# Variables. Data Types.

The usefulness of the "Hello World" programs shown in the previous section is quite questionable. We had to write several lines of code, compile them, and then execute the resulting program just to obtain a simple sentence written on the screen as result. It certainly would have been much faster to type the output sentence by ourselves. However, programming is not limited only to printing simple texts on the screen. In order to go a little further on and to become able to write programs that perform useful tasks that really save us work we need to introduce the concept of variable.

Let us think that I ask you to retain the number 5 in your mental memory, and then I ask you to memorize also the number 2 at the same time. You have just stored two different values in your memory. Now, if I ask you to add 1 to the first number I said, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Values that we could now for example subtract and obtain 4 as result.

The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:

```
a = 5;
b = 2;
a = a + 1;
result = a - b;
```

Obviously, this is a very simple example since we have only used two small integer values, but consider that your computer can store millions of numbers like these at the same time and conduct sophisticated mathematical operations with them.

Therefore, we can define a variable as a portion of memory to store a determined value.

Each variable needs an identifier that distinguishes it from the others, for example, in the previous code the variable identifiers were `a`, `b` and `result`, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

## Identifiers

A valid identifier is a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline character (_ ), but in some cases these may be reserved for compiler specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case they can begin with a digit.

Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C++ language nor your compiler's specific ones, which are *reserved keywords*. The standard reserved keywords are:

```
asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete,
do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto,
if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register,
reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template,
this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void,
volatile, wchar_t, while
```

Additionally, alternative representations for some operators cannot be used as identifiers since they are reserved words under some circumstances:

```
and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq
```

Your compiler may also include some additional specific reserved keywords.

**Very important:** The C++ language is a "case sensitive" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the `RESULT` variable is not the same as the `result` variable or the `Result` variable. These are three different variable identifiers.

# Fundamental data types

When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

| Name | Description | Size* | Range* |
|---|---|---|---|
| `char` | Character or small integer. | 1byte | signed: -128 to 127<br>unsigned: 0 to 255 |
| `short int` (`short`) | Short Integer. | 2bytes | signed: -32768 to 32767<br>unsigned: 0 to 65535 |
| `int` | Integer. | 4bytes | signed: -2147483648 to 2147483647<br>unsigned: 0 to 4294967295 |
| `long int` (`long`) | Long integer. | 4bytes | signed: -2147483648 to 2147483647<br>unsigned: 0 to 4294967295 |
| `bool` | Boolean value. It can take one of two values: true or false. | 1byte | true or false |
| `float` | Floating point number. | 4bytes | +/- 3.4e +/- 38 (~7 digits) |
| `double` | Double precision floating point number. | 8bytes | +/- 1.7e +/- 308 (~15 digits) |
| `long double` | Long double precision floating point number. | 8bytes | +/- 1.7e +/- 308 (~15 digits) |
| `wchar_t` | Wide character. | 2 *or* 4 bytes | 1 wide character |

* The values of the columns **Size** and **Range** depend on the system the program is compiled for. The values shown above are those found on most 32-bit systems. But for other systems, the general specification is that `int` has the natural size suggested by the system architecture (one "*word*") and the four integer types `char`, `short`, `int` and `long` must each one be at least as large as the one preceding it, with `char` being always 1 byte in size. The same applies to the floating point types `float`, `double` and `long double`, where each one must provide at least as much precision as the preceding one.

