное число ";

    return 0;
}
```

Теперь сумма 9+1 вычисляется до выполнения операции деления по модулю. В общем случае лучше всегда заключать параметры макроопределения в круглые скобки, чтобы избежать непредвиденных результатов, подобных описанному выше.

Использование макроопределений вместо настоящих функций имеет одно существенное достоинство: поскольку код макроопределения расширяется в строке, и нет никаких затрат системных ресурсов на вызов функции, скорость работы вашей программы будет выше по сравнению с применением обычной функции. Но повышение скорости является платой за увеличение размера программы (из-за дублирования кода функции).

Несмотря на то что макроопределения все еще встречаются в C++-коде, макросы, действующие подобно функциям, можно заменить спецификатором `inline`, который справляется с той же ролью лучше и безопаснее. (Вспомните: спецификатор `inline` обеспечивает вместо вызова функции расширение ее тела в строке.) Кроме того, `inline`-функции не требуют дополнительных круглых скобок, без которых не могут обойтись макроопределения. Однако макросы, действующие подобно функциям, все еще остаются частью C++-программ, поскольку многие C/C++-программисты продолжают использовать их по привычке.

## Директива `#error`

Директива `#error` представляет собой указание компилятору остановить компиляцию. Она используется в основном для отладки. Общий формат ее записи таков.

`#error сообщение`

Обратите внимание на то, что элемент `сообщение` не заключен в двойные кавычки. При встрече с директивой `#error` отображается заданное `сообщение` и другая информация (она зависит от конкретной реализации рабочей среды), после чего компиляция прекращается. Чтобы узнать, какую информацию отображает в этом случае компилятор, достаточно провести эксперимент.

## Директива `#include`

Директива препроцессора `#include` обязывает компилятор включить в файл, который ее содержит, либо стандартный заголовок, либо другой исходный файл, заданный в директиве `#include`. Имя стандартного заголовка заключается в угловые скобки, как показано в примерах, приведенных в этой книге. Например, эта директива

```
#include <fstream>
```

включает стандартный заголовок для реализации файловых операций ввода-вывода.

При включении другого исходного файла его имя может быть указано в двойных кавычках или угловых скобках. Например, следующие две директивы обзывают C++ прочитать и скомпилировать файл с именем `sample.h`:

```
#include <sample.h>
#include "sample.h"
```

Если имя файла заключено в угловые скобки, то поиск файла будет осуществляться в одном или нескольких специальных каталогах, определенных конкретной реализацией. Если же имя файла заключено в кавычки, поиск файла выполняется, как правило, в текущем каталоге (что также определено конкретной реализацией). Во многих случаях это означает поиск текущего рабочего каталога. Если заданный файл не найден, поиск повторяется с использованием первого способа (как если бы имя файла было заключено в угловые скобки). Поскольку маршрут поиска файлов определяется конкретной реализацией, то за детальным описанием этого процесса лучше обратиться к руководству пользователя, прилагаемому к вашему компилятору.

## Директивы условной компиляции

Существуют директивы, которые позволяют избирательно скомпилировать части исходного кода. Этот процесс, именуемый *условной компиляцией*, широко используется коммерческими фирмами по разработке программного обеспечения, которые создают и поддерживают множество различных версий одной программы.

### Директивы `#if`, `#else`, `#elif` и `#endif`

Главная идея директивы `#if` состоит в том, что если стоящее после нее выражение оказывается истинным, то будет скомпилирован код, расположенный между нею и директивой `#endif`; в противном случае данный код будет опущен. Директива `#endif` используется для обозначения конца блока `#if`.

Общая форма записи директивы `#if` выглядит так.

```
#if константное_выражение
    последовательность инструкций
#endif
```

Если `константное_выражение` является истинным, код, расположенный непосредственно за этой директивой, будет скомпилирован. Рассмотрим пример.

```
// Простой пример использования директивы #if.  
#include <iostream>  
using namespace std;  
  
#define MAX 100  
  
int main()  
{  
#if MAX>10  
    cout << "Требуется дополнительная память\n";  
#endif  
  
    // ...  
    return 0;  
}
```

При выполнении эта программа отобразит на экране сообщение Требуется дополнительная память, поскольку, как определено в программе, значение константы MAX больше 10. Этот пример иллюстрирует важный момент: выражение, которое стоит после директивы `#if`, вычисляется во время компиляции. Следовательно, оно должно содержать только идентификаторы, которые были предварительно определены, или константы. Использование же переменных здесь исключено.

Поведение директивы `#else` во многом подобно поведению инструкции `else`, которая является частью языка C++: она определяет альтернативу на случай не выполнения директивы `#if`. Чтобы показать, как работает директива `#else`, воспользуемся предыдущим примером, немного его расширив.

```
// Пример использования директив #if/#else.  
#include <iostream>  
using namespace std;  
  
#define MAX 6  
  
int main()  
{  
#if MAX>10  
    cout << "Требуется дополнительная память.\n";  
#else  
    cout << "Достаточно имеющейся памяти.\n";  
}
```

```
#endif
// ...
return 0;
}
```

В этой программе для имени MAX определено значение, которое меньше 10, поэтому `#if`-ветвь кода не скомпилируется, но зато скомпилируется альтернативная `#else`-ветвь. В результате отобразится сообщение достаточно имеющейся памяти..

Обратите внимание на то, что директива `#else` используется для индикации одновременно как конца `#if`-блока, так и начала `#else`-блока. В этом есть логическая необходимость, поскольку только одна директива `#endif` может быть связана с директивой `#if`.

Директива `#elif` эквивалентна связке инструкций `else-if` и используется для формирования многозначной "лестницы" `if-else-if`, представляющей несколько вариантов компиляции. После директивы `#elif` должно стоять константное выражение. Если это выражение истинно, скомпилируется расположенный за ним блок кода, и уже никакие другие `#elif`-выражения не будут тестироваться или компилироваться. В противном случае будет проверено следующее по очереди `#elif`-выражение. Вот как выглядит общий формат использования директивы `#elif`.

