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 unsigned: 0 to 255
`short int` (`short`)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
`int`	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`long int` (`long`)	Long integer.	4bytes	signed: -2147483648 to 2147483647 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: