ГЛАВА 3. ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННЫХ И ТИПИЗАЦИЯ ДАННЫХ

Оглавление

§3.1 Определение переменной и константы

Переменная (variable) – это средство, позволяющее программисту временно сохранить данные. Переменная представляет именованную область памяти, в которой хранится значение определенного типа. Переменная имеет тип, имя и значение. Тип определяет, какого рода информацию может хранить переменная.

Константа – это средство, позволяющее программисту определить элемент, которому не позволено изменяться.

Все компьютеры, смартфоны и другие программируемые устройства имеют микропроцессор и определенный объем памяти для временного хранения данных, называемый оперативной памятью (Random Access Memory – RAD). Кроме того, многие устройства позволяют сохранять данные на долгосрочном запоминающем устройстве, таком как жесткий диск. Микропроцессор выполняет ваше приложение и использует при этом оперативную память для его загрузки, а также для связанных с ним данных, включая те, которые отображаются на экране и вводятся пользователем.

Саму оперативную память, являющуюся областью хранения, можно сравнить с рядом шкафчиков в общежитии, каждый из которых имеет свой номер, т.е. адрес. Чтобы получить доступ к области в памяти, скажем 876-й, процессору нужно с помощи специальных инструкций попросить выбрать оттуда значение или записать значение в нее.

Литеральные константы (или просто «литералы») — это значения, которые вставляются непосредственно в код. Поскольку они являются константами, то их значения изменить нельзя. Литералы бывают логическими, целочисленными, вещественными, символьными и строчными. И отдельный литерал представляет ключевое слово `null`. Например:

```
bool myNameIsAlex = true; // true – это логический литерал
int x = 5; // 5 – это целочисленный литерал
int y = 2 * 3; // 2 и 3 – это целочисленные литералы
cout << ‘a’; // a – символьный литерал
```

```
Console.WriteLine("hello"); // hello – это строковый литерал
```

Строковые литералы заключаются в двойные кавычки, а символьные литералы – в одинарные кавычки.

§3.2 Объявление переменных для получения доступа и использования памяти

Предположим, что надо написать программу умножения для двух чисел, предоставляемых пользователем. Пользователя просят ввести множитель и множимое, один за другим, и каждое из этих значений необходимо хранить до момента умножения. В зависимости от того, что вы хотите делать с результатом умножения, их может понадобится хранить и для более позднего использования в программе.

При программировании на таких языках, как C++ и C#, для хранения значений определяют переменные. Определение переменной очень просто и осуществляется по следующему шаблону:

```
ТИП_ПЕРЕМЕННОЙ ИМЯ_ПЕРЕМЕННОЙ;
```

или

```
ТИП_ПЕРЕМЕННОЙ ИМЯ_ПЕРЕМЕННОЙ = ИСХОДНОЕ_ЗНАЧЕНИЕ;
ТИП_ПЕРЕМЕННОЙ ИМЯ_ПЕРЕМЕННОЙ(ИСХОДНОЕ_ЗНАЧЕНИЕ);
```

Атрибут «ТИП_ПЕРЕМЕННОЙ» указывает компилятору характер данных, который может хранить переменная, и компилятор резервирует для этого необходимое пространство.

Атрибут «ИМЯ_ПЕРЕМЕННОЙ» является более осмысленной заменой для адреса области в памяти, где хранится значение переменной.

Если исходное значение не применяется, вы не можете быть уверены в содержимом этой области памяти, что может быть плохо для программы. Поэтому, будучи необязательной, инициализация зачастую является хорошей практикой программирования. Для инициализации переменной исходным значением применяется **копирующая инициализация**, выполняемая с помощью знака равенства **=**, или **прямая инициализация** с помощью круглых скобок **()**.

Листинг 3.1 демонстрирует объявление переменных, их инициализацию и использование в программе, которая умножает два числа, предоставленных пользователем.

Листинг 3.1.

```
1  #include <iostream>
2  using namespace std;
3  int main ()
4  {
5  cout << "This program will help you multiply two numbers" << endl;
6  cout << "Enter the first number: ";
7  int FirstNumber = 0;
8  cin >> FirstNumber;
9  cout << "Enter the second number: ";
10 int SecondNumber = 0;
11 cin >> SecondNumber;
12 // Умножение двух чисел, сохранение результата в переменной
13 int MultiplicationResult = FirstNumber * SecondNumber;
14 // Отображение результата
15 cout << FirstNumber << " x " << SecondNumber;
16 cout << " = " << MultiplicationResult << endl;
17 return 0;
18 }
```

Результат выполнения программы:

This program will help you multiply two numbers

Enter the first number: 5

Enter the second number: 2

5 x 2 = 10

Это приложение просит пользователя ввести два числа, результат умножения которых и отображается. Чтобы приложение могло использовать введенные пользователем числа, оно должно хранить их в памяти. Переменные `FirstNumber` и `SecondNumber`, объявленные в строках 7 и 10, решают задачу временного хранения введенных пользователем целочисленных значений. Операторы `cin` в строках 8 и 11 используются для получения введенных пользователем значений и сохранения их в двух целочисленных переменных. Оператор `cout` в строке 21 используется для отображения результата на консоли.

Давайте проанализируем объявление переменной подробнее:

```
7  int FirstNumber = 0;
```

Эта строка объявляет переменную типа `int`, который означает целое число, по имени `FirstNumber`. В качестве исходного значения переменной присваивается нулевой значение.

Таким образом, по сравнению с программированием на ассемблере, где необходимо явно просить процессор сохранить множитель в области памяти, скажем 876, высокоуровневые языки программирования, такие как C++ и C#, позволяют обратиться к области памяти для сохранения и получения данных, используя боле дружественные концепции, такие как переменная по имени `FirstNumber`. Компилятор сам выполнит задачу по сопоставлению имени этой переменной с адресом области памяти и позаботится о соответствующих действиях за вас.

Имена переменных важны для написания хорошего, понятного и удобного в сопровождении кода.

Имена же переменных задаются согласно следующим правилам:

1. Имя переменной не может начинаться с цифры.
2. В имени переменной не может быть пробелов, а также специальных символов (вроде ; № # % или /).

3. Имя переменной не может совпадать с другими, ранее объявленными именами (функций, переменных, стандартных операторов и т. п.).

Именами переменных не могут быть также зарезервированные ключевые слова. Например, переменная по имени `return` приведет к ошибке при компиляции.

Переменные в листинге 3.1 имеют одинаковый тип (целочисленный тип), но объявляются в трех отдельных строках. Запись определения этих трех переменных можно уплотнить до одной строки:

```
int FirstNumber = 0, SecondNumber = 0, MultiplicationResult
= 0;
```

§3.3 Область видимости переменной

У обычных переменных есть четко определенная **область видимости** (scope), в пределах которой они допустимы и применимы. При использовании вне своих областей видимости имена переменных не будут распознаны компилятором, и ваша программа не будет откомпилирована. Вне своей области видимости переменная – неопознанная сущность, о которой компилятор ничего не знает.

В C++ существуют отдельные блоки, которые начинаются с открывающей скобки ({) и заканчиваются соответственно закрывающей скобкой (}). Такими блоками являются циклы (for, while, do while) и функции.

Листинг 3.2.

```
1  int func () {
2  // блок (функция - func)
3  }
4  int main() {
5  // блок (функция - main)
6  for (int i = 0; i < 10; i++) {
```

```
7  // блок (цикл - for), также является дочерним блоком функции main
8  for (int j = 0; j < 5; j++) {
9  // блок (цикл - for), но он еще является и дочерним блоком для первого цикла
10 }
11 }
12 return 0;
13 }
```

Если переменная была создана в таком блоке, то ее областью видимости будет являться этот блок от его начала (от открывающей скобки — **{**) и до его конца (до закрывающей скобки — **}**) включая все дочерние блоки созданные в этом блоке.

В примере ниже, программист ошибся с областью видимости:

1. Он создал переменную `j` во втором цикле;
2. Использовав ее в первом цикле `for` (строка 7) он вынудил компилятор сообщить об ошибке (переменной `j` больше нет, поскольку второй цикл закончил свою работу).

Листинг 3.3.

```
1  int main() {
2  for (int i = 0; i < 10; i++) {
3  int b = i;
4  for (int j = 0; j < 5; j++) {
5  cout << b + j;
6  }
7  cout << j;  // ошибка: так как переменная j была создана в другом блоке
8  }
9  return 0;
```

```
10 }
```

А вот ошибки в строке 5 нет, поскольку второй цикл находится в первом цикле (является дочерним блоком первого цикла) и поэтому переменная b может спокойно там использоваться.

§3.3.1 Глобальные переменные

Глобальными переменными называются те переменные, которые были созданы вне тела какого-то блока. Они имеют самую широкую область видимости, так как их можно всегда использовать во всей вашей программе, вплоть до ее окончания работы.

В примере ниже мы создали две глобальные переменные `global` и `global_too` и использовали их в функции `summa`:

Листинг 3.4.

```
1  int global = 5;        // глобальные
2  int global_too = 10;  // переменные
3  int summa() {
4  cout << global + global_too;  // суммируем числа = 15
5  }
6  int main() {
7  summa();  // вызываем функцию summa
8  return 0;
9  }
```

Как видите глобальные переменные видны везде. В нашем примере внутри функции `summa` мы не создавали никакие переменные, мы лишь использовали две глобальные переменные, которые были созданы раньше.

Безосновательное использование глобальных переменных обычно считается плохой практикой программирования.

Значение глобальной переменной может быть присвоено в любой функции, и это значение может оказаться непредсказуемо, особенно если разные функциональные модули разрабатываются разными программистами.

§3.3.2 Локальные переменные

Локальные переменные — это переменные, которые применимы в функции от места ее объявления до конца функции. Областью видимости таких переменных является блоки (и все их дочерние), а также их область видимости не распространяется на другие блоки. Фигурная скобка (**}**), означающая конец функции, означает также конец области видимости объявленных в ней переменных. Когда функция заканчивается, все ее локальные переменные ликвидируются, а занимаемая ей память освобождается.

Из этого можно сделать вывод: у нас есть возможность создавать переменные с одинаковыми именами, но в разных блоках (или другими словами, чтобы их область видимости не совпадала друг с другом).

Листинг 3.5.

```
1  int main() {
2  for (int i = 0; i < 2; i++) {
3  int b = i;  // локальная переменная (она находится в блоке for)
4  cout << b;
5  }
6  return 0;
7  }
```

В примере выше блоком, где была создана локальная переменная `b` является цикл `for` (2 — 5). А вот если бы мы захотели вывести переменную `b` вне блока `for`, компилятор сообщил бы нам об ошибке.