```
#if выражение
    последовательность инструкций
#elif выражение 1
    последовательность инструкций
#elif выражение 2
    последовательность инструкций
#elif выражение 3
    последовательность инструкций
// ...
#elif выражение N
    последовательность инструкций
#endif
```

Например, в этом фрагменте кода используется идентификатор `COMPILED_BY`, который позволяет определить, кем компилируется программа.

```
#define JOHN 0
#define BOB 1
```

```
#define TOM 2

#define COMPILED_BY JOHN

#if COMPILED_BY == JOHN
    char who[] = "John";
#elif COMPILED_BY == BOB
    char who[] = "Bob";
#else
    char who[] = "Tom";
#endif
```

Директивы `#if` и `#elif` могут быть вложенными. В этом случае директива `#endif`, `#else` или `#elif` связывается с ближайшей директивой `#if` или `#elif`. Например, следующий фрагмент кода совершенно допустим.

```
#if COMPILED_BY == BOB
    #if DEBUG == FULL
        int port = 198;
    #elif DEBUG == PARTIAL
        int port = 200;
    #endif
#else
    cout << "Боб должен скомпилировать код "
        << "для отладки вывода данных.\n";
#endif
```

## Директивы `#ifdef` и `#ifndef`

Директивы `#ifdef` и `#ifndef` предлагают еще два варианта условной компиляции, которые можно выразить как “если определено” и “если не определено” соответственно.

Общий формат использования директивы `#ifdef` таков.

```
#ifdef макроимя
    последовательность инструкций
#endif
```

Если макроимя предварительно определено с помощью какой-нибудь директивы `#define`, то последовательность инструкций, расположенная между директивами `#ifdef` и `#endif`, будет скомпилирована.

Общий формат использования директивы `#ifndef` таков.

```
#ifndef макроимя
    последовательность инструкций
#endif
```

Если *макроимя* не определено с помощью какой-нибудь директивы `#define`, то *последовательность инструкций*, расположенная между директивами `#ifdef` и `#endif`, будет скомпилирована. Как директива `#ifdef`, так и директива `#ifndef` может иметь директиву `#else` или `#elif`. Кроме того, директивы `#ifdef` и `#ifndef` можно вкладывать точно так же, как и директивы `#if`.

## Директива `#undef`

Директива `#undef` используется для удаления предыдущего определения некоторого макроимени. Ее общий формат таков.

```
#undef макроимя
```

Рассмотрим пример.

```
#define TIMEOUT 100
#define WAIT 0

// ...
```

```
#undef TIMEOUT
#undef WAIT
```

Здесь имена `TIMEOUT` и `WAIT` определены до тех пор, пока не выполнится соответствующая директива `#undef`. Основное назначение директивы `#undef` – разрешить локализацию макроимен для тех частей кода, в которых они нужны.

## Использование оператора `defined`

Помимо директивы `#ifdef` существует еще один способ выяснить, определено ли в программе некоторое макроимя. Для этого можно использовать директиву `#if` в сочетании с оператором времени компиляции `defined`. Например, чтобы узнать, определено ли макроимя `MYFILE`, можно использовать такие команды препроцессорной обработки.

```
#if defined MYFILE
```

или

```
#ifdef MYFILE
```

При необходимости, чтобы реверсировать условие проверки, можно предварить оператор `defined` символом “!”. Например, следующий фрагмент кода скомпилируется только в том случае, если макроимя DEBUG не определено.

```
#if !defined DEBUG
    cout << "Окончательная версия!\n";
#endif
```

## Директива `#line`

Директива `#line` используется для изменения содержимого псевдопеременных `__LINE__` и `__FILE__`, которые являются зарезервированными идентификаторами (макроименами). Псевдопеременная `__LINE__` содержит номер скомпилированной строки, а псевдопеременная `__FILE__` — имя компилируемого файла. Базовая форма записи директивы `#line` имеет следующий вид.

```
#line номер "имя_файла"
```

Здесь *номер* — это любое положительное целое число, а *имя\_файла* — любой допустимый идентификатор файла. Значение элемента *номер* становится номером текущей строки, а значение элемента *имя\_файла* — именем исходного файла. Имя файла — элемент необязательный. Директива `#line` используется, главным образом, в целях отладки и в специальных приложениях.

## Директива `#pragma`

Работа директивы `#pragma` зависит от конкретной реализации компилятора. Она позволяет выдавать компилятору различные инструкции, предусмотренные создателем компилятора. Общий формат ее использования таков.

```
#pragma имя
```

Здесь элемент *имя* представляет имя желаемой `#pragma`-инструкции. Если указанное имя не распознается компилятором, директива `#pragma` попросту игнорируется без сообщения об ошибке.

Для получения подробной информации о возможных вариантах использования директивы `#pragma` стоит обратиться к системной документации по используемому вами компилятору. Вы можете найти для себя очень полезную информацию. Обычно `#pragma`-инструкции позволяют определить, какие предупреждающие сообщения выдает компилятор, как генерируется код и какие библиотеки компонуются с вашими программами.

## Операторы препроцессора “#” и “##”

В C++ предусмотрена поддержка двух операторов препроцессора: “#” и “##”. Эти операторы используются совместно с директивой `#define`. Оператор “#” преобразует следующий за ним аргумент в строку, заключенную в кавычки. Рассмотрим, например, следующую программу.

```
#include <iostream>
using namespace std;

#define mkstr(s) # s

int main()
{
    cout << mkstr(Я в восторге от C++) ;

    return 0;
}
```

Препроцессор C++ преобразует строку

```
cout << mkstr(Я в восторге от C++) ;
в строку
cout << "Я в восторге от C++";
```

Оператор “##” используется для конкатенации двух лексем. Рассмотрим пример.