# Declaration of variables

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier. For example:

```
int a;
float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type int with the identifier a. The second one declares a variable of type float with the identifier mynumber. Once declared, the variables a and mynumber can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

```
int a, b, c;
```

This declares three variables (a, b and c), all of them of type int, and has exactly the same meaning as:

```
int a;
int b;
int c;
```

The integer data types char, short, long and int can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier signed or the specifier unsigned before the type name. For example:

```
unsigned short int NumberOfSisters;
signed int MyAccountBalance;
```

By default, if we do not specify either signed or unsigned most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

```
int MyAccountBalance;
```

with exactly the same meaning (with or without the keyword signed)

An exception to this general rule is the char type, which exists by itself and is considered a different fundamental data type from signed char and unsigned char, thought to store characters. You should use either signed or unsigned if you intend to store numerical values in a char-sized variable.

short and long can be used alone as type specifiers. In this case, they refer to their respective integer fundamental types: short is equivalent to short int and long is equivalent to long int. The following two variable declarations are equivalent:

```
short Year;
short int Year;
```

Finally, signed and unsigned may also be used as standalone type specifiers, meaning the same as signed int and unsigned int respectively. The following two declarations are equivalent:

```
unsigned NextYear;
unsigned int NextYear;
```

To see what variable declarations look like in action within a program, we are going to see the C++ code of the example about your mental memory proposed at the beginning of this section:

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;
}
```

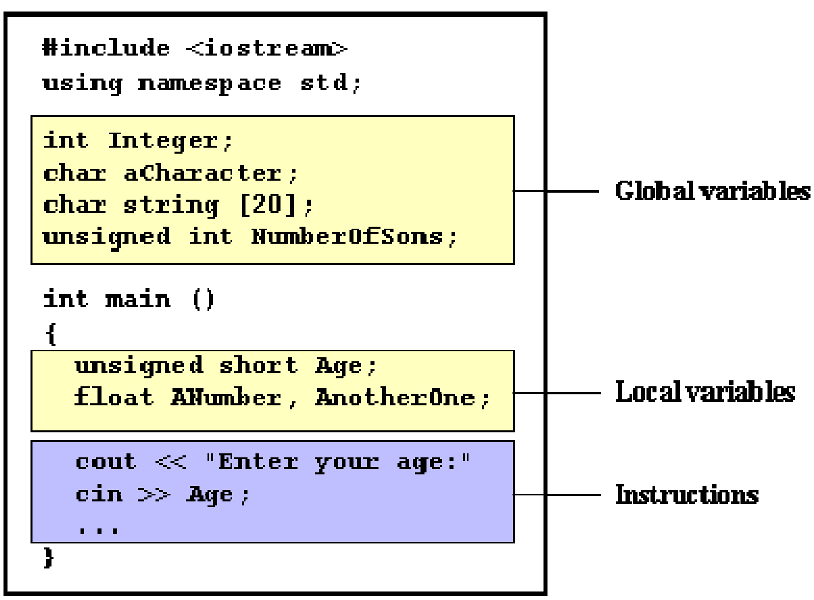
```
4
```

Do not worry if something else than the variable declarations themselves looks a bit strange to you. You will see the rest in detail in coming sections.

## Scope of variables

All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that `a`, `b`, and `result` were of type `int`.

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.




Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

The scope of local variables is limited to the block enclosed in braces ({}) where they are declared. For example, if they are declared at the beginning of the body of a function (like in function `main`) their scope is between its declaration point and the end of that function. In the example above, this means that if another function existed in addition to `main`, the local variables declared in `main` could not be accessed from the other function and vice versa.

# Initialization of variables

When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++:

The first one, known as c-like, is done by appending an equal sign followed by the value to which the variable will be initialized:

```
type identifier = initial_value ;
```

For example, if we want to declare an int variable called a initialized with a value of 0 at the moment in which it is declared, we could write:

```
int a = 0;
```

The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses (`()`):

```
type identifier (initial_value) ;
```

For example:

```
int a (0);
```

Both ways of initializing variables are valid and equivalent in C++.

