

# Содержание

Благодарности	23
Об авторе	23
Поддержка читателя	24
Ждем ваших отзывов!	24
<b>ВВЕДЕНИЕ</b>	<b>25</b>
Для кого написана эта книга	25
Структура книги	25
Соглашения, принятые в книге	26
Примеры кода	27
<b>ЧАСТЬ I. Основы C++</b>	<b>29</b>
<b>ЗАНЯТИЕ 1. Первые шаги</b>	<b>31</b>
Краткий экскурс в историю языка C++	32
Связь с языком C	32
Преимущества языка C++	32
Развитие стандарта C++	33
Кто использует программы, написанные на C++	33
Создание приложения C++	33
Этапы создания исполнимого файла	34
Анализ и устранение ошибок	34
Интегрированные среды разработки	34
Создание первого приложения на C++	35
Построение и запуск вашего первого приложения C++	36
Понятие ошибок компиляции	38
Что нового в C++	38
Резюме	39
Вопросы и ответы	39
Коллоквиум	40
Контрольные вопросы	40
Упражнения	40
<b>ЗАНЯТИЕ 2. Структура программы на C++</b>	<b>41</b>
Части программы Hello World	42
Директива препроцессора <code>#include</code>	42
Тело программы — функция <code>main()</code>	43
Возврат значения	44
Концепция пространств имен	44
Комментарии в коде C++	46
Функции в C++	47
Ввод-вывод с использованием потоков <code>std::cin</code> и <code>std::cout</code>	50
Резюме	52
Вопросы и ответы	52
Коллоквиум	52
Контрольные вопросы	53
Упражнения	53

<b>ЗАНЯТИЕ 3. Использование переменных и констант</b>	<b>55</b>
Что такое переменная	56
Коротко о памяти и адресации	56
Объявление переменных для получения доступа и использования памяти	56
Объявление и инициализация нескольких переменных одного типа	58
Понятие области видимости переменной	59
Глобальные переменные	61
Соглашения об именовании	62
Распространенные типы переменных, поддерживаемые компилятором C++	63
Использование типа <code>bool</code> для хранения логических значений	64
Использование типа <code>char</code> для хранения символьных значений	64
Концепция знаковых и беззнаковых целых чисел	65
Знаковые целочисленные типы <code>short</code> , <code>int</code> , <code>long</code> и <code>long long</code>	66
Беззнаковые целочисленные типы <code>unsigned short</code> , <code>unsigned int</code> , <code>unsigned long</code> и <code>unsigned long long</code>	66
Избегайте переполнения, выбирая подходящие типы	67
Типы с плавающей точкой <code>float</code> и <code>double</code>	69
Определение размера переменной с использованием оператора <code>sizeof</code>	69
Запрет сужающего преобразования при использовании инициализации списком	71
Автоматический вывод типа с использованием <code>auto</code>	72
Использование ключевого слова <code>typedef</code> для замены типа	73
Что такое константа	74
Литеральные константы	74
Объявление переменных как констант с использованием ключевого слова <code>const</code>	75
Объявление констант с использованием ключевого слова <code>constexpr</code>	76
Перечисления	78
Определение констант с использованием директивы <code>#define</code>	80
Ключевые слова, недопустимые для использования в качестве имен переменных и констант	80
Резюме	81
Вопросы и ответы	82
Коллоквиум	84
Контрольные вопросы	84
Упражнения	84
<b>ЗАНЯТИЕ 4. Массивы и строки</b>	<b>85</b>
Что такое массив	86
Необходимость в массивах	86
Объявление и инициализация статических массивов	87
Как данные хранятся в массиве	88
Доступ к данным, хранимым в массиве	89
Изменение данных в массиве	90
Многомерные массивы	93
Объявление и инициализация многомерных массивов	93
Доступ к элементам многомерного массива	94
Динамические массивы	95
Строки символов в стиле C	97

Строки C++: использование <code>std::string</code>	100
Резюме	101
Вопросы и ответы	102
Коллоквиум	103
Контрольные вопросы	103
Упражнения	103
<b>ЗАНЯТИЕ 5. Выражения, инструкции и операторы</b>	<b>105</b>
Выражения	106
Составные инструкции, или блоки	107
Использование операторов	107
Оператор присваивания (=)	107
Понятие l- и r-значений	107
Операторы сложения (+), вычитания (-), умножения (*), деления (/) и деления по модулю (%)	108
Операторы инкремента (++) и декремента (--)	109
Что значит “постфиксный” и “префиксный”	109
Операторы равенства (==) и неравенства (!=)	111
Операторы сравнения	111
Логические операции НЕ, И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ	114
Использование логических операторов C++ !, && и	115
Побитовые операторы ~, &,   и ^	119
Побитовые операторы сдвига вправо (>>) и влево (<<)	121
Составные операторы присваивания	122
Использование оператора <code>sizeof</code> для определения объема памяти, занимаемого переменной	124
Приоритет операторов	125
Резюме	127
Вопросы и ответы	127
Коллоквиум	128
Контрольные вопросы	128
Упражнения	128
<b>ЗАНЯТИЕ 6. Управление потоком выполнения программы</b>	<b>129</b>
Условное программирование с использованием конструкции <code>if...else</code>	131
Условное выполнение нескольких инструкций	133
Вложенные инструкции <code>if</code>	134
Условная обработка с использованием конструкции <code>switch-case</code>	138
Тернарный условный оператор (?:)	141
Выполнение кода в циклах	142
Рудиментарный цикл с использованием инструкции <code>goto</code>	143
Цикл <code>while</code>	145
Цикл <code>do...while</code>	146
Цикл <code>for</code>	148
Цикл <code>for</code> для диапазона	151
Изменение поведения цикла с использованием операторов <code>continue</code> и <code>break</code>	153
Бесконечные циклы, которые никогда не заканчиваются	154
Управление бесконечными циклами	154

Программирование вложенных циклов	157
Использование вложенных циклов для перебора многомерного массива	158
Использование вложенных циклов для вычисления чисел Фибоначчи	160
Резюме	161
Вопросы и ответы	162
Коллоквиум	162
Контрольные вопросы	163
Упражнения	163
<b>ЗАНЯТИЕ 7. Организация кода с помощью функций</b>	<b>165</b>
Потребность в функциях	166
Что такое прототип функции	167
Что такое определение функции	168
Что такое вызов функции и аргументы	168
Создание функций с несколькими параметрами	169
Создание функций без параметров и возвращаемых значений	170
Параметры функций со значениями по умолчанию	171
Рекурсия — функция, вызывающая сама себя	173
Функции с несколькими операторами <code>return</code>	175
Использование функций для работы с данными различных видов	176
Перегрузка функций	177
Передача в функцию массива значений	178
Передача аргументов по ссылке	180
Как процессор обрабатывает вызовы функций	182
Встраиваемые функции	183
Автоматический вывод возвращаемого типа	184
Лямбда-функции	186
Резюме	187
Вопросы и ответы	188
Коллоквиум	188
Контрольные вопросы	188
Упражнения	189
<b>ЗАНЯТИЕ 8. Указатели и ссылки</b>	<b>191</b>
Что такое указатель	192
Объявление указателя	192
Определение адреса переменной с использованием оператора получения адреса <code>&amp;</code>	193
Использование указателей для хранения адресов	194
Доступ к данным с использованием оператора разыменования <code>*</code>	197
Значение <code>sizeof()</code> для указателя	199
Динамическое распределение памяти	201
Использование <code>new</code> и <code>delete</code> для выделения и освобождения памяти	201
Указатели и операции инкремента и декремента	204
Использование ключевого слова <code>const</code> с указателями	207
Передача указателей в функции	208
Сходство между массивами и указателями	209
Наиболее распространенные ошибки при использовании указателей	212
Утечки памяти	212

Когда указатели указывают на недопустимые области памяти	213
Висячие (беспризорные, дикие) указатели	214
Проверка успешности запроса с использованием оператора <code>new</code>	215
Полезные советы по применению указателей	218
Что такое ссылка	218
Зачем нужны ссылки	220
Использование ключевого слова <code>const</code> со ссылками	221
Передача аргументов в функции по ссылке	221
Резюме	223
Вопросы и ответы	223
Коллоквиум	225
Контрольные вопросы	225
Упражнения	225
<b>часть II. Объектно-ориентированное программирование на C++</b>	<b>227</b>
<b>ЗАНЯТИЕ 9. Классы и объекты</b>	<b>229</b>
Концепция классов и объектов	230
Объявление класса	230
Объект как экземпляр класса	231
Доступ к членам класса с использованием оператора точки ( <code>.</code> )	232
Обращение к членам класса с использованием оператора указателя ( <code>-&gt;</code> )	232
Ключевые слова <code>public</code> и <code>private</code>	234
Абстракция данных с помощью ключевого слова <code>private</code>	236
Конструкторы	237
Объявление и реализация конструктора	237
Когда и как использовать конструкторы	238
Перегрузка конструкторов	240
Класс без конструктора по умолчанию	242
Параметры конструктора со значениями по умолчанию	243
Конструкторы со списками инициализации	244
Деструктор	246
Объявление и реализация деструктора	246
Когда и как использовать деструкторы	247
Копирующий конструктор	250
Поверхностное копирование и связанные с ним проблемы	250
Глубокое копирование с использованием копирующего конструктора	252
Перемещающий конструктор улучшает производительность	257
Способы использования конструкторов и деструктора	258
Класс, который не разрешает себя копировать	258
Класс-синглтон, обеспечивающий наличие только одного экземпляра	259
Класс, запрещающий создание экземпляра в стеке	262
Применение конструкторов для преобразования типов	264
Указатель <code>this</code>	266
Размер класса	267
Чем структура отличается от класса	269
Объявление друзей класса	270