```
#include <iostream>
using namespace std;

#define concat(a, b) a ## b

int main()
{
    int xy = 10;

    cout << concat(x, y);

    return 0;
}
```

Препроцессор преобразует строку

```
cout << concat(x, y);
в строку
cout << xy;
```

Если эти операторы вам кажутся странными, помните, что они не являются операторами “повседневного спроса” и редко используются в программах. Их основное назначение — позволить препроцессору обрабатывать некоторые специальные ситуации.

## Зарезервированные макроимена

В языке C++ определено шесть встроенных макроимен.

```
--LINE--
--FILE--
--DATE--
--TIME--
--STDC--
__cplusplus
```

Макросы `--LINE--` и `--FILE--` описаны при рассмотрении директивы `#line` выше в этом приложении. Они содержат номер текущей строки и имя файла компилируемой программы.

Макрос `--DATE--` представляет собой строку в формате `месяц/день/год`, которая означает дату трансляции исходного файла в объектный код.

Время трансляции исходного файла в объектный код содержится в виде строки в макросе `--TIME--`. Формат этой строки следующий: `часы минуты секунды`.

Точное назначение макроса `--STDC--` зависит от конкретной реализации компилятора. Как правило, если макрос `--STDC--` определен, то компилятор примет только стандартный C/C++-код, который не содержит никаких нестандартных расширений.

Компилятор, соответствующий ANSI/ISO-стандарту C++, определяет макрос `--cplusplus` как значение, содержащее по крайней мере шесть цифр. “Нестандартные” компиляторы должны использовать значение, содержащее пять (или даже меньше) цифр.

# Приложение Б

## Использование устаревшего С++- компилиатора

При использовании современного компилятора у вас не должно быть проблем с компиляцией программ из этой книги. В этом случае вам вообще не понадобится информация, представленная в настоящем приложении. Как разъяснялось выше, программы, приведенные в этой книге, полностью соответствуют стандарту ANSI/ISO для C++ и могут быть скомпилированы практически любым современным C++-компилятором, включая Visual C++ (Microsoft) и C++ Builder (Borland).

Но если вы используете компилятор, созданный несколько лет назад, то при попытке скомпилировать наши примеры он может выдать целый список ошибок, не распознав ряд новых C++-средств. И в этом случае не стоит волноваться. Для того чтобы эти программы заработали со старыми компиляторами, нужно внести в них небольшие изменения. Чаще всего старые и новые C++-программы отличаются использованием двух средств: заголовков и пространств имен. Вот об этом и пойдет здесь речь.

Как упоминалось в модуле 1, инструкция `#include` включает в программу заданный заголовок. Для более ранних версий C++ под заголовками понимались файлы с расширением `.h`. Например, в старой C++-программе для включения заголовка `iostream` была бы использована следующая инструкция.

```
#include <iostream.h>
```

В этом случае в программу был бы включен заголовочный файл `iostream.h`. Таким образом, для того, чтобы включить в C++-программу заголовок в расчете на более "древний" компилятор, необходимо задать имя файла с расширением `.h`.

В новых C++-программах в соответствии со стандартом ANSI/ISO для C++ используются заголовки другого типа. Современные заголовки определяют не имена файлов, а стандартные идентификаторы, которые могут совпадать с таковыми, но не всегда. Современные C++-заголовки представляют собой абстракцию, которая попросту гарантирует включение в программу требуемой информации.

Поскольку современные заголовки необязательно являются именами файлов, они не должны иметь расширение `.h`. Они определяют имя заголовка, заключенного в угловые скобки. Вот, например, как выглядят два современных заголовка, поддерживаемых стандартом C++.

```
<iostream>
<fstream>
```

Чтобы преобразовать эти "новые" заголовки в "старые" заголовочные файлы, достаточно добавить расширение `.h`.

Включая современный заголовок в программу, необходимо помнить, что его содержимое относится к пространству имен `std`. Как упоминалось выше, пространство имен — это просто декларативная область. Ее назначение — локализовать имена идентификаторов во избежание коллизий с именами. В старых версиях C++ имена библиотечных функций помещаются в глобальное пространство имен, а не в про-

пространство имен `std`, используемое современными компиляторами. Таким образом, работая со старым компилятором, не нужно использовать эту инструкцию:

```
using namespace std;
```

В действительности большинство старых компиляторов вообще не воспримут инструкцию `using namespace`.

## Два простых изменения

Если ваш компилятор не поддерживает пространства имен и новые заголовки, он выдаст одно или несколько сообщений об ошибках при попытке скомпилировать первые несколько строк программ, приведенных в данной книге. В этом случае в программы необходимо внести только два простых изменения: использовать заголовок старого типа и удалить `namespace`-инструкцию. Например, замените эти инструкции

```
#include <iostream>
using namespace std;
```

такой.