§3.4 l-values и r-values

Все переменные являются **l-values**. l-value (в переводе «л-значение», произносится как «ел-валью») — это значение, которое имеет свой собственный адрес в памяти. Поскольку все переменные имеют адреса, то они все являются l-values (например, переменные a, b, c — все они являются l-values). l от слова «left», так как только значения l-values могут находиться в

левой стороне в операциях присваивания (в противном случае, мы получим ошибку). Например, стэйтмент 9 = 10; вызовет ошибку компилятора, так как 9 не является l-value. Число 9 не имеет своего адреса в памяти и, таким образом, мы ничего не можем ему присвоить (9 = 9 и ничего здесь не изменить).

Противоположностью l-value является r-value (в переводе «р-значение», произносится как «ер-валью»). **r-value** — это значение, которое не имеет постоянного адреса в памяти. Примерами могут быть единичные числа (например, 7, которое имеет значение 7) или выражения (например, 3 + x, которое имеет значение x плюс 3).

Вот несколько примеров операций присваивания с использованием r-values:

Листинг 3.6.

```
1   int a          // объявляем целочисленную переменную a
2   a = 5          // 5 имеет значение 5, которое затем присваивается переменной a
3   a = 4 + 6      // 4 + 6 имеет значение 10, которое затем присваивается переменной a
4   int b          // объявляем целочисленную переменную b
5   b = a          // a имеет значение 10 (исходя из предыдущих операций), которое затем присваивается переменной b
6   b = b          // b имеет значение 10, которое затем присваивается переменной b (ничего не происходит)
7   b = b + 2      // b + 2 имеет значение 12, которое затем присваивается переменной b
```

Давайте детальнее рассмотрим последнюю операцию присваивания:

```
b = b + 2;
```

Здесь переменная b используется в двух различных контекстах. Слева b используется как l-value (переменная с адресом в памяти), а справа b используется как r-value и имеет отдельное значение (в данном случае, 12). При выполнении этого стэйтмента, компилятор видит следующее:

```
b = 10 + 2;
```

И здесь уже понятно, какое значение присваивается переменной b.

В левой стороне операции присваивания всегда должно находиться l-value (которое имеет свой собственный адрес в памяти), а в правой стороне операции присваивания — r-value (которое имеет какое-то значение).

§3.5 Типы данных

Тип данных - фундаментальное понятие теории программирования. **Тип данных** определяет множество значений, набор операций, которые можно применять к таким значениям и, возможно, способ реализации хранения значений и выполнения операций

Язык программирования C++ является расширяемым языком программирования. Понятие расширяемый означает то, что кроме встроенных типов данных, можно создавать свои типы данных.

В таблице 3.1 представлены основные встроенные типы данных в C++. Вся таблица делится на три столбца. В первом столбце указывается зарезервированное слово, которое будет определять, каждое свой, тип данных. Во втором столбце указывается количество байт, которое отводится под переменную с соответствующим типом данных. В третьем столбце показан диапазон допустимых значений.

Таблица 3.1 Типы данных

Тип данных	**байт**	**Диапазон принимаемых значений**
логический тип данных		
bool	1	0 / 255
символьный тип данных		
char	1	0 / 255

Тип данных	байт	Диапазон принимаемых значений
целочисленные типы данных		
short int	2	-32 768 / 32 767
unsigned short int	2	0 / 65 535
int	4	-2 147 483 648 / 2 147 483 647
unsigned int	4	0 / 4 294 967 295
long int	4	-2 147 483 648 / 2 147 483 647
unsigned long int	4	0 / 4 294 967 295
типы данных с плавающей точкой		
float	4	-2 147 483 648.0 / 2 147 483 647.0
long float	8	-9 223 372 036 854 775 808 .0 / 9 223 372 036 854 775 807.0
double	8	-9 223 372 036 854 775 808 .0 / 9 223 372 036 854 775 807.0

§3.5.1 Тип данных bool

Язык C++ предоставляет тип, специально созданный для хранения логических значений `true` или `false`, оба из которых являются зарезервированными ключевыми словами C++. Этот тип особенно полезен при хранении параметров и флагов, которые могут быть установлены или сброшены, существовать или отсутствовать, могут быть доступными или недоступными.

Типичное объявление инициализированной логической переменной имеет следующий вид:

```
bool AlwaysOnTop = false;
```

Но так как диапазон допустимых значений типа данных bool от 0 до 255, то необходимо было как-то сопоставить данный диапазон с определёнными в языке программирования логическими константами true и false. Таким образом, константе true эквивалентны все числа от 1 до 255 включительно, тогда как константе false эквивалентно только одно целое число — 0.

Выражение, обрабатывающее логический тип, выглядит так:

```
bool DeleteFile = (UserSelection == "yes");
// истино, только если UserSelection содержит "yes",
// в противном случае ложно
```

§3.5.2 Тип данных char

Тип `char` используется для хранения одного символа. Типичное объявление показано ниже.

```
char Userlnput = 'Y'; // инициализированный символ 'Y'
```

Обратите внимание, что память состоит из битов и байтов. Биты могут содержать значения 0 или 1, а байты могут хранить числовые представления, используя эти биты. Таким образом, работая или присваивая символьные данные, как показано в примере, компилятор преобразует символы в числовое представление, которое может быть помещено в память. Числовое представление латинских символов A-Z, a-z, чисел 0-9, некоторых специальных клавиш (например, <DEL>) и специальных символов (таких, как возврат на один символ) было стандартизировано в стандартный американский код обмена информацией (American Standard Code for Information Interchange), называемый также **ASCII**.

