# Лекция 2

**Преобразование типов**

При вычислении выражений некоторые операции требуют, чтобы операнды имели соответствующий тип, иначе – при компиляции - сообщение об ошибке.

Например, операция взятия остатка от деления (%) требует целочисленных операндов.

Преобразование типов – это приведение значения переменной одного типа в значение другого типа.

Выделяют явное и неявное приведения типов.

При явном приведении указывается тип переменной, к которому необходимо преобразовать исходную переменную.

При неявном приведении преобразование происходит автоматически, по правилам, заложенным в C++.

Формат операции явного преобразования типов: имя_типа (операнд)

int(x), float(2/5), long(x+y/0.5)

Пример.

```
//Взятие цифры разряда сотых в дробном числе

#include "stdafx.h"

#include <iostream>

   using namespace std;

   int main( ){

   float s,t;

   long int a,b;

   printf("Введите вещественное число\n");

   scanf("%f", &s);

   t=s*100;

   a=(int)t; //переменная t приводится к типу int в переменную a

   b=a%10;

   printf("\n Цифра разряда сотых числа %f равна %d.", s, b);

   system("pause");

   return 0;

}
```

При выполнении математических операций производится неявное (автоматическое) преобразование типов, чтобы привести операнды

выражений к общему типу или чтобы расширить короткие величины до размера целых величин, используемых в машинных командах.

Выполнение преобразования зависит от специфики операций и от типа операнда или операндов.

Пример 1. Пусть необходимо разделить целых числа.

```
int x,y;

x=15;

y=2;
```

Делим: `x/y=7`

Если сделать преобразования: `x/y.=7.5`

Тогда число 2 - вещественное, результат тоже будет вещественный.

Само число 2 с точки зрения математики не изменилось, ведь 2=2.0.

Этот же можно было применить к 15, результат был бы тем же, а можно было сразу к двум числам, но зачем ,если хватает одного.

Пример 2.

```
int i1 = 11;

int i2 = 3;

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

**Арифметические операции**

Таблица 1.4. Арифметические операции

| Операция | Обозначение | Комментарий |
|---|---|---|
| Сложение | + | |
| Вычитание | - | |
| Умножение | * | |
| Деление | / | Если оба операнда целочисленные - то операция интерпретируется как целочисленное деление.<br><br>Если хотя бы один из операндов является числом с плавающей точкой, операция - обычное деление |
| Остаток от деления | % | Результат операции - остаток, образующийся при целочисленном делении делимого на делитель |

| Операция | Обозначение | Комментарий |
|---|---|---|
| Инкремент | ++ | Увеличение значения переменной на единицу. Выделяют префиксный (++x) и постфиксный (x++) инкременты. В первом случае значение переменной сначала увеличивается на единицу, а затем уже обновленное значение используется в вычислении содержащего инкремент выражения. В случае постфиксного инкремента сначала вычисляется выражение, а уже затем увеличивается значение инкрементируемой переменной |
| Декремент | -- | Уменьшение значения на единицу. Аналогично бывает префиксным и постфиксным |
| Побитовая инверсия | ~ | Изменение значения бита на противоположное |
| Побитовое И | & | Побитовая конъюнкция двух операндов |

| Операция | Обозначение | Комментарий |
|---|---|---|
| Побитовое ИЛИ | \| | Побитовая дизъюнкция двух операндов |
| Побитовое исключающее ИЛИ | ^ | Побитовое исключающее ИЛИ (сложение по модулю 2) двух операндов |
| Побитовые сдвиг влево (вправо) | << (>>) | Сдвиг на несколько битов влево (вправо). Например, запись a << 2 обозначает сдвиг битов переменной a на 2 разряда влево. |
| Присваивание | = | Оператор присваивания (не путать с сравнением) |
| Совместное присваивание | +=<br>-=<br>*=<br>/=<br>%=<br>&=<br>\|=<br><<= (>>=) | При использовании операторов составного присваивания сначала над двумя операндами выполняется операция, указанная до знака равно, а затем первому операнду присваивается результат данной операции.<br>Так, например, запись<br>X += Y<br>эквивалентна следующей записи: |

| Операция | Обозначение | Комментарий |
| --- | --- | --- |
| | | X = X + Y |

**Условия в языке С++**