Специальный механизм хранения данных — <code>union</code>	272
Объявление объединения	272
Где используется объединение	273
Агрегатная инициализация классов и структур	275
<code>constexpr</code> с классами и объектами	278
Резюме	279
Вопросы и ответы	279
Коллоквиум	280
Контрольные вопросы	280
Упражнения	281
<b>ЗАНЯТИЕ 10. Реализация наследования</b>	<b>283</b>
Основы наследования	284
Наследование и порождение	284
Синтаксис наследования C++	286
Модификатор доступа <code>protected</code>	288
Инициализация базового класса — передача параметров базовому классу	290
Перекрытие методов базового класса в производном	293
Вызов перекрытых методов базового класса	295
Вызов методов базового класса в производном классе	296
Производный класс, скрывающий методы базового класса	298
Порядок конструирования	300
Порядок деструкции	300
Закрытое наследование	303
Защищенное наследование	305
Проблема срезки	308
Множественное наследование	309
Запрет наследования с помощью ключевого слова <code>final</code>	311
Резюме	313
Вопросы и ответы	313
Коллоквиум	313
Контрольные вопросы	314
Упражнения	314
<b>ЗАНЯТИЕ 11. Полиморфизм</b>	<b>315</b>
Основы полиморфизма	316
Потребность в полиморфном поведении	316
Полиморфное поведение, реализованное с помощью виртуальных функций	318
Необходимость виртуальных деструкторов	320
Как работают виртуальные функции. Понятие таблицы виртуальных функций	324
Абстрактные классы и чисто виртуальные функции	328
Использование виртуального наследования для решения проблемы ромба	330
Ключевое слово <code>override</code> для указания преднамеренного перекрытия	335
Использование ключевого слова <code>final</code> для предотвращения	
перекрытия функции	336
Виртуальные копирующие конструкторы?	336
Резюме	340
Вопросы и ответы	340

Коллоквиум	341
Контрольные вопросы	341
Упражнения	342
<b>ЗАНЯТИЕ 12. Типы операторов и их перегрузка</b>	<b>343</b>
Что такое операторы C++	344
Унарные операторы	345
Типы унарных операторов	345
Программирование унарного оператора инкремента или декремента	345
Создание операторов преобразования	348
Создание оператора разыменования (*) и оператора выбора члена (->)	351
Бинарные операторы	353
Типы бинарных операторов	353
Создание бинарных операторов сложения (a+b) и вычитания (a-b)	354
Реализация операторов сложения с присваиванием (+=)	
и вычитания с присваиванием (-=)	357
Перегрузка операторов равенства (==) и неравенства (!=)	359
Перегрузка операторов <, >, <= и >=	361
Перегрузка оператора копирующего присваивания (=)	363
Оператор индексации ([])	366
Оператор функции ()	369
Перемещающий конструктор и оператор перемещающего присваивания	370
Проблема излишнего копирования	370
Объявление перемещающих конструктора и оператора присваивания	371
Пользовательские литералы	376
Операторы, которые не могут быть перегружены	378
Резюме	379
Вопросы и ответы	379
Коллоквиум	380
Контрольные вопросы	380
Упражнения	380
<b>ЗАНЯТИЕ 13. Операторы приведения</b>	<b>381</b>
Потребность в приведении типов	382
Почему приведения в стиле C не нравятся некоторым программистам C++	383
Операторы приведения C++	383
Использование оператора <code>static_cast</code>	384
Использование оператора <code>dynamic_cast</code> и идентификация типа	
времени выполнения	385
Использование оператора <code>reinterpret_cast</code>	388
Использование оператора <code>const_cast</code>	389
Проблемы с операторами приведения C++	390
Резюме	392
Вопросы и ответы	392
Коллоквиум	392
Контрольные вопросы	393
Упражнения	393

<b>ЗАНЯТИЕ 14. Введение в макросы и шаблоны</b>	<b>395</b>
Препроцессор и компилятор	396
Использование <code>#define</code> для определения констант	396
Использование макроса для защиты от множественного включения	399
Использование директивы <code>#define</code> для написания макрофункции	400
Зачем все эти скобки?	402
Использование макроса <code>assert</code> для проверки выражений	402
Преимущества и недостатки использования макрофункций	404
Введение в шаблоны	405
Синтаксис объявления шаблона	406
Типы объявлений шаблонов	406
Шаблонные функции	407
Шаблоны и безопасность типов	409
Шаблонные классы	409
Объявление шаблонов с несколькими параметрами	410
Объявление шаблонов параметрами по умолчанию	411
Простой шаблон класса <code>holdsPair</code>	411
Инстанцирование и специализация шаблона	413
Шаблонные классы и статические члены	415
Шаблоны с переменным количеством параметров (вариадические шаблоны)	416
Использование <code>static_assert</code> для выполнения проверок времени компиляции	420
Использование шаблонов в практическом программировании на C++	421
Резюме	422
Вопросы и ответы	422
Коллоквиум	423
Контрольные вопросы	423
Упражнения	423
<b>ЧАСТЬ III. Стандартная библиотека шаблонов</b>	<b>425</b>
<b>ЗАНЯТИЕ 15. Введение в стандартную библиотеку шаблонов</b>	<b>427</b>
Контейнеры STL	428
Последовательные контейнеры	428
Ассоциативные контейнеры	429
Адаптеры контейнеров	431
Итераторы STL	431
Алгоритмы STL	432
Взаимодействие контейнеров и алгоритмов с использованием итераторов	432
Использование ключевого слова <code>auto</code> для определения типа	434
Выбор правильного контейнера	435
Классы строк библиотеки STL	437
Резюме	437
Вопросы и ответы	437
Коллоквиум	438
Контрольные вопросы	438



<b>ЗАНЯТИЕ 16. Класс строки библиотеки STL</b>	<b>439</b>
Потребность в классах обработки строк	440
Работа с классом строки STL	441
Создание экземпляров и копий строк STL	441
Доступ к символу в строке <code>std::string</code>	443
Конкатенация строк	445
Поиск символа или подстроки в строке	446
Усечение строк STL	448
Обращение строки	450
Смена регистра символов	451
Реализация строки на базе шаблона STL	453
Оператор <code>""s</code> в <code>std::string</code> в C++14	453
Резюме	454
Вопросы и ответы	455
Коллоквиум	455
Контрольные вопросы	455
Упражнения	455
<b>ЗАНЯТИЕ 17. Классы динамических массивов библиотеки STL</b>	<b>457</b>
Характеристики класса <code>std::vector</code>	458
Типичные операции с вектором	458
Создание экземпляра вектора	458
Вставка элементов в конец вектора с помощью <code>push_back()</code>	460
Инициализация списком	461
Вставка элементов в определенную позицию с помощью <code>insert()</code>	461
Доступ к элементам вектора с использованием семантики массива	464
Доступ к элементам вектора с использованием семантики указателя	465
Удаление элементов из вектора	466
Концепции размера и емкости	468
Класс <code>deque</code> библиотеки STL	470
Резюме	473
Вопросы и ответы	473
Коллоквиум	474
Контрольные вопросы	474
Упражнения	474
<b>ЗАНЯТИЕ 18. Классы <code>list</code> и <code>forward_list</code></b>	<b>475</b>
Характеристики класса <code>std::list</code>	476
Основные операции со списком	476
Инстанцирование класса <code>std::list</code>	476
Вставка элементов в начало и в конец списка	478
Вставка в середину списка	479
Удаление элементов из списка	482
Обращение списка и сортировка его элементов	483
Обращение элементов списка с помощью <code>list::reverse()</code>	484
Сортировка элементов	485
Сортировка и удаление элементов из списка, который содержит объекты класса	487
Шаблон класса <code>std::forward_list</code>	490

