
جامعة الانبار

كلية علوم الحاسوب وتكنولوجيا المعلومات

قسم نظم المعلومات

# The C++ Language Tutorial

م.م يقين سعد علي

## Basics of C++

# Structure of a program

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our first program:

```
// my first program in C++

#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World!";
  return 0;
}
```

```
Hello World!
```

The first panel shows the source code for our first program. The second one shows the result of the program once compiled and executed. The way to edit and compile a program depends on the compiler you are using. Depending on whether it has a Development Interface or not and on its version. Consult the compilers section and the manual or help included with your compiler if you have doubts on how to compile a C++ console program.

The previous program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!" sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

**`// my first program in C++`**

This is a comment line. All lines beginning with two slash signs (`//`) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

**`#include <iostream>`**

Lines beginning with a hash sign (`#`) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive `#include <iostream>` tells the preprocessor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

**`using namespace std;`**

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library, and in fact it will be included in most of the source codes included in these tutorials.

**`int main ()`**

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it - the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a `main` function.

The word `main` is followed in the code by a pair of parentheses (`()`). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them.

Right after these parentheses we can find the body of the main function enclosed in braces (`{}`). What is contained within these braces is what the function does when it is executed.

`cout << "Hello World!";`

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.

`cout` represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the `Hello World` sequence of characters) into the standard output stream (which usually is the screen).

`cout` is declared in the `iostream` standard file within the `std` namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

Notice that the statement ends with a semicolon character (`;`). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

`return 0;`

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code `0`). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by `//`). There were lines with directives for the compiler's preprocessor (those beginning by `#`). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into `cout`), which were all included within the block delimited by the braces (`{}`) of the main function.

The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, instead of

```
int main ()
{
  cout << " Hello World!";
  return 0;
}
```

We could have written:

```
int main () { cout << "Hello World!"; return 0; }
```

All in just one line and this would have had exactly the same meaning as the previous code.

In C++, the separation between statements is specified with an ending semicolon (`;`) at the end of each one, so the separation in different code lines does not matter at all for this purpose. We can write many statements per line or write a single statement that takes many code lines. The division of code in different lines serves only to make it more legible and schematic for the humans that may read it.

Let us add an additional instruction to our first program:

```
// my second program in C++

#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World! ";
  cout << "I'm a C++ program";
  return 0;
}
```

```
Hello World! I'm a C++ program
```

In this case, we performed two insertions into cout in two different statements. Once again, the separation in different lines of code has been done just to give greater readability to the program, since `main` could have been perfectly valid defined this way:

```
int main () { cout << " Hello World! "; cout << " I'm a C++ program "; return 0; }
```

We were also free to divide the code into more lines if we considered it more convenient:

```
int main ()
{
  cout <<
    "Hello World!";
  cout
    << "I'm a C++ program";
  return 0;
}
```

And the result would again have been exactly the same as in the previous examples.

Preprocessor directives (those that begin by `#`) are out of this general rule since they are not statements. They are lines read and processed by the preprocessor and do not produce any code by themselves. Preprocessor directives must be specified in their own line and do not have to end with a semicolon (`;`).

# Comments

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.

C++ supports two ways to insert comments:

```
// line comment
/* block comment */
```

The first of them, known as line comment, discards everything from where the pair of slash signs (`//`) is found up to the end of that same line. The second one, known as block comment, discards everything between the `/*` characters and the first appearance of the `*/` characters, with the possibility of including more than one line.
We are going to add comments to our second program:

```
/* my second program in C++
   with more comments */

#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World! ";      // prints Hello
World!
  cout << "I'm a C++ program"; // prints I'm a
C++ program
  return 0;
}
```

```
Hello World! I'm a C++ program
```

If you include comments within the source code of your programs without using the comment characters combinations `//`, `/*` or `*/`, the compiler will take them as if they were C++ expressions, most likely causing one or several error messages when you compile it.

# 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: