

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.

Содержание

Об авторе	15
Введение	17
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
Модуль 1. Основы C++	21
1.1. Из истории создания C++	22
Язык C: 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
Компилирование программы	34
Выполнение программы	35
Построчный “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 управления 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
Модуль 2. Типы данных и операторы	69
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

Модуль 3. Инструкции управления	109
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
Модуль 4. Массивы, строки и указатели	157
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
Модуль 5. Введение в функции	211
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
Модуль 6. О функциях подробнее	261
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. Ссылочные 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
Модуль 7	303
Еще о типах данных и операторах	303
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
Модуль 8. Классы и объекты	349
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
Модуль 9. О классах подробнее	397
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 операндов	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
Модуль 10. Наследование	465
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
Модуль 11. C++-система ввода-вывода	521
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
Модуль 12. Исключения, шаблоны и кое-что еще	571
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 исключения	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
Приложение А. Препроцессор	639
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
Приложение Б. Использование устаревшего C++-компилятора	653
Два простых изменения	655
Предметный указатель	656

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 программировали, 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 C, 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 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 третьи 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 попадетсся 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 объявления 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 компоуются 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++-компоновку. 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` объявляется 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: "
```

312 Модуль 7. Еще о типах данных и операторах

```
        << 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;
}
```

314 Модуль 7. Еще о типах данных и операторах

```
}

// ----- 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 «держат 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 супердостоинством, 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 C-программистами, поскольку в C 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 максимально возможной 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

1. Локальная переменная, объявленная с использованием модификатора `static`, _____ свое значение между вызовами функции.
2. Спецификатор `extern` используется для объявления переменной без ее определения. Верно ли это?
3. Какой спецификатор служит для компилятора запросом на оптимизацию обработки переменной с целью повышения быстродействия программы?*

1. Локальная переменная, объявленная с использованием модификатора `static`, сохраняет свое значение между вызовами функции.
2. Верно, спецификатор `extern` действительно используется для объявления переменной без ее определения.
3. 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 C (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 };
```

320 Модуль 7. Еще о типах данных и операторах

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;
```

Не забывайте, 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++ ориентирован 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 операндов 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
<code>&</code>	Поразрядное И (AND)
<code> </code>	Поразрядное ИЛИ (OR)
<code>^</code>	Поразрядное исключающее ИЛИ (XOR)
<code>~</code>	Дополнение до 1 (унарный оператор НЕ)
<code>>></code>	Сдвиг вправо
<code><<</code>	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;`

324 Модуль 7. Еще о типах данных и операторах

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 операндов (нулей и единиц).

p	q	p & q	p q	p ^ q	~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 операндов; в противном случае результат принимает значение ЛОЖЬ (0).

Поразрядный оператор И можно представить как способ подавления битовой информации. Это значит, что 0 в любом операнде 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)
{
```

326 Модуль 7. Еще о типах данных и операторах

```
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:

```

0111 1111
^ 1011 1001
-----
1100 0110

```

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 исходное 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 (храняемое в переменной A) представляет собой двоичный код 1001 0110, то в результате операции $\sim A$ 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: $\&$, $|$, \wedge и \sim .
2. Верно, поразрядный оператор работает на побитовой основе.
3. $x \ll 2$

циклического 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 “вдвинут” 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 конца.
    }

    /* 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 впра-
во:\n";
    for(int i=0; i < 8; i++) {
        ch = rrotate(ch, 1);
        show_binary(ch);
    }

    cout << "Результат 8-кратного циклического сдвига вле-
во:\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. */
```

7

Еще о типах данных и операторах

340 Модуль 7. Еще о типах данных и операторах

```
#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 дру-
    кое”.

    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 операндами.
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 сгенерировать 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 (type)
```

```
sizeof имя_переменной
```

344 Модуль 7. Еще о типах данных и операторах

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;
}
```

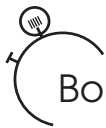
Оператор `sizeof` можно применить к любому типу данных. Например, в случае применения к массиву он возвращает количество байтов, занимаемых массивом. Рассмотрим следующий фрагмент кода.

```
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

1. Покажите, как присвоить переменным `t1`, `t2` и `t3` значение 10, используя только одну инструкцию присваивания.
2. Как по-другому можно переписать эту инструкцию?
`x = x + 100;`
3. 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).

1. `t1 = t2 = t3 = 10;`
2. `x += 100;`
3. Оператор `sizeof` возвращает размер заданной переменной или типа в байтах.

Таблица 7.2. Приоритет C++-операторов

Наивысший	() [] -> :: . ! ~ ++ -- - * & sizeof new delete typeid операторы приведения типа . * -> * * / % + - << >> < <= > >= == != & ^ && ?: = += -= *= /= %= >>= <<= &= ^= =
Низший	,

Тест для самоконтроля по модулю 7

1. Покажите, как объявить `int`-переменную с именем `test`, которую невозможно изменить в программе. Присвойте ей начальное значение 100.
2. Спецификатор `volatile` сообщает компилятору о том, что значение переменной может быть изменено за пределами программы. Так ли это?
3. Какой спецификатор применяется в одном из файлов многофайловой программы, чтобы сообщить об использовании глобальной переменной, объявленной в другом файле?
4. Каково наиглавнейшее свойство статической локальной переменной?
5. Напишите программу, содержащую функцию с именем `counter()`, которая просто подсчитывает количество вызовов. Позаботьтесь о том, чтобы функция возвращала текущее значение счетчика вызовов.
6. 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.

7

Еще о типах данных и операторах

На заметку



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