C++_Lecture Note 1

Structure of C++ Program

The best way to learn a programming language is by writing programs. Typically, the first program beginners write is a program called "Hello World", which simply prints "Hello World" to your computer screen. Although it is very simple, it contains all the fundamental components C++ programs have:

```
1 // my first program in C++
2 #include <iostream>
3
4 int main()
5 {
6   std::cout << "Hello World!";
7 }
```

```
Hello World!
```

The left panel above shows the C++ code for this program. The right panel shows the result when the program is executed by a computer. The grey numbers to the left of the panels are line numbers to make discussing programs and researching errors easier. They are not part of the program.

Let's examine this program line by line:

Line 1: `// my first program in C++`

Two slash signs indicate that the rest of the line is a comment inserted by the programmer but which has no effect on the behavior of the program. Programmers use them to include short explanations or observations concerning the code or program. In this case, it is a brief introductory description of the program.

Line 2: `#include <iostream>`

Lines beginning with a hash sign (`#`) are directives read and interpreted by what is known as the *preprocessor*. They are special lines interpreted before the compilation of the program itself begins. In this case, the directive `#include <iostream>`, instructs the preprocessor to include a section of standard C++ code, known as *header iostream*, that allows to perform standard input and output operations, such as writing the output of this program (`Hello World`) to the screen.

Line 3: A blank line.

Blank lines have no effect on a program. They simply improve readability of the code.

Line 4: `int main ()`

This line initiates the declaration of a function. Essentially, a function is a group of code statements which are given a name: in this case, this gives the name "main" to the group of code statements that follow. Functions will be discussed in detail in a later chapter, but essentially, their definition is introduced with a succession of a type (`int`), a name (`main`) and a pair of parentheses (`()`), optionally including parameters.

The function named `main` is a special function in all C++ programs; it is the function called when the program is run. The execution of all C++ programs begins with the `main` function, regardless of where the function is actually located within the code.

Lines 5 and 7: `{` and `}`

The open brace (`{`) at line 5 indicates the beginning of `main`'s function definition, and the closing brace (`}`) at line 7, indicates its end. Everything between these braces is the function's body that defines what happens when `main` is called. All functions use braces to indicate the beginning and end of their definitions.

Line 6: `std::cout << "Hello World!";`

This line is a C++ statement. A statement is an expression that can actually produce some effect. It is the meat of a program, specifying its actual behavior. Statements are executed in the same order that they appear within a function's body.

This statement has three parts: First, `std::cout`, which identifies the **st**andar**d** **c**haracter **out**put device (usually, this is the computer screen). Second, the insertion operator (`<<`), which indicates that what follows is inserted into `std::cout`. Finally, a sentence within quotes ("Hello world!"), is the content inserted into the standard output.

Notice that the statement ends with a semicolon (`;`). This character marks the end of the statement, just as the period ends a sentence in English. All C++ statements must end with a semicolon character. One of the most common syntax errors in C++ is forgetting to end a statement with a semicolon.

You may have noticed that not all the lines of this program perform actions when the code is executed. There is a line containing a comment (beginning with `//`). There is a line with a directive for the preprocessor (beginning with `#`). There is a line that defines a function (in this case, the `main` function). And, finally, a line with a statements ending with a semicolon (the insertion into `cout`), which was within the block delimited by the braces (`{ }`) of the `main` function.

The program has been structured in different lines and properly indented, in order to make it easier to understand for the humans reading it. But C++ does not have strict rules on indentation or on how to split instructions in different lines. For example, instead of

```
int main ()
{
  std::cout << " Hello World!";
}
```

Edit & Run

We could have written:

```
int main () { std::cout << "Hello World!"; }
```
Edit & Run

all in a single line, and this would have had exactly the same meaning as the preceding code.

In C++, the separation between statements is specified with an ending semicolon (`;`), with the separation into different lines not mattering at all for this purpose. Many statements can be written in a single line, or each statement can be in its own line. The division of code in different lines serves only to make it more legible and schematic for the humans that may read it, but has no effect on the actual behavior of the program.

Now, let's add an additional statement to our first program:

```
// my second program in C++
#include <iostream>

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

```
Hello World! I'm a C++ program
```
Edit & Run

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

```
int main () { std::cout << " Hello World! "; std::cout << " I'm a
C++ program "; }
```
Edit & Run

The source code could have also been divided into more code lines instead:

```
int main ()
{
  std::cout <<
    "Hello World!";
  std::cout
    << "I'm a C++ program";
}
```
Edit & Run

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 before proper compilation begins. Preprocessor directives must be specified in their own line and, because they are not statements, do not have to end with a semicolon (`;`).

Comments

As noted above, comments do not affect the operation of the program; however, they provide an important tool to document directly within the source code what the program does and how it operates.

C++ supports two ways of commenting code:

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

The first of them, known as *line comment*, discards everything from where the pair of slash signs (`//`) are 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 multiple lines.

Let's add comments to our second program:

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

#include <iostream>

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

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

Edit & Run

If comments are included within the source code of a program without using the comment characters combinations `//`, `/*` or `*/`, the compiler takes them as if they were C++ expressions, most likely causing the compilation to fail with one, or several, error messages.

Using namespace std

If you have seen C++ code before, you may have seen `cout` being used instead of `std::cout`. Both name the same object: the first one uses its *unqualified name* (`cout`), while the second qualifies it directly within the *namespace* `std` (as `std::cout`).

`cout` is part of the standard library, and all the elements in the standard C++ library are declared within what is called a *namespace*: the namespace `std`.

In order to refer to the elements in the `std` namespace a program shall either qualify each and every use of elements of the library (as we have done by prefixing `cout` with `std::`), or introduce visibility of its components. The most typical way to introduce visibility of these components is by means of *using declarations*:

```
using namespace std;
```

The above declaration allows all elements in the `std` namespace to be accessed in an *unqualified* manner (without the `std::` prefix).

With this in mind, the last example can be rewritten to make unqualified uses of `cout` as:

```
// my second program in C++
#include <iostream>
using namespace std;

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

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

Edit & Run

Both ways of accessing the elements of the `std` namespace (explicit qualification and *using* declarations) are valid in C++ and produce the exact same behavior. For simplicity, and to improve readability, the examples in these tutorials will more often use this latter approach with *using* declarations, although note that *explicit qualification* is the only way to guarantee that name collisions never happen.

Variables and types

The usefulness of the "Hello World" programs shown in the previous chapter is rather questionable. We had to write several lines of code, compile them, and then execute the resulting program, just to obtain the result of a simple sentence written on the screen. It certainly would have been much faster to type the output sentence 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 *variables*.

Let's imagine that I ask you to remember the number 5, and then I ask you to also memorize the number 2 at the same time. You have just stored two different values in your memory (5 and 2). 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. Then we could, for example, subtract these values and obtain 4 as result.

The whole process described above is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following set of statements:

```
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.

We can now define *variable* as a portion of memory to store a value.

Each variable needs a name that identifies it and distinguishes it from the others. For example, in the previous code the variable names were `a`, `b`, and `result`, but we could have called the variables any names we could have come up with, as long as they were valid C++ identifiers.

Identifiers

A *valid identifier* is a sequence of one or more letters, digits, or underscore characters (_). Spaces, punctuation marks, and symbols cannot be part of an identifier. In addition, identifiers shall always begin with a letter. They can also begin with an underline character (_), but such identifiers are -on most cases- considered reserved for compiler-specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case can they begin with a digit.

C++ uses a number of keywords to identify operations and data descriptions; therefore, identifiers created by a programmer cannot match these keywords. The standard reserved keywords that cannot be used for programmer created identifiers are:

```
alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case,
catch, char, char16_t, char32_t, class, compl, const, constexpr, const_cast,
continue, decltype, default, delete, do, double, dynamic_cast, else, enum,
explicit, export, extern, false, float, for, friend, goto, if, inline, int,
long, mutable, namespace, new, noexcept, not, not_eq, nullptr, operator, or,
or_eq, private, protected, public, register, reinterpret_cast, return, short,
signed, sizeof, static, static_assert, static_cast, struct, switch, template,
this, thread_local, throw, true, try, typedef, typeid, typename, union,
unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq
```

Specific compilers may also have 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 identifiers identifiying three different variables.