These tutorials explain the C++ language from its basics up to the newest features introduced by C++11. Chapters have a practical orientation, with example programs in all sections to start practicing what is being explained right away.

Compilers

The essential tools needed to follow these tutorials are a computer and a compiler toolchain able to compile C++ code and build the programs to run on it.

C++ is a language that has evolved much over the years, and these tutorials explain many features added recently to the language. Therefore, in order to properly follow the tutorials, a recent compiler is needed. It shall support (even if only partially) the features introduced by the 2011 standard.

Many compiler vendors support the new features at different degrees. See the bottom of this page for some compilers that are known to support the features needed. Some of them are free!

If for some reason, you need to use some older compiler, you can access an older version of these tutorials here(no longer updated).

What is a compiler?

Computers understand only one language and that language consists of sets of instructions made of ones and zeros. This computer language is appropriately called *machine language*.

A single instruction to a computer could look like this:

00000	10011110

A particular computer's machine language program that allows a user to input two numbers, adds the two numbers together, and displays the total could include these machine code instructions:

00000	10011110
00001	11110100

00010	10011110
00011	11010100
00100	10111111
00101	00000000

As you can imagine, programming a computer directly in machine language using only ones and zeros is very tedious and error prone. To make programming easier, high level languages have been developed. High level programs also make it easier for programmers to inspect and understand each other's programs easier.

This is a portion of code written in C++ that accomplishes the exact same purpose:

```
int a, b, sum;

cin >> a;
cin >> b;

sum = a + b;
cout << sum << endl;
```

Even if you cannot really understand the code above, you should be able to appreciate how much easier it will be to program in the C++ language as opposed to machine language.

Because a computer can only understand machine language and humans wish to write in high level languages high level languages have to be re-written (translated) into machine language at some point. This is done by special programs called compilers, interpreters, or assemblers that are built into the various programming applications.

C++ is designed to be a compiled language, meaning that it is generally translated into machine language that can be understood directly by the system, making the generated program highly efficient. For that, a set of tools are needed, known as the development toolchain, whose core are a compiler and its linker.

Console programs

Console programs are programs that use text to communicate with the user and the environment, such as printing text to the screen or reading input from a keyboard.

Console programs are easy to interact with, and generally have a predictable behavior that is identical across all platforms. They are also simple to implement and thus are very useful to learn the basics of a programming language: The examples in these tutorials are all console programs.

The way to compile console programs depends on the particular tool you are using.

The easiest way for beginners to compile C++ programs is by using an Integrated Development Environment (IDE). An IDE generally integrates several development tools, including a text editor and tools to compile programs directly from it.

Here you have instructions on how to compile and run console programs using different free Integrated Development Interfaces (IDEs):

IDE	Platform	Console programs
Code::blocks	Windows/Linux/MacOS	Compile console programs using Code::blocks
Visual Studio Express	Windows	Compile console programs using VS Express 2013
Dev-C++	Windows	Compile console programs using Dev-C++

If you happen to have a Linux or Mac environment with development features, you should be able to compile any of the examples directly from a terminal just by including C++11 flags in the command for the compiler:

Compiler	Platform	Command
GCC	Linux, among others...	`g++ -std=c++0x example.cpp -o example_program`
Clang	OS X, among others...	`clang++ -std=c++11 -stdlib=libc++ example.cpp -o example_program`

Structure of a 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:

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

int main()
```

```
Hello World!
```

Edit
&
Run

```
5 {
6   std::cout << "Hello World!";
7 }
```

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
```

Edi
&
Rur

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
```

Edi
&
Rur

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 a 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
```

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.

Namespaces are explained in more detail in a later chapter.

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 *variable*.

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.

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	Can represent 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**	
	double	Precision not less than float
	long double	Precision not less than double
Boolean type	**bool**	
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 properties of fundamental types in a particular system and compiler implementation can be obtained by using the `numeric_limits` classes (see standard header `<limits>`). If for some reason, types of specific sizes are needed, the library defines certain fixed-size type aliases in header `<cstdint>`.

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.

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
```

Edit & Run

Don't be worried if something else than the variable declarations themselves look a bit strange to you. Most of it will be explained in more detail in coming chapters.

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
```

Edit
&
Run

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`.

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
```

Edit
&
Run

```

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
```

Edit
&
Run

Note: inserting the `endl` manipulator **end**s the **l**ine (printing a newline character and flushing the stream).

The `string` class is a *compound type*. As you can see in the example above, *compound types* are used in the same way as *fundamental types*: the same syntax is used to declare variables and to initialize them.

For more details on standard C++ strings, see the `string` class reference.

Constants

Constants are expressions with a fixed value.

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.

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.

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.

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
`\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 ASCIIcharacter 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 formsa 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"
```

All the character literals and string literals described above are made of characters of type `char`. A different character type can be specified by using one of the following prefixes:

Prefix	Character type
`u`	`char16_t`
`U`	`char32_t`
`L`	`wchar_t`

Note that, unlike type suffixes for integer literals, these prefixes are *case sensitive*: lowercase for `char16_t` and uppercase for `char32_t` and `wchar_t`.

For string literals, apart from the above `u`, `U`, and `L`, two additional prefixes exist:

Prefix	Description
`u8`	The string literal is encoded in the executable using UTF-8
`R`	The string literal is a raw string

In raw strings, backslashes and single and double quotes are all valid characters; the content of the literal is delimited by an initial `R"`*`sequence`*`(` and a final `)`*`sequence`*`"`, where *`sequence`* is any sequence of characters (including an empty sequence). The content of the string is what lies inside the parenthesis, ignoring the delimiting sequence itself. For example:

```
R"(string with \backslash)"
R"&%$(string with \backslash)&%$"
```

Both strings above are equivalent to `"string with \\backslash"`. The `R` prefix can be combined with any other prefixes, such as `u`, `L` or `u8`.

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;
```

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;
```

```
31.4159
```

Edit
&

```

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;
}
```

Rur

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.

For example:

```
#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
```

Edi
&
Rur

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.

Operators

Once introduced to variables and constants, we can begin to operate with them by using *operators*. What follows is a complete list of operators. At this point, it is likely not necessary to know all of them, but they are all listed here to also serve as reference.

Assignment operator (=)

The assignment operator assigns a value to a variable.

```
x = 5;
```

This statement assigns the integer value `5` to the variable `x`. The assignment operation always takes place from right to left, and never the other way around:

```
x = y;
```

This statement assigns to variable `x` the value contained in variable `y`. The value of `x` at the moment this statement is executed is lost and replaced by the value of `y`.

Consider also that we are only assigning the value of `y` to `x` at the moment of the assignment operation. Therefore, if `y` changes at a later moment, it will not affect the new value taken by `x`.

For example, let's have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
// assignment operator
#include <iostream>
using namespace std;

int main ()
{
  int a, b;         // a:?,  b:?
  a = 10;           // a:10, b:?
  b = 4;            // a:10, b:4
  a = b;            // a:4,  b:4
  b = 7;            // a:4,  b:7
```

```
a:4 b:7
```

Edit
&
Run

```

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;
}
```

This program prints on screen the final values of `a` and `b` (4 and 7, respectively). Notice how `a` was not affected by the final modification of `b`, even though we declared `a = b` earlier.

Assignment operations are expressions that can be evaluated. That means that the assignment itself has a value, and -for fundamental types- this value is the one assigned in the operation. For example:

```
y = 2 + (x = 5);
```

In this expression, `y` is assigned the result of adding 2 and the value of another assignment expression (which has itself a value of 5). It is roughly equivalent to:

```
x = 5;
y = 2 + x;
```

With the final result of assigning 7 to `y`.

The following expression is also valid in C++:

```
x = y = z = 5;
```

It assigns 5 to the all three variables: `x`, `y` and `z`; always from right-to-left.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by C++ are:

operator	description
+	addition
-	subtraction

`*`	multiplication
`/`	division
`%`	modulo

Operations of addition, subtraction, multiplication and division correspond literally to their respective mathematical operators. The last one, *modulo operator*, represented by a percentage sign (`%`), gives the remainder of a division of two values. For example:

```
x = 11 % 3;
```

results in variable `x` containing the value 2, since dividing 11 by 3 results in 3, with a remainder of 2.

Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

Compound assignment operators modify the current value of a variable by performing an operation on it. They are equivalent to assigning the result of an operation to the first operand:

expression	equivalent to...
`y += x;`	`y = y + x;`
`x -= 5;`	`x = x - 5;`
`x /= y;`	`x = x / y;`
`price *= units + 1;`	`price = price * (units+1);`

and the same for all other compound assignment operators. For example:

```
// compound assignment operators
#include <iostream>
using namespace std;

int main ()
{
  int a, b=3;
  a = b;
  a+=2;             // equivalent to a=a+2
  cout << a;
}
```

```
5
```

Edit
&
Run

Increment and decrement (++, --)

Some expression can be shortened even more: the increase operator (`++`) and the decrease operator (–
–) increase or reduce by one the value stored in a variable. They are equivalent to `+=1` and to `-=1`, respectively. Thus:

```
++x;
x+=1;
x=x+1;
```

are all equivalent in its functionality; the three of them increase by one the value of `x`.

In the early C compilers, the three previous expressions may have produced different executable code depending on which one was used. Nowadays, this type of code optimization is generally performed automatically by the compiler, thus the three expressions should produce exactly the same executable code.

A peculiarity of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable name (`++x`) or after it (`x++`). Although in simple expressions like `x++` or `++x`, both have exactly the same meaning; in other expressions in which the result of the increment or decrement operation is evaluated, they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (`++x`) of the value, the expression evaluates to the final value of `x`, once it is already increased. On the other hand, in case that it is used as a suffix (`x++`), the value is also increased, but the expression evaluates to the value that x had before being increased. Notice the difference:

Example 1	**Example 2**

`x = 3;`
`y = ++x;`
`// x contains 4, y contains 4`
	`x = 3;`
	`y = x++;`
	`// x contains 4, y contains 3`

In *Example 1*, the value assigned to `y` is the value of `x` after being increased. While in *Example 2*, it is the value `x`had before being increased.

Relational and comparison operators (==, !=, >, <, >=, <=)

Two expressions can be compared using relational and equality operators. For example, to know if two values are equal or if one is greater than the other.

The result of such an operation is either true or false (i.e., a Boolean value).

The relational operators in C++ are:

operator	description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Here there are some examples:

```
(7 == 5)     // evaluates to false
(5 > 4)      // evaluates to true
(3 != 2)     // evaluates to true
(6 >= 6)     // evaluates to true
(5 < 5)      // evaluates to false
```

Of course, it's not just numeric constants that can be compared, but just any value, including, of course, variables. Suppose that `a=2`, `b=3` and `c=6`, then:

```
(a == 5)     // evaluates to false, since a is not equal to 5
(a*b >= c)   // evaluates to true, since (2*3 >= 6) is true
(b+4 > a*c)  // evaluates to false, since (3+4 > 2*6) is false
((b=2) == a) // evaluates to true
```

Be careful! The assignment operator (operator `=`, with one equal sign) is not the same as the equality comparison operator (operator `==`, with two equal signs); the first one (`=`) assigns the value on the right-hand to the variable on its left, while the other (`==`) compares whether the values on both sides of the operator are equal. Therefore, in the last expression (`(b=2) == a`), we first assigned the value `2` to `b` and then we compared it to `a` (that also stores the value 2), yielding `true`.

Logical operators (!, &&, ||)

The operator `!` is the C++ operator for the Boolean operation NOT. It has only one operand, to its right, and inverts it, producing `false` if its operand is `true`, and `true` if its operand is `false`. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

```
!(5 == 5)   // evaluates to false because the expression at its right (5 == 5) is true
!(6 <= 4)   // evaluates to true because (6 <= 4) would be false
!true       // evaluates to false
!false      // evaluates to true
```

The logical operators `&&` and `||` are used when evaluating two expressions to obtain a single relational result. The operator `&&` corresponds to the Boolean logical operation AND, which yields `true` if both its operands are `true`, and`false` otherwise. The following panel shows the result of operator `&&` evaluating the expression `a&&b`:

&& OPERATOR (and)		
a	**b**	**a && b**
true	true	true
true	false	false
false	true	false
false	false	false

The operator `||` corresponds to the Boolean logical operation OR, which yields `true` if either of its operands is `true`, thus being false only when both operands are false. Here are the possible results of `a||b`:

|| OPERATOR (or)		
a	**b**	**a || b**
true	true	true
true	false	true
false	true	true

`false`	`false`	`false`

For example:

```
( (5 == 5) && (3 > 6) )  // evaluates to false ( true && false )
( (5 == 5) || (3 > 6) )  // evaluates to true ( true || false )
```

When using the logical operators, C++ only evaluates what is necessary from left to right to come up with the combined relational result, ignoring the rest. Therefore, in the last example (`(5==5)||(3>6)`), C++ evaluates first whether `5==5` is `true`, and if so, it never checks whether `3>6` is `true` or not. This is known as *short-circuit evaluation*, and works like this for these operators:

operator	**short-circuit**
`&&`	if the left-hand side expression is `false`, the combined result is `false` (the right-hand side expression is never evaluated).
`||`	if the left-hand side expression is `true`, the combined result is `true` (the right-hand side expression is never evaluated).

This is mostly important when the right-hand expression has side effects, such as altering values:

```
if ( (i<10) && (++i<n) ) { /*...*/ }   // note that the condition increments i
```

Here, the combined conditional expression would increase `i` by one, but only if the condition on the left of `&&` is`true`, because otherwise, the condition on the right-hand side (`++i<n`) is never evaluated.

Conditional ternary operator (?)

The conditional operator evaluates an expression, returning one value if that expression evaluates to `true`, and a different one if the expression evaluates as `false`. Its syntax is:

```
condition ? result1 : result2
```

If `condition` is `true`, the entire expression evaluates to `result1`, and otherwise to `result2`.

```
7==5 ? 4 : 3     // evaluates to 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3   // evaluates to 4, since 7 is equal to 5+2.
5>3 ? a : b      // evaluates to the value of a, since 5 is greater than 3.
a>b ? a : b      // evaluates to whichever is greater, a or b.
```

For example:

```
// conditional operator
#include <iostream>
using namespace std;

int main ()
{
  int a,b,c;

  a=2;
  b=7;
  c = (a>b) ? a : b;

  cout << c << '\n';
}
```

```
7
```

Edit & Run

In this example, `a` was 2, and `b` was 7, so the expression being evaluated (`a>b`) was not `true`, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was `b`(with a value of 7).

Comma operator (,)

The comma operator (`,`) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the right-most expression is considered.

For example, the following code:

```
a = (b=3, b+2);
```

would first assign the value 3 to `b`, and then assign `b+2` to variable `a`. So, at the end, variable `a` would contain the value 5 while variable `b` would contain value 3.

Bitwise operators (&, |, ^, ~, <<, >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

operator	asm equivalent	description

&	AND	Bitwise AND
|	OR	Bitwise inclusive OR
^	XOR	Bitwise exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift bits left
>>	SHR	Shift bits right

Explicit type casting operator

Type casting operators allow to convert a value of a given type to another type. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the floating-point number `3.14` to an integer value (`3`); the remainder is lost. Here, the typecasting operator was `(int)`. Another way to do the same thing in C++ is to use the functional notation preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int (f);
```

Both ways of casting types are valid in C++.

sizeof

This operator accepts one parameter, which can be either a type or a variable, and returns the size in bytes of that type or object:

```
x = sizeof (char);
```

Here, `x` is assigned the value `1`, because `char` is a type with a size of one byte.

The value returned by `sizeof` is a compile-time constant, so it is always determined before program execution.

Other operators

Later in these tutorials, we will see a few more operators, like the ones referring to pointers or the specifics for object-oriented programming.