Например: код буквы 'a' — 97, код буквы 'b' — 98. Символы всегда помещаются в одинарные кавычки.

§3.5.3 Тип данных int

Целочисленные типы данных используются для представления чисел. В таблице 1 их аж шесть штук: short int, unsigned short int, int, unsigned int, long int, unsigned long int**.** Все они имеют свой собственный размер занимаемой памяти и диапазоном принимаемых значений. В зависимости от компилятора, размер занимаемой памяти и диапазон принимаемых значений могут изменяться. Диапазон принимаемых значений, так или иначе, зависит от размера занимаемой памяти. Соответственно, чем больше размер занимаемой памяти, тем больше диапазон принимаемых значений. Также диапазон принимаемых значений меняется в случае, если тип данных объявляется с приставкой unsigned — без знака. Приставка unsigned говорит о том, что тип

данных не может хранить знаковые значения, тогда и диапазон положительных значений увеличивается в два раза, например, типы данных short int и unsigned short int.

Приставки целочисленных типов данных:

1. short — приставка укорачивает тип данных, к которому применяется, путём уменьшения размера занимаемой памяти;
2. long — приставка удлиняет тип данных, к которому применяется, путём увеличения размера занимаемой памяти;
3. unsigned (без знака) — приставка увеличивает диапазон положительных значений в два раза, при этом диапазон отрицательных значений в таком типе данных храниться не может.

Так, что, по сути, мы имеем один целочисленный тип для представления целых чисел — это тип данных `int`. Благодаря приставкам `short`, `long`, `unsigned` появляется некоторое разнообразие типов данных `int`, различающихся размером занимаемой памяти и (или) диапазоном принимаемых значений.

§3.5.4 Тип данных с плавающей точкой

Целочисленные типы данных отлично подходят для работы с целыми числами, но есть ведь ещё и дробные числа. И тут нам на помощь приходит **тип данных с плавающей точкой** (или «тип данных с плавающей запятой», от англ. «floating point»). Переменная такого типа может хранить любые действительные дробные значения, например: 4320.0, -3.33 или 0.01226. Почему точка «плавающая»? Дело в том, точка/запятая перемещается («плавает») между цифрами, разделяя целую и дробную части значения.

Есть два типа данных с плавающей точкой:

- float;
- double.

Язык C++ определяет только их минимальный размер (как и с целочисленными типами). Типы данных с плавающей точкой всегда являются signed (т.е. могут хранить как положительные, так и отрицательные числа).

Объявление переменных разных типов данных с плавающей точкой:

```
float fValue;
double dValue;
```

Если нужно использовать целое число с переменной типа с плавающей точкой, то тогда после этого числа нужно поставить разделительную точку и нуль. Это позволяет различать переменные целочисленных типов от переменных типов с плавающей запятой:

```
int n(5); // 5 - это целочисленный тип
double d(5.0); // 5.0 - это тип данных с плавающей точкой (по умолчанию double)
float f(5.0f); // 5.0 - это тип данных с плавающей точкой, "f" от "float"
```

Обратите внимание, литералы типа с плавающей точкой по умолчанию относятся к типу `double`. `f` в конце числа означает тип `float`.

§3.5.5 Дополнительные типы данных

В C# вводятся также дополнительные типы данных, которые отсутствуют в C++. Дополнительные типы и их описание представлено в таблице 3.2.

Таблица 3.2. Дополнительные типы данных

Тип данных	**Описание типа данных**	**Пример использования**
string	Хранит набор символов. Этому типу соответствуют символьные литералы.	string hello = "Hello"; string word = "world";

object	Может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе.	object a = 22;
		object b = 3.14;
		object c = "hello code";

§3.6 Определение размера переменной

Размер (size) — это объем памяти, резервируемый компилятором при объявлении программистом переменной для содержания присваиваемых ей данных. Размер переменной зависит от ее типа, и в языке C++ есть очень удобный оператор `sizeof`, который сообщает размер в байтах переменной или типа.

Чтобы определить размер целого числа, вызывайте оператор `sizeof` с параметром `int` (тип) или укажите наименование переменной, для которой необходимо определить размер.

```
cout << "Size of an int: " << sizeof(int); // 4
```

Вывод данного оператора демонстрирует размер различных типов в байтах и зависит от конкретной платформы: компилятора, операционной системы и аппаратных средств.

§3.7 Неявное определение типов

В некоторых случаях, когда тип переменной очевиден по присваиваемому при инициализации значению, именно он и применяется. Например, если переменная инициализируется значением true, типом переменной может быть только bool. В языке C++11 есть возможность определять тип неявно, с использованием вместо него ключевого слова auto: auto Flag = true;

Здесь задача определения конкретного типа переменной Flag оставлена компилятору. Компилятор просто проверяет характер значения, которым инициализируется переменная, а затем выбирает тип, наилучшим образом подходящий для этой переменной. В данном случае для инициализирующего значения true лучше всего подходит тип bool. Таким образом, компилятор определяет тип bool как наилучший для переменной Flag и внутренне рассматривает ее как имеющую тип bool, что демонстрирует листинг 3.7.