```
...
if(x>=0 && x<=5){
    counter1++;
}
else if(x>=10 && x<=15){
    counter2++;
}
else{
    cout << "Точка не попала";
}
...
```

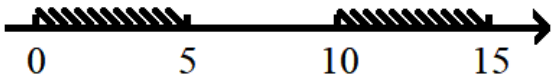


Рис. 1.7. Условия в С++

| Логическая операция | Обозначение |
|---|---|
| НЕ | ! |
| И | && |
| ИЛИ | \|\| |

| Операция сравнения | Обозначение |
|---|---|
| Равно | == |
| Не равно | != |
| Больше (меньше) | > (<) |
| Больше или равно (меньше или равно) | >= (<=) |

Рис. 1.8. Операции в условиях

**Оператор switch**

```
...
switch (x) {
    case 5:
        cout << "Отлично";
        break;
    case 4:
        cout << "Хорошо";
        break;
    case 3:
        cout << "Удовлетворительно";
        break;
    default:
        cout << "Неудовлетворительно";
}
...
```

Рис. 1.9. Пример использования оператора switch

**Параметрический цикл**

```
for(int i = 0; i < 10; i++) {
    cout << "i = " << i << endl;
}
```

Рис 1.10. Пример параметрического цикла

**Итерационные циклы**

**Цикл с предусловием:**

```
...
int i = 0;
while(i < 5) {
    cout << "i = " << i << endl;
    i++;
}
...
```

**Цикл с постусловием:**

```
...
int i = 0;
do{
    cout << "i = " << i << endl;
    i++;
} while(i < 5);
...
```

Рис. 1.11. Пример итерационных циклов

**Комментарии**

```
/* Это
многострочный
комментарий*/

int i = 0; // счетчик
while(i < 5) {
    cout << "i = " << i << endl;
    i++; // увеличиваем на единицу
}
```

Рис. 1.12. Пример использования комментариев

**Типы int, float, double**

Большинство компьютеров хранят int, используя 32 бита (4 байта), можно управлять int в диапазоне [-2147483648.. 2147483647].

Может быть, что:

• не нужны такие большие значения;

• нужны гораздо большие значения;

• не нужны отрицательные числа.

Язык C++ предоставляет методы для точного определения того, как хранить большие/малые числа. Это позволяет компилятору выделять память, либо меньшую, чем обычно (например, 16 бит вместо 32), либо большую (64 бита вместо 32).

Можно заявить, что значение переменной будет неотрицательным.

В этом случае ширина диапазона переменной не меняется, а смещается в сторону положительных чисел. Вместо диапазона [-2,147,483,648 .. 2,147,483,647] мы получаем диапазон [0..4294967295].

Дополнительные ключевые слова - модификаторы:

длинный long ;  короткий short ;

без знака unsigned, его можно использовать с типом char;

short int counter;  или  short counter;(длина 16 бит)

long int ants;  или  long ants;  (длина 64 бита)

Без знака (никогда не будет иметь отрицательного значения):

unsigned int positive;  или  unsigned positive;

Можно:

unsigned long int big_number;

unsigned long big_number;

unsigned short int lambs;   unsigned short lambs;

Модификаторы long  и short  не должны использоваться в сочетании с типом char  (длина которого всегда является минимально возможной).

Обычно при использовании целочисленных литералов предполагают, что они имеют тип int. Бывают случаи, когда компилятор распознает литералы типа long.

Это произойдет, если:

• буквальное значение выходит за пределы допустимого диапазона типа int;

• буква L или l добавляется к литералу: 0L или 1981l – эти литералы имеют тип long.

Модификаторы short и long  нельзя использовать вместе с float

Есть: double и long double - двойной и длинный двойной

Переменные типов double и long double могут отличаться от переменных типа float не только по диапазону, но и по точности.

Данные, хранящиеся в переменной float, имеют конечную точность, т.е. в переменной точно хранится только определенное количество цифр.

Переменная float сохраняет 8 точных цифр (длина 32 бита/4 байта).

Переменная double сохраняет 15-17 цифр  (длина 64 бита/8 байт)

Переменная long double  хранит 33-36 значащих цифр (128 бит/16 байт), этот тип иногда называют quadruple.

Некоторые аппаратные платформы предлагают еще более длинные типы с плавающей запятой long long double (длина 256 бит)

Обычный литерал с плавающей запятой, такой как 3.1415, распознается компилятором как double и занимает 64 бита памяти компьютера.

Чтобы любой ваш литерал был обычным float, следует добавить к нему суффикс "f" или "F".

3.1515f и 6.626E-34f относятся к типу float.

Компьютерное сложение:если очень маленькое значение float складывается с очень большим, то в результате меньшее значение может исчезнуть. Это явление - числовая аномалия. Замена float на double не всегда помогает.

Пример - Одна из самых известных неточностей с double.

Код кажется тривиальным. Очевидно, что 0,1+0,2=0,3. Так и есть, но не всегда!

Это следствие того, что числа, хранящиеся в виде данных с плавающей запятой, могут отличаться от их реальных (точных) значений.