```
#include <iostream.h>
```

Это изменение преобразует "новую" программу в "старую". Поскольку при использовании "старого" заголовка в глобальное пространство имен считывается все содержимое заголовочного файла, необходимость в использовании `namespace`-инструкции отпадает. После внесения этих изменений программу можно скомпилировать с помощью старого компилятора.

Иногда приходится вносить и другие изменения. C++ наследует ряд заголовков из языка С. Язык С не поддерживает современный стиль C++-заголовков, используя вместо них заголовочные `.h`-файлы. Для разрешения обратной совместимости стандарт C++ по-прежнему поддерживает заголовочные С-файлы. Однако стандарт C++ также определяет современные заголовки, которые можно использовать вместо заголовочных С-файлов. В C++-версиях стандартных С-заголовков к имени С-файла просто добавляется префикс 'c' и опускается расширение `.h`. Например, C++-заголовком для файла `math.h` служит заголовок `<cmath>`, а для файла `string.h` — заголовок `<cstring>`. Несмотря на то что в C++-программу разрешено включать заголовочный С-файл, против такого подхода у разработчиков стандарта есть существенные возражения (другими словами, это не рекомендовано). Поэтому в настоящей книге используются современные C++-заголовки во всех инструкциях `#include`. Но если ваш компилятор не поддерживает C++-эквиваленты для С-заголовков, просто замените их "старыми" заголовочными файлами.

# Предметный указатель

## Символы

#define, директива 640  
 #elif, директива 647  
 #endif, директива 645  
 #егор, директива 644  
 #if, директива 645  
 #ifdef, директива 648  
 #ifndef, директива 648  
 #include, директива 644, 654  
 #pragma, директива 650  
 #undef, директива 649  
 \_\_cplusplus, макрос 652  
 \_\_DATE\_\_, макрос 652  
 \_\_FILE\_\_, макрос 652  
 \_\_LINE\_\_, макрос 652  
 \_\_STDC\_\_, макрос 652  
 \_\_TIME\_\_, макрос 652  
 FILE 650  
 LINE 650

## А

Абстрактный класс 518  
 Аргумент 63  
 Аргументы функции  
     по умолчанию 292  
 Аргументы функций 215

## Б

Байт-код 28  
 Библиотеки C++ 64  
 Блок 225  
 Блок кода 56

## В

Виртуальная машина Java 28  
 Виртуальные функции 502  
 Вложенный класс 376  
 Выражение  
     условное 112  
 Выражения 101

## Д

Декремент 90  
 Деструктор 368  
 Директива  
     #define 640  
     #elif 647  
     #endif 645  
     #егор 644  
     #if 645  
     #ifdef 648  
     #ifndef 648  
     #include 644, 654  
     #line 650  
     #pragma 650  
     #undef 649

Директивы препроцессора 640  
 Дополнительный код 73

## З

Заголовки 250  
 Заголовок  
     <cctype> 176  
     <cmath> 78, 127  
     <cstdio> 171  
     <cstdlib> 64, 111  
     <cstring> 173  
     <fstream> 545  
     <iostream> 37, 522, 524, 557  
     <typeinfo> 629

Заголовочный файл  
     <iostream.h> 522

## И

Идентификатор 66  
 Индекс 159  
 Инициализация  
     массивов 177  
     переменных 87  
 Инкапсуляция 30  
 Инкремент 90  
 Инструкция  
     cin 45

continue 145  
 cout 45  
 do-while 136  
 for 54, 125  
 goto 152  
 if 52, 110  
 switch 117  
 while 134  
**Исключение** 572  
 bad\_cast 634  
 bad\_typeid 632  
**Исключительная ситуация** 572

**K**

**Класс** 30, 350  
 basic\_ios 525  
 basic\_iostream 525  
 basic\_istream 525  
 basic\_ostream 525  
 basic\_streambuf 525  
 fstream 546  
 ifstream 546  
 ios 533, 547  
 ios\_base 525  
 ofstream 546  
 string 84, 169  
 type\_info 629  
 абстрактный 518  
 базовый 466  
 вложенный 376  
 полиморфный 503, 630  
 производный 466  
**Классы**  
 обобщенные 594  
**Ключевые слова C++** 66  
**Комментарий** 36  
**Компаранд** 255  
**Компилятор**  
 C++ Builder 34  
 Visual C++ 34  
**Константы** 83  
**Конструктор** 368, 480  
 копии 405, 411  
 параметризованный 370  
**Куча** 605  
**Кэш** 315

**Л**

**Литерал** 83  
 вещественный 83  
 восьмеричный 84  
 символьный 83  
 строковый 84, 170  
 целочисленный 83  
 шестнадцатеричный 84

**M**

**Макроимена** 652  
**Макроимя** 640  
**Макроподстановка** 640  
**Макрос**  
 \_\_cplusplus 652  
 \_\_DATE\_\_ 652  
 \_\_FILE\_\_ 652  
 \_\_LINE\_\_ 652  
 \_\_STDC\_\_ 652  
 \_\_TIME\_\_ 652  
**Манипулятор**  
 boolalpha 541  
 dec 541  
 endl 541  
 ends 541  
 fixed 541  
 flush 541  
 hex 541  
 internal 541  
 left 541  
 noboolalpha 541  
 noshowbase 541  
 noshowpoint 541  
 noshowpos 541  
 noskipws 541  
 nounitbuf 541  
 nouppercase 541  
 oct 541  
 resetiosflags() 541  
 right 541  
 scientific 541  
 setbase() 541  
 setfill() 541  
 setiosflags() 541, 542  
 setprecision() 541  
 setw() 541

- showbase 541  
showpoint 541  
showpos 541  
skipws 541  
unitbuf 541  
uppercase 541  
ws 541, 543
- Манипуляторные функции 543  
Манипуляторы 540  
Массив 158, 193  
    двумерный 163  
    инициализация 177  
    многомерный 165  
    одномерный 158  
    строк 182
- Массивы  
    объектов 388  
    указателей 203
- Метка 152  
Метод 30  
Многоуровневая непрямая адресация 206
- Модификатор  
    const 635  
    inline 377  
    long 71  
    short 71  
    signed 71  
    static 310, 312  
    unsigned 71  
    volatile 635
- Н**
- Наследование 32, 465  
    виртуальных функций 506  
    синтаксис 468
- О**
- Обобщенные  
    классы 594  
    функции 587
- Объединения 424  
    анонимные 426
- Объект 30
- Объектно-ориентированное  
    программирование 25, 29