Листинг 3.7. Использование ключевого слова auto для выведения типов компилятором

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  auto Flag = true;
6  auto Number = 2500000000000;
7  cout << "Flag = " << Flag;
8  cout << " , sizeof(Flag) = " << sizeof(Flag) << endl;
9  cout << "Number = " << Number;
10 cout « " , sizeof(Number) = " << sizeof(Number) <<
   endl;
11 return 0;
12 }
```

Результат выполнения программы:

Flag = 1 , sizeof(Flag) = 1

Number = 2500000000000 , sizeof(Number) = 8

Как можно заметить, вместо явного указания типа `bool` для переменной `Flag` и типа `long long` для переменной `Number` в строках 5 и 6, где они объявляются, было использовано ключевое слово `auto`. Это делегирует решение о типе переменных компилятору, который использует для этого инициализирующее значение. Чтобы проверить, создал ли компилятор фактически предполагаемые типы, используется оператор `sizeof`, позволяющий увидеть в выводе листинга 3.7, что это действительно так.

Использование ключевого слова `auto` требует инициализации переменной, поскольку компилятор нуждается в инициализирующем значении, чтобы принять решение о наилучшем типе для переменной.

Если вы не инициализируете переменную, то применение ключевого слова `auto` приведет к ошибке при компиляции.

§3.8 Константа

Самый важный тип констант, с практической и программной точек зрения, объявляется при помощи ключевого слова `const`, расположенного перед типом переменной. В общем виде объявление выглядит следующим образом:

```
const ИМЯ_ТИПА ИМЯ_КОНСТАТНЫ;
```

Рассмотрим простое приложение, которое отображает значение константы по имени `Pi` (листинг 3.8).

Листинг 3.8. Объявление константы

```
1  #include <iostream>
2  int main()
3  {
4  using namespace std;
5  const double Pi = 22.0 / 7;
6  cout « "The value of constant Pi is: " « Pi « endl;
7  // Снятие комментария со следующей строки приведет к ошибке
8  // Pi = 345;
9  return 0;
10 }
```

Результат выполнения программы:

The value of constant Pi is: 3.14286

Обратите внимание на объявление константы `Pi` в строке 5. Ключевое слово `const` позволяет указать компилятору, что `Pi` — это константа типа `double`. Если снять комментарий со строки 8, где осуществляется попытка присвоить значение переменной, которую вы определили как константу, произойдет ошибка при компиляции примерно с таким сообщением: «You cannot assign to a variable that is const (Вы не можете присвоить значение

переменной, которая является константой)». Таким образом, константы — это прекрасный способ гарантировать неизменность определенных данных.

§3.9 Арифметические операторы

Арифметические операции производятся над числами. Значения, которые участвуют в операции, называются **операндами**. Арифметические операции бывают *бинарными* (производятся над двумя операндами) и *унарными* (выполняются над одним операндом).

К бинарным операциям относят следующие (см. Таблицу 3.3):

Таблица 3.3. Бинарные арифметические числа

Оператор	**Название**	**Назначение**	**Пример**

+	оператор сложения	Операция сложения возвращает сумму двух чисел	int a = 10;
			int b = 7;
			int c = a + b; // 17
			int d = 4 + b; // 11

-	оператор вычитания	Операция вычитания возвращает разность двух чисел	int a = 10;
			int b = 7;
			int c = a - b; // 3
			int d = 41 - b; // 34

*	оператор умножения	Операция умножения возвращает произведение двух чисел	int a = 10;
			int b = 7;
			int c = a * b; // 70
			int d = b * 5; // 35

/	оператор деления	Операция деления возвращает частное двух чисел	int a = 20;
			int b = 5;
			int c = a / b; // 4
			double d = 22.5 / 4.5; // 5

Оператор	Название	Назначение	Пример

%	оператор получения остатка	Операция получения остатка от целочисленного деления	int a = 33;
			int b = 5;
			int c = a % b; // 3
			int d = 22 % 4; // 2 (22 - 4*5 = 2)

Оператор деления имеет два режима. Если оба операнда являются целыми числами, то оператор выполняет целочисленное деление. Т.е. любая дробь (больше/меньше) отбрасывается и возвращается целое значение без остатка, например, 7 / 4 = 1.

Если один или оба операнда типа с плавающей точкой, то тогда будет выполняться деление типа с плавающей точкой. Здесь уже дробь присутствует. Например, выражения 7.0 / 3 = 2.333, 7 / 3.0 = 2.333 или 7.0 / 3.0 = 2.333 имеют один и тот же результат.

Попытки деления на 0 (или на 0.0) станут причиной сбоя в вашей программе.

Также есть две унарные арифметические операции, которые производятся над одним числом: ++ (инкремент – увеличение на единицу) и -- (декремент – уменьшение на единицу). Каждая из операций имеет две разновидности: префиксная и постфиксная (см. Таблицу 3.4):

Таблица 3.4 Унарные арифметические операции

Оператор	Название	Назначение	Пример

++x	Префиксный инкремент	Увеличивает значение переменной на единицу и полученный результат используется как значение выражения ++x	int a = 8;
			int b = ++a;
			cout << a << "\n"; // 9
			cout << b << "\n"; // 9

x++	Постфиксный инкремент	Увеличивает значение переменной на единицу,	int a = 8; int b = a++;

Оператор	Название	Назначение	Пример
		но значением выражения x++ будет то, которое было до увеличения на единицу	cout << a << "\n"; // 9 cout << b << "\n"; // 8

--x	Префиксный декремент	Уменьшает значение переменной на единицу, и полученное значение используется как значение выражения --x	int a = 8;
			int b = --a;
			cout << a << "\n"; // 7
			cout << b << "\n"; // 7

x--	Постфиксный декремент	Уменьшает значение переменной на единицу, но значением выражения x-- будет то, которое было до уменьшения на единицу	int a = 8;
			int b = a--;
			cout << a << "\n"; // 7
			cout << b << "\n"; // 8

Рассмотрим пример постфиксного варианта подробнее, на основе примера, представленного в таблице 3.4. Во-первых, компилятор создает временную копию a, которая имеет то же значение, что и оригинал (8). Затем увеличивается первоначальная a с 8 до 9. После этого компилятор возвращает временную копию, значением которой является 8, и присваивает её переменной b. Только после этого копия a уничтожается. Следовательно, в вышеприведенном примере, мы получим b = 8 и a = 9.

§3.10 Операторы сравнения и присваивания

Для сокращённой записи выражений есть специальные операции, которые называются операциями присваивания. Рассмотрим фрагмент кода, с использованием операции присваивания.

Листинг 3.9.