```
#include <iostream>
```

```
using namespace std;

int main()

{

    double a = 0.1;

    double b = 0.2;

    double c = 0.3;

    if (a + b != c)

    cout << "Your computer is out of order";

}
```

Результат:

```
Your computer is out of order
```

**Цикл while**

Цикл while:

- проверяется условие перед входом в тело цикла
- если условие истинно, то выполняется тело цикла
- если условие ложно, то тело цикла не выполняется

Пример: цикл, который не завершит свое выполнение, будет бесконечно печатать на экране «I am stuck inside a loop».

```
while(true) {

  cout << "I am stuck inside a loop" << endl;
```

```
}
```

В C++ эти две формы эквивалентны:

```
while(number != 0) {...}
```

```
while(number) {...}
```

Условие, которое проверяет, является ли число нечетным:

```
if(number % 2 == 1)...
```

```
if(number % 2)...
```

### Цикл do

```
do {
  оператор_1;
     …
     оператор_n;
}
while (условное выражение)
```

Цикл «do» :

- условие проверяется в конце выполнения тела, тело цикла выполняется по крайней мере один раз, даже если условие не выполнено.

```
int main()
{
     int number;
     int max = -100000;
     int counter = 0;
```

```
    do {
    cin >> number;
    if (number != -1)
         counter++;
    if (number > max)
         max = number;
    } while (number != -1);
    if (counter)
    cout << "The largest number is " << max << endl;
    else
    cout << "You haven't entered any number!" << endl;
}
```

```
5
6
-1
The largest number is 6
```

**Параметрический цикл for**

```
int i = 0;
while(i < 100) {
  /* тело цикла */
  i++;
}
```

```
int i = 0;
 for (i = 0; i < 100;
i++) {
   /* тело цикла */
}
```

```
initialization;
while(checking) {
  /* the body goes
here */
```

```
for (initialization;
checking; modifying) {
```

```
        modifying;              /* the body goes here
      }                        */
                               }
```

Особенность цикла for: если опустить любой из его трех компонентов, предполагается, что вместо этого там есть 1.

Бесконечный цикл:

```
for(;;) {

    /* тело цикла */

  }
```

(Условного выражения нет, поэтому оно автоматически считается истинным. Условие никогда не становится ложным, цикл становится бесконечным).

Допускается запись:

```
for (i = 5,j = 10; i + j < 20;i++,j++)
```

```
int main(){

    int i, j;

    for ( i = 5, j = 10 ; i + j < 20; i++, j++ ) {

    cout << "i + j = " << (i + j) << '\n';

