# 1.7 C++ OPERATORS

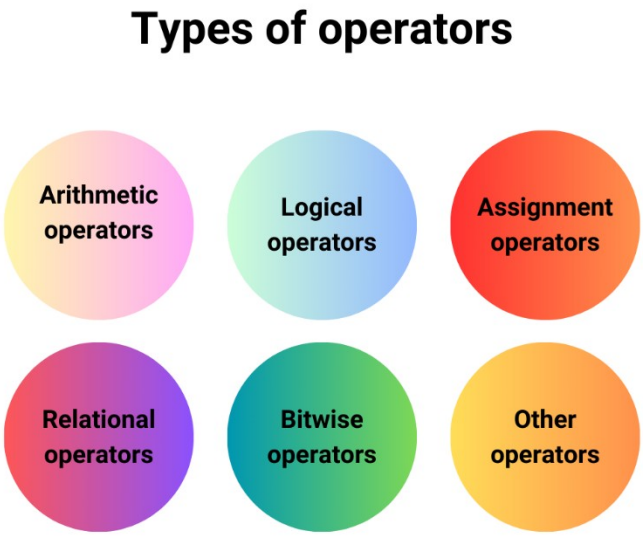
Computers process data! They read them from the keyboard, store them in variables (or constants), display them on the screen. And we do various **operations** with them. We are used to doing **arithmetic operations** (addition, subtraction, etc.), but in C++ there are **many other operations**.

## Introduction

An operation consists of **operands** and **operator**. The operands represent the **data** with which the operations are done, and the operator is the **symbol** that determines what operation is done with the operands. From the point of view of the number of operands, the operations (operators) can be:

- **unary** – applies to a single operand (for example -7, the operation to change the sign of a number);
- **binary** – applies to two operands (for example addition of numbers, 2+5);
- **ternar**y – applies to three operands. There is only one ternary operator in C++, the conditional operator, and it will be discussed below.

Operands can be **variables, constants, literals, the results of some functions, the results of other operations**. An operation that has other operations as operands is called an **expression**.




## Arithmetic operators

These are: +, -, *, /, %.

In the examples below, we consider the variables:

- N = 11 and M = 3 of type *int*
- X = 11 and Y = -3.5 of type *double*

### Unary arithmetic operators

In C++ there are the unary operations + and -:

- **+** returns the value of the operand
- **-** returns the value of the operand with changed sign

e.g.: + X = 11

- Y = 3.5

- + N = -11

### Binary arithmetic operators

- **+** : addition of two numbers;
- **-** : subtraction of two numbers;
- ***** : multiplication of two numbers;
- **/** : division of two numbers;
- **%** : remainder of division of two integers (modulo);
- There is no exponentiation operator in C++!

Addition, subtraction, and multiplication behave as expected, just like math. The division operation and the modulo operation require some further explanation.

### Integer division and decimal division

The division operation has two ways of working, depending on the type of operands.

- If the operands are of type integer (*int, short, char*, etc.), the integer division will be performed, and the result of the / operation is the **quotient** of the integer division.
- If the operands are of real type (*float, double, long double*), the decimal division will be performed, and the result of the / operation is the **result** of this division, **"with a comma"**.

e.g.:

- N / M = 3
- X / Y = -3.14286
- X / 2.0 = 5.5
- M / 2 = 1
- M / 2.0 = 1.5

The last division is special. The two operands have different types: M = 3 is of type *int*, and 2.0 is of type *double*. This is where the implicit conversion operation comes in: automatically, the value of the operand M is considered to be of type *double*, the division is real division, and the result is 1.5.

### The modulo operator (%)

The modulo operation only makes sense if **both operands** are of type **integer** - division by remainder only makes sense in this situation. Here are some examples:

- N % M = 2 : The remainder of dividing 11 by 3 is 2
- 30% 10 = 0

The modulo operator is useful in many situations. It can be used to find the **last digit of a natural number**: the last digit of 276 is 276 % 10 i.e. 6, or to check **if a number N is a divisor of M**. If so, M % N is 0.

## Relational operators

They are: <, >, <=, >=, ==, != .

A relational operator determines whether **a certain relationship occurs between two numbers** (the operands). The result of this operation is true or false. The result of relational operations can be 0 or 1:

- the result is 1 if the relation is true

- the result is 0 if the relation is false

Let N = 11 and M = 3. The relational operations are:

- comparison operations (comparisons)
- smaller <; N < M is false, i.e. 0
- greater >; N > M is true, i.e. 1
- less than or equal to <=; M <= N is 1
- greater than or equal to >=; M >= N is 0
- equality operation == ; N == M is false, i.e. 0
- the inequality operation (different, not equal) !=; N != M is true, i.e. 1.

## Logical operators

They are: !, ||, &&.

Logical operators have **truth-valued operands and truth-valued results**. Historically, logical operations are linked to the name of the English mathematician **George Boole**, who laid the foundations of this branch of mathematics and invented Boolean algebra and propositional calculus.

In C++, logical operators can be applied to any numeric values, and result in one of the values **0 or 1**. In the examples below we will use the bool literals **true** and **false**.

### The negation: !

- *! true* is *false*. Any negated non-zero value becomes 0.
- *! false* is *true*. 0 negated becomes 1.

### Disjunction: ||

- *false || false → false*
- *false || true → true*
- *true || false → true*
- *true || true → true*

### Conjunction: &&

- *false && false → false*
- *false && true → false*
- *true && false → false*
- *true && true → true*

### De Morgan's Laws

Consider *p* and *q* two boolean values (they can be the results of some expressions, for example). Then:

- *!(p && q) == !p || q*
- *!(p || q) == !p && !q*

Let's take as an example the membership of a value in a range:

- $x \in [a,b]$

The equivalent C++ expression is $x \geq a$ && $x \leq b$

According to **De Morgan's laws**, by negation we obtain: $!(x \geq a)$ || $!(x \leq b)$

Since !(x ≥ a) is equivalent to x<a, we get: $x < a$ || $x > b$

Using the intervals we get: $x \in (-\infty,a) \cup (b,+\infty)$, i.e. $x \notin [a,b]$

## The assignment operator: =

Assignment is the operation by which a variable receives the value of an expression:

- *variable = expression*

The **expression** can have any kind of result, if its type is identical to that of the variable or it can be converted to the type of the **variable**. In the case of **integer, real, bool types**, any of these can be converted to **any other**, possibly with truncation of some values.

e.g.:

main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n , m; // random values
    double x , y; // random values
    n = 5; // value of n becomes 5
    cout << n << endl;
    m = n + 2; // value of m becomes 7
    cout << m << endl;
    n = n + 3; // the value of n becomes 5 + 3, i.e. 8
    cout << n << endl;
    x = m / 5; // the value of x becomes 8 / 5, that is 1. ATTENTION! is whole division
    cout << x << endl;
    y = 5; // the value of y becomes 5, of type double. The conversion of 5 of type int to double takes place
    cout << y << endl;
    x = m / y; // the value of x becomes 1.4, because the division is decimal. The value of m is converted to double, then the division is done
    cout << x << endl;
    return 0;
}
```

Multiple assignments are also possible, as below:

main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    int a,b,c;
    a=b=c=10;
}
```

All variables will be given the value 10.

The following assignment is more interesting:

main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n;
    n=n+4;
}
```

It is performed as follows (let us consider, as an example, that the initial value of n is 5):

- first the expression on the right is calculated, where the current value of n is used: n + 4 is 5 + 4 i.e. 9
- the value of the expression on the right is assigned to the variable on the left, so n becomes 9.

## Compound assignment operators

They are: +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=.

In programming, assignments of the form are very common:

main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n;
    n=n*5;
}
```

where a certain arithmetic operation is applied to a variable (in the above example *) and the result is stored in that variable itself. To make it easier to write code in these situations, in C++ there is compound assignment:

- *var OP= expression*, equivalent to *var = var OP expression*

Thus, the assignment **x = x * 5** is equivalent to **x *= 5**.

## Interchange

Interchange is the operation by which **the values of two variables are exchanged**. For example, if the variables A and B have the values 5 and 7 respectively, after swapping B will have the value 5 and A will have the value 7.

The exchange is done by **successive exchanges**. There are several methods, but the most widely used one is known as the rule of glasses, because it is similar to how the contents of two glasses change. In this case, one more glass is needed, and in the case of alternating variables, an auxiliary variable is needed.

The exchange scheme is as follows:

main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
   int A = 5 , B = 7;
    int aux = A;
    A = B;
    B = aux;
    cout << A << " " << B; // 7 5
}
```

## The operators ++, --

They are called **increment** (++) and **decrement** (--) operators.

Incrementing a variable means increasing its value by 1. Similarly, decrementing a variable means **decreasing its value by 1**.

The operation **to increment** the variable X can be:

- **postincrement**: X ++. The effect of the expression is to increase the value of X by 1, and the result of the operation is the original value of X.

main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x = 5 , y = 10;
    y = x ++; // y receives the value of (x++), i.e. the initial value of x
    cout << x << " " << y; // 6 5
}
```

- **preincrement**: ++ X. The effect of the expression is to increment the value of X by 1, and the result of the operation is the variable X itself.

main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x = 5 , y = 10;
    y = ++ x; // y receives the value of (++x), i.e. the increased value of x
    cout << x << " " << y; // 6 6
}
```

The **decrement** operation of the variable X can be:

- **postdecrement**: X --. The effect of the expression is to decrement the value of X by 1, and the result of the operation is the original value of X.
- **predecrement**: -- X. The effect of the expression is to decrement the value of X by 1, and the result of the operation is the variable X itself.

## The conditional operator (?)

The conditional operator is the **only ternary operator** (with three operands) in C++. Its syntax is:

- *expression1 ? expression2 : expression3*

And is evaluated as follows:

- expression1 is evaluated
- if the result of expression1 is null (true), expression2 is evaluated and the result of this expression will be the result of the ?
- if the result of expression1 is null (false), expression3 is evaluated and the result of this expression will be the result of the ?
- expression2 and expression3 must have results of the same type, or of compatible types.

e.g.:

main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x;
    cin >> x;
    cout << (x % 2 == 0? "even" : "odd");
}
```