```
1  int value = 256
```

2	`value = value + 256 // обычное выражение с использованием двух операций:  = и +`
3	`value += 256 // сокращённое эквивалентное выражение с использованием операции присваивания`

В строке 2 переменной `value` присваивается значение 512, полученное в результате суммы значения, содержащегося в переменной `value` с числом 256. В строке 3 выражение выполняет аналогичную операцию, что и в строке 2, но выражение записано в упрощённом виде. В этом выражении присутствует операция присваивания со знаком плюс **+=**. Таким образом, операция **+=** суммирует значение переменной `value` со значением, которое находится правее: 256, и присваивает результат суммы этой же переменной. Как видно из примера оператор в строке 3 короче оператора в строке 2, хоть и выполняет аналогичную операцию. Так что, если некоторую переменную нужно изменить, то рекомендуется использовать операции присваивания.

Существует пять операций присваивания, не считая основную операцию присваивания: =.

1. += операция присваивания-сложения;
2. -= операция присваивания-вычитания;
3. *= операция присваивания-умножения;
4. /= операция присваивания-деления;
5. %= операция присваивания-остатка от деления.

В Таблице 3.5 наглядно показаны примеры использования операторов присваивания. Во всех примерах результат операции сохраняется в переменную `var`.

Таблица 3.5 Операторы присваивания

Операция	Оператор	Пример	Экв. пример	Пояснение
операция присваивания-сложения	+=	var += 16	var = var + 16	Прибавляем к значению переменной var число 16
операция присваивания-вычитания	-=	var -= 16	var = var — 16	Вычитаем из переменной var число 16
операция присваивания-умножения	*=	var *= 16	var = var * 16	Умножаем значение переменной var на 16
операция присваивания-деления	/=	var /= 16	var = var / 16	Делим значение переменной var на 16
операция присваивания-остатка от деления	%=	var %= 16	var = var % 16	Находим остаток от деления

Разработаем программу, которая будет использовать операции присваивания.

Листинг 3.10.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int value = 256;
6  cout << "value = " << value << endl;
```

```
7   value += 256; // сокращённое выражение с использованием операции присваивания - сложения
8   cout << "value += 256; >> " << value << endl;
9   value -= 256; // сокращённое выражение с использованием операции присваивания - вычитания
10  cout << "value -= 256; >> " << value << endl;
11  value *= 2; // сокращённое выражение с использованием операции присваивания - умножения
12  cout << "value *= 2;   >> " << value << endl;
13  value /= 8; // сокращённое выражение с использованием операции присваивания - деления
14  cout << "value /= 8;   >> " << value << endl;
15  return 0;
16  }
```

Для начала в строке 5 была объявлена переменная `value`, и инициализирована значением 256. В строках 7, 9, 11, 13, прописаны операции присваивания – сложения, вычитания, умножения и деления соответственно. После выполнения каждой операции присваивания оператор `cout` печатает результат.

Результат работы программы (см. Рисунок 3.1):

```
value = 256
value += 256; >> 512
value -= 256; >> 256
value *= 2;   >> 512
value /= 8;   >> 64
```

Рисунок 3.1

Существует 6 различных операторов сравнения в своем арсенале. Также существуют такие операторы, которые являются комбинациями других. Все они вам должны быть знакомы из курса математики, поэтому их изучение не должно вызвать у вас проблем. Операторы сравнения представлены в Таблице 3.6.

Таблица 3.6 Операторы сравнения

Оператор	Символ	Пример	Операция
Больше	>	x > y	true, если x больше y, в противном случае — false
Меньше	<	x < y	true, если x меньше y, в противном случае — false
Больше или равно	>=	x >= y	true, если x больше/равно y, в противном случае — false
Меньше или равно	<=	x <= y	true, если x меньше/равно y, в противном случае — false
Равно	==	x == y	true, если x равно y, в противном случае — false
Не равно	!=	x != y	true, если x не равно y, в противном случае — false

Каждый из этих операторов возвращает логическое значение true (1, истина) или false (0, ложь).

§3.11 Логические операторы

В то время как операторы сравнения используются для проверки конкретного условия: ложное оно или истинное, они могут проверить только одно условие за определенный промежуток времени. Но бывают ситуации, когда нужно протестировать сразу несколько условий. Например, чтобы узнать, выиграли ли мы в лотерею, нам нужно сравнить все цифры купленного билета с выигрышными. Если в лотерее 6 цифр, то нужно выполнить 6 сравнений, все из которых должны быть true. Проверить истинность одновременно всех шести условий помогут логические операторы.

