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
char	Character or small integer.	1 byte
int	Integer.	4 bytes
short int (short)	Short Integer.	2 bytes
long int (long)	Long integer.	4 bytes
bool	Boolean value. It can take one of two values: true or false.	1 byte
float	Floating point number.	4 bytes
double	Double precision floating point number.	8 bytes
long_double	Long double precision floating point number.	8 bytes
Wchar_t	Wide character.	2 *or* 4 bytes

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;
}
```

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.

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;
}
```

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. 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;
}
```

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;
}
```