Резюме	492
Вопросы и ответы	492
Коллоквиум	493
Контрольные вопросы	493
Упражнения	493
<b>ЗАНЯТИЕ 19. Классы множеств STL</b>	<b>495</b>
Введение в классы множеств STL	496
Фундаментальные операции с классами <code>set</code> и <code>multiset</code>	496
Инстанцирование объекта <code>std::set</code>	497
Вставка элементов в множество и мультимножество	499
Поиск элементов в множестве и мультимножестве	500
Удаление элементов из множества и мультимножества	502
Преимущества и недостатки использования множеств и мультимножеств	507
Реализация хеш-множеств <code>std::unordered_set</code>	
и <code>std::unordered_multiset</code>	507
Резюме	511
Вопросы и ответы	511
Коллоквиум	511
Контрольные вопросы	512
Упражнения	512
<b>ЗАНЯТИЕ 20. Классы отображений библиотеки STL</b>	<b>513</b>
Введение в классы отображений библиотеки STL	514
Фундаментальные операции с классами <code>std::map</code> и <code>std::multimap</code>	515
Инстанцирование классов <code>std::map</code> и <code>std::multimap</code>	515
Вставка элементов в <code>map</code> и <code>multimap</code>	517
Поиск элементов в отображении	519
Поиск элементов в мультиотображении STL	522
Удаление элементов из <code>map</code> и <code>multimap</code>	522
Применение пользовательского предиката	524
Контейнер для пар “ключ–значение” на базе хеш-таблиц	528
Как работают хеш-таблицы	528
Использование <code>unordered_map</code> и <code>unordered_multimap</code>	529
Резюме	533
Вопросы и ответы	533
Коллоквиум	534
Контрольные вопросы	534
Упражнения	534
<b>ЧАСТЬ IV. Углубляемся в STL</b>	<b>535</b>
<b>ЗАНЯТИЕ 21. Понятие о функциональных объектах</b>	<b>537</b>
Концепция функциональных объектов и предикатов	538
Типичные приложения функциональных объектов	538
Унарные функции	538
Унарный предикат	543
Бинарные функции	545
Бинарный предикат	547

Резюме	550
Вопросы и ответы	550
Коллоквиум	550
Контрольные вопросы	550
Упражнения	551
<b>ЗАНЯТИЕ 22. Лямбда-выражения языка C++11</b>	<b>553</b>
Что такое лямбда-выражение	554
Как определить лямбда-выражение	555
Лямбда-выражение для унарной функции	555
Лямбда-выражение для унарного предиката	557
Лямбда-выражения с состоянием и списки захвата [ . . . ]	558
Обобщенный синтаксис лямбда-выражений	560
Лямбда-выражение для бинарной функции	561
Лямбда-выражение для бинарного предиката	563
Резюме	565
Вопросы и ответы	565
Коллоквиум	566
Контрольные вопросы	566
Упражнения	566
<b>ЗАНЯТИЕ 23. Алгоритмы библиотеки STL</b>	<b>567</b>
Что такое алгоритмы STL	568
Классификация алгоритмов STL	568
Не изменяющие алгоритмы	568
Изменяющие алгоритмы	569
Использование алгоритмов STL	571
Поиск элементов по заданному значению или условию	571
Подсчет элементов с использованием значения или условия	573
Поиск элемента или диапазона в коллекции	575
Инициализация элементов в контейнере заданным значением	577
Использование алгоритма <code>std::generate()</code>	
для инициализации значениями, генерируемыми во время выполнения	579
Обработка элементов диапазона с использованием алгоритма <code>for_each()</code>	581
Выполнение преобразований с помощью алгоритма <code>std::transform()</code>	583
Операции копирования и удаления	585
Замена значений и элементов с использованием условия	588
Сортировка, поиск в отсортированной коллекции и удаление дубликатов	590
Разделение диапазона	592
Вставка элементов в отсортированную коллекцию	594
Резюме	597
Вопросы и ответы	597
Коллоквиум	598
Контрольные вопросы	598
Упражнения	598
<b>ЗАНЯТИЕ 24. Адаптивные контейнеры: стек и очередь</b>	<b>599</b>
Поведенческие характеристики стеков и очередей	600
Стеки	600
Очереди	600

Использование класса STL <code>stack</code>	601
Создание экземпляра стека	601
Функции-члены класса <code>stack</code>	602
Вставка и извлечение с помощью методов <code>push()</code> и <code>pop()</code>	603
Использование класса STL <code>queue</code>	605
Создание экземпляра очереди	605
Функции-члены класса <code>queue</code>	606
Вставка в конец и извлечение из начала очереди с использованием методов <code>push()</code> и <code>pop()</code>	607
Использование класса STL <code>priority_queue</code>	608
Создание экземпляра очереди с приоритетами	608
Функции-члены класса <code>priority_queue</code>	610
Вставка в конец и извлечение из начала очереди с приоритетами с использованием методов <code>push()</code> и <code>pop()</code>	611
Резюме	613
Вопросы и ответы	613
Коллоквиум	613
Контрольные вопросы	614
Упражнения	614
<b>ЗАНЯТИЕ 25. Работа с битовыми флагами при использовании библиотеки STL</b>	<b>615</b>
Класс <code>bitset</code>	616
Инстанцирование класса <code>std::bitset</code>	616
Использование класса <code>std::bitset</code> и его членов	617
Полезные операторы, предоставляемые классом <code>std::bitset</code>	618
Методы класса <code>std::bitset</code>	618
Класс <code>vector&lt;bool&gt;</code>	621
Создание экземпляра класса <code>vector&lt;bool&gt;</code>	621
Функции и операторы класса <code>vector&lt;bool&gt;</code>	622
Резюме	623
Вопросы и ответы	623
Коллоквиум	624
Контрольные вопросы	624
Упражнения	624
<b>ЧАСТЬ V. Сложные концепции C++</b>	<b>625</b>
<b>ЗАНЯТИЕ 26. Понятие интеллектуальных указателей</b>	<b>627</b>
Что такое интеллектуальный указатель	628
Проблемы обычных указателей	628
Чем могут помочь интеллектуальные указатели	628
Как реализованы интеллектуальные указатели	629
Типы интеллектуальных указателей	630
Глубокое копирование	631
Механизм копирования при записи	633
Интеллектуальные указатели со счетчиком ссылок	633
Интеллектуальный указатель со списком ссылок	634

Деструктивное копирование	634
Использование интеллектуального указателя <code>std::unique_ptr</code>	637
Популярные библиотеки интеллектуальных указателей	639
Резюме	639
Вопросы и ответы	639
Коллоквиум	640
Контрольные вопросы	640
Упражнения	640
<b>ЗАНЯТИЕ 27. Применение потоков для ввода и вывода</b>	<b>641</b>
Концепция потоков	642
Важнейшие классы и объекты потоков C++	643
Использование <code>std::cout</code> для вывода форматированных данных на консоль	644
Изменение формата представления чисел	645
Выравнивание текста и установка ширины поля	647
Использование <code>std::cin</code> для ввода	648
Использование <code>std::cin</code> для ввода простых старых типов данных	648
Использование метода <code>std::cin::get()</code> для ввода в буфер <code>char*</code>	649
Использование <code>std::cin</code> для ввода в переменную типа <code>std::string</code>	650
Использование потока <code>std::fstream</code> для работы с файлом	652
Открытие и закрытие файла с помощью методов <code>open()</code> и <code>close()</code>	652
Создание и запись текстового файла с использованием метода <code>open()</code> и оператора <code>&lt;&lt;</code>	653
Чтение текстового файла с использованием метода <code>open()</code> и оператора <code>&gt;&gt;</code>	655
Запись и чтение из бинарного файла	656
Использование <code>std::stringstream</code> для преобразования строк	658
Резюме	660
Вопросы и ответы	660
Коллоквиум	660
Контрольные вопросы	660
Упражнения	661
<b>ЗАНЯТИЕ 28. Обработка исключений</b>	<b>663</b>
Что такое исключение	664
Что вызывает исключения	664
Реализация безопасности в отношении исключений с помощью блоков <code>try</code> и <code>catch</code>	665
Использование блока <code>catch(...)</code> для обработки всех исключений	665
Обработка исключения конкретного типа	666
Генерация исключения с помощью оператора <code>throw</code>	668
Как работает обработка исключений	669
Класс <code>std::exception</code>	671
Пользовательский класс исключения, производный от <code>std::exception</code>	672
Резюме	674
Вопросы и ответы	675
Коллоквиум	675
Контрольные вопросы	676
Упражнения	676

<b>ЗАНЯТИЕ 29. Что дальше</b>	<b>677</b>
Чем отличаются современные процессоры	678
Как лучше использовать несколько ядер	679
Что такое поток	679
Зачем создавать многопоточные приложения	680
Как потоки осуществляют транзакцию данных	681
Использование мьютексов и семафоров для синхронизации потоков	682
Проблемы, вызываемые многопоточностью	682
Как писать отличный код C++	683
C++17: что новенького	684
Инициализация в if и switch	684
Гарантия устранения копирования	685
Устранение накладных расходов выделения памяти с помощью <code>std::string_view</code>	686
<code>std::variant</code> как безопасная с точки зрения типов альтернатива объединению	686
Условная компиляция с использованием <code>if constexpr</code>	687
Усовершенствованные лямбда-выражения	688
Автоматический вывод типа для конструкторов <code>template&lt;auto&gt;</code>	688
Изучение C++ на этом не заканчивается	688
Документация в вебе	688
Сетевые сообщества и помощь	689
Резюме	689
Вопросы и ответы	689
Коллоквиум	690
Контрольные вопросы	690
<b>ЧАСТЬ VI. Приложения</b>	<b>691</b>
<b>ПРИЛОЖЕНИЕ А. Двоичные и шестнадцатеричные числа</b>	<b>693</b>
Десятичная система счисления	694
Двоичная система счисления	694
Почему компьютеры используют двоичные числа	695
Что такое биты и байты	695
Сколько байтов в килобайте	695
Шестнадцатеричная система счисления	696
Зачем нужна шестнадцатеричная система	696
Преобразование в различные системы счисления	697
Обобщенный процесс преобразования	697
Преобразование десятичного числа в двоичное	697
Преобразование десятичного числа в шестнадцатеричное	698
<b>ПРИЛОЖЕНИЕ Б. Ключевые слова языка C++</b>	<b>699</b>
<b>ПРИЛОЖЕНИЕ В. Приоритет операторов</b>	<b>701</b>
<b>ПРИЛОЖЕНИЕ Г. Коды ASCII</b>	<b>703</b>
Таблица ASCII отображаемых символов	704

<b>ПРИЛОЖЕНИЕ Д. Ответы</b>	<b>707</b>
Ответы к занятию 1	707
Контрольные вопросы	707
Упражнения	707
Ответы к занятию 2	708
Контрольные вопросы	708
Упражнения	708
Ответы к занятию 3	709
Контрольные вопросы	709
Упражнения	709
Ответы к занятию 4	710
Контрольные вопросы	710
Упражнения	711
Ответы к занятию 5	711
Контрольные вопросы	711
Упражнения	712
Ответы к занятию 6	712
Контрольные вопросы	712
Упражнения	713
Ответы к занятию 7	716
Контрольные вопросы	716
Упражнения	716
Ответы к занятию 8	717
Контрольные вопросы	717
Упражнения	717
Ответы к занятию 9	717
Контрольные вопросы	717
Упражнения	718
Ответы к занятию 10	719
Контрольные вопросы	719
Упражнения	719
Ответы к занятию 11	720
Контрольные вопросы	720
Упражнения	720
Ответы к занятию 12	722
Контрольные вопросы	722
Упражнения	722
Ответы к занятию 13	723
Контрольные вопросы	723
Упражнения	723
Ответы к занятию 14	724
Контрольные вопросы	724
Упражнения	724
Ответы к занятию 15	725
Контрольные вопросы	725
Ответы к занятию 16	726
Контрольные вопросы	726
Упражнения	726

Ответы к занятию 17	729
Контрольные вопросы	729
Упражнения	729
Ответы к занятию 18	732
Контрольные вопросы	732
Упражнения	733
Ответы к занятию 19	734
Контрольные вопросы	734
Упражнения	734
Ответы к занятию 20	737
Контрольные вопросы	737
Упражнения	738
Ответы к занятию 21	738
Контрольные вопросы	738
Упражнения	738
Ответы к занятию 22	740
Контрольные вопросы	740
Упражнения	740
Ответы к занятию 23	741
Контрольные вопросы	741
Упражнения	742
Ответы к занятию 24	743
Контрольные вопросы	743
Упражнения	743
Ответы к занятию 25	743
Контрольные вопросы	743
Упражнения	744
Ответы к занятию 26	744
Контрольные вопросы	744
Упражнения	744
Ответы к занятию 27	745
Контрольные вопросы	745
Упражнения	746
Ответы к занятию 28	746
Контрольные вопросы	746
Упражнения	746
Ответы к занятию 29	746
Контрольные вопросы	746



## ЗАНЯТИЕ 5

# Выражения, инструкции и операторы

Основой программ является набор последовательно выполняемых команд. Эти команды формируются в выражения и инструкции и используют операторы для выполнения определенных вычислений или действий.

*На этом занятии...*

- Что такое выражения
- Что такое блоки, или составные выражения
- Что такое операторы
- Как выполнять простые арифметические и логические операции

## Выражения

Языки программирования состоят из *инструкций* (statement), которые следуют одна за другой. Давайте проанализируем первую инструкцию, которую вы изучили:

```
cout << "Hello World" << endl;
```

Эта инструкция использует поток `cout` для вывода текста на консоль (т.е. на экран). Все инструкции в языке C++ заканчиваются точкой с запятой (;), определяющей границу инструкции. Эта точка с запятой подобна точке, которую вы добавляете в конце предложения разговорного языка. Следующая инструкция может начинаться непосредственно после точки с запятой, но для удобства и удобочитаемости программисты, как правило, записывают инструкции с новой строки. Вот, например, две инструкции в одной строке:

```
cout << "Hello World" << endl; cout << "Another hello" << endl;  
// Одна строка, две инструкции
```

### ПРИМЕЧАНИЕ

Пробельные символы обычно не воспринимаются компилятором. К ним относятся пробелы, символы табуляции, символы новой строки и т.д. Тем не менее пробельные символы в составе строковых литералов отображаются при выводе.

Поэтому следующий код недопустим:

```
cout << "Hello  
World" << endl;  
// Символ новой строки в строковом литерале недопустим
```

Такой код обычно заканчивается сообщением об ошибке, указывающим, что компилятор не обнаружил в первой строке закрывающую кавычку (") и завершающую инструкцию точку с запятой (;). Если по каким-то причинам необходимо распространить инструкцию на несколько строк, достаточно добавить последним символом обратный косой черты (\):

```
cout << "Hello \  
World" << endl; // Разделение строки на две вполне допустимо
```

Еще один способ разместить приведенную выше инструкцию в двух строках — это использовать два строковых литерала вместо одного:

```
cout << "Hello "  
"World" << endl; // Два строковых литерала подряд вполне допустимы
```

Встретив такой код, компилятор обратит внимание на два соседних строковых литерала и сам объединит их.

**ПРИМЕЧАНИЕ**

Разделение инструкций на несколько строк может быть полезным, если у вас есть длинные текстовые элементы или сложные инструкции, состоящие из множества переменных, которые делают инструкцию намного длиннее, чем может вместить большинство экранов.

## Составные инструкции, или блоки

Сгруппировав инструкции в фигурных скобках `{ ... }`, вы создаете *составную инструкцию* (compound statement), или *блок* (block).

```
{
    int Number = 365;
    cout << "Этот блок содержит две инструкции" << endl;
}
```

Как правило, блок объединяет несколько связанных инструкций. Блоки особенно полезны при применении условной инструкции `if` и циклов, которые рассматриваются на занятии 6, “Управление потоком выполнения программы”.

## Использование операторов

*Операторы* (operator) в C++ представляют собой инструменты, предоставляемые языком для работы с данными, их преобразования, обработки и принятия решений на их основе.

### Оператор присваивания (=)

*Оператор присваивания* (assignment operator) мы уже использовали в этой книге. Он вполне интуитивно понятен:

```
int daysInYear = 365;
```

Приведенное выше выражение использует оператор присваивания для инициализации целочисленной переменной значением 365. Оператор присваивания заменяет значение, содержащееся в операнде слева от оператора присваивания (называемого *l-значением* (l-value)), значением операнда справа (называемого *r-значением* (r-value)).

### Понятие l- и r-значений

Зачастую l-значения называют областями памяти. Такая переменная, как `daysInYear`, из приведенного выше примера фактически является дескриптором области памяти и, соответственно, l-значением. С другой стороны, r-значения могут быть самим содержимым области памяти.

Все l-значения могут быть г-значениями, но не все г-значения могут быть l-значениями. Чтобы понять это лучше, рассмотрим следующий пример, который не имеет никакого смысла, а потому не будет компилироваться:

```
365 = daysInYear;
```

## Операторы сложения (+), вычитания (-), умножения (\*), деления (/) и деления по модулю (%)

Вы можете выполнять арифметические операции между двумя операндами, используя оператор + для сложения, оператор - для вычитания, оператор \* для умножения, оператор / для деления и оператор % для деления по модулю:

```
int num1 = 22;
int num2 = 5;
int addNums      = num1 + num2; // 27
int subtractNums = num1 - num2; // 17
int multiplyNums = num1 * num2; // 110
int divideNums   = num1 / num2; // 4
int moduloNums  = num1 % num2; // 2
```

Оператор деления (/) возвращает результат деления двух операндов. Однако в случае целых чисел результат не содержит дробной части, поскольку целые числа по определению не могут ее содержать. Оператор деления по модулю (%) возвращает остаток от деления и применим только к целочисленным значениям. В листинге 5.1 содержится простая программа, демонстрирующая выполнение арифметических действий с двумя введенными пользователем числами.

