Содержание

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 символ обратной 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 связных 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 `daysIn Year`, 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 r-значениями, но не все r-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

Бо́льшая 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, контролирующих 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 `true`, если один операнд меньше другого (`Op1<Op2`), в противном случае возвращает значение `false`
Больше (>)	Возвращает значение `true`, если один операнд больше другого (`Op1>Op2`), в противном случае возвращает значение `false`
Меньше или равно (<=)	Возвращает значение `true`, если один операнд меньше или равен другому, в противном случае возвращает значение `false`
Больше или равно (>=)	Возвращает значение `true`, если один операнд больше или равен другому, в противном случае возвращает значение `false`

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"
`false`	`true`
`true`	`false`

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"
`false`	`false`	`false`
`true`	`false`	`false`
`false`	`true`	`false`
`true`	`true`	`true`

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"
`false`	`false`	`false`
`true`	`false`	`true`
`false`	`true`	`true`
`true`	`true`	`true`

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, содержавшихся 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	`num1 >>= num2;`	`num1 = num1 >> num2;`
Присваивание с побитовым И	`num1 &= num2;`	`num1 = num1 & num2;`
Присваивание с побитовым ИЛИ	`num1 |= num2;`	`num1 = num1 | num2;`
Присваивание с побитовым XOR	`num1 ^= num2;`	`num1 = num1 ^ num2;`

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	НЕ РЕКОМЕНДУЕТСЯ
Используйте круглые скобки, чтобы сделать свой код более понятным.	**Не создавайте** сложные выражения, полагающиеся на таблицу приоритета операторов; ваш код должен быть понятен, в первую очередь, людям.
Используйте правильные типы переменных и убедитесь, что они никогда не приведут к переполнению.	**Не забывайте**, что выражения *++Переменная* и *Переменная++* отличаются одно от другого при использовании в присваивании.
Помните, что все l-значения (например, переменные) могут быть r-значениями, но не все r-значения (например, `"Hello World"`) могут быть l-значениями.	

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 выбирать 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. Попытайтесь 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.