## The comma operator (,)

In some situations, the syntax rules of the C++ language require the **presence of a single operation**, but the logic of the program requires the presence of several operations. They can be grouped using the , operator. The syntax of this operation is:

- *expression1 , expression2*

The evaluation method is:

- expression1 is evaluated first, then expression2 – important, if variables appear in expression2 that change in expression1
- the result of the operation is the result of expression2

e.g.:

main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x , y , z;
    x = 1, y = 2, z = 3;
    x ++, y = x + 2, z -= x; // the order in which the three expressions were evaluated is significant
    cout << x << " " << y << " " << z; // 2 4 1
}
```

## The bitwise operators

They are: &, |, ^, ~, <<, >>.

Bitwise operators are an **advanced programming topic**. They allow direct and very fast manipulation of the bits that make up the memory representation of a data.

## The explicit conversion operator

In certain situations we must consider an expression of one type to be of another type. This can be done via the **conversion operator**:

- *(new_type) expression*

e.g.:

1. main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x = 2;
    cout << 7 / x << endl; // 3 - is whole division
    cout << 7 / (double) x; // 3.5 - is decimal division
}
```

2. main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    char p = 'a';
    cout << (int)p << endl; // 97, the ASCII code of 'a'
    cout << p - 32 << endl; // 65
    cout << (char)(p - 32); // A - character with ASCII code 65
}
```

## The sizeof operator

The *sizeof* operator is a unary operator that applies to a data type or an expression. Its result is the number of bytes it occupies once of that type, i.e. **the result of the expression**.

e.g.:

main.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    cout << sizeof(double) << endl; // 8: a double date occupies 8 bytes
    cout << sizeof(3 + 5) << endl; // 4: 3 + 5 is of type int; an int type date occupies 4 bytes
}
```

## Other operators

The C++ language also contains other operators, including:

- ( ) – changing the priority of an operation, function call
- [ ] – indexing an array
- ., -> – access to the members of a structure
- &, * – referencing (determining the address of a variable), dereferencing (accessing the variable from an address)
- new, delete – memory allocation and deallocation
- <<, >> – insert and extract from stream
- :: the resolution operator

## Operator priority

Operator precedence determines **the order in which an expression containing multiple operators of various kinds is evaluated**—the order in which operations are performed.

**Operator association** determines the order in which an expression containing multiple operators with the same precedence is evaluated. It can be **left to right or right to left**.

Both precedence and association of operators can be modified using parentheses ()

For the operators presented above, the priority is as follows:

**Priority level 1.**

Association: left right

Resolution operator ::

**Priority level 2.**

Association: left right

Postincrement / postdecrement ++ --

Access to the elements of an array []

Changing the priority of an operation, function call()

Access to the field of a structure. ->

**Priority level 3.**

Association: right left

Preincrement / predecrement ++ --

Bitwise negation ~, logical negation !

Unary arithmetic operators + -

Referencing / dereferencing & *

Memory allocation / deallocation new delete

The sizeof operator

The explicit conversion operator

**Priority level 4.**

Association: left right

Pointers to members of a structure.* ->*

**Priority level 5.**

Association: left right

Multiplicative arithmetic operations * / %

**Priority level 6.**

Association: left right

Additive arithmetic operations + -

**Priority level 7.**

Association: left right

Bitwise shift <<, >>

**Priority level 8.**

Association: left right

Comparison relational operators < > <= >=

**Priority level 9.**

Association: left right

The relational operators of equality / inequality == !=

**Priority level 10.**

Association: left right

AND bitwise &

**Priority level 11.**

Association: left right

EXCLUSIVE OR bitwise ^

**Priority level 12.**

Association: left right

OR (OR INCLUDING) bitwise |

**Priority level 13.**

Association: left right

Logical conjunction &&

**Priority level 14.**

Association: left right

Logical disjunction ||

**Priority level 15.**

Association: right left

Assignment, compound assignment = += -= *= /= %= >>= <<= &= ^= |=

The conditional operator ?

**Priority level 16.**

Association: left right

Operator ,

## Pay attention to!

- Every C++ operation has a different result!
- One of the most common errors is using the = operator instead of == for the equality operation. The = operator represents the assignment operation!
- Another common error occurs when comparing multiple numbers. From mathematics we are used to comparing numbers like this: a < b < c – the condition is true if the numbers are in strictly ascending order. In C++, the result of this operation may be different than expected due to the way the operations are done.
- Assignment is an operation, so it has a result! The result of the assignment operation is the very variable that receives the value.
- Do not confuse the = assignment operation with the == equality operation.
-