5. Variables and Constants

5.1 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.

5.2 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.

5.3 Fundamental data types

The values of variables are stored somewhere in an unspecified location in the computer memory as zeros and ones. Our program does not need to know the exact location where a variable is stored; it can simply refer to it by its name. What the program needs to be aware of is the kind of data stored in the variable. It's not the same to store a simple integer as it is to store a letter or a large floating-point number; even though they are all represented using

zeros and ones, they are not interpreted in the same way, and in many cases, they don't occupy the same amount of memory.

Fundamental data types are basic types implemented directly by the language that represent the basic storage units supported natively by most systems. They can mainly be classified into:

- **Character types:** They can represent a single character, such as 'A' or '$'. The most basic type is char, which is a one-byte character. Other types are also provided for wider characters.
- **Numerical integer types:** They can store a whole number value, such as 7 or 1024. They exist in a variety of sizes, and can either be signed or unsigned, depending on whether they support negative values or not.
- **Floating-point types:** They can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used.
- **Boolean type:** The boolean type, known in C++ as bool, can only represent one of two states, true or false.

Here is the complete list of fundamental types in C++:

Group	Type names*	Notes on size / precision
Character types	**char**	Exactly one byte in size. At least 8 bits.
	char16_t	Not smaller than char. At least 16 bits.
	char32_t	Not smaller than char16_t. At least 32 bits.
	wchar_t *float* mynumber;	Represents the largest supported character set.
Integer types (signed)	**signed char**	Same size as char. At least 8 bits.
	signed **short** *int*	Not smaller than char. At least 16 bits.
	signed **int**	Not smaller than short. At least 16 bits.
	signed **long** *int*	Not smaller than int. At least 32 bits.
	signed **long long** *int*	Not smaller than long. At least 64 bits.
Integer types (unsigned)	**unsigned char**	(same size as their signed counterparts)
	unsigned short *int*	
	unsigned *int*	
	unsigned long *int*	
	unsigned long long *int*	

Floating-point types	**`float`**	At least 32 bits.
	`double`	Precision not less than float. At least 64 bits.
	`long double`	Precision not less than double. At least 64 bits.
Boolean type	**`bool`**	At least 8 bits.
Void type	**`void`**	no storage
Null pointer	**`decltype(nullptr)`**	

> ** The names of certain integer types can be abbreviated without their signed and int components - only the part not in italics is required to identify the type, the part in italics is optional. I.e., signed short int can be abbreviated as signed short, short int, or simply short; they all identify the same fundamental type.*

Within each of the groups above, the difference between types is only their size (i.e., how much they occupy in memory): the first type in each group is the smallest, and the last is the largest, with each type being at least as large as the one preceding it in the same group. Other than that, the types in a group have the same properties.

Note in the panel above that other than char (which has a size of exactly one byte), none of the fundamental types has a standard size specified (but a minimum size, at most). Therefore, the type is not required (and in many cases is not) exactly this minimum size. This does not mean that these types are of an undetermined size, but that there is no standard size across all compilers and machines; each compiler implementation may specify the sizes for these types that fit the best the architecture where the program is going to run. This rather generic size specification for types gives the C++ language a lot of flexibility to be adapted to work optimally in all kinds of platforms, both present and future.

Type sizes above are expressed in bits; the more bits a type has, the more distinct values it can represent, but at the same time, also consumes more space in memory:

Size	Unique representable values	Notes
8-bit	256	$= 2^8$
16-bit	65 536	$= 2^{16}$
32-bit	4 294 967 296	$= 2^{32}$ (~4 billion)
64-bit	18 446 744 073 709 551 616	$= 2^{64}$ (~18 billion billion)

For integer types, having more representable values means that the range of values they can represent is greater; for example, a 16-bit unsigned integer would be able to represent 65536 distinct values in the range 0 to 65535, while its signed counterpart would be able to

represent, on most cases, values between -32768 and 32767. Note that the range of positive values is approximately halved in signed types compared to unsigned types, due to the fact that one of the 16 bits is used for the sign; this is a relatively modest difference in range, and seldom justifies the use of unsigned types based purely on the range of positive values they can represent.

For floating-point types, the size affects their precision, by having more or less bits for their significant and exponent.

If the size or precision of the type is not a concern, then char, int, and double are typically selected to represent characters, integers, and floating-point values, respectively. The other types in their respective groups are only used in very particular cases.

The types described above (characters, integers, floating-point, and boolean) are collectively known as arithmetic types. But two additional fundamental types exist: void, which identifies the lack of type; and the type nullptr, which is a special type of pointer. Both types will be discussed further in a coming chapter about pointers.

C++ supports a wide variety of types based on the fundamental types discussed above; these other types are known as compound data types, and are one of the main strengths of the C++ language. We will also see them in more detail in future chapters.

5.4 Declaration of variables

C++ is a strongly-typed language, and requires every variable to be declared with its type before its first use. This informs the compiler the size to reserve in memory for the variable and how to interpret its value. The syntax to declare a new variable in C++ is straightforward: we simply write the type followed by the variable name (i.e., its 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 declaring more than one variable of the same type, they can all be declared 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;
```

To see what variable declarations look like in action within a program, let's have a look at the entire C++ code of the example about your mental memory proposed at the beginning of this chapter:

```
// 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
```

5.5 Initialization of variables

When the variables in the example above are declared, they have an undetermined value until they are assigned a value for the first time. But it is possible for a variable to have a specific value from the moment it is declared. This is called the initialization of the variable.

In C++, there are three ways to initialize variables. They are all equivalent and are reminiscent of the evolution of the language over the years:

The first one, known as c-like initialization (because it is inherited from the C language), consists of appending an equal sign followed by the value to which the variable is initialized:

```
type identifier = initial_value;
```

For example, to declare a variable of type int called x and initialize it to a value of zero from the same moment it is declared, we can write:

```
int x = 0;
```

A second method, known as constructor initialization (introduced by the C++ language), encloses the initial value between parentheses (()):

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

For example:

```
int x (0);
```

Finally, a third method, known as uniform initialization, similar to the above, but using curly braces ({}) instead of parentheses (this was introduced by the revision of the C++ standard, in 2011):

```
type identifier {initial_value};
```

For example:

```
int x {0};
```

All three 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(3);     // initial value: 3
  int c{2};     // initial value: 2
  int result;   // initial value undetermined
  a = a + b;
  result = a - c;
  cout << result;
  return 0;
}
```

```
6
```

5.6 Type deduction: auto and decltype

When a new variable is initialized, the compiler can figure out what the type of the variable is automatically by the initializer. For this, it suffices to use **auto** as the type specifier for the variable:

```
int foo = 0;
auto bar = foo;  // the same as: int bar = foo;
```

Here, bar is declared as having an **auto** type; therefore, the type of bar is the type of the value used to initialize it: in this case it uses the type of foo, which is int.

Variables that are not initialized can also make use of type deduction with the **decltype** specifier:

```
int foo = 0;
decltype(foo) bar;  // the same as: int bar;
```

Here, bar is declared as having the same type as foo.

auto and **decltype** are powerful features recently added to the language. But the type deduction features they introduce are meant to be used either when the type cannot be obtained by other means or when using it improves code readability. The two examples above were likely neither of these use cases. In fact they probably decreased readability, since, when reading the code, one has to search for the type of foo to actually know the type of bar.

5.7 Introduction to strings

Fundamental types represent the most basic types handled by the machines where the code may run. But one of the major strengths of the C++ language is its rich set of compound types, of which the fundamental types are mere building blocks.

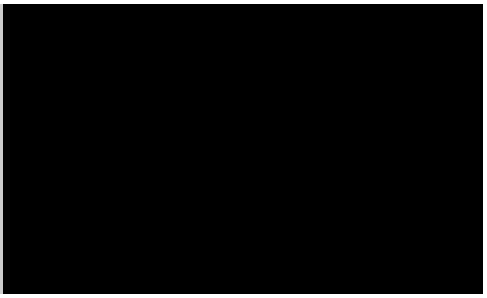
An example of compound type is the string class. Variables of this type are able to store sequences of characters, such as words or sentences. A very useful feature!

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type, the program needs to include the header where the type is defined within the standard library (header <string>):

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

```
This is a string!
```

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



As you can 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. As with fundamental types, all initialization formats are valid with strings:

```
string mystring = "This is a string";
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 change its value 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
```

5.8 Constants

Constants are expressions with a fixed value.

5.9 Literals

Literals are the most obvious kind of constants. They are used to express particular values within the source code of a program. We have already used some in previous chapters to give specific 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 classified into: integer, floating-point, characters, strings, Boolean, pointers, and user-defined literals.

5.9.1 Integer Numerals

```
1776
707
-273
```

These are numerical constants that identify integer values. Notice that they are not enclosed in quotes or any other special character; they are a simple succession of digits representing a whole number in decimal base; for example, 1776 always represents the value one thousand seven hundred seventy-six.

In addition to decimal numbers (those that most of us use every day), C++ allows the use of octal numbers (base 8) and hexadecimal numbers (base 16) as literal constants. For octal literals, the digits are preceded with a 0 (zero) character. And for hexadecimal, they are preceded by 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.

These literal constants have a type, just like variables. By default, integer literals are of type int. However, certain suffixes may be appended to an integer literal to specify a different integer type:

Suffix	Type modifier
u *or* U	unsigned
l *or* L	long
ll *or* LL	long long

Unsigned may be combined with any of the other two in any order to form unsigned long or unsigned long long.

For example:

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

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

5.9.2 Floating Point Numerals

They express real values, 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. Floating-point literals of type float or long double can be specified by adding one of the following suffixes:

Suffix	Type
f *or* F	float
l *or* L	long double

For example:

```
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 with no difference in meaning.

5.9.3 Character and string literals

Character and string literals are enclosed in quotes:

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

The first two expressions represent single-character literals, 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 the characters between double quotes (").

Both single-character and string literals require quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

```
x
'x'
```

Here, x alone would refer to an identifier, such as the name of a variable or a compound type, whereas 'x' (enclosed within single quotation marks) would refer to the character literal 'x' (the character that represents a lowercase x letter).

Character and string literals can also represent special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (\n) or tab (\t). These special characters are all of them preceded by a backslash character (\).

Here you have a list of the single character escape codes:

Escape code	Description
\n	newline
\r	carriage return

Escape code	Description
\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"
```

Internally, computers represent characters as numerical codes: most typically, they use one extension of the ASCII character encoding system (see ASCII code for more info). Characters can also be represented in literals using its numerical code by writing a backslash character (\) followed by the code expressed as an octal (base-8) or hexadecimal (base-16) number. For an octal value, the backslash is followed directly by the digits; while for hexadecimal, an x character is inserted between the backslash and the hexadecimal digits themselves (for example: \x20 or \x4A).

Several string literals can be concatenated to form a single string literal simply by separating them by one or more blank spaces, including tabs, newlines, and other valid blank characters. For example:

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

The above is a string literal equivalent to:

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

Note how spaces within the quotes are part of the literal, while those outside them are not.

Some programmers also use a trick to include long string literals in multiple lines: In C++, a backslash (\) at the end of line is considered a line-continuation character that merges both that line and the next into a single line. Therefore the following code:

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

is equivalent to:

```
x = "string expressed in two lines"
```

5.9.4 Other literals

Three keyword literals exist in C++: **true, false** and **nullptr**:

- **true** and **false** are the two possible values for variables of type bool.
- **nullptr** is the null pointer value.

```
bool foo = true;
bool bar = false;
int* p = nullptr;
```

5.10 Typed constant expressions

Sometimes, it is just convenient to give a name to a constant value:

```
const double pi = 3.1415926;
const char tab = '\t';
```

We can then use these names instead of the literals they were defined to:

```
#include <iostream>
using namespace std;

const double pi = 3.14159;
const char newline = '\n';

int main ()
{
  double r = 5.0;       // radius
  double circle;
  circle = 2 * pi * r;
  cout << circle;
  cout << newline;
}
```

```
31.4159
```

5.11 Preprocessor definitions (#define)

Another mechanism to name constant values is the use of preprocessor definitions. They have the following form:

```
#define identifier replacement
```

After this directive, any occurrence of identifier in the code is interpreted as replacement, where replacement is any sequence of characters (until the end of the line). This replacement is performed by the preprocessor, and happens before the program is compiled, thus causing a sort of blind replacement: the validity of the types or syntax involved is not checked in any way.

```
#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;

}
```

```
31.4159
```

Note that the #define lines are preprocessor directives, and as such are single-line instructions that -unlike C++ statements- do not require semicolons (;) at the end; the directive extends automatically until the end of the line. If a semicolon is included in the line, it is part of the replacement sequence and is also included in all replaced occurrences.