- Объявление  
    класса 478  
    опережающее 419
- ООП 25
- Оператор  
    != 629  
    & 185  
    \* 185  
    == 629  
    "##" 651  
    #" 651  
    "запятая" 128, 340  
    "знак вопроса" 339  
    "исключающее ИЛИ" (XOR) 96  
    "точка" 353  
    const\_cast 635  
    defined 649  
    delete 605  
    dynamic\_cast 633  
    new 605  
    reinterpret\_cast 635  
    sizeof 343  
    static\_cast 635  
    typeid 629  
    XOR 324, 327  
    ввода 44, 526  
    вывода 38, 526  
    декремента 56  
    деления по модулю 89  
    дополнения до 1 328  
    И, поразрядный 324  
    ИЛИ, поразрядный 326  
    инкремента 55, 436  
    исключающее ИЛИ 324, 327  
    НЕ 328  
    присваивания 41  
    разрешения контекста 358, 419  
    разрешения области видимости 358, 419
- Операторы 89  
    арифметические 89  
    декремента 90  
    инкремента 90  
    логические 92  
    отношений 92  
    поразрядные 322

приведения типов 633  
присваивания, составные 99, 342  
сдвига 330  
Операция  
    приведения типов 102  
Операторющее объявление 419  
Очередь 382

**П**

Параметры  
    сырьевые 266  
Перегрузка  
    конструкторов 398  
    операторов 430  
    операторов ввода-вывода 526  
Перегрузка функций 277  
Переменная 40  
Переменные  
    инициализация 87  
Перечисление  
    fmtflags 533  
    iostate 568  
    openmode 547  
    seekdir 565  
Перечисления 318  
Поле  
    сборное:adjustfield 535  
    сборное:basefield 534  
    сборное:floatfield 535  
Полиморфизм 31, 502  
Полиморфный класс 503, 630  
Порожденная функция 590  
Поток 523  
    cerr 524  
    cin 524  
    clog 524  
    cout 524  
    wcerr 524  
    wcin 524  
    wclog 524  
    wcout 524  
    двоичный 523  
    текстовый 523  
Предупреждения 39  
Препроцессор 639  
Промежуточный язык Microsoft 28

Пространство имен 614  
    std 37, 522, 622  
    неименованное 622  
Прототип функций 248  
Прототип функции 214  
Псевдокодом 28  
Псевдопеременные  
    FILE 650  
    LINE 650

**Р**

Расширение типа 102  
Реализация 590  
Рекурсия 250  
Ритчи, Дэвис 23

**С**

Символ  
    новой строки 46  
Система счисления  
    восьмеричная 84  
    шестнадцатеричная 84  
Сортировка массива 166  
Спагетти-код 23  
Специализация  
    класса:явная 597  
    функции 590  
    функции:явная 591  
Спецификатор  
    inline 644  
    private 474  
    public 351, 469, 474  
    компоновки 310  
Спецификатор класса памяти  
    auto 308  
    extern 308  
    register 315  
Спецификатор типа  
    const 304  
    volatile 307  
Ссылки  
    на производные типы 502  
    независимые 275  
Стек 382  
Страуструп, Бирн 25, 92  
Строка 38, 84

# 660 Предметный указатель

Строки 169

Строковый литерал 170

Структурированное программирование  
23

Структуры 421

## T

Тип

    bool 102  
    off\_type 565  
    pos\_type 567  
    streamsize 538

Тип данных 41

    bool 78  
    char 75  
    double 77  
    float 77  
    int 72  
    long double 78  
    void 80  
    wchar\_t 76

## У

Указатели

    на объекты 391  
    на производные типы 501

Указатель 158, 183

    this 428

Управляющие последовательности 85

Условное выражение 112

## Ф

Файл 523

Факториал числа 251

Флаг

    boolalpha 534  
    dec 534  
    fixed 534  
    hex 534  
    internal 534  
    left 534  
    oct 534  
    right 534  
    scientific 534  
    showbase 534  
    showpoint 534

showpos 534

skipws 534

unithbuf 534

uppercase 534

Флаг знака 73

Функции

    “друзья” 415  
    виртуальные 502  
    встраиваемые 376  
    доступа 471  
    манипуляторные 543  
    обобщенные 587  
    перегрузка 277  
    чисто виртуальные 515

Функция 37, 62, 212

    abs() 63  
    atof() 246  
    bad() 569  
    before() 629  
    clear() 569  
    close() 549  
    eof() 559  
    fail() 569  
    fill() 538  
    flags() 536  
    flush() 559  
    free() 607  
    get() 552, 557  
    getline() 558  
    gets() 171  
    good() 569  
    isalpha() 177  
    islower() 177, 195  
    isupper() 177  
    main() 37, 243  
    malloc() 607  
    name() 629  
    open() 546  
    operator 430  
    peek() 559  
    pow() 104  
    precision() 538  
    put() 552  
    putback() 559  
    qsort() 169, 255  
    rand() 110

rdstate() 568  
 read() 554  
 seekg() 565, 567  
 seekp() 565, 567  
 setf() 535  
 sqrt() 126  
 strcat() 173  
 strcmp() 173  
 strcpy() 173  
 strlen() 174  
 tolower() 177  
 toupper() 176  
 unsetf() 536  
 width() 538  
 write() 554  
 операторная 430  
 порожденная 590  
 шаблонная 590

**П**

Цикл

do-while 136  
 for 54, 125  
 while 134  
 бесконечный 130  
 вложенный 151

**Ч**

Чисто виртуальная функция 515

**III**

Шаблон 587

Шаблонная функция 590

**A**

Array 158  
 atof() 246  
 auto 231  
 auto, спецификатор 308

**B**

B 23  
 bad() 569  
 bad\_cast, исключение 634  
 basic\_ios, класс 525

basic\_iostream, класс 525  
 basic\_istream, класс 525  
 basic\_ostream, класс 525  
 basic\_streambuf, класс 525  
 BCPL 23  
 before() 629  
 bool 71  
 boolalpha, флаг 534  
 break 143  
 Bytecode 28

**C**

C 22  
 C# 27  
 C++ Builder 26, 34  
 Call-by-reference 262  
 Call-by-value 262  
 Cast 102  
 catch 572  
 cerr 524  
 char 71, 72, 73  
 cin 44, 524  
 class 350  
 clear() 569  
 clog 524  
 close() 549  
 CLR (Common Language Runtime) 28  
 Common Language Runtime 28  
 const, спецификатор типа 304  
 const\_cast, оператор 635  
 continue 145  
 cout 38, 524

**D**

dec, флаг 534  
 delete 605  
 delete, оператор 605  
 do-while 136  
 double 71  
 dynamic\_cast, оператор 633

**E**

enum 318  
 eof() 559  
 extern 308

**F**

fail() 569  
false, константа 78  
FIFO 382  
fill() 538  
FILO 382  
First in First Out 382  
First in Last Out 382  
fixed, флаг 534  
flags() 536  
float 71  
flush() 559  
fmtflags, перечисление 533  
for, инструкция 54  
for, цикл 125  
FORTRAN 24  
free() 607  
friend 415  
Function overloading 277

**G**

Generated function 590  
get() 552, 557  
getline() 558  
gets() 171  
good() 569  
goto 152  
GUI 36

**H**

Heap 604  
hex, флаг 534

**I**

IDE (Integrated Development Environment) 34  
if 52, 110  
if-else-if 115  
inline 644  
inline, модификатор 377  
Inline function 377  
Instantiating 590  
int 41, 71, 72, 73  
Integral promotion 102  
Integrated Development Environment 34

internal, флаг 534  
ios, класс 533  
ios\_base, класс 525  
iostate, перечисление 568  
isalpha() 177  
islower() 177, 195  
isupper() 177

**J**

Java 27  
Java Virtual Machine 28  
JVM (Java Virtual Machine) 28

**L**

left, флаг 534  
long, модификатор 71  
long double 72, 73  
long int 72, 73

**M**

main() 37, 243  
malloc() 607  
Manipulator 533  
MFC 509  
Microsoft Foundation Classes 509  
Microsoft Intermediate Language 28  
MSIL (Microsoft Intermediate Language) 28  
Multiple indirection 206

**N**

name() 629  
Namespace 37  
namespace 614  
new, оператор 605

**O**

ost, флаг 534  
open() 546  
openmode, перечисление 547  
Operator 89  
operator 430  
overload 65

**P**

Pascal 23  
 peek() 559  
 Plain Old Data 424  
 POD 424  
 pow() 104  
 precision() 538  
 Preprocessor 640  
 private 474  
 protected, модификатор доступа 477  
 public 469, 474  
 public, спецификатор 351  
 put() 552  
 putback() 559

**Q**

qsort() 169, 255  
 Queue 382  
 QuickSort, алгоритм сортировки 254

**R**

rand() 110  
 rdstate() 568  
 read() 554  
 Reference parameter 267  
 register, спецификатор 315  
 reinterpret\_cast, оператор 635  
 return 38  
 right, флаг 534  
 RTTI 628

**S**

scientific, флаг 534  
 seekdir, перечисление 565  
 seekg() 565, 567  
 seekp() 565, 567  
 setf() 535  
 short, модификатор 71  
 short int 72, 73  
 showbase, флаг 534  
 showpoint, флаг 534  
 showpos, флаг 534  
 signed, модификатор 71  
 signed char 72, 73  
 signed int 72, 73

signed long int 72, 73  
 signed short int 72, 73  
 sizeof 343  
 skipws, флаг 534  
 sqrt() 126  
 Stack 382  
 Standard C++ 26  
 Standard Template Library 26  
 static 623  
 static, модификатор 310, 312  
 static\_cast, оператор 635  
 std 522  
 std, пространство имен 37  
 STL 26  
 strcat() 173  
 strcmp() 173  
 strcpy() 173  
 Stream 523  
 streamsize, тип 538  
 strlen() 174  
 struct 421  
 switch 117

**T**

template 588, 595  
 template<> 593, 597  
 this 428  
 throw 572  
 throw-выражение 583  
 tolower() 177  
 toupper() 176  
 true, константа 78  
 try 572  
 type\_info, класс 629  
 typedef 322  
 typeid 629  
 typename 588  
 Type promotion 102

**U**

union 424  
 unitbuf, флаг 534  
 UNIX 23  
 unsetf() 536  
 unsigned, модификатор 71  
 unsigned char 72, 73

unsigned int 72, 73  
unsigned long int 72, 73  
unsigned short int 72, 73  
uppercase, флаг 534  
using 37, 619

**V**

virtual 502  
Visual C++ 26, 34  
Visual Studio .NET, среда разработки  
программ 35  
void 71, 212, 214  
volatile, спецификатор 307  
volatile, спецификатор типа 307

**W**

wcerr 524  
wchar\_t 71  
wcin 524  
wclog 524  
wcout 524  
while 134  
width() 538  
write() 554

*Научно-популярное издание*

Герберт Шилдт

# C++:руководство для начинающих

## 2-е издание

Литературный редактор

Верстка

Художественный редактор

Корректоры

*О.Ю. Белозовская*

*О.В. Линник*

*В.Г. Павлютин*

*З.В. Александрова, Л.А. Гордиенко,*

*О.В. Мишутина, Л.В. Чернокозинская*

Издательский дом "Вильямс"  
101509, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать с готовых диапозитивов 14.06.05.

Формат 70 × 100<sup>1/16</sup>. Печать офсетная.

Гарнитура Times. Усл. печ. л. 54, 18.

Уч.-изд. л. 23,5. Тираж 3 000 экз. Заказ № 201.

ОАО «Санкт-Петербургская типография № 6».  
191144, Санкт-Петербург, ул. Моисеенко, 10.  
Телефон отдела маркетинга 271-35-42.

# C++: базовый курс, третье издание

Герберт Шилдт

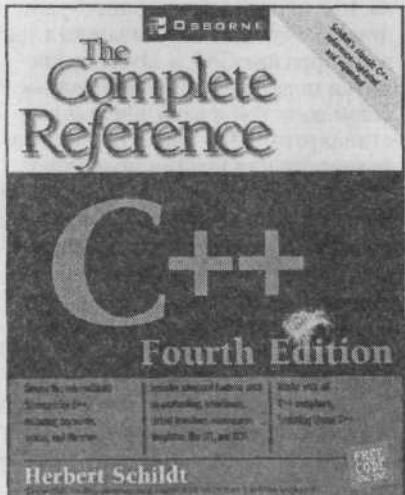


ISBN 5-8459-0768-3  
в продаже

В этой книге описаны все основные средства языка C++: от элементарных понятий до супервозможностей. После рассмотрения основ программирования на C++ (переменных, операторов, инструкций управления, функций, классов и объектов) читатель освоит такие более сложные средства языка, как механизм обработки исключительных ситуаций (исключений), шаблоны, пространства имён, динамическая идентификация типов, стандартная библиотека шаблонов (STL), а также познакомится с расширенным набором ключевых слов, используемых в программировании для .NET. Автор справочника — общепризнанный авторитет в области программирования на языках С и С++, Java и С# — включил в текст своей книги и советы программистам, которые позволят повысить эффективность их работы.

# ПОЛНЫЙ СПРАВОЧНИК по C++, 4-Е ИЗДАНИЕ

Герберт Шилдт



[www.williamspublishing.com](http://www.williamspublishing.com)

В четвертом издании "Полного справочника по C++" полностью описаны и проиллюстрированы все ключевые слова, функции, классы и свойства языка C++, соответствующие стандарту ANSI/ISO. Информацию, изложенную в книге, можно использовать во всех современных средах программирования. Освещены все аспекты языка C++, включая его основу — язык C. Справочник состоит из пяти частей:

- 1) подмножество C;
  - 2) язык C++;
  - 3) библиотека стандартных функций;
  - 4) библиотека стандартных классов;
  - 5) приложения на языке C++.
- Для широкого круга программистов.

# ПОЛНЫЙ СПРАВОЧНИК по С, 4-Е ИЗДАНИЕ

Герберт Шилдт



[www.williamspublishing.com](http://www.williamspublishing.com)

в продаже

В данной книге, задуманной как справочник для всех программистов, работающих на языке С, независимо от их уровня подготовки, подробно описаны все аспекты языка С и его библиотеки стандартных функций. Главный акцент сделан на стандарте ANSI/ISO языка С. Приведено описание как стандарта С89, так и С99. В книге особое внимание уделяется учету характеристик трансляторов, среды программирования и операционных систем, использующихся в настоящее время. Уже в самом начале подробно представлены все средства языка С, такие как ключевые слова, инструкции препроцессора и другие. Вначале описывается главным образом С89, а затем приводится подробное описание новых возможностей языка, введенных стандартом С99. Такая последовательность изложения позволяет облегчить практическое программирование на языке С, так как в настоящее время именно эта версия для большинства программистов представляется как "собственно С", к тому же это самый распространенный в мире язык программирования. Кроме того, эта последовательность изложения облегчает освоение С++, который является надмножеством С89. В описании библиотеки стандартных функций С приведены как функции стандарта С89, так и С99, причем функции, введенные стандартом С99, отмечены специально. Все книги Шилдта программисты всегда горячо любили за наличие содержательных, нетривиальных примеров. В книге рассматриваются наиболее важные и распространенные алгоритмы и приложения, необходимые для каждого программиста, а также применение методов искусственного интеллекта и программирование для Windows 2000.

# ПОЛНЫЙ СПРАВОЧНИК ПО C#

Герберт Шилдт



Созданный компанией Microsoft для поддержки среды .NET Framework, язык C# опирается на богатое наследие в области программирования. C# — прямой потомок двух самых успешных в мире компьютерных языков: C++ и Java. При изложении материала о языке C# труднее всего понять, когда следует поставить точку. Сам по себе язык C# очень большой, а библиотека классов C# еще больше. Чтобы читателю было легче совладать с таким огромным объемом материала, эта книга была разделена на три части, посвященных языку C#, его библиотеке и применению этого языка. Для работы с книгой никакого предыдущего опыта в области программирования не требуется. Если вы уже знакомы с C++ или Java, то с освоением C# у вас не будет никаких проблем, поскольку у C# много общего с этими языками. Если вы не считаете себя опытным программистом, книга поможет изучить C#, но для этого вам придется тщательно разобраться в примерах, приведенных в каждой главе.

[www.williamspublishing.com](http://www.williamspublishing.com)

ISBN 5-8459-0563-X

в продаже

# Искусство программирования на Java

Герберт Шилдт, Джеймс Холмс



ISBN 5-8459-0786-1  
в продаже

**Э**та книга отличается от множества других книг по языку Java. В то время как другие книги обучаются основам языка, эта книга показывает, как использовать язык наиболее эффективно, с большой пользой и отдачей для решения запутанных задач программирования. На страницах книги постепенно раскрывается мощь, универсальность и элегантность языка Java. Как и можно ожидать, несколько описанных приложений связаны непосредственно с Internet. Многие главы посвящены анализу кода. В них показаны потрясающие возможности языка Java при создании приложений для Internet. Легкость, с которой эти программы могут быть написаны на языке Java, подтверждает гибкость и элегантность языка. В каждой главе рассматриваются фрагменты кода, которые без изменений вы сможете использовать в своих программах. Например, синтаксический анализатор может послужить отличным дополнением для многих разработок. Однако наибольшую пользу от этих программ можно получить, если на их основе создавать собственные приложения. Например, Web-червь, подробное описание которого приводится в книге, может послужить основой для разработки архиватора Web-сайта или детектора разрыва связи. Книга рассчитана на студентов, преподавателей и специалистов в области компьютерных технологий.

# Параллельное и распределенное программирование с использованием C++

Камерон Хьюз, Трейси Хьюз



ISBN 5-8459-0686-5

в продаже

В этой книге представлен архитектурный подход к распределенному и параллельному программированию с использованием языка C++. Здесь описаны простые методы программирования параллельных виртуальных машин и основы разработки кластерных приложений.

Многолетний опыт авторов книги в области разработки программного обеспечения убедил их в том, что для успешного проектирования программных средств и эффективной их реализации без универсальности применяемых средств уже не обойтись. Чтобы эффективно решать задачи, стоящие перед современным программистом, необходимо сочетать различные программные и инженерные подходы. Например, для решения проблем "гонки" данных и синхронизации доступа к ним можно использовать методы объектно-ориентированного программирования. При многозадачном и многопоточном управлении авторы считают наиболее перспективной агентно-ориентированную архитектуру, а для минимизации затрат на обеспечение связей между объектами предлагают применять методологию "классной доски". Помимо объектно-ориентированного, агентно-ориентированного и AI-ориентированного программирования, они используют параметризованное программирование для реализации обобщенных C++-алгоритмов, которые применяются именно там, где нужен параллелизм.

Эта книга не только научит писать программные компоненты, предназначенные для совместной работы в сетевой среде, но и послужит надежным "путеводителем" по стандартам для программистов, которые занимаются многозадачными и многопоточными приложениями.

Основные темы книги:

- возможность использования агентов и технологии "классной доски" для упрощения параллельного программирования;
- объектно-ориентированные подходы к многозадачности и многопоточности;
- использование средств языка UML в разработке проектов, требующих применения параллельного и распределенного программирования;
- новый стандарт POSIX/UNIX IEEE для библиотеки Pthreads.

Эта книга адресована программистам, проектировщикам и разработчикам программных продуктов, а также научным работникам, преподавателям и студентам, которых интересует введение в параллельное и распределенное программирование с использованием языка C++.

# C++ для "чайников" 5-е издание

С. Дэвис



ISBN 5-8459-0723-3

В продаже

**К**нига представляет собой введение в язык программирования C++. Основное отличие данной книги от предыдущих изданий C++ для "чайников" в том, что это издание не требует от читателя каких-либо дополнительных знаний, в то время как предыдущие издания опирались на знание читателем языка программирования С. Книга отличается также тем, что несмотря на простоту изложения материала, он подан достаточно строго. Поэтому, изучив основы программирования на C++, читателю не придется пересматривать свои знания при дальнейшем изучении языка.

Книга отличается широким охватом тем — от простейших объявлений переменных до концепций объектно-ориентированного программирования, вопросов перегрузки операторов и множественного наследования, причем весь материал снабжен множеством практических примеров.

Эта книга не учит программированию в Windows или созданию красивого интерфейса двумя движениями мышью; изложенный в ней материал не привязан к какому-то определенному компилятору или операционной системе. Она вряд ли будет полезна профессиональному программисту, но если ваша цель — глубокое знание языка программирования и вы не знаете, с чего начать — эта книга для вас.

*Основы основ — проще простого!*

# C++

## руководство для начинающих Второе издание

Профессиональный программист Герберт Шилдт усовершенствовал один из своих бестселлеров — обучающее пособие по C++, языку, который позволяет создавать высокоеффективные программы. Теперь научиться программировать на C++ стало еще легче! Герберт на простых примерах доходчиво разъясняет основные понятия C++: инструкции управления, типы данных, массивы, строки, классы, объекты и функции. После освоения элементарных вещей вы легко перейдете к рассмотрению и таких более сложных тем, как перегрузка операторов, механизм обработки исключительных ситуаций (исключений), наследование, полиморфизм, виртуальные функции, средства ввода-вывода и шаблоны. Вы не ошиблись, решив сделать свои первые шаги в программировании на C++ под руководством такого "гига", как Герберт Шилдт!

### Об авторе

Герберт Шилдт — всемирно известный автор книг по программированию. Среди них такие бестселлеры, как *Полный справочник по C++, Полный справочник по C#, Искусство программирования на Java*.



Чтобы облегчить процесс изучения языка C++, в книгу включены такие разделы:



**МОДУЛИ.** Книга разделена на логически организованные модули (главы), рассчитанные на самостоятельное освоение материала.



**ВАЖНО!** Каждый модуль начинается с перечня основных тем, которые необходимо изучить, прежде чем двигаться дальше.



**ТЕСТ ДЛЯ САМОКОНТРОЛЯ ПО МОДУлю.** В конце каждого модуля предлагается тест для проверки степени освоения материала, состоящий из вопросов, требующих коротких ответов, выбора правильного варианта из предложенных либо написания небольших программ.



**СПРОСИМ У ОПЫТНОГО ПРОГРАММИСТА.** В разделах "вопросов и ответов" вы найдете дополнительную информацию и полезные советы.



**ВОПРОСЫ ДЛЯ ТЕКУЩЕГО КОНТРОЛЯ.** Разделы блиц-опросов позволят вам быстро проверить, насколько хорошо вы разобрались в конкретной теме.



**ПРОЕКТЫ.** Именно с помощью практических упражнений лучше всего понять, как применяются знания (которые "важно не пропустить"), полученные в каждом модуле.



**КОММЕНТАРИИ.** Примеры программ снабжены многочисленными комментариями, в которых описываются демонстрируемые методы программирования.

*The McGraw-Hill Companies*

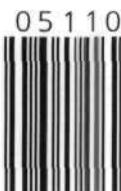


**OSBORNE**

[www.williamspublishing.com](http://www.williamspublishing.com)  
[www.osborne.com](http://www.osborne.com)

C++ / ПРОГРАММИРОВАНИЕ

ISBN 5-8459-0840-X



9 785845 908407