### ЛИСТИНГ 5.1. Демонстрация арифметических операторов с введенными пользователем целыми числами

---

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите два целых числа: ";
6:     int num1 = 0, num2 = 0;
7:     cin >> num1;
8:     cin >> num2;
9:
10:    cout << num1 << " + " << num2 << " = " << num1+num2 << endl;
11:    cout << num1 << " - " << num2 << " = " << num1-num2 << endl;
12:    cout << num1 << " * " << num2 << " = " << num1*num2 << endl;
13:    cout << num1 << " / " << num2 << " = " << num1/num2 << endl;
14:    cout << num1 << " % " << num2 << " = " << num1%num2 << endl;
15:
16:    return 0;
17: }
```

---

## Результат

```
Введите два целых числа: 365 25
365 + 25 = 390
365 - 25 = 340
365 * 25 = 9125
365 / 25 = 14
365 % 25 = 15
```

## Анализ

Большая часть программы говорит сама за себя. Интереснее всего, вероятно, строка, использующая оператор деления по модулю `%`. Она возвращает остаток деления значения переменной `num1` (365) на значение переменной `num2` (25).

## Операторы инкремента (`++`) и декремента (`--`)

Иногда в программе необходим *инкремент* (increment), т.е. простое увеличение значения переменной на единицу. Это особенно важно для переменных, контролируемых циклы, в которых значение переменной должно увеличиваться или уменьшаться на единицу при каждом выполнении цикла.

Для сокращения записей наподобие `num=num+1` или `num=num-1` язык C++ предоставляет операторы `++` (инкремента) и `--` (декремента).

Синтаксис их использования следующий:

```
int num1 = 101;
int num2 = num1++; // Постфиксный оператор инкремента
int num2 = ++num1; // Префиксный оператор инкремента
int num2 = num1--; // Постфиксный оператор декремента
int num2 = --num1; // Префиксный оператор декремента
```

Пример кода демонстрирует два разных способа применения операторов инкремента и декремента: до и после операнда. Операторы, которые располагаются перед операндом, называются *префиксными* (prefix) операторами инкремента или декремента, а те, которые располагаются после, — *постфиксными* (postfix).

## Что значит “постфиксный” и “префиксный”

Сначала следует понять различие между префиксными и постфиксными операторами, а затем использовать тот, который нужен вам в каждом конкретном случае. Результат выполнения постфиксных операторов заключается в том, что сначала `l`-значению присваивается `r`-значение, а потом `r`-значение увеличивается или уменьшается. Это значит, что во всех случаях использования постфиксного оператора значением переменной `num2` будет прежнее значение переменной `num1` (т.е. то значение, которое она имела до операции инкремента или декремента).

Действие префиксных операторов прямо противоположно: сначала изменяется `r`-значение, а затем оно присваивается `l`-значению. В этих случаях переменные `num2` и `num1` имеют одинаковые значения. Листинг 5.2 демонстрирует результат выполнения префиксных и постфиксных операторов инкремента и декремента для определенного целого числа.

**ЛИСТИНГ 5.2.** Различия между постфиксными и префиксными операторами

---

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int startValue = 101;
6:     cout << "Начальное значение: " << startValue << endl;
7:
8:     int postfixIncrement = startValue++;
9:     cout << "Постфиксный ++ = " << postfixIncrement << endl;
10:    cout << "После постфиксного ++ startValue = "
11:        << startValue << endl;
12:    startValue = 101; // Сброс
13:    int prefixIncrement = ++startValue;
14:    cout << "Префиксный ++ = " << prefixIncrement << endl;
15:    cout << "После префиксного ++ startValue = "
16:        << startValue << endl;
17:    startValue = 101; // Сброс
18:    int postfixDecrement = startValue--;
19:    cout << "Постфиксный -- = " << postfixDecrement << endl;
20:    cout << "После постфиксного -- startValue = "
21:        << startValue << endl;
22:    startValue = 101; // Сброс
23:    int prefixDecrement = --startValue;
24:    cout << "Префиксный -- = " << prefixDecrement << endl;
25:    cout << "После префиксного -- startValue = "
26:        << startValue << endl;
27:    return 0;
28: }
```

---

**Результат**

```
Начальное значение: 101
Префиксный ++ = 101
После постфиксного ++ startValue = 102
Постфиксный ++ = 102
После префиксного ++ startValue = 102
Постфиксный -- = 101
После постфиксного -- startValue = 100
Префиксный -- = 100
После префиксного -- startValue = 100
```

**Анализ**

Результаты показывают, чем постфиксные операторы отличаются от префиксных. При использовании постфиксных операторов в строках 8 и 18 l-значения содержат

исходные значения целого числа, — какими они были до операций инкремента или декремента. Использование префиксных операторов в строках 13 и 23, напротив, присваивает результат инкремента или декремента. Это самое важное различие, о котором следует помнить, выбирая правильный тип оператора.

В следующих выражениях префиксные или постфиксные операторы никак не влияют на результат:

```
startValue++; // То же, что и...
++startValue;
```

Дело в том, что здесь нет присваивания исходного значения и конечный результат в обоих случаях — увеличенное на единицу значение переменной `startValue`.

## ПРИМЕЧАНИЕ

Нередко приходится слышать о ситуациях, когда префиксные операторы инкремента или декремента являются более предпочтительными с точки зрения производительности, т.е. `++startValue` предпочтительнее, чем `startValue++`.

Это правда, по крайней мере теоретически, поскольку при постфиксных операторах компилятор должен временно хранить исходное значение на случай его присваивания. Влияние на производительность в случае целых чисел незначительно, но в случае некоторых классов этот аргумент мог бы иметь смысл. Интеллектуальные компиляторы могут полностью устранить различия, оптимизируя код.

## Операторы равенства (==) и неравенства (!=)

Зачастую необходимо проверить выполнение или не выполнение определенного условия прежде, чем предпринять некое действие. Операторы равенства `==` (операнды равны) и неравенства `!=` (операнды не равны) позволяют сделать именно это.

Результат проверки равенства имеет логический тип `bool`, т.е. `true` (истина) или `false` (ложь).

```
int personAge = 20;
bool checkEquality      = (personAge == 20); // true
bool checkInequality    = (personAge != 100); // true
bool checkEqualityAgain = (personAge == 200); // false
bool checkInequalityAgain = (personAge != 20); // false
```

## Операторы сравнения

Кроме проверки на равенство и неравенство, может возникнуть необходимость в сравнении, значение какой переменной больше, а какой меньше. Для этого язык C++ предоставляет операторы сравнения, приведенные в табл. 5.1.

ТАБЛИЦА 5.1. Операторы сравнения

Оператор	Назначение
Меньше (<)	Возвращает значение <code>true</code> , если один операнд меньше другого ( $Op1 < Op2$ ), в противном случае возвращает значение <code>false</code>
Больше (>)	Возвращает значение <code>true</code> , если один операнд больше другого ( $Op1 > Op2$ ), в противном случае возвращает значение <code>false</code>
Меньше или равно (<=)	Возвращает значение <code>true</code> , если один операнд меньше или равен другому, в противном случае возвращает значение <code>false</code>
Больше или равно (>=)	Возвращает значение <code>true</code> , если один операнд больше или равен другому, в противном случае возвращает значение <code>false</code>

Как свидетельствует табл. 5.1, результатом операции сравнения всегда является значение `true` или `false`, другими словами, значение типа `bool`. Следующий пример кода демонстрирует применение операторов сравнения, приведенных в табл. 5.1:

```
int personAge = 20;
bool checkLessThan          = (personAge < 100); // true
bool checkGreaterThan      = (personAge > 100); // false
bool checkLessThanEqualTo  = (personAge <= 20); // true
bool checkGreaterThanEqualTo = (personAge >= 20); // true
bool checkGreaterThanEqualToAgain = (personAge >= 100); // false
```

Код листинга 5.3 демонстрирует использование этих операторов при отображении результата на экране.

ЛИСТИНГ 5.3. Операторы равенства и сравнения

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите два целых числа:" << endl;
6:     int num1 = 0, num2 = 0;
7:     cin >> num1;
8:     cin >> num2;
9:
10:    bool Equality = (num1 == num2);
11:    cout << "Проверка равенства: " << Equality << endl;
12:
13:    bool Inequality = (num1 != num2);
14:    cout << "Проверка неравенства: " << Inequality << endl;
15:
16:    bool GreaterThan = (num1 > num2);
17:    cout << "Результат сравнения " << num1 << " > " << num2;
18:    cout << ": " << GreaterThan << endl;
19:
20:    bool LessThan = (num1 < num2);
```



```
21:     cout << "Результат сравнения " << num1 << " < " << num2;
22:     cout << ": " << LessThan << endl;
23:
24:     bool GreaterThanEquals = (num1 >= num2);
25:     cout << "Результат сравнения " << num1 << " >= " << num2;
26:     cout << ": " << GreaterThanEquals << endl;
27:
28:     bool LessThanEquals = (num1 <= num2);
29:     cout << "Результат сравнения " << num1 << " <= " << num2;
30:     cout << ": " << LessThanEquals << endl;
31:
32:     return 0;
33: }
```

---

## Результат

Введите два целых числа:

**365**

**-24**

Проверка равенства: 0

Проверка неравенства: 1

Результат сравнения 365 > -24: 1

Результат сравнения 365 < -24: 0

Результат сравнения 365 >= -24: 1

Результат сравнения 365 <= -24: 0

Следующий запуск:

Введите два целых числа:

**101**

**101**

Проверка равенства: 1

Проверка неравенства: 0

Результат сравнения 101 > 101: 0

Результат сравнения 101 < 101: 0

Результат сравнения 101 >= 101: 1

Результат сравнения 101 <= 101: 1

## Анализ

Программа отображает результат различных операций в двоичном виде. Интересно отметить в выводе случай, когда сравниваются два одинаковых целых числа. Операторы ==, >= и <= дают идентичные результаты.

Тот факт, что результат операторов равенства и сравнения является логическим значением, делает их отлично подходящими для использования в операторах принятия решения и в выражениях условий циклов, гарантирующих выполнение цикла, только пока условие истинно. Более подробная информация об условном выполнении и циклах приведена на занятии 6, “Управление потоком выполнения программы”.

**ПРИМЕЧАНИЕ**

В листинге 5.3 булево значение `false` выводится как 0. Значение `true` выводится как 1. С точки зрения компилятора выражение равно `false`, если его вычисляемое значение нулевое. Проверка на равенство `false` представляет собой проверку того, что данное значение нулевое. Выражение с ненулевым значением рассматривается как логическое значение `true`.

## Логические операции НЕ, И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ

Логическая операция НЕ выполняется с помощью оператора `!` и выполняется над одним операндом. Таблица истинности логической операции НЕ, которая просто инвертирует значение логического флага, приведена в табл. 5.2.

**ТАБЛИЦА 5.2. Таблица истинности логической операции НЕ**

Операнд	Результат операции “НЕ Операнд”
<code>false</code>	<code>true</code>
<code>true</code>	<code>false</code>

Для других операций, таких как И, ИЛИ или ИСКЛЮЧАЮЩЕЕ ИЛИ, необходимы два операнда. Логическая операция И возвращает значение `true` только тогда, когда каждый операнд содержит значение `true`. Таблица истинности логической операции И приведена в табл. 5.3.

**ТАБЛИЦА 5.3. Таблица истинности логической операции И**

Операнд 1	Операнд 2	Результат операции “Операнд 1 И Операнд 2”
<code>false</code>	<code>false</code>	<code>false</code>
<code>true</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>
<code>true</code>	<code>true</code>	<code>true</code>

Логическая операция И выполняется с помощью оператора `&&`.

Логическая операция ИЛИ возвращает значение `true` тогда, когда по крайней мере один из операндов содержит значение `true`. Таблица истинности логической операции ИЛИ приведена в табл. 5.4.

**ТАБЛИЦА 5.4. Таблица истинности логической операции ИЛИ**

Операнд 1	Операнд 2	Результат операции “Операнд 1 ИЛИ Операнд 2”
<code>false</code>	<code>false</code>	<code>false</code>
<code>true</code>	<code>false</code>	<code>true</code>
<code>false</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>true</code>	<code>true</code>

Логическая операция **ИЛИ** выполняется с помощью оператора `||`.

Логическая операция **ИСКЛЮЧАЮЩЕГО ИЛИ (XOR)** немного отличается от логической операции **ИЛИ** и возвращает значение `true` тогда, когда любой из операндов содержит значение `true`, но не оба одновременно (т.е. когда логические значения операндов не равны). Таблица истинности логической операции **ИСКЛЮЧАЮЩЕГО ИЛИ** приведена в табл. 5.5.

**ТАБЛИЦА 5.5.** Таблица истинности логической операции **ИСКЛЮЧАЮЩЕГО ИЛИ**

Операнд 1	Операнд 2	Результат операции "Операнд 1 XOR Операнд 2"
false	false	false
true	false	true
false	true	true
true	true	false

Логическая операция **ИСКЛЮЧАЮЩЕГО ИЛИ** выполняется с помощью оператора `^`. Результат получается путем выполнения операции **ИСКЛЮЧАЮЩЕГО ИЛИ** над битами операндов.

## Использование логических операторов **C++ !, && и ||**

Рассмотрим следующие утверждения.

- “Если идет дождь **И** если нет автобуса, то я не смогу попасть на работу”.
- “Если есть большая скидка **ИЛИ** если я получу премию, то смогу купить этот автомобиль”.

В программировании часто необходима некоторая логическая конструкция, когда результат двух операций используется в логическом контексте для принятия решения о выполнении последующего потока программы. Язык **C++** предоставляет логические операторы **И** и **ИЛИ**, которые можно использовать в условных инструкциях, а следовательно, в зависимости от условий изменять поток выполнения программы.

В листинге 5.4 демонстрируется работа логических операторов **И** и **ИЛИ**.

**ЛИСТИНГ 5.4.** Анализ логических операторов **C++ && и ||**

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите true(1) или false(0) "
6:         "для двух операндов:";
7:     bool Op1 = false, Op2 = false;
8:     cin >> Op1;
9:     cin >> Op2;
10:    cout << Op1 << " И " << Op2 << " = " << (Op1&&Op2) << endl;
```

```
11:     cout << Op1 << " ИЛИ " << Op2 << " = " << (Op1||Op2) << endl;
12:
13:     return 0;
14: }
```

---

## Результат

Введите true(1) или false(0) для двух операндов: 1 0  
1 И 0 = 0  
1 ИЛИ 0 = 1

Следующий запуск:

Введите true(1) или false(0) для двух операндов: 1 1  
1 И 1 = 1  
1 ИЛИ 1 = 1

## Анализ

Программа демонстрирует, что позволяют делать логические операции И и ИЛИ. Однако она не показывает, как их использовать для принятия решений.

В листинге 5.5 представлена программа, которая, используя условные и логические операторы, выполняет разные строки кода в зависимости от значений, содержащихся в переменных.

### ЛИСТИНГ 5.5. Использование логических операторов !

и && в условных инструкциях для изменения потока выполнения

---

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите 0 или 1 для ответа на вопрос" << endl;
6:     cout << "Идет дождь? ";
7:     bool isRaining = false;
8:     cin >> isRaining;
9:
10:    cout << "На улице есть автобус? ";
11:    bool busesPly = false;
12:    cin >> busesPly;
13:
14:    // Условный оператор использует операторы && и !
15:    if (isRaining && !busesPly)
16:        cout << "Вы не попадете на работу" << endl;
17:    else
18:        cout << "Вы попадете на работу" << endl;
19:
20:    if (isRaining && busesPly)
```

```
21:         cout << "Возьмите зонтик" << endl;
22:
23:     if (!isRaining) && busesPly)
24:         cout << "Приятного дня и хорошей погоды!" << endl;
25:
26:     return 0;
27: }
```

---

## Результат

Введите 0 или 1 для ответа на вопрос  
Идет дождь? **1**  
На улице есть автобус? **1**  
Вы попадете на работу  
Возьмите зонтик

### Следующий запуск:

Введите 0 или 1 для ответа на вопрос  
Идет дождь? **1**  
На улице есть автобус? **0**  
Вы не попадете на работу

### Последний запуск:

Введите 0 или 1 для ответа на вопрос  
Идет дождь? **0**  
На улице есть автобус? **1**  
Вы попадете на работу  
Приятного дня и хорошей погоды!

## Анализ

Код в листинге 5.5 использует условную инструкцию `if`, которая пока еще не рассматривалась. Но вы все же попробуйте понять поведение этой инструкции, сопоставив ее с выводом на консоль. Строка 15 содержит логическое выражение `(isRaining && !busesPly)`, которое можно прочитать как “идет дождь И НЕТ автобуса”. Логический оператор И здесь использован для объединения отсутствия автобусов (обозначенного логическим оператором НЕ перед наличием автобусов) и присутствия дождя.

## ПРИМЕЧАНИЕ

Более подробная информация об инструкции `if` будет приведена на занятии 6, “Управление потоком выполнения программы”.

Код листинга 5.6 использует логические операторы `!` и `||` для демонстрации условной обработки.

**ЛИСТИНГ 5.6.** Использование логических операторов !  
и || для решения о возможности купить автомобиль

---

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите 0 или 1 для ответа на вопрос" << endl;
6:     cout << "Есть ли скидка на автомобиль? ";
7:     bool onDiscount = false;
8:     cin >> onDiscount;
9:
10:    cout << "Вы получили премию? ";
11:    bool fantasticBonus = false;
12:    cin >> fantasticBonus;
13:
14:    if (onDiscount || fantasticBonus)
15:        cout << "Вы можете купить автомобиль!" << endl;
16:    else
17:        cout << "Покупку придется отложить..." << endl;
18:
19:    if (!onDiscount)
20:        cout << "Скидки на автомобиль нет" << endl;
21:
22:    return 0;
23: }
```

---

**Результат**

Введите 0 или 1 для ответа на вопрос  
Есть ли скидка на автомобиль? **0**  
Вы получили премию? **1**  
Вы можете купить автомобиль!  
Скидки на автомобиль нет

**Следующий запуск:**

Введите 0 или 1 для ответа на вопрос  
Есть ли скидка на автомобиль? **0**  
Вы получили премию? **0**  
Покупку придется отложить...  
Скидки на автомобиль нет

**Последний запуск:**

Введите 0 или 1 для ответа на вопрос  
Вы получили премию? **1**  
Есть ли скидка на автомобиль? **1**  
Вы можете купить автомобиль!

## Анализ

Программа сообщает о возможности купить автомобиль, если на него есть скидка или если вы получили премию. Инструкция в строке 19, кроме того, сообщает, когда скидки на автомобиль нет. В строке 14 инструкция `if` используется вместе с конструкцией `else` в строке 16. Часть `if` выполняет инструкцию в строке 15, если условие `(onDiscount || fantasticBonus)` истинно (имеет значение `true`). Это выражение содержит логический оператор **ИЛИ** и возвращает значение `true`, если есть скидка на ваш любимый автомобиль или если вы получили фантастическую премию. Если рассматриваемое выражение возвращает значение `false`, выполняется инструкция в строке 17, идущая после конструкции `else`.

## Побитовые операторы `~`, `&`, `|` и `^`

Различие между логическими и побитовыми операторами в том, что они возвращают не логический результат, а значение, отдельные биты которого получены в результате выполнения оператора над соответствующими битами операндов. Язык C++ позволяет выполнять такие операции, как **НЕ**, **ИЛИ**, **И** и **ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR)** в побитовом режиме, позволяя работать с отдельными битами, инвертируя их с помощью оператора `~`, применяя операцию **ИЛИ** с помощью оператора `|`, операцию **И** с помощью оператора `&` и операцию **XOR** с помощью оператора `^`. Последние три операции обычно выполняются с некоторой специально подготовленной битовой маской.

Некоторые битовые операции полезны в тех случаях, когда, например, каждый из битов, содержащихся в целом числе, определяет состояние некоего флага. Так, целое число размером 32 бита можно использовать для хранения 32-х логических флагов. Использование побитовых операторов продемонстрировано в листинге 5.7.

### **ЛИСТИНГ 5.7.** Использование побитовых операторов для работы с отдельными битами целого числа

```
0: #include <iostream>
1: #include <bitset>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Введите число (0-255): ";
7:     unsigned short inputNum = 0;
8:     cin >> inputNum;
9:
10:    bitset<8> inputBits(inputNum);
11:    cout << inputNum << " в бинарном виде равно "
12:        << inputBits << endl;
13:    bitset<8> BitwiseNOT = (~inputNum);
14:    cout << "Побитовое НЕ ~" << endl;
15:    cout << "~" << inputBits << " = "
```

```
16:         << BitwiseNOT << endl;
17:     cout << "Логическое И (&) с 00001111" << endl;
18:     bitset<8> BitwiseAND = (0x0F&inputNum); // 0x0F == 0001111
19:     cout << "0001111 & " << inputBits << " = "
20:         << BitwiseAND << endl;
21:     cout << "Логическое ИЛИ (|) с 00001111" << endl;
22:     bitset<8> BitwiseOR = (0x0F | inputNum);
23:     cout << "00001111 | " << inputBits << " = "
24:         << BitwiseOR << endl;
25:     cout << "Логическое XOR (^) с 00001111" << endl;
26:     bitset<8> BitwiseXOR = (0x0F ^ inputNum);
27:     cout << "00001111 ^ " << inputBits << " = "
28:         << BitwiseXOR << endl;
29:     return 0;
30: }
```

---

## Результат

```
Введите число (0-255): 181
181 в бинарном виде равно 10110101
Побитовое НЕ ~
~10110101 = 01001010
Логическое И (&) с 00001111
0001111 & 10110101 = 00000101
Логическое ИЛИ (|) с 00001111
00001111 | 10110101 = 10111111
Логическое XOR (^) с 00001111
00001111 ^ 10110101 = 10111010
```

## Анализ

Эта программа использует *битовое множество* (bitset) — тип, который нами еще не рассматривался, — для облегчения отображения двоичных данных. Роль класса `std::bitset` здесь исключительно вспомогательная — он помогает с отображением данных и не более того. В строках 10, 13, 18 и 22 вы фактически присваиваете целое число объекту битового множества, используемому для отображения этого целочисленного значения в двоичном виде. Все операции выполняются с целыми числами. Сначала обратите внимание на вывод введенного пользователем исходного числа 181 в двоичном виде, а затем переходите к результату выполнения различных побитовых операторов — `~`, `&`, `|` и `^` — с этим целым числом. Как можно заметить, побитовое НЕ, использованное в строке 13, просто инвертирует все биты числа. Программа демонстрирует также работу операторов `&`, `|` и `^`, создающих результат путем выполнения вычислений с каждым битом двух операндов. Сопоставьте представленные результаты с приведенными ранее таблицами истинности, и работа побитовых операторов станет вам понятнее.



**ПРИМЕЧАНИЕ**

Если вы хотите узнать больше о работе с битовыми флагами в языке C++, обратитесь к занятию 25, “Работа с битовыми флагами при использовании библиотеки STL”, на котором класс `std::bitset` обсуждается подробнее.

## Побитовые операторы сдвига вправо (>>) и влево (<<)

Операторы сдвига перемещают всю последовательность битов вправо или влево, позволяя осуществить умножение или деление на степень двойки, а также имеют многие другие применения.

Вот типичный пример применения оператора сдвига для умножения на два:

```
int doubledValue = Num << 1; // Для удвоения значения биты
                          // сдвигаются на одну позицию влево
```

Вот типичный пример применения оператора сдвига для деления на два:

```
int halvedValue = Num >> 2; // Для деления значения на два биты
                          // сдвигаются на одну позицию вправо
```

Использование операторов сдвига для быстрого умножения и деления целочисленных значений продемонстрировано в листинге 5.8.

**ЛИСТИНГ 5.8.** Использование побитового оператора сдвига вправо >> для получения четверти и половины значения, а оператора сдвига влево << для умножения значения на два и четыре

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите число: ";
6:     int inputNum = 0;
7:     cin >> inputNum;
8:
9:     int halfNum      = inputNum >> 1;
10:    int quarterNum   = inputNum >> 2;
11:    int doubleNum    = inputNum << 1;
12:    int quadrupleNum = inputNum << 2;
13:
14:    cout << "Четверть: "      << quarterNum << endl;
15:    cout << "Половина: "     << halfNum << endl;
16:    cout << "Удвоенное: "    << doubleNum << endl;
17:    cout << "Учетверенное: " << quadrupleNum << endl;
18:
19:    return 0;
20: }
```

## Результат

Введите число: **16**  
 Четверть: 4  
 Половина: 8  
 Удвоенное: 32  
 Учетверенное: 64

## Анализ

Пользователь вводит число 16, которое в двоичном представлении имеет вид 1000. В строке 9 осуществляется его смещение вправо на один бит, и получается 0100, что в десятичном виде составляет 8 — фактически половина исходного значения. В строке 10 осуществляется смещение вправо на два бита, 1000 превращается в 0010, что составляет 4. Результат операторов сдвига влево в строках 11 и 12 прямо противоположен. Смещение на один бит влево дает значение 10000 или 32, а на два бита — соответственно 100000 или 64, фактически удваивая и учетверяя исходное значение.

## ПРИМЕЧАНИЕ

Побитовые операторы сдвига не выполняют циклический сдвиг. Кроме того, результат сдвига знаковых чисел зависит от конкретного компилятора.

## Составные операторы присваивания

*Составные операторы присваивания* (compound assignment operator) — это операторы присваивания, в которых результат операции присваивается левому операнду.

Рассмотрим следующий код:

```
int num1 = 22;
int num2 = 5;
num1 += num2; // После операции num1 содержит значение 27
```

Этот код эквивалентен следующей строке кода:

```
num1 = num1 + num2;
```

Таким образом, результат оператора += — это сумма этих двух операндов, присвоенная затем левому операнду (num1). Краткий перечень составных операторов присваивания с объяснением их работы приведен в табл. 5.6.

**ТАБЛИЦА 5.6. Составные операторы присваивания**

Оператор	Применение	Эквивалент
Присваивание с добавлением	num1 += num2;	num1 = num1 + num2;
Присваивание с вычитанием	num1 -= num2;	num1 = num1 - num2;
Присваивание с умножением	num1 *= num2;	num1 = num1 * num2;
Присваивание с делением	num1 /= num2;	num1 = num1 / num2;
Присваивание с делением по модулю	num1 %= num2;	num1 = num1 % num2;
Присваивание с побитовым сдвигом влево	num1 <<= num2;	num1 = num1 << num2;

Окончание табл. 5.6

Оператор	Применение	Эквивалент
Присваивание с побитовым сдвигом вправо	<code>num1 &gt;&gt;= num2;</code>	<code>num1 = num1 &gt;&gt; num2;</code>
Присваивание с побитовым И	<code>num1 &amp;= num2;</code>	<code>num1 = num1 &amp; num2;</code>
Присваивание с побитовым ИЛИ	<code>num1  = num2;</code>	<code>num1 = num1   num2;</code>
Присваивание с побитовым XOR	<code>num1 ^= num2;</code>	<code>num1 = num1 ^ num2;</code>

Применение этих операторов продемонстрировано в листинге 5.9.

#### ЛИСТИНГ 5.9. Использование составных операторов присваивания

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите число: ";
6:     int value = 0;
7:     cin >> value;
8:
9:     value += 8;
10:    cout << "После += 8, value = " << value << endl;
11:    value -= 2;
12:    cout << "После -= 2, value = " << value << endl;
13:    value /= 4;
14:    cout << "После /= 4, value = " << value << endl;
15:    value *= 4;
16:    cout << "После *= 4, value = " << value << endl;
17:    value %= 1000;
18:    cout << "После %= 1000, value = " << value << endl;
19:
20:    // Примечание: далее присваивание происходит в пределах cout
21:    cout << "После <<= 1, value = " << (value <<= 1) << endl;
22:    cout << "После >>= 2, value = " << (value >>= 2) << endl;
23:
24:    cout << "После |= 0x55, value = " << (value |= 0x55) << endl;
25:    cout << "После ^= 0x55, value = " << (value ^= 0x55) << endl;
26:    cout << "После &= 0x0F, value = " << (value &= 0x0F) << endl;
27:
28:    return 0;
29: }
```

#### Результат

```
Введите число: 440
После += 8, value = 448
После -= 2, value = 446
```

```
После /= 4, value = 111
После *= 4, value = 444
После %= 1000, value = 444
После <<= 1, value = 888
После >>= 2, value = 222
После |= 0x55, value = 223
После ^= 0x55, value = 138
После &= 0x0F, value = 10
```

## Анализ

Обратите внимание, как последовательно изменяется значение переменной `value` по мере применения в программе различных составных операторов присваивания. Каждая операция осуществляется с использованием переменной `value`, и ее результат снова присваивается переменной `value`. Так, в строке 9 ко введенному пользователем значению 440 прибавляется 8, а результат, 448, снова присваивается переменной `value`. При следующей операции в строке 11 из 448 вычитается 2, что дает значение 446, которое снова присваивается переменной `value`, и т.д.

## Использование оператора `sizeof` для определения объема памяти, занимаемого переменной

Этот оператор возвращает объем памяти в байтах, используемой определенным типом или переменной. Оператор `sizeof` имеет следующий синтаксис:

```
sizeof (Переменная);
или
sizeof (Тип);
```

### ПРИМЕЧАНИЕ

Оператор `sizeof(...)` выглядит как вызов функции, но это не функция, а оператор. Данный оператор не может быть определен программистом, а следовательно, не может быть перегружен.

Более подробная информация об определении собственных операторов рассматривается на занятии 12, “Типы операторов и их перегрузка”.

В листинге 5.10 демонстрируется применение оператора `sizeof` для определения объема памяти, занятого массивом. Кроме того, можно вернуться к листингу 3.5 и проанализировать применение оператора `sizeof` для определения объема памяти, занятого переменными наиболее распространенных типов.

**ЛИСТИНГ 5.10.** Использование оператора `sizeof` для определения количества байтов, занятых массивом из 100 целых чисел и каждым его элементом

```
0: #include <iostream>
1: using namespace std;
2:
```

```
3: int main()
4: {
5:     cout << "Использование sizeof для массива" << endl;
6:     int myNumbers[100] = {0};
7:
8:     cout << "Байт для типа int: " << sizeof(int) << endl;
9:     cout << "Байт для массива myNumbers: "
10:         << sizeof(myNumbers) << endl;
11:     cout << "Байт для элемента массива: "
12:         << sizeof(myNumbers[0]) << endl;
13:     return 0;
14: }
```

---

## Результат

```
Использование sizeof для массива
Байт для типа int: 4
Байт для массива myNumbers: 400
Байт для элемента массива: 4
```

## Анализ

Программа демонстрирует, как оператор `sizeof` возвращает размер в байтах массива из 100 целых чисел, составляющий 400 байтов, а также что размер каждого его элемента составляет 4 байта.

Оператор `sizeof` может быть весьма полезен, когда необходимо динамически разместить в памяти  $N$  объектов, особенно если их тип создан вами самостоятельно. Вы можете использовать результат выполнения оператора `sizeof` для определения объема памяти, занимаемого каждым объектом, а затем динамически выделить память, используя оператор `new`.

Более подробная информация о динамическом распределении памяти рассматривается на занятии 8, “Указатели и ссылки”.

## Приоритет операторов

Возможно, вы помните из школы, что арифметические операции в сложном арифметическом выражении выполняются в определенном порядке.

В языке C++ также используются сложные выражения, например:

```
int myNumber = 10 * 30 + 20 - 5 * 5 << 2;
```

Вопрос: какое значение будет содержать переменная `myNumber`? Здесь нет места догадкам. Порядок выполнения различных операторов строго определен стандартом C++. Этот порядок определяется *приоритетом операторов*, приведенным в табл. 5.7.

ТАБЛИЦА 5.7. Приоритет операторов

Приоритет	Название	Оператор
1	Область видимости	::
2	Прямое и косвенное обращение к члену класса, вызов функции	. -> ()
3	Инкремент и декремент, логические операторы отрицания и побитового дополнения, унарные “минус” и “плюс”, получение адреса и разыменование, а также операторы new, new[], delete, delete[], sizeof и операторы приведения типов	++ -- ~ ! - + & *
4	Обращение к элементу по указателю	.* ->*
5	Умножение, деление, деление по модулю	* / %
6	Сложение, вычитание	+ -
7	Сдвиг влево, сдвиг вправо	<< >>
8	Меньше, меньше или равно, больше, больше или равно	< <= > >=
9	Равно, не равно	== !=
10	Побитовое И	&
11	Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ	^
12	Побитовое ИЛИ	
13	Логическое И	&&
14	Логическое ИЛИ	
15	Тернарный условный оператор	?:
16	Операторы присваивания	= *= /= %= += -= <<= >>= &=  = ^=
17	Оператор “запятая”	,

Давайте теперь еще раз рассмотрим сложное выражение, приведенное ранее в качестве примера:

```
int myNumber = 10 * 30 + 20 - 5 * 5 << 2;
```

При вычислении результата этого выражения необходимо использовать правила приоритета операторов, приведенные в табл. 5.7, чтобы понять, как их выполняет компилятор. Так, умножение и деление имеют более высокий приоритет, чем сложение и вычитание, приоритет которых, в свою очередь, выше приоритета оператора сдвига. В результате после первого упрощения мы получаем:

```
int myNumber = 300 + 20 - 25 << 2;
```

Поскольку сложение и вычитание имеют более высокий приоритет по сравнению со сдвигом, это выражение упрощается до

```
int myNumber = 295 << 2;
```

И наконец выполняется операция сдвига. Зная, что сдвиг влево на один бит удваивает число, а сдвиг влево на два бита умножает его на 4, можно утверждать, что выражение сводится к  $295 * 4$ , а результат равен 1180.

**ВНИМАНИЕ!**

Чтобы код был более понятным, используйте круглые скобки. Приведенное выше выражение просто написано плохо. Компилятору не составляет труда в нем разобраться, но написанный код должен быть понятен, в первую очередь, людям.

То же выражение будет намного понятнее, если записать его следующим образом:

```
int myNumber = ((10*30) - (5*5) + 20) << 2; // 1180
```

**РЕКОМЕНДУЕТСЯ**

**Используйте** круглые скобки, чтобы сделать свой код более понятным.

**Используйте** правильные типы переменных и убедитесь, что они никогда не приведут к переполнению.

**Помните**, что все l-значения (например, переменные) могут быть r-значениями, но не все r-значения (например, "Hello World") могут быть l-значениями.

**НЕ РЕКОМЕНДУЕТСЯ**

**Не создавайте** сложные выражения, полагающиеся на таблицу приоритета операторов; ваш код должен быть понятен, в первую очередь, людям.

**Не забывайте**, что выражения ++*Переменная* и *Переменная*++ отличаются одно от другого при использовании в присваивании.

## Резюме

На этом занятии вы узнали, что такое инструкции, выражения и операторы языка C++. Вы научились выполнять простые арифметические операции, такие как сложение, вычитание, умножение и деление. Был также приведен краткий обзор таких логических операторов, как НЕ, И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ. Мы рассмотрели логические операторы !, && и ||, используемые в условных выражениях, и побитовые операторы, такие как ~, &, | и ^, которые позволяют работать с отдельными битами.

Вы узнали о приоритете операторов, а также о том, почему так важно использовать круглые скобки при написании кода, который должен быть понятен, в первую очередь, поддерживающим его программистам. Было дано общее представление о переполнении целочисленных переменных и о способах его избегания.

## Вопросы и ответы

■ Почему некоторые программы используют тип `unsigned int`, если тип `unsigned short` занимает меньше памяти и код вполне компилируется?

Тип `unsigned short` обычно имеет предел 65535, а при его превышении происходит переполнение, дающее неверное значение. Чтобы избежать этого, если нет абсолютной уверенности, что рабочие значения всегда останутся ниже указанного предела, следует выбрать более емкий тип, например `unsigned int`.

- Я должен удвоить результат деления на три. Нет ли в моем коде каких-либо проблем: `int result = Number / 3 << 1;`?

Есть. Почему бы вам не использовать круглые скобки, чтобы сделать эту строку проще для понимания другими программистами? Комментарий тоже не повредил бы.

- Мое приложение делит два целочисленных значения — 5 и 2:

```
int num1 = 5, num2 = 2;
int result = num1 / num2;
```

- При выполнении `result` содержит значение 2. Разве это не ошибка?

Нет. Целые числа не предназначены для хранения вещественных чисел. Поэтому результат этой операции — 2, а не 2,5. Если вам нужен результат 2,5, то измените все типы данных на `float` или `double`, так как именно они предназначены для операций с плавающей точкой.

## Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

## Контрольные вопросы

1. Я пишу приложение для деления чисел. Какой тип данных подойдет мне лучше: `int` или `float`?
2. Каков результат выражения `32/7`?
3. Каков результат выражения `32.0/7`?
4. Является ли `sizeof(...)` функцией?
5. Я должен вычислить удвоенное число, добавить к нему 5, а затем снова удвоить результат. Все ли я сделал правильно?  

```
int Result = number << 1 + 5 << 1;
```
6. Каков результат операции ИСКЛЮЧАЮЩЕЕ ИЛИ, если оба операнда содержат значение `true`?

## Упражнения

1. Исправьте код в контрольном вопросе 5, используя круглые скобки для устранения неоднозначности.
2. Каким будет значение переменной `result` в этом выражении?  

```
int result = number << 1 + 5 << 1;
```
3. Напишите программу, которая запрашивает у пользователя два логических значения и демонстрирует результаты различных побитовых операций над ними.