    }

}
```

Возможны случаи:

- нет необходимости продолжать цикл в целом; нужно прекратить выполнение тела цикла и идти дальше;

- нужно начать тестирование условий без завершения выполнения текущего кода.

**break и continue**

Язык C++ предоставляет две специальные инструкции для реализации этих задач.

В их существовании нет необходимости - опытный программист может запрограммировать любой алгоритм без этих инструкций.

Эти дополнения не улучшают выразительную силу языка, а только упрощают работу разработчика, их называют синтаксическим сахаром.

Это:

• **break** - выход из цикла и завершение работы цикла (прерывание выполнения цикла) ; программа начинает выполнять ближайшую инструкцию после тела цикла;

• **continue** – выполнение текущей итерации цикла останавливается; происходит переход к следующей итерации цикла.

**Компьютеры и их логика**

Оператор && - это оператор конъюнкции, ключевое слово and – синоним &&.

Приоритет у && ниже, чем у операторов сравнения, поэтому можно кодировать сложные условия без использования скобок:

```
counter > 0 && value == 100
```

Оператор || - это оператор дизъюнкции, синоним or.

Оператор ! - это оператор отрицания, синоним not.

**Логические выражения**

Законы Де Моргана:

Отрицание конъюнкции - это дизъюнкция отрицаний.

Отрицание дизъюнкции - это конъюнкция отрицаний.

На языке C++:

```
!(p && q) == !p || !q

!(p || q) == !p && !q
```

Ни один из операторов с двумя аргументами не может использоваться в сокращенной форме `op=`.

**Работа с одиночными битами**

Логические операторы принимают свои аргументы целиком, независимо от того, сколько битов они содержат.

Операторам известно только одно значение:

- 0 (все биты сброшены) означает “ложь”;
- не 0 (установлен хотя бы один бит) означает “истина”.

Результатом операций является одно из значений: 0 или 1.

Фрагмент:

```
bool i = false;

bool j = !!i;
```

присвоит переменной j значение 1, если i не равно нулю; в противном случае оно будет равно 0

Побитовые операторы (bitwise operators) - позволяют манипулировать отдельными битами данных.

Они охватывают все операции (И, ИЛИ, НЕ) и один дополнительный оператор - xor (исключающее ИЛИ), обозначается ^.

Разница в работе логических и битовых операторов: логические операторы не проникают на битовый уровень своего аргумента. Их интересует только конечное логическое значение.

Побитовые операторы обрабатывают каждый бит отдельно.

Пример - разница в работе между логическими и битовыми операциями.

Пусть выполнено объявление:

```
int i = 15, j = 22;
```

Предположим, что int хранятся с 32 битами, тогда побитовое изображение переменных:

```
i: 00000000000000000000000000001111

j: 00000000000000000000000000010110
```

Логическая связь: `int log = i && j;`

Ход вычислений: обе переменные i и j не являются нулями, поэтому будут считаться как true “истина”.

true && true - результат будет true “истинна”, это целое число =1.

Побитовое изображение переменной:

```
log:00000000000000000000000000000001
```

Побитовая операция: `int bit = i & j;`

Оператор & будет работать с каждой парой битов отдельно. Результат:

```
bit:00000000000000000000000000000110
```

Это соответствует целому числу = 6.

Логический: `int i = 15; logneg = !i;`

Переменной logneg будет присвоено значение 0, поэтому ее изображение будет состоять только из нулей. Результат:

```
logneg: 00000000000000000000000000000000
```

Побитовое отрицание: int i = 15, bitneg = ~i;

Результат:

```
bitneg: 11111111111111111111111111110000
```

Значение переменной bitneg = -16.

Если ответ не понятен, то повторить двоичную системы счисления

**Применение одиночных бит**

Пусть вам нужно написать часть ОС. Вы должны использовать переменную:
`int flag_register;`

Она хранит информацию о различных аспектах работы ОС. Каждый бит переменной хранит одно значение "да"/"нет". Пусть только один из этих битов ваш – бит номер три. Остальные биты изменять нельзя.

Ваш «кусочек» помечен буквой “x”:

00000000000000000000000000x000

Могут быть задачи:

**#1**: Проверить состояние вашего бита (узнать значение бита x). Сравнение всей переменной с нулем ничего не даст, т.к. оставшиеся биты могут иметь непредсказуемые значения. Можно использовать свойство побитовое И:

```
x & 1 = x   x & 0 = 0
```

Применим операцию & к переменной flag_register вместе со битовым изображением:

```
 00000000000000000000000000001000
```

("1" - в позиции вашего бита), в результате получим одну из битовых строк:

`00000000000000000000000000001000` Если ваш бит был установлен “1”

`00000000000000000000000000000000` Если ваш бит был установлен “0”.

Последовательность 0 и 1, задача которой - захватить значение или изменить выбранные биты, называется битовой маской.

Создадим битовую маску для определения состояние вашего бита. Это 3-ий бит, имеет вес 2^3 = 8. Маска:

```
int the_mask = 8;  или: int the_mask = 0b1000;
```

Последовательность инструкций в зависимости от состояния вашего бита:

```
if (flag_register & the_mask) {
    /* my bit is set */
} else {
    /* my bit is reset */
```

**#2**: Сбросить свой бит – присвоить биту ноль, все остальные биты остаются неизменными. Используем побитовое И, но другую маску:

```
11111111111111111111111111110111
```

Маска создана отрицанием всех битов переменной the_mask .

Сброс бита (два варианта) (~ поразрядное отрицание):

```
flag_register = flag_register & ~the_mask;
flag_register &= ~the_mask;
```

**#3**: Установить свой бит – назначить “единицу” своему биту. Все остальные биты должны оставаться неизменными. Используем свойство дизъюнкции:

```
 x | 1 = 1   x | 0 = x
```

Установка бита - с помощью одной из инструкций:

```
flag_register = flag_register | the_mask;

flag_register |= the_mask;
```

**#4**: Отрицание своего бита – заменить “1” на “0”, а “0” на “1”.

Используем свойство оператора xor (исключающее ИЛИ):

```
x ^ 1 = !x   x ^ 0 = x
```

Отрицание бита:

```
flag_register = flag_register ^ the_mask;

flag_register ^= the_mask;
```

**Сдвиг битов**

Операция, связанная с отдельными битами - сдвиг **shifting**.

Применяется только к целочисленным значениям аргументов.

Сдвиг битов:

• логический – все биты переменной сдвинуты (применяется к целым числам без знака);

• арифметический - сдвиг пропускает знаковый бит, роль знакового бита играет старший бит переменной; если он равен "1", значение обрабатывается как отрицательное; арифметический сдвиг не может изменить знак сдвинутого значения.

Операторы сдвига в языке C++ представляют собой пару: << и >>, четко указывающих, в каком направлении будет действовать сдвиг. Левый аргумент целочисленное значение, биты которого сдвигаются, правый - определяет

размер сдвига. Приоритет этих операторов очень высок. value << bits  value >> bits

**switch-case и if**

Каскад if - это название для построения кода, в котором множество инструкций if размещаются последовательно одна за другой.

```
if(i == 1)
   cout << "Only one?" << endl;
else if(i == 2)
   cout << "I want more" <<endl;
else if(i == 3)
   cout << "Not bad" << endl;
else
   cout << "OK" << endl;
```

```
switch(i) {
  case 1:
    cout << "Only one?" << endl;
    break;
  case 2:
    cout<<"I want more" << endl;
    break;
  case 3:
    cout << "Not bad" << endl;
    break;
  case 4:
    cout << "OK"<< endl;
  default:
    cout<<"Don't care"<<endl;
}
```

- Значение после switch не должно быть выражением, содержащим переменные или другие сущности, значения которых неизвестны во время компиляции.

- Значение по умолчанию default может быть расположено в любом месте (даже в качестве метки первого переключателя), хотя размещение его в последней позиции облегчает анализ кода;

- Все метки должны быть уникальными.

- Значение по умолчанию default не может быть использовано более одного раза.