

To whom is this tutorial directed?

This tutorial is for those people who want to learn programming in C++ and do not necessarily have any previous knowledge of other programming languages. Of course any knowledge of other programming languages or any general computer skill can be useful to better understand this tutorial, although it is not essential.

If you are familiar with C language you can take the first 3 parts of this tutorial (from 1.1 to 3.4) as a review, since they mainly explain the C part of C++.

Part 4 describes object-oriented programming.

Part 5 mostly describes the new features introduced by ANSI-C++ standard.

Structure of this tutorial

The tutorial is divided in 6 parts and each part is in several different sections. You can access any section directly from the [main index](#) or begin the tutorial from any point and follow the links at the bottom of each section.

Many sections include an additional page with specific examples that describe the use of the newly acquired knowledge in that chapter. It is recommended to read these examples and be able to understand each of the code lines that constitute it before passing to the next chapter.

A good way to gain experience with a programming language is by modifying and adding new functionalities on your own to the example programs that you fully understand. Don't be scared to modify the examples provided with this tutorial. There are no reports of people whose computer has been destroyed due to that.

Compatibility Notes

The ANSI-C++ standard accepted as an international standard is relatively recent. It was published in November 1997, nevertheless the C++ language exists from long ago (1980s). Therefore there are many compilers which do not support all the new capabilities included in ANSI-C++, specially those released prior to the publication of the standard.

During this tutorial, the concepts that have been added by ANSI-C++ standard which are not included in most older C++ compilers are indicated by the following icon:



<- new in ANSI C++

Also, given the enormous extension that the C language enjoys (the language from which C++ was derived), an icon will also be included when the topic explained is a concept whose implementation is clearly different between C and C++ or that is exclusive of C++:



<- different implementation in C and C++

Compilers

The examples included in this tutorial are all console programs. That means they use text to communicate with the user and to show results.

All C++ compilers support the compilation of console programs. If you want to get more information on how to compile the examples that appear in this tutorial, check the document [Compilation of Console Programs](#), where you will find specific information about this subject for several C++ compilers existing in the market.

Section 1.1

Structure of a C++ program



Probably the best way to start learning a programming language is with a program. So here is our first program:

```
// my first program in C++  
  
#include <iostream.h>  
  
int main ()  
{  
    cout << "Hello World!";  
    return 0;  
}
```

```
Hello World!
```

The left side shows the source code for our first program, which we can name, for example, `hiworld.cpp`. The right side shows the result of the program once compiled and executed. The way to edit and compile a program depends on the compiler you are using. Depending on whether it has a Development Interface or not and on its version. Consult section [compilers](#) and the manual or help included with your compiler if you have doubts on how to compile a C++ console program.

The previous program is the first program that most programming apprentices write, and its result is the printing on screen of the "Hello World!" sentence. It is one of the simpler programs that can be written in C++, but it already includes the basic components that every C++ program has. We are going to take a look at them one by one:

```
// my first program in C++
```

This is a comment line. All the lines beginning with two slash signs (`//`) are considered comments and do not have any effect on the behavior of the program. They

can be used by the programmer to include short explanations or observations within the source itself. In this case, the line is a brief description of what our program does.

```
#include <iostream.h>
```

Sentences that begin with a pound sign (#) are directives for the preprocessor. They are not executable code lines but indications for the compiler. In this case the sentence `#include <iostream.h>` tells the compiler's preprocessor to include the **iostream** standard header file. This specific file includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is used later in the program.

```
int main ()
```

This line corresponds to the beginning of the **main** function declaration. The **main** function is the point where all C++ programs begin their execution. It is independent of whether it is at the beginning, at the end or in the middle of the code - its content is always the first to be executed when a program starts. In addition, for that same reason, it is essential that all C++ programs have a **main** function.

main is followed by a pair of parenthesis () because it is a function. In C++ all functions are followed by a pair of parenthesis () that, optionally, can include arguments within them. The content of the **main** function immediately follows its formal declaration and it is enclosed between curly brackets ({}), as in our example.

```
cout << "Hello World";
```

This instruction does the most important thing in this program. **cout** is the standard output stream in C++ (usually the screen), and the full sentence inserts a sequence of characters (in this case "Hello World") into this output stream (the screen). **cout** is declared in the `iostream.h` header file, so in order to be able to use it that file must be included.