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
  int a=5;               // initial value = 5
  int b(2);              // initial value = 2
  int result;            // initial value
undetermined

  a = a + 3;
  result = a - b;
  cout << result;

  return 0;
}
```

```
6
```

# Introduction to strings

Variables that can store non-numerical values that are longer than one single character are known as strings.

The C++ language library provides support for strings through the standard `string` class. This is not a fundamental type, but it behaves in a similar way as fundamental types do in its most basic usage.

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type we need to include an additional header file in our source code: `<string>` and have access to the `std` namespace (which we already had in all our previous programs thanks to the `using namespace` statement).

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring = "This is a string";
  cout << mystring;
  return 0;
}
```

```
This is a string
```

As you may see in the previous example, strings can be initialized with any valid string literal just like numerical type variables can be initialized to any valid numerical literal. Both initialization formats are valid with strings:

```
string mystring = "This is a string";
string mystring ("This is a string");
```

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and being assigned values during execution:

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is the initial string content";
  cout << mystring << endl;
  mystring = "This is a different string content";
  cout << mystring << endl;
  return 0;
}
```

```
This is the initial string content
This is a different string content
```

For more details on C++ strings, you can have a look at the string class reference.

# Constants

Constants are expressions with a fixed value.

## Literals

Literals are used to express particular values within the source code of a program. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
a = 5;
```

the 5 in this piece of code was a literal constant.

Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

### *Integer Numerals*

```
1776
707
-273
```

They are numerical constants that identify integer decimal values. Notice that to express a numerical constant we do not have to write quotes (") nor any special character. There is no doubt that it is a constant: whenever we write 1776 in a program, we will be referring to the value 1776.

In addition to decimal numbers (those that all of us are used to use every day) C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we have to precede it with a 0 (zero character). And in order to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
75          // decimal
0113        // octal
0x4b        // hexadecimal
```

All of these represent the same number: 75 (seventy-five) expressed as a base-10 numeral, octal numeral and hexadecimal numeral, respectively.

Literal constants, like variables, are considered to have a specific data type. By default, integer literals are of type int. However, we can force them to either be unsigned by appending the u character to it, or long by appending l:

```
75          // int
75u         // unsigned int
75l         // long
75ul        // unsigned long
```

In both cases, the suffix can be specified using either upper or lowercase letters.

### *Floating Point Numbers*

They express numbers with decimals and/or exponents. They can include either a decimal point, an e character (that expresses "by ten at the Xth height", where X is an integer value that follows the e character), or both a decimal point and an e character:

```
3.14159    // 3.14159
6.02e23    // 6.02 x 10^23
1.6e-19    // 1.6 x 10^-19
3.0        // 3.0
```

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated- and the last one is the number three expressed as a floating-point numeric literal.

The default type for floating point literals is `double`. If you explicitly want to express a `float` or `long double` numerical literal, you can use the `f` or `l` suffixes respectively:

```
3.14159L   // long double
6.02e23f   // float
```

Any of the letters that can be part of a floating-point numerical constant (`e`, `f`, `l`) can be written using either lower or uppercase letters without any difference in their meanings.

## *Character and string literals*

There also exist non-numerical constants, like:

```
'z'
'p'
"Hello world"
"How do you do?"
```

The first two expressions represent single character constants, and the following two represent string literals composed of several characters. Notice that to represent a single character we enclose it between single quotes (`'`) and to express a string (which generally consists of more than one character) we enclose it between double quotes (`"`).

When writing both single character and string literals, it is necessary to put the quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

```
x
'x'
```

`x` alone would refer to a variable whose identifier is `x`, whereas `'x'` (enclosed within single quotation marks) would refer to the character constant `'x'`.

Character and string literals have certain peculiarities, like the escape codes. These are special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (`\n`) or tab (`\t`). All of them are preceded by a backslash (`\`). Here you have a list of some of such escape codes:

| | |
|---|---|
| `\n` | newline |
| `\r` | carriage return |
| `\t` | tab |
| `\v` | vertical tab |
| `\b` | backspace |
| `\f` | form feed (page feed) |
| `\a` | alert (beep) |
| `\'` | single quote (') |
| `\"` | double quote (") |
| `\?` | question mark (?) |
| `\\` | backslash (\) |

For example:

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

Additionally, you can express any character by its numerical ASCII code by writing a backslash character (\) followed by the ASCII code expressed as an octal (base-8) or hexadecimal (base-16) number. In the first case (octal) the digits must immediately follow the backslash (for example \23 or \40), in the second case (hexadecimal), an x character must be written before the digits themselves (for example \x20 or \x4A).

String literals can extend to more than a single line of code by putting a backslash sign (\) at the end of each unfinished line.

```
"string expressed in \
two lines"
```

You can also concatenate several string constants separating them by one or several blank spaces, tabulators, newline or any other valid blank character:

```
"this forms" "a single" "string" "of characters"
```

Finally, if we want the string literal to be explicitly made of wide characters (wchar_t), instead of narrow characters (char), we can precede the constant with the L prefix:

```
L"This is a wide character string"
```

Wide characters are used mainly to represent non-English or exotic character sets.

### *Boolean literals*

There are only two valid Boolean values: true and false. These can be expressed in C++ as values of type bool by using the Boolean literals true and false.

## Defined constants (#define)

You can define your own names for constants that you use very often without having to resort to memory-consuming variables, simply by using the #define preprocessor directive. Its format is:

```
#define identifier value
```

For example:

```
#define PI 3.14159
#define NEWLINE '\n'
```

This defines two new constants: `PI` and `NEWLINE`. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

```
// defined constants: calculate circumference

#include <iostream>
using namespace std;

#define PI 3.14159
#define NEWLINE '\n'

int main ()
{
  double r=5.0;               // radius
  double circle;

  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;

  return 0;
}
```

```
31.4159
```

In fact the only thing that the compiler preprocessor does when it encounters `#define` directives is to literally replace any occurrence of their identifier (in the previous example, these were `PI` and `NEWLINE`) by the code to which they have been defined (`3.14159` and `'\n'` respectively).

The `#define` directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (`;`) at its end. If you append a semicolon character (`;`) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

# Declared constants (const)

With the `const` prefix you can declare constants with a specific type in the same way as you would do with a variable:

```
const int pathwidth = 100;
const char tabulator = '\t';
```

Here, `pathwidth` and `tabulator` are two typed constants. They are treated just like regular variables except that their values cannot be modified after their definition.

# Operators

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.

You do not have to memorize all the content of this page. Most details are only provided to serve as a later reference in case you need it.

## Assignment (=)

The assignment operator assigns a value to a variable.

```
a = 5;
```

This statement assigns the integer value 5 to the variable `a`. The part at the left of the assignment operator (`=`) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The lvalue has to be a variable whereas the rvalue can be either a constant, a variable, the result of an operation or any combination of these. The most important rule when assigning is the *right-to-left* rule: The assignment operation always takes place from right to left, and never the other way:

```
a = b;
```

This statement assigns to variable `a` (the lvalue) the value contained in variable `b` (the rvalue). The value that was stored until this moment in `a` is not considered at all in this operation, and in fact that value is lost.

Consider also that we are only assigning the value of `b` to `a` at the moment of the assignment operation. Therefore a later change of `b` will not affect the new value of `a`.

For example, let us have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
// assignment operator

#include <iostream>
using namespace std;

int main ()
{
  int a, b;         // a:?,  b:?
  a = 10;           // a:10, b:?
  b = 4;            // a:10, b:4
  a = b;            // a:4,  b:4
  b = 7;            // a:4,  b:7

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;

  return 0;
}
```

```
a:4 b:7
```

This code will give us as result that the value contained in `a` is `4` and the one contained in `b` is `7`. Notice how `a` was not affected by the final modification of `b`, even though we declared `a = b` earlier (that is because of the *right-to-left rule*).

A property that C++ has over other programming languages is that the assignment operation can be used as the rvalue (or part of an rvalue) for another assignment operation. For example:

```
a = 2 + (b = 5);
```

is equivalent to:

```
b = 5;
a = 2 + b;
```

that means: first assign `5` to variable `b` and then assign to `a` the value `2` plus the result of the previous assignment of `b` (i.e. 5), leaving `a` with a final value of `7`.