Существует три логические операции:

1. Логическая операция И &&;
2. Логическая операция ИЛИ ||;
3. Логическая операция НЕ ! или логическое отрицание.

Логические операции образуют сложное (составное) условие из нескольких простых (двух или более) условий. Эти операции упрощают структуру программного кода в несколько раз.

В следующей таблице 3.7 кратко охарактеризованы все логические операции, используемые для построения логических условий.

Таблица 3.1 Логические операторы

Операции	**Обозначение**	**Условие**	**Краткое описание**
И	&&	a == 3 && b > 4	Составное условие истинно, если истинны оба простых условия
ИЛИ	||	a == 3 || b > 4	Составное условие истинно, если истинно, хотя бы одно из простых условий
НЕ	!	!(a == 3)	Условие истинно, если а не равно 3

Рассмотрим принцип работы следующей программы, и все будет понятно со всеми этими логическими операциями.

Листинг 3.10. Таблица истинности

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  bool a1 = true, a2 = false;
6  bool a3 = true, a4 = false;
7  cout << "Tablica istinnosti log operacii &&" << endl;
8  cout << "true && false: " << (a1 && a2) << endl
   << "false && true: " << (a2 && a1) << endl
   << "true && true: " << (a1 && a3) << endl
   << "false && false: " << (a2 && a4) << endl;
9  cout << "Tablica istinnosti log operacii ||" << endl;
```

```
10  cout << "true || false: " << (a1 || a2) << endl
    << "false || true: " << (a2 || a1) << endl
    << "true || true: " << (a1 || a3) << endl
    << "false || false: " << (a2 || a4) << endl;
11  cout << "Tablica istinnosti log operacii !"  << endl;
12  cout << "!true: " << (!a1) << endl << "!false: "<< (!a2)
    << endl;
13  return 0;
14  }
```

В строках 5-6 инициализируются переменные логического типа. Причем каждой переменной присваивается значение `true` или `false`. Начиная с 7-й строки и заканчивая 12-й, показано использование логических операций. Результат работы программы (см. Рисунок 3.2):

```
Tablica istinnosti log operacii &&
true  && false: 0
false && true: 0
true  && true: 1
false && false: 0
Tablica istinnosti log operacii ||
true  || false: 1
false || true: 1
true  || true: 1
false || false: 0
Tablica istinnosti log operacii !
!true: 0
!false: 1
```

Рисунок 3.2

§3.12 Приоритет операций

Приоритет операций — очерёдность выполнения операций в выражении, при условии, что в выражении нет явного указания порядка следования выполнения операций (с помощью круглых скобок).

Все операторы (операции) имеют свой уровень приоритета. Те, в которых он выше, выполняются первыми. В таблице, приведенной ниже, можно увидеть, что приоритет операций умножения и деления (5) выше, чем в операциях сложения и вычитания (6). Компилятор использует приоритет операторов для определения порядка обработки выражений.

Если две операции в выражении имеют одинаковый приоритет, то работает **правило ассоциативности**, которое указывает направление выполнения операций – справа налево ($R \to L$) или слева направо ($L \to R$).

§3.13 Определение типа переменной

Для определения типа переменной в C++ существует функция `typeid(ИМЯ_ПЕРЕМЕННОЙ).name()` из библиотеки `typeinfo` которую нужно предварительно подключить с помощью. В C# есть похожая функция `ИМЯ_ПЕРЕМЕННОЙ.GetType()`. В листинге 3.11 представлен пример работы метода, определяющего тип данных.

```
1  #include <string>
2  #include <iostream>
3  #include <typeinfo>
4  using namespace std;
5  int main()
6  {
7  // Создадим четыре переменные разных типов
8  int int_var = 1;
9  float float_var = 1.0;
10 char char_var = '0';
11 string str1 = "www.heihei.ru";
12 // Выведем на экран результат работы typeid
13 cout << typeid(int_var).name() << endl;
14 cout << typeid(float_var).name() << endl;
15 cout << typeid(char_var).name() << endl;
16 cout << typeid(str1).name() << endl;
17 return 0;
18 }
```

Результат выполнения программы:

int

float

char

class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >

§3.14 Неявное преобразование типов

Различают явное и неявное преобразование типов данных. Неявное преобразование типов данных выполняет компилятор, ну а явное преобразование данных выполняет сам программист.

Неявное преобразование типов (или «автоматическое преобразование типов») выполняется всякий раз, когда требуется один фундаментальный тип данных, но предоставляется другой, и пользователь не указывает компилятору, как выполнить конвертацию (не использует явное преобразование типов через операторы явного преобразования).

Есть 2 основных способа неявного преобразования типов:

- числовое расширение;
- числовая конверсия.

Числовое расширение

Когда значение из одного типа данных конвертируется в другой тип данных побольше (по размеру и по диапазону значений), то это называется числовым расширением. Например, тип `int` может быть расширен в тип `long`, а тип `float` может быть расширен в тип `double`:

```
long l(65); // расширяем значение типа int (65) в тип long
```

В языке C++ есть два варианта расширений:

- **Интегральное расширение** (или «целочисленное расширение»). Включает в себя преобразование целочисленных типов, меньших, чем int (bool, char, unsigned char, signed char, unsigned short, signed short) в int (если это возможно) или unsigned int.

- **Расширение типа с плавающей точкой**. Конвертация из типа float в тип double.

Интегральное расширение и расширение типа с плавающей точкой используются для преобразования «меньших по размеру» типов данных в типы int/unsigned int или double (они наиболее эффективны для выполнения разных операций).

Преимуществом данного варианта неявного преобразования является то, что числовые расширения всегда безопасны и не приводят к потере данных.

Числовая конверсия

Когда мы конвертируем значение из более крупного типа данных в аналогичный, но более мелкий тип данных, или конвертация происходит между разными типами данных, то это называется **числовой конверсией**. Например:

```
double d = 4; // конвертируем 4 (тип int) в double
```

В отличие от расширений, которые всегда безопасны, конверсии могут привести к потере данных. Поэтому в любой программе, где выполняется неявная конверсия, компилятор будет выдавать предупреждение.

Обработка арифметических выражений

При обработке выражений компилятор разбивает каждое выражение на отдельные подвыражения. Арифметические операторы требуют, чтобы их операнды были одного типа данных. Чтобы это гарантировать, компилятор использует следующие правила:

- Если операндом является целое число меньше (по размеру/диапазону) типа `int`, то оно подвергается интегральному расширению в `int` или в `unsigned int`;
- Если операнды разных типов данных, то компилятор вычисляет операнд с наивысшим приоритетом и неявно конвертирует тип другого операнда в такой же тип, как у первого.

Приоритет типов операндов:

1. long double (самый высокий);
2. double;
3. float;
4. unsigned long long;
5. long long;
6. unsigned long;
7. long;

8. unsigned int;

9. int (самый низкий).

Рассмотрим пример:

```
1  #include <iostream>
2  #include <typeinfo> // для typeid()
3  using namespace std;
4  int main()
5  {
6  double a(3.0);
7  short b(2);
8  cout << typeid(a + b).name() << " " << a + b << endl;
9  return 0;
10 }
```

Здесь `short` подвергается интегральному расширению в `int`. Однако int и `double` по-прежнему не совпадают. Поскольку `double` находится выше в иерархии типов, то целое число 2 преобразовывается в 2.0 (тип `double`), и сложение двух чисел типа `double` дадут число типа `double`:

double 5

§3.15 Явное преобразование типов

В языке C++ есть 5 видов операций явного преобразования типов:

- конвертация C-style;
- применение оператора static_cast;
- применение оператора const_cast;
- применение оператора dynamic_cast;
- применение оператора reinterpret_cast.

Подробнее рассмотрим первые два варианта явного преобразования типов.

Конвертация C-style

В программировании на языке Си явное преобразование типов данных выполняется с помощью оператора (). Внутри круглых скобок мы пишем тип, в который нужно конвертировать. Этот способ конвертации типов называется **конвертацией C-style**. Например:

```
int i1 = 11;
```

```
int i2 = 3;
float x = (float)i1 / i2;
```

В программе, приведенной выше, мы используем круглые скобки, чтобы сообщить компилятору о необходимости преобразования переменной i1 (типа int) в тип float. Поскольку переменная i1 станет типа float, то i2 также затем автоматически преобразуется в тип float, и выполнится деление типа с плавающей точкой!

Также позволяется использовать этот оператор следующим образом:

```
int i1 = 11;
int i2 = 3;
float x = float(i1) / i2;
```

Оператор static_cast

Оператор `static_cast` лучше всего использовать для конвертации одного фундаментального типа данных в другой:

```
int i1 = 11;
int i2 = 3;
float x = static_cast<float>(i1) / i2;
```

Основным преимуществом оператора `static_cast` является проверка его выполнения компилятором во время компиляции, что усложняет возможность возникновения непреднамеренных проблем.

§3.16 Математические функции

В языке программирования C++ в заголовочном файле <cmath> определены функции, решающие некоторые математические задачи. Например, нахождение корня, возведение в степень, sin(), cos() и многие другие. В Таблице 3.6 показаны основные математические функции, прототипы которых содержатся в заголовочном файле <cmath>.

Таблица 3.2 Математические функции

Функция	**Описание**	**Пример**
abs(a)	модуль или абсолютное значение от a	abs(-3.0)= 3.0 abs(5.0)= 5.0
sqrt(a)	корень квадратный из a, причём a не отрицательно	sqrt(9.0)=3.0
pow(a, b)	возведение a в степень b	pow(2,3)=8
ceil(a)	округление a до наименьшего целого, но не меньше чем a	ceil(2.3)=3.0 ceil(-2.3)=-2.0
floor(a)	округление a до наибольшего целого, но не больше чем a	floor(12.4)=12 floor(-2.9)=-3
fmod(a, b)	вычисление остатка от a/b	fmod(4.4, 7.5) = 4.4 fmod(7.5, 4.4) = 3.1
exp(a)	вычисление экспоненты e^a	exp(0)=1
sin(a)	a задаётся в радианах	
cos(a)	a задаётся в радианах	
log(a)	натуральный логарифм a (основанием является экспонента)	log(1.0)=0.0
log10(a)	десятичный логарифм a	Log10(10)=1
asin(a)	арксинус a, где -1.0 < a < 1.0	asin(1)=1.5708

Необходимо запомнить то, что операнды данных функций всегда должны быть вещественными, то есть a и b числа с плавающей точкой.