Precedence of operators

A single expression may have multiple operators. For example:

```
x = 5 + 7 % 2;
```

In C++, the above expression always assigns 6 to variable `x`, because the `%` operator has a higher precedence than the `+` operator, and is always evaluated before. Parts of the expressions can be enclosed in parenthesis to override this precedence order, or to make explicitly clear the intended effect. Notice the difference:

```
x = 5 + (7 % 2);     // x = 6 (same as without parenthesis)
x = (5 + 7) % 2;     // x = 0
```

From greatest to smallest priority, C++ operators are evaluated in the following order:

Level	Precedence group	Operator	Description	Grouping
1	Scope	`::`	scope qualifier	Left-to-right
2	Postfix (unary)	`++ --`	postfix increment / decrement	Left-to-right
		`()`	functional forms	
		`[]`	subscript	
		`. ->`	member access	
3	Prefix (unary)	`++ --`	prefix increment / decrement	Right-to-left
		`~ !`	bitwise NOT / logical NOT	

		`+ -`	unary prefix	
		`& *`	reference / dereference	
		`new delete`	allocation / deallocation	
		`sizeof`	parameter pack	
		`(`*`type`*`)`	C-style type-casting	
4	Pointer-to-member	`.* ->*`	access pointer	Left-to-right
5	Arithmetic: scaling	`* / %`	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	`+ -`	addition, subtraction	Left-to-right
7	Bitwise shift	`<< >>`	shift left, shift right	Left-to-right
8	Relational	`< > <= >=`	comparison operators	Left-to-right
9	Equality	`== !=`	equality / inequality	Left-to-right
10	And	`&`	bitwise AND	Left-to-right
11	Exclusive or	`^`	bitwise XOR	Left-to-right
12	Inclusive or	`|`	bitwise OR	Left-to-right
13	Conjunction	`&&`	logical AND	Left-to-right
14	Disjunction	`||`	logical OR	Left-to-right
15	Assignment-level expressions	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment / compound assignment	Right-to-left
		`?:`	conditional operator	

16	Sequencing	,	comma separator	Left-to-right

When an expression has two operators with the same precedence level, *grouping* determines which one is evaluated first: either left-to-right or right-to-left.

Enclosing all sub-statements in parentheses (even those unnecessary because of their precedence) improves code readability.

Basic Input/Output

The example programs of the previous sections provided little interaction with the user, if any at all. They simply printed simple values on screen, but the standard library provides many additional ways to interact with the user via its input/output features. This section will present a short introduction to some of the most useful.

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen, the keyboard or a file. A *stream* is an entity where a program can either insert or extract characters to/from. There is no need to know details about the media associated to the stream or any of its internal specifications. All we need to know is that streams are a source/destination of characters, and that these characters are provided/accepted sequentially (i.e., one after another).

The standard library defines a handful of stream objects that can be used to access what are considered the standard sources and destinations of characters by the environment where the program runs:

stream	description
`cin`	standard input stream
`cout`	standard output stream
`cerr`	standard error (output) stream
`clog`	standard logging (output) stream

We are going to see in more detail only `cout` and `cin` (the standard output and input streams); `cerr` and `clog` are also output streams, so they essentially work like `cout`, with the only difference being that they identify streams for specific purposes: error messages and logging; which, in many cases, in most environment setups, they actually do the exact same thing: they print on screen, although they can also be individually redirected.

Standard output (cout)

On most program environments, the standard output by default is the screen, and the C++ stream object defined to access it is `cout`.

For formatted output operations, `cout` is used together with the *insertion operator*, which is written as `<<` (i.e., two "less than" signs).

```
cout << "Output sentence"; // prints Output sentence on screen
cout << 120;               // prints number 120 on screen
cout << x;                 // prints the value of x on screen
```

The `<<` operator inserts the data that follows it into the stream that precedes it. In the examples above, it inserted the literal string `Output sentence`, the number `120`, and the value of variable `x` into the standard output stream`cout`. Notice that the sentence in the first statement is enclosed in double quotes (`"`) because it is a string literal, while in the last one, `x` is not. The double quoting is what makes the difference; when the text is enclosed between them, the text is printed literally; when they are not, the text is interpreted as the identifier of a variable, and its value is printed instead. For example, these two sentences have very different results:

```
cout << "Hello";   // prints Hello
cout << Hello;     // prints the content of variable Hello
```

Multiple insertion operations (<<) may be chained in a single statement:

```
cout << "This " << " is a " << "single C++ statement";
```

This last statement would print the text `This is a single C++ statement`. Chaining insertions is especially useful to mix literals and variables in a single statement:

```
cout << "I am " << age << " years old and my zipcode is " << zipcode;
```

Assuming the *age* variable contains the value 24 and the *zipcode* variable contains 90064, the output of the previous statement would be:

```
I am 24 years old and my zipcode is 90064
```

What cout does not do automatically is add line breaks at the end, unless instructed to do so. For

example, take the following two statements inserting into `cout`:
cout << "This is a sentence.";
cout << "This is another sentence.";

The output would be in a single line, without any line breaks in between. Something like:

```
This is a sentence.This is another sentence.
```

To insert a line break, a new-line character shall be inserted at the exact position the line should be broken. In C++, a new-line character can be specified as `\n` (i.e., a backslash character followed by a lowercase `n`). For example:

```
cout << "First sentence.\n";
cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

```
First sentence.
Second sentence.
Third sentence.
```

Alternatively, the `endl` manipulator can also be used to break lines. For example:

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

This would print:

```
First sentence.
Second sentence.
```

The `endl` manipulator produces a newline character, exactly as the insertion of `'\n'` does; but it also has an additional behavior: the stream's buffer (if any) is flushed, which means that the output is requested to be physically written to the device, if it wasn't already. This affects mainly *fully buffered* streams, and `cout` is (generally) not a *fully buffered* stream. Still, it is generally a good idea to use `endl` only when flushing the stream would be a feature and `'\n'` when it would not. Bear in mind that a flushing operation incurs a certain overhead, and on some devices it may produce a delay.

Standard input (cin)

In most program environments, the standard input by default is the keyboard, and the C++ stream object defined to access it is `cin`.

For formatted input operations, `cin` is used together with the extraction operator, which is written as `>>` (i.e., two "greater than" signs). This operator is then followed by the variable where the extracted data is stored. For example:

```cpp
int age;
cin >> age;
```

The first statement declares a variable of type `int` called `age`, and the second extracts from `cin` a value to be stored in it. This operation makes the program wait for input from `cin`; generally, this means that the program will wait for the user to enter some sequence with the keyboard. In this case, note that the characters introduced using the keyboard are only transmitted to the program when the `ENTER` (or `RETURN`) key is pressed. Once the statement with the extraction operation on `cin` is reached, the program will wait for as long as needed until some input is introduced.

The extraction operation on `cin` uses the type of the variable after the `>>` operator to determine how it interprets the characters read from the input; if it is an integer, the format expected is a series of digits, if a string a sequence of characters, etc.

```cpp
// i/o example

#include <iostream>
using namespace std;

int main ()
{
  int i;
  cout << "Please enter an integer
value: ";
  cin >> i;
  cout << "The value you entered is " <<
i;
  cout << " and its double is " << i*2
<< ".\n";
  return 0;
}
```

```
Please enter an integer value: 702
The value you entered is 702 and its double
is 1404.
```

As you can see, extracting from `cin` seems to make the task of getting input from the standard input pretty simple and straightforward. But this method also has a big drawback. What happens in the example above if the user enters something else that cannot be interpreted as an integer? Well, in this case, the extraction operation fails. And this, by default, lets the program continue without setting a value for variable `i`, producing undetermined results if the value of `i` is used later.

This is very poor program behavior. Most programs are expected to behave in an expected manner no matter what the user types, handling invalid values appropriately. Only very simple programs should rely on values extracted directly from `cin` without further checking. A little later we will see how *stringstreams* can be used to have better control over user input.
Extractions on `cin` can also be chained to request more than one datum in a single statement:

```
cin >> a >> b;
```

This is equivalent to:

```
cin >> a;
cin >> b;
```

In both cases, the user is expected to introduce two values, one for variable `a`, and another for variable `b`. Any kind of space is used to separate two consecutive input operations; this may either be a space, a tab, or a new-line character.

cin and strings

The extraction operator can be used on `cin` to get strings of characters in the same way as with fundamental data types:

```
string mystring;
cin >> mystring;
```

However, `cin` extraction always considers spaces (whitespaces, tabs, new-line...) as terminating the value being extracted, and thus extracting a string means to always extract a single word, not a phrase or an entire sentence.

To get an entire line from `cin`, there exists a function, called `getline`, that takes the stream (`cin`) as first argument, and the string variable as second. For example:

```
// cin with strings
#include <iostream>
#include <string>
using namespace std;

int main ()
```

```
What's your name? Homer Simpson
Hello Homer Simpson.
What is your favorite team? The Isotopes
I like The Isotopes too!
```

Edi
&
Rur

```
{
  string mystr;
  cout << "What's your name? ";
  getline (cin, mystr);
  cout << "Hello " << mystr << ".\n";
  cout << "What is your favorite team? ";
  getline (cin, mystr);
  cout << "I like " << mystr << "
too!\n";
  return 0;
}
```

Notice how in both calls to `getline`, we used the same string identifier (`mystr`). What the program does in the second call is simply replace the previous content with the new one that is introduced.

The standard behavior that most users expect from a console program is that each time the program queries the user for input, the user introduces the field, and then presses `ENTER` (or `RETURN`). That is to say, input is generally expected to happen in terms of lines on console programs, and this can be achieved by using `getline` to obtain input from the user. Therefore, unless you have a strong reason not to, you should always use `getline` to get input in your console programs instead of extracting from `cin`.

stringstream

The standard header <sstream> defines a type called stringstream that allows a string to be treated as a stream, and thus allowing extraction or insertion operations from/to strings in the same way as they are performed on `cin`and `cout`. This feature is most useful to convert strings to numerical values and vice versa. For example, in order to extract an integer from a string we can write:

```
string mystr ("1204");
int myint;
stringstream(mystr) >> myint;
```

This declares a `string` with initialized to a value of `"1204"`, and a variable of type `int`. Then, the third line uses this variable to extract from a `stringstream` constructed from the string. This piece of code stores the numerical value`1204` in the variable called `myint`.

```
// stringstreams
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main ()
```

```
Enter price: 22.25
Enter quantity: 7
Total price: 155.75
```

Edit
&
Rur

```
{
  string mystr;
  float price=0;
  int quantity=0;

  cout << "Enter price: ";
  getline (cin,mystr);
  stringstream(mystr) >> price;
  cout << "Enter quantity: ";
  getline (cin,mystr);
  stringstream(mystr) >> quantity;
  cout << "Total price: " << price*quantity
<< endl;
  return 0;
}
```

In this example, we acquire numeric values from the *standard input* indirectly: Instead of extracting numeric values directly from `cin`, we get lines from it into a string object (`mystr`), and then we extract the values from this string into the variables `price` and `quantity`. Once these are numerical values, arithmetic operations can be performed on them, such as multiplying them to obtain a total price.

With this approach of getting entire lines and extracting their contents, we separate the process of getting user input from its interpretation as data, allowing the input process to be what the user expects, and at the same time gaining more control over the transformation of its content into useful data by the program.

Statements and flow control

A simple C++ statement is each of the individual instructions of a program, like the variable declarations and expressions seen in previous sections. They always end with a semicolon (`;`), and are executed in the same order in which they appear in a program.

But programs are not limited to a linear sequence of statements. During its process, a program may repeat segments of code, or take decisions and bifurcate. For that purpose, C++ provides flow control statements that serve to specify what has to be done by our program, when, and under which circumstances.

Many of the flow control statements explained in this section require a generic (sub)statement as part of its syntax. This statement may either be a simple C++ statement, -such as a single instruction, terminated with a semicolon (`;`) - or a compound statement. A compound statement is a group of statements (each of them terminated by its own semicolon), but all grouped together in a block, enclosed in curly braces: {}:

```
{ statement1; statement2; statement3; }
```

The entire block is considered a single statement (composed itself of multiple substatements). Whenever a generic statement is part of the syntax of a flow control statement, this can either be a simple statement or a compound statement.

Selection statements: if and else

The `if` keyword is used to execute a statement or block, if, and only if, a condition is fulfilled. Its syntax is:

```
if (condition) statement
```

Here, `condition` is the expression that is being evaluated. If this `condition` is true, `statement` is executed. If it is false, `statement` is not executed (it is simply ignored), and the program continues right after the entire selection statement.
For example, the following code fragment prints the message `(x is 100)`, only if the value stored in the `x` variable is indeed 100:

```
if (x == 100)
  cout << "x is 100";
```

If `x` is not exactly 100, this statement is ignored, and nothing is printed.

If you want to include more than a single statement to be executed when the condition is fulfilled, these statements shall be enclosed in braces (`{}`), forming a block:

```
if (x == 100)
{
   cout << "x is ";
   cout << x;
}
```

As usual, indentation and line breaks in the code have no effect, so the above code is equivalent to:

```
if (x == 100) { cout << "x is "; cout << x; }
```

Selection statements with `if` can also specify what happens when the condition is not fulfilled, by using the `else`keyword to introduce an alternative statement. Its syntax is:

```
if (condition) statement1 else statement2
```

where `statement1` is executed in case condition is true, and in case it is not, `statement2` is executed.

For example:

```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

This prints `x is 100`, if indeed x has a value of 100, but if it does not, and only if it does not, it prints `x is not 100`instead.
Several if + else structures can be concatenated with the intention of checking a range of values. For example:

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

This prints whether x is positive, negative, or zero by concatenating two if-else structures. Again, it would have also been possible to execute more than a single statement per case by grouping them into blocks enclosed in braces: `{}`.

Iteration statements (loops)

Loops repeat a statement a certain number of times, or while a condition is fulfilled. They are introduced by the keywords `while`, `do`, and `for`.

The while loop

The simplest kind of loop is the while-loop. Its syntax is:

```
while (expression) statement
```

The while-loop simply repeats `statement` while `expression` is true. If, after any execution of `statement`, `expression`is no longer true, the loop ends, and the program continues right after the loop. For example, let's have a look at a countdown using a while-loop:

```
// custom countdown using while
#include <iostream>
using namespace std;

int main ()
{
  int n = 10;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "liftoff!\n";
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

Edi
&
Rur

The first statement in `main` sets *n* to a value of 10. This is the first number in the countdown. Then the while-loop begins: if this value fulfills the condition `n>0` (that *n* is greater than zero), then the block that follows the condition is executed, and repeated for as long as the condition (`n>0`) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in `main`):

1. `n` is assigned a value
2. The `while` condition is checked (`n>0`). At this point there are two possibilities:
   - condition is true: the statement is executed (to step 3)
   - condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
   ```
   cout << n << ", ";
   --n;
   ```
   (prints the value of `n` and decreases `n` by 1)
4. End of block. Return automatically to step 2.
5. Continue the program right after the block:
   print `liftoff!` and end the program.

A thing to consider with while-loops is that the loop should end at some point, and thus the statement shall alter values checked in the condition in some way, so as to force it to become false at some point. Otherwise, the loop will continue looping forever. In this case, the loop includes `--n`, that decreases the value of the variable that is being evaluated in the condition (`n`) by one - this will eventually make the condition (`n>0`) false after a certain number of loop iterations. To be more specific, after 10 iterations, `n` becomes 0, making the condition no longer true, and ending the while-loop.

Note that the complexity of this loop is trivial for a computer, and so the whole countdown is performed instantly, without any practical delay between elements of the count (if interested, see sleep_for for a countdown example with delays).

The do-while loop

A very similar loop is the do-while loop, whose syntax is:

```
do statement while (condition);
```

It behaves like a while-loop, except that `condition` is evaluated after the execution of `statement` instead of before, guaranteeing at least one execution of `statement`, even if `condition` is never fulfilled. For example, the following example program echoes any text the user introduces until the user enters goodbye:

```
// echo machine
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str;
  do {
    cout << "Enter text: ";
    getline (cin,str);
    cout << "You entered: " << str << 
'\n';
  } while (str != "goodbye");
}
```

```
Enter text: hello
You entered: hello
Enter text: who's there?
You entered: who's there?
Enter text: goodbye
You entered: goodbye
```

Edi
&
Ru

The do-while loop is usually preferred over a while-loop when the `statement` needs to be executed at least once, such as when the condition that is checked to end of the loop is determined within the loop statement itself. In the previous example, the user input within the block is what will determine if the loop ends. And thus, even if the user wants to end the loop as soon as possible by entering `goodbye`, the block in the loop needs to be executed at least once to prompt for input, and the condition can, in fact, only be determined after it is executed.

The for loop

The `for` loop is designed to iterate a number of times. Its syntax is:

```
for (initialization; condition; increase) statement;
```

Like the while-loop, this loop repeats `statement` while `condition` is true. But, in addition, the for loop provides specific locations to contain an `initialization` and an `increase` expression, executed before the loop begins the first time, and after each iteration, respectively. Therefore, it is especially useful to use counter variables as`condition`.

It works in the following way:

1. `initialization` is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
2. `condition` is checked. If it is true, the loop continues; otherwise, the loop ends, and `statement` is skipped, going directly to step 5.
3. `statement` is executed. As usual, it can be either a single statement or a block enclosed in curly braces `{ }`.
4. `increase` is executed, and the loop gets back to step 2.
5. the loop ends: execution continues by the next statement after it.

Here is the countdown example using a for loop:

```cpp
// countdown using a for loop
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

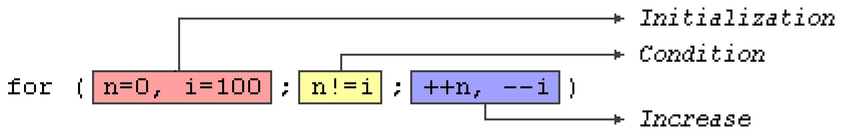
Edit
&
Run

The three fields in a for-loop are optional. They can be left empty, but in all cases the semicolon signs between them are required. For example, `for (;n<10;)` is a loop without *initialization* or *increase* (equivalent to a while-loop); and `for (;n<10;++n)` is a loop with *increase*, but no *initialization* (maybe because the variable was already initialized before the loop). A loop with no *condition* is equivalent to a loop with `true` as condition (i.e., an infinite loop).

Because each of the fields is executed in a particular time in the life cycle of a loop, it may be useful to execute more than a single expression as any of *initialization*, *condition*, or *statement*. Unfortunately, these are not statements, but rather, simple expressions, and thus cannot be replaced by a block. As expressions, they can, however, make use of the comma operator (`,`): This operator is an expression separator, and can separate multiple expressions where only one is generally expected. For example, using it, it would be possible for a for loop to handle two counter variables, initializing and increasing both:

```cpp
for ( n=0, i=100 ; n!=i ; ++n, --i )
{
   // whatever here...
}
```

This loop will execute 50 times if neither `n` or `i` are modified within the loop:



`n` starts with a value of 0, and `i` with 100, the condition is `n!=i` (i.e., that `n` is not equal to `i`). Because `n` is increased by one, and `i` decreased by one on each iteration, the loop's condition will become false after the 50th iteration, when both `n` and `i` are equal to 50.

Range-based for loop

The for-loop has another syntax, which is used exclusively with ranges:

```
for ( declaration : range ) statement;
```

This kind of for loop iterates over all the elements in `range`, where `declaration` declares some variable able to take the value of an element in this range. Ranges are sequences of elements, including arrays, containers, and any other type supporting the functions `begin` and `end`; Most of these types have not yet been introduced in this tutorial, but we are already acquainted with at least one kind of range: strings, which are sequences of characters.

An example of range-based for loop using strings:

```
// range-based for loop
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str {"Hello!"};
  for (char c : str)
  {
    std::cout << "[" << c << "]";
  }
  std::cout << '\n';
}
```

```
[H][e][l][l][o][!]
```

Edi
&
Ru

Note how what precedes the colon (`:`) in the for loop is the declaration of a `char` variable (the elements in a string are of type `char`). We then use this variable, `c`, in the statement block to represent the value of each of the elements in the range.

This loop is automatic and does not require the explicit declaration of any counter variable.

Range based loops usually also make use of type deduction for the type of the elements with `auto`. Typically, the range-based loop above can also be written as:

```
for (auto c : str)
  std::cout << "[" << c << "]";
```

Here, the type of `c` is automatically deduced as the type of the elements in `str`.

Jump statements

Jump statements allow altering the flow of a program by performing jumps to specific locations.

The break statement

`break` leaves a loop, even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, let's stop the countdown before its natural end:

```
// break loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown
aborted!
```

Edit
&
Rur

The continue statement

The `continue` statement causes the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, let's skip number 5 in our countdown:

```
// continue loop example
#include <iostream>
using namespace std;
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, liftoff!
```

Edit
&

```

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

Run

The goto statement

`goto` allows to make an absolute jump to another point in the program. This unconditional jump ignores nesting levels, and does not cause any automatic stack unwinding. Therefore, it is a feature to use with care, and preferably within the same block of statements, especially in the presence of local variables.

The destination point is identified by a *label*, which is then used as an argument for the `goto` statement. A *label* is made of a valid identifier followed by a colon (`:`).

`goto` is generally deemed a low-level feature, with no particular use cases in modern higher-level programming paradigms generally used with C++. But, just as an example, here is a version of our countdown loop using goto:

```
// goto loop example
#include <iostream>
using namespace std;

int main ()
{
  int n=10;
mylabel:
  cout << n << ", ";
  n--;
  if (n>0) goto mylabel;
  cout << "liftoff!\n";
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

Edit & Run

Another selection statement: switch.

The syntax of the switch statement is a bit peculiar. Its purpose is to check for a value among a number of possible constant expressions. It is something similar to concatenating `if-else` statements, but limited to constant expressions. Its most typical syntax is:

```
switch (expression)
{
  case constant1:
     group-of-statements-1;
     break;
  case constant2:
     group-of-statements-2;
     break;
  .
  .
  .
  default:
     default-group-of-statements
}
```

It works in the following way: `switch` evaluates `expression` and checks if it is equivalent to `constant1`; if it is, it executes `group-of-statements-1` until it finds the `break` statement. When it finds this `break` statement, the program jumps to the end of the entire `switch` statement (the closing brace).

If expression was not equal to `constant1`, it is then checked against `constant2`. If it is equal to this, it executes`group-of-statements-2` until a `break` is found, when it jumps to the end of the switch.

Finally, if the value of expression did not match any of the previously specified constants (there may be any number of these), the program executes the statements included after the `default:` label, if it exists (since it is optional).

Both of the following code fragments have the same behavior, demonstrating the if-else equivalent of a switch statement:

switch example	**if-else equivalent**
`switch (x) {`	`if (x == 1) {`
`  case 1:`	`  cout << "x is 1";`
`    cout << "x is 1";`	`}`
`    break;`	`else if (x == 2) {`
`  case 2:`	`  cout << "x is 2";`
`    cout << "x is 2";`	`}`
`    break;`	`else {`
`  default:`	`  cout << "value of x unknown";`
`    cout << "value of x unknown";`	`}`
`  }`	

The `switch` statement has a somewhat peculiar syntax inherited from the early times of the first C compilers, because it uses labels instead of blocks. In the most typical use (shown above), this means that `break` statements are needed after each group of statements for a particular label. If `break` is not included, all statements following the case (including those under any other labels) are also executed, until the end of the switch block or a jump statement (such as `break`) is reached.

If the example above lacked the break statement after the first group for case one, the program would not jump automatically to the end of the switch block after printing `x is 1`, and would instead continue executing the statements in case two (thus printing also `x is 2`). It would then continue doing so until a `break` statement is encountered, or the end of the `switch` block. This makes unnecessary to enclose the statements for each case in braces `{}`, and can also be useful to execute the same group of statements for different possible values. For example:

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
  }
```

Notice that `switch` is limited to compare its evaluated expression against labels that are constant expressions. It is not possible to use variables as labels or ranges, because they are not valid C++ constant expressions.

To check for ranges or values that are not constant, it is better to use concatenations of `if` and `else if`statements.

Functions

Functions allow to structure programs in segments of code to perform individual tasks.

In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

```
type name ( parameter1, parameter2, ...) { statements }
```

Where:
- `type` is the type of the value returned by the function.
- `name` is the identifier by which the function can be called.
- `parameters` (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma. Each parameter looks very much like a regular variable declaration (for example: `int x`), and in fact acts within the function as a regular variable which is local to the function. The purpose of parameters is to allow passing arguments to the function from the location where it is called from.
- `statements` is the function's body. It is a block of statements surrounded by braces { } that specify what the function actually does.

Let's have a look at an example:

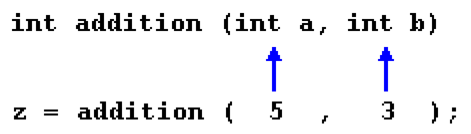
```cpp
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
}
```

```
The result is 8
```

Edi & Ru

This program is divided in two functions: `addition` and `main`. Remember that no matter the order in which they are defined, a C++ program always starts by calling `main`. In fact, `main` is the only function called automatically, and the code in any other function is only executed if its function is called from `main` (directly or indirectly).

In the example above, `main` begins by declaring the variable `z` of type `int`, and right after that, it performs the first function call: it calls `addition`. The call to a function follows a structure very similar to its declaration. In the example above, the call to `addition` can be compared to its definition just a few lines earlier:



```
int addition (int a, int b)
                  ↑      ↑
z = addition (  5   ,  3   );
```

The parameters in the function declaration have a clear correspondence to the arguments passed in the function call. The call passes two values, `5` and `3`, to the function; these correspond to the parameters `a` and `b`, declared for function `addition`.

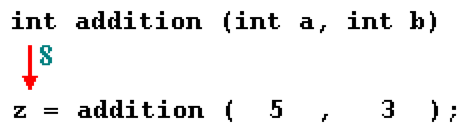
At the point at which the function is called from within main, the control is passed to function `addition`: here, execution of `main` is stopped, and will only resume once the `addition` function ends. At the moment of the function call, the value of both arguments (`5` and `3`) are copied to the local variables `int a` and `int b` within the function.

Then, inside `addition`, another local variable is declared (`int r`), and by means of the expression `r=a+b`, the result of `a` plus `b` is assigned to `r`; which, for this case, where `a` is 5 and `b` is 3, means that 8 is assigned to `r`.

The final statement within the function:

```
return r;
```

Ends function `addition`, and returns the control back to the point where the function was called; in this case: to function `main`. At this precise moment, the program resumes its course on `main` returning exactly at the same point at which it was interrupted by the call to `addition`. But additionally, because `addition` has a return type, the call is evaluated as having a value, and this value is the value specified in the return statement that ended `addition`: in this particular case, the value of the local variable `r`, which at the moment of the `return` statement had a value of 8.


```
int addition (int a, int b)
  8
z = addition (  5  ,   3  );
```

Therefore, the call to `addition` is an expression with the value returned by the function, and in this case, that value, 8, is assigned to `z`. It is as if the entire function call (`addition(5,3)`) was replaced by the value it returns (i.e., 8).

Then main simply prints this value by calling:

```
cout << "The result is " << z;
```

A function can actually be called multiple times within a program, and its argument is naturally not limited just to literals:

```
// function example
#include <iostream>
using namespace std;

int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return r;
}

int main ()
{
  int x=5, y=3, z;
  z = subtraction (7,2);
```

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

Edi
&
Rur

```
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction
(7,2) << '\n';
  cout << "The third result is " << subtraction (x,y)
<< '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " << z << '\n';
}
```

Similar to the `addition` function in the previous example, this example defines a `subtract` function, that simply returns the difference between its two parameters. This time, `main` calls this function several times, demonstrating more possible ways in which a function can be called.

Let's examine each of these calls, bearing in mind that each function call is itself an expression that is evaluated as the value it returns. Again, you can think of it as if the function call was itself replaced by the returned value:

```
z = subtraction (7,2);
cout << "The first result is " << z;
```

If we replace the function call by the value it returns (i.e., 5), we would have:

```
z = 5;
cout << "The first result is " << z;
```

With the same procedure, we could interpret:

```
cout << "The second result is " << subtraction (7,2);
```

as:

```
cout << "The second result is " << 5;
```

since 5 is the value returned by `subtraction (7,2)`.

In the case of:

```
cout << "The third result is " << subtraction (x,y);
```

The arguments passed to subtraction are variables instead of literals. That is also valid, and works fine. The function is called with the values `x` and `y` have at the moment of the call: 5 and 3 respectively, returning 2 as result.

The fourth call is again similar:

```
z = 4 + subtraction (x,y);
```

The only addition being that now the function call is also an operand of an addition operation. Again, the result is the same as if the function call was replaced by its result: 6. Note, that thanks to the commutative property of additions, the above can also be written as:

```
z = subtraction (x,y) + 4;
```

With exactly the same result. Note also that the semicolon does not necessarily go after the function call, but, as always, at the end of the whole statement. Again, the logic behind may be easily seen again by replacing the function calls by their returned value:

```
z = 4 + 2;    // same as z = 4 + subtraction (x,y);
z = 2 + 4;    // same as z = subtraction (x,y) + 4;
```

Functions with no type. The use of void

The syntax shown above for functions:

```
type name ( argument1, argument2 ...) { statements }
```

Requires the declaration to begin with a type. This is the type of the value returned by the function. But what if the function does not need to return a value? In this case, the type to be used is `void`, which is a special type to represent the absence of value. For example, a function that simply prints a message may not need to return any value:

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
  cout << "I'm a function!";
}

```

```
I'm a function!
```

Edi
&
Rur

```
int main ()
{
  printmessage ();
}
```

void can also be used in the function's parameter list to explicitly specify that the function takes no actual parameters when called. For example, printmessage could have been declared as:

```
void printmessage (void)
{
  cout << "I'm a function!";
}
```

In C++, an empty parameter list can be used instead of void with same meaning, but the use of void in the argument list was popularized by the C language, where this is a requirement.

Something that in no case is optional are the parentheses that follow the function name, neither in its declaration nor when calling it. And even when the function takes no parameters, at least an empty pair of parentheses shall always be appended to the function name. See how printmessage was called in an earlier example:

```
printmessage ();
```

The parentheses are what differentiate functions from other kinds of declarations or statements. The following would not call the function:

```
printmessage;
```

The return value of main

You may have noticed that the return type of main is int, but most examples in this and earlier chapters did not actually return any value from main.

Well, there is a catch: If the execution of main ends normally without encountering a return statement the compiler assumes the function ends with an implicit return statement:

```
return 0;
```

Note that this only applies to function `main` for historical reasons. All other functions with a return type shall end with a proper `return` statement that includes a return value, even if this is never used.

When `main` returns zero (either implicitly or explicitly), it is interpreted by the environment as that the program ended successfully. Other values may be returned by `main`, and some environments give access to that value to the caller in some way, although this behavior is not required nor necessarily portable between platforms. The values for `main` that are guaranteed to be interpreted in the same way on all platforms are:

value	description
0	The program was successful
EXIT_SUCCESS	The program was successful (same as above). This value is defined in header `<cstdlib>`.
EXIT_FAILURE	The program failed. This value is defined in header `<cstdlib>`.

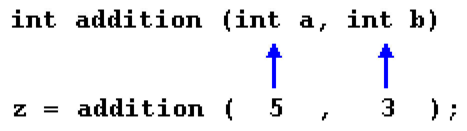
Because the implicit `return 0;` statement for `main` is a tricky exception, some authors consider it good practice to explicitly write the statement.

Arguments passed by value and by reference

In the functions seen earlier, arguments have always been passed *by value*. This means that, when calling a function, what is passed to the function are the values of these arguments on the moment of the call, which are copied into the variables represented by the function parameters. For example, take:

```
int x=5, y=3, z;
z = addition ( x, y );
```

In this case, function addition is passed 5 and 3, which are copies of the values of `x` and `y`, respectively. These values (5 and 3) are used to initialize the variables set as parameters in the function's definition, but any modification of these variables within the function has no effect on the values of the variables x and y outside it, because x and y were themselves not passed to the function on the call, but only copies of their values at that moment.



```
int addition (int a, int b)

z = addition ( 5 , 3 );
```

In certain cases, though, it may be useful to access an external variable from within a function. To do that, arguments can be passed *by reference*, instead of *by value*. For example, the function `duplicate` in this code duplicates the value of its three arguments, causing the variables used as arguments to actually be modified by the call:

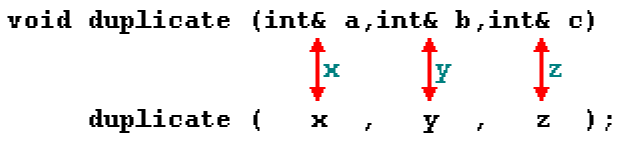
```
// passing parameters by reference
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ",
z=" << z;
  return 0;
}
```

```
x=2, y=6, z=14
```

Edi
&
Rur

To gain access to its arguments, the function declares its parameters as *references*. In C++, references are indicated with an ampersand (`&`) following the parameter type, as in the parameters taken by `duplicate` in the example above.

When a variable is passed *by reference*, what is passed is no longer a copy, but the variable itself, the variable identified by the function parameter, becomes somehow associated with the argument passed to the function, and any modification on their corresponding local variables within the function are reflected in the variables passed as arguments in the call.



```
void duplicate (int& a,int& b,int& c)

     duplicate (   x  ,  y  ,  z  );
```

In fact, `a`, `b`, and `c` become aliases of the arguments passed on the function call (`x`, `y`, and `z`) and any change on `a`within the function is actually modifying variable `x` outside the function. Any change on `b` modifies `y`, and any change on `c` modifies `z`. That is why when, in the example, function `duplicate` modifies the values of variables `a`, `b`, and `c`, the values of `x`, `y`, and `z` are affected.

If instead of defining duplicate as:

```
void duplicate (int& a, int& b, int& c)
```

Was it to be defined without the ampersand signs as:

```
void duplicate (int a, int b, int c)
```

The variables would not be passed *by reference*, but *by value*, creating instead copies of their values. In this case, the output of the program would have been the values of `x`, `y`, and `z` without being modified (i.e., 1, 3, and 7).

Efficiency considerations and const references

Calling a function with parameters taken by value causes copies of the values to be made. This is a relatively inexpensive operation for fundamental types such as `int`, but if the parameter is of a large compound type, it may result on certain overhead. For example, consider the following function:

```
string concatenate (string a, string b)
{
  return a+b;
}
```

This function takes two strings as parameters (by value), and returns the result of concatenating them. By passing the arguments by value, the function forces `a` and `b` to be copies of the arguments passed to the function when it is called. And if these are long strings, it may mean copying large quantities of data just for the function call.

But this copy can be avoided altogether if both parameters are made *references*:

```
string concatenate (string& a, string& b)
{
  return a+b;
}
```

Arguments by reference do not require a copy. The function operates directly on (aliases of) the strings passed as arguments, and, at most, it might mean the transfer of certain pointers to the function. In this regard, the version of `concatenate` taking references is more efficient than the version taking values, since it does not need to copy expensive-to-copy strings.

On the flip side, functions with reference parameters are generally perceived as functions that modify the arguments passed, because that is why reference parameters are actually for.

The solution is for the function to guarantee that its reference parameters are not going to be modified by this function. This can be done by qualifying the parameters as constant:

```
string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

By qualifying them as `const`, the function is forbidden to modify the values of neither `a` nor `b`, but can actually access their values as references (aliases of the arguments), without having to make actual copies of the strings.

Therefore, `const` references provide functionality similar to passing arguments by value, but with an increased efficiency for parameters of large types. That is why they are extremely popular in C++ for arguments of compound types. Note though, that for most fundamental types, there is no noticeable difference in efficiency, and in some cases, const references may even be less efficient!

Inline functions

Calling a function generally causes a certain overhead (stacking arguments, jumps, etc...), and thus for very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of formally calling a function.

Preceding a function declaration with the `inline` specifier informs the compiler that inline expansion is preferred over the usual function call mechanism for a specific function. This does not change at all the behavior of a function, but is merely used to suggest the compiler that the code generated by the function body shall be inserted at each point the function is called, instead of being invoked with a regular function call.

For example, the concatenate function above may be declared inline as:

```
inline string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

This informs the compiler that when `concatenate` is called, the program prefers the function to be expanded inline, instead of performing a regular call. `inline` is only specified in the function declaration, not when it is called.

Note that most compilers already optimize code to generate inline functions when they see an opportunity to improve efficiency, even if not explicitly marked with the `inline` specifier. Therefore, this specifier merely indicates the compiler that inline is preferred for this function, although the compiler is free to not inline it, and optimize otherwise. In C++, optimization is a task delegated to the compiler, which is free to generate any code for as long as the resulting behavior is the one specified by the code.

Default values in parameters

In C++, functions can also have optional parameters, for which no arguments are required in the call, in such a way that, for example, a function with three parameters may be called with only two. For this, the function shall include a default value for its last parameter, which is used by the function when called with fewer arguments. For example:

```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12) << '\n';
  cout << divide (20,4) << '\n';
  return 0;
}
```

```
6
5
```

Edit
&
Run

In this example, there are two calls to function `divide`. In the first one:

```
divide (12)
```

The call only passes one argument to the function, even though the function has two parameters. In this case, the function assumes the second parameter to be 2 (notice the function definition, which declares its second parameter as `int b=2`). Therefore, the result is 6.

In the second call:

```
divide (20,4)
```

The call passes two arguments to the function. Therefore, the default value for b (int b=2) is ignored, and b takes the value passed as argument, that is 4, yielding a result of 5.

Declaring functions

In C++, identifiers can only be used in expressions once they have been declared. For example, some variable xcannot be used before being declared with a statement, such as:

```
int x;
```

The same applies to functions. Functions cannot be called before they are declared. That is why, in all the previous examples of functions, the functions were always defined before the main function, which is the function from where the other functions were called. If main were defined before the other functions, this would break the rule that functions shall be declared before being used, and thus would not compile.

The prototype of a function can be declared without actually defining the function completely, giving just enough details to allow the types involved in a function call to be known. Naturally, the function shall be defined somewhere else, like later in the code. But at least, once declared like this, it can already be called.

The declaration shall include all types involved (the return type and the type of its arguments), using the same syntax as used in the definition of the function, but replacing the body of the function (the block of statements) with an ending semicolon.

The parameter list does not need to include the parameter names, but only their types. Parameter names can nevertheless be specified, but they are optional, and do not need to necessarily match those in the function definition. For example, a function called protofunction with two int parameters can be declared with either of these statements:

```
int protofunction (int first, int second);
int protofunction (int, int);
```

Anyway, including a name for each parameter always improves legibility of the declaration.

```
// declaring functions prototypes
#include <iostream>
using namespace std;
```

```
Please, enter number (0 to exit): 9
It is odd.
Please, enter number (0 to exit): 6
```

Edi
&

```

void odd (int x);
void even (int x);

int main()
{
  int i;
  do {
    cout << "Please, enter number (0 to
exit): ";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}

void odd (int x)
{
  if ((x%2)!=0) cout << "It is odd.\n";
  else even (x);
}

void even (int x)
{
  if ((x%2)==0) cout << "It is even.\n";
  else odd (x);
}
```

```
It is even.
Please, enter number (0 to exit): 1030
It is even.
Please, enter number (0 to exit): 0
It is even.
```

Rur

This example is indeed not an example of efficiency. You can probably write yourself a version of this program with half the lines of code. Anyway, this example illustrates how functions can be declared before its definition:

The following lines:

```
void odd (int a);
void even (int a);
```

Declare the prototype of the functions. They already contain all what is necessary to call them, their name, the types of their argument, and their return type (`void` in this case). With these prototype declarations in place, they can be called before they are entirely defined, allowing for example, to place the function from where they are called (`main`) before the actual definition of these functions.

But declaring functions before being defined is not only useful to reorganize the order of functions within the code. In some cases, such as in this particular case, at least one of the declarations is required, because `odd` and `even`are mutually called; there is a call to `even` in `odd` and a call to `odd` in `even`. And, therefore, there is no way to structure the code so that `odd` is defined before `even`, and `even` before `odd`.

Recursivity

Recursivity is the property that functions have to be called by themselves. It is useful for some tasks, such as sorting elements, or calculating the factorial of numbers. For example, in order to obtain the factorial of a number (`n!`) the mathematical formula would be:

```
n! = n * (n-1) * (n-2) * (n-3) ... * 1
```

More concretely, `5!` (factorial of 5) would be:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

And a recursive function to calculate this in C++ could be:

```
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

int main ()
{
  long number = 9;
  cout << number << "! = " << factorial
(number);
  return 0;
}
```

```
9! = 362880
```

Edi
&
Ru

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since, otherwise, the function would perform an infinite recursive loop, in which once it arrived to 0, it would continue multiplying by all the negative numbers (probably provoking a stack overflow at some point during runtime).

Overloads and templates

Overloaded functions

In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type. For example:

```cpp
// overloading functions
#include <iostream>
using namespace std;

int operate (int a, int b)
{
  return (a*b);
}

double operate (double a, double b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  double n=5.0,m=2.0;
  cout << operate (x,y) << '\n';
  cout << operate (n,m) << '\n';
  return 0;
}
```

```
10
2.5
```

Edit
&
Run

In this example, there are two functions called `operate`, but one of them has two parameters of type `int`, while the other has them of type `double`. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two `int` arguments, it calls to the function that has two`int` parameters, and if it is called with two `double`s, it calls the one with two `double`s.

In this example, both functions have quite different behaviors, the `int` version multiplies its arguments, while the`double` version divides them. This is generally not a good idea. Two functions with the same name are generally expected to have -at least- a similar behavior, but this example demonstrates that is entirely possible for them not to. Two overloaded functions (i.e., two functions with the same name) have entirely different definitions; they are, for all purposes, different functions, that only happen to have the same name.

Note that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

Function templates

Overloaded functions may have the same definition. For example:

```cpp
// overloaded functions
#include <iostream>
using namespace std;
```

```
30
2.5
```

Edit
&

```cpp

int sum (int a, int b)
{
  return a+b;
}

double sum (double a, double b)
{
  return a+b;
}

int main ()
{
  cout << sum (10,20) << '\n';
  cout << sum (1.0,1.5) << '\n';
  return 0;
}
```

Rur

Here, `sum` is overloaded with different parameter types, but with the exact same body.

The function `sum` could be overloaded for a lot of types, and it could make sense for all of them to have the same body. For cases such as this, C++ has the ability to define functions with generic types, known as *function templates*. Defining a function template follows the same syntax as a regular function, except that it is preceded by the `template` keyword and a series of template parameters enclosed in angle-brackets <>:

`template <template-parameters> function-declaration`

The template parameters are a series of parameters separated by commas. These parameters can be generic template types by specifying either the `class` or `typename` keyword followed by an identifier. This identifier can then be used in the function declaration as if it was a regular type. For example, a generic `sum` function could be defined as:

```cpp
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

It makes no difference whether the generic type is specified with keyword `class` or keyword `typename` in the template argument list (they are 100% synonyms in template declarations).

In the code above, declaring `SomeType` (a generic type within the template parameters enclosed in angle-brackets) allows `SomeType` to be used anywhere in the function definition, just as any other type; it can be used as the type for parameters, as return type, or to declare new variables of this type. In all cases, it represents a generic type that will be determined on the moment the template is instantiated.

Instantiating a template is applying the template to create a function using particular types or values for its template parameters. This is done by calling the *function template*, with the same syntax as calling a regular function, but specifying the template arguments enclosed in angle brackets:

```
name <template-arguments> (function-arguments)
```

For example, the `sum` function template defined above can be called with:

```
x = sum<int>(10,20);
```

The function `sum<int>` is just one of the possible instantiations of function template `sum`. In this case, by using `int`as template argument in the call, the compiler automatically instantiates a version of `sum` where each occurrence of`SomeType` is replaced by `int`, as if it was defined as:

```
int sum (int a, int b)
{
  return a+b;
}
```

Let's see an actual example:

```
// function template
#include <iostream>
using namespace std;

template <class T>
T sum (T a, T b)
{
  T result;
  result = a + b;
  return result;
}

int main () {
  int i=5, j=6, k;
  double f=2.0, g=0.5, h;
  k=sum<int>(i,j);
  h=sum<double>(f,g);
  cout << k << '\n';
  cout << h << '\n';
  return 0;
}
```

```
11
2.5
```

Edi
&
Rur

In this case, we have used `T` as the template parameter name, instead of `SomeType`. It makes no difference, and `T`is actually a quite common template parameter name for generic types.

In the example above, we used the function template `sum` twice. The first time with arguments of type `int`, and the second one with arguments of type `double`. The compiler has instantiated and then called each time the appropriate version of the function.

Note also how `T` is also used to declare a local variable of that (generic) type within `sum`:

```
T result;
```

Therefore, result will be a variable of the same type as the parameters `a` and `b`, and as the type returned by the function.
In this specific case where the generic type `T` is used as a parameter for `sum`, the compiler is even able to deduce the data type automatically without having to explicitly specify it within angle brackets. Therefore, instead of explicitly specifying the template arguments with:

```
k = sum<int> (i,j);
h = sum<double> (f,g);
```

It is possible to instead simply write:

```
k = sum (i,j);
h = sum (f,g);
```

without the type enclosed in angle brackets. Naturally, for that, the type shall be unambiguous. If `sum` is called with arguments of different types, the compiler may not be able to deduce the type of `T` automatically.

Templates are a powerful and versatile feature. They can have multiple template parameters, and the function can still use regular non-templated types. For example:

```
// function templates
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T a, U b)
{
  return (a==b);
}

int main ()
{
  if (are_equal(10,10.0))
    cout << "x and y are equal\n";
```

```
x and y are equal
```

Edit
&
Run

```
  else
    cout << "x and y are not equal\n";
  return 0;
}
```

Note that this example uses automatic template parameter deduction in the call to `are_equal`:

```
are_equal(10,10.0)
```

Is equivalent to:

```
are_equal<int,double>(10,10.0)
```

There is no ambiguity possible because numerical literals are always of a specific type: Unless otherwise specified with a suffix, integer literals always produce values of type `int`, and floating-point literals always produce values of type `double`. Therefore `10` has always type `int` and `10.0` has always type `double`.

Non-type template arguments

The template parameters can not only include types introduced by `class` or `typename`, but can also include expressions of a particular type:

```
// template arguments
#include <iostream>
using namespace std;

template <class T, int N>
T fixed_multiply (T val)
{
  return val * N;
}

int main() {
  std::cout << fixed_multiply<int,2>(10) <<
'\n';
  std::cout << fixed_multiply<int,3>(10) <<
'\n';
}
```

```
20
30
```

Edi
&
Ru

The second argument of the `fixed_multiply` function template is of type `int`. It just looks like a regular function parameter, and can actually be used just like one.

But there exists a major difference: the value of template parameters is determined on compile-time to generate a different instantiation of the function `fixed_multiply`, and thus the value of that argument is never passed during runtime: The two calls to `fixed_multiply` in `main` essentially call two versions of the function: one that always multiplies by two, and one that always multiplies by three. For that same reason, the second template argument needs to be a constant expression (it cannot be passed a variable).