The following expression is also valid in C++:

```
a = b = c = 5;
```

It assigns `5` to the all the three variables: `a`, `b` and `c`.

## Arithmetic operators ( +, -, *, /, % )

The five arithmetical operations supported by the C++ language are:

| | |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| % | modulo |

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is *modulo*; whose operator is the percentage sign (`%`). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

```
a = 11 % 3;
```

the variable `a` will contain the value `2`, since `2` is the remainder from dividing `11` between `3`.

## Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

| expression | is equivalent to |
|---|---|
| `value += increase;` | `value = value + increase;` |
| `a -= 5;` | `a = a - 5;` |
| `a /= b;` | `a = a / b;` |
| `price *= units + 1;` | `price = price * (units + 1);` |

and the same for all other operators. For example:

| | |
|---|---|
| `// compound assignment operators`<br><br>`#include <iostream>`<br>`using namespace std;`<br><br>`int main ()`<br>`{`<br>`  int a, b=3;`<br>`  a = b;`<br>`  a+=2;              // equivalent to a=a+2`<br>`  cout << a;`<br>`  return 0;`<br>`}` | 5 |

## Increase and decrease (++, --)

Shortening even more some expressions, the increase operator (`++`) and the decrease operator (`--`) increase or reduce by one the value stored in a variable. They are equivalent to `+=1` and to `-=1`, respectively. Thus:

```
c++;
c+=1;
c=c+1;
```

are all equivalent in its functionality: the three of them increase by one the value of c.

In the early C compilers, the three previous expressions probably produced different executable code depending on which one was used. Nowadays, this type of code optimization is generally done automatically by the compiler, thus the three expressions should produce exactly the same executable code.

A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (`++a`) or after it (`a++`). Although in simple expressions like `a++` or `++a` both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (`a++`) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression. Notice the difference:

| Example 1 | Example 2 |
|---|---|
| `B=3;`<br>`A=++B;`<br>`// A contains 4, B contains 4` | `B=3;`<br>`A=B++;`<br>`// A contains 3, B contains 4` |

In Example 1, `B` is increased before its value is copied to `A`. While in Example 2, the value of `B` is copied to `A` and then `B` is increased.

## Relational and equality operators ( ==, !=, >, <, >=, <= )

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.

We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in C++:

| | |
|---|---|
| == | Equal to |
| != | Not equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |

Here there are some examples:

```
(7 == 5)     // evaluates to false.
(5 > 4)      // evaluates to true.
(3 != 2)     // evaluates to true.
(6 >= 6)     // evaluates to true.
(5 < 5)      // evaluates to false.
```

Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that `a=2`, `b=3` and `c=6`,

```
(a == 5)     // evaluates to false since a is not equal to 5.
(a*b >= c)   // evaluates to true since (2*3 >= 6) is true.
(b+4 > a*c)  // evaluates to false since (3+4 > 2*6) is false.
((b=2) == a) // evaluates to true.
```

Be careful! The operator `=` (one equal sign) is not the same as the operator `==` (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (`==`) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, in the last expression `((b=2) == a)`, we first assigned the value `2` to `b` and then we compared it to `a`, that also stores the value `2`, so the result of the operation is true.

# Logical operators ( !, &&, || )

The Operator `!` is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

```
!(5 == 5)    // evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4)    // evaluates to true because (6 <= 4) would be false.
!true        // evaluates to false
!false       // evaluates to true.
```

The logical operators `&&` and `||` are used when evaluating two expressions to obtain a single relational result. The operator `&&` corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator `&&` evaluating the expression `a && b`:

**&& OPERATOR**

| a | b | a && b |
|---|---|---|
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

The operator `||` corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of `a || b`:

**|| OPERATOR**

| a | b | a \|\| b |
|---|---|---|
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

For example:

```
( (5 == 5) && (3 > 6) )  // evaluates to false ( true && false ).
( (5 == 5) || (3 > 6) )  // evaluates to true ( true || false ).
```

# Conditional operator ( ? )

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:

```
condition ? result1 : result2
```

If `condition` is true the expression will return `result1`, if it is not it will return `result2`.

```
7==5 ? 4 : 3     // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3   // returns 4, since 7 is equal to 5+2.
5>3 ? a : b      // returns the value of a, since 5 is greater than 3.
a>b ? a : b      // returns whichever is greater, a or b.
```

```
// conditional operator

#include <iostream>
using namespace std;

int main ()
{
  int a,b,c;

  a=2;
  b=7;
  c = (a>b) ? a : b;

  cout << c;

  return 0;
}
```

```
7
```

In this example `a` was `2` and `b` was `7`, so the expression being evaluated (`a>b`) was not true, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was `b`, with a value of `7`.

# Comma operator ( , )

The comma operator (`,`) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

For example, the following code:

```
a = (b=3, b+2);
```

Would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

## Bitwise Operators ( &, |, ^, ~, <<, >> )

Bitwise operators modify variables considering the bit patterns that represent the values they store.

| operator | asm equivalent | description |
|---|---|---|
| & | AND | Bitwise AND |
| \| | OR | Bitwise Inclusive OR |
| ^ | XOR | Bitwise Exclusive OR |
| ~ | NOT | Unary complement (bit inversion) |
| << | SHL | Shift Left |
| >> | SHR | Shift Right |

## Explicit type casting operator

Type casting operators allow you to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++.

## sizeof()

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

```
a = sizeof (char);
```

This will assign the value 1 to a because char is a one-byte long type.
The value returned by sizeof is a constant, so it is always determined before program execution.

## Other operators

Later in these tutorials, we will see a few more operators, like the ones referring to pointers or the specifics for object-oriented programming. Each one is treated in its respective section.

## Precedence of operators

When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:

```
a = 5 + 7 % 2
```

we may doubt if it really means:

```
a = 5 + (7 % 2)    // with a result of 6, or
a = (5 + 7) % 2    // with a result of 0
```

The correct answer is the first of the two expressions, with a result of `6`. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for all the operators which can appear in C++. From greatest to lowest priority, the priority order is as follows:

| Level | Operator | Description | Grouping |
|---|---|---|---|
| 1 | `::` | scope | Left-to-right |
| 2 | `() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid` | postfix | Left-to-right |
| 3 | `++ -- ~ ! sizeof new delete` | unary (prefix) | Right-to-left |
| | `* &` | indirection and reference (pointers) | |
| | `+ -` | unary sign operator | |
| 4 | `(type)` | type casting | Right-to-left |
| 5 | `.* ->*` | pointer-to-member | Left-to-right |
| 6 | `* / %` | multiplicative | Left-to-right |
| 7 | `+ -` | additive | Left-to-right |
| 8 | `<< >>` | shift | Left-to-right |
| 9 | `< > <= >=` | relational | Left-to-right |
| 10 | `== !=` | equality | Left-to-right |
| 11 | `&` | bitwise AND | Left-to-right |
| 12 | `^` | bitwise XOR | Left-to-right |
| 13 | `\|` | bitwise OR | Left-to-right |
| 14 | `&&` | logical AND | Left-to-right |
| 15 | `\|\|` | logical OR | Left-to-right |
| 16 | `?:` | conditional | Right-to-left |
| 17 | `= *= /= %= += -= >>= <<= &= ^= \|=` | assignment | Right-to-left |
| 18 | `,` | comma | Left-to-right |

Grouping defines the precedence order in which operators are evaluated in the case that there are several operators of the same level in an expression.

All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using parentheses signs `(` and `)`, as in this example:

```
a = 5 + 7 % 2;
```

might be written either as:

```
a = 5 + (7 % 2);
```

or

```
a = (5 + 7) % 2;
```

depending on the operation that we want to perform.

So if you want to write complicated expressions and you are not completely sure of the precedence levels, always include parentheses. It will also become a code easier to read.