Notice that the sentence ends with a semicolon character (;). This character signifies the end of the instruction and must be included after every instruction in any C++ program (one of the most common errors of C++ programmers is indeed to forget to include a semicolon ; at the end of each instruction).

```
return 0;
```

The **return** instruction causes the **main()** function finish and return the code that the instruction is followed by, in this case **0**. This is the most usual way to terminate a program that has not found any errors during its execution. As you will see in coming examples, all C++ programs end with a sentence similar to this.

Therefore, you may have noticed that not all the lines of this program did an action. There were lines containing only comments (those beginning by //), lines with instructions for the compiler's preprocessor (those beginning by #), then there were lines that initiated the declaration of a function (in this case, the **main** function) and, finally lines with instructions (like the call to `cout <<`), these last ones were all included within the block delimited by the curly brackets ({}) of the **main** function.

The program has been structured in different lines in order to be more readable, but it is not compulsory to do so. For example, instead of

```
int main ()
{
    cout << " Hello World ";
```

```
    return 0;
}
```

we could have written:

```
int main () { cout << " Hello World "; return 0; }
```

in just one line and this would have had exactly the same meaning.

In C++ the separation between instructions is specified with an ending semicolon (;) after each one. The division of code in different lines serves only to make it more legible and schematic for humans that may read it.

Here is a program with some more instructions:

```
// my second program in C++
```

```
#include <iostream.h>
```

```
int main ()
```

```
{
```

```
    cout << "Hello World! ";
```

```
    cout << "I'm a C++ program";
```

```
    return 0;
```

```
}
```

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

In this case we used the `cout <<` method twice in two different instructions. Once again, the separation in different lines of the code has just been done to give greater readability to the program, since `main` could have been perfectly defined thus:

```
int main () { cout << " Hello World! "; cout << " I'm to C++ program ";
return 0; }
```

We were also free to divide the code into more lines if we considered it convenient:

```
int main ()
```

```
{
```

```
    cout <<
```

```
        "Hello World!";
```

```
    cout
```

```
        << "I'm a C++ program";
```

```
    return 0;
```

```
}
```

And the result would have been exactly the same than in the previous examples.

Preprocessor directives (those that begin by #) are out of this rule since they are not true instructions. They are lines read and discarded by the preprocessor and do not produce any code. These must be specified in their own line and do not require the include a semicolon (;) at the end.

Comments.

Comments are pieces of source code discarded from the code by the compiler. They do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.

C++ supports two ways to insert comments:

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

The first of them, the line comment, discards everything from where the pair of slash signs (//) is found up to the end of that same line. The second one, the block comment, discards everything between the /* characters and the next appearance of the */ characters, with the possibility of including several lines.

We are going to add comments to our second program:

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

```
#include <iostream.h>
```

```
int main ()
```

```
{
    cout << "Hello World! ";    //
    says Hello World!
    cout << "I'm a C++ program"; //
    says I'm a C++ program
    return 0;
}
```

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

If you include comments within the sourcecode of your programs without using the comment characters combinations //, /* or */, the compiler will take them as if they were C++ instructions and, most likely causing one or several error messages.

Section 1.2

Variables. Data types. Constants.



The usefulness of the "Hello World" programs shown in the previous section are something more than questionable. We had to write several lines of code, compile them, and then execute the resulting program just to obtain a sentence on the screen as the result. It is true that it would have been much faster to simply write the output sentence by ourselves, but programming is not limited only to printing texts on 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 the **variable**.

Let's think that I ask you to retain the number 5 in your mental memory, and then I ask you to also memorize the number 2. You have just stored two values in your memory. Now, if I ask you to add 1 to the first number I said, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Values that we could now subtract and obtain 4 as the result.

All this process that you have made is a simile of what a computer can do with two variables. This same process can be expressed in C++ with the following instruction set:

```
a = 5;
b = 2;
a = a + 1;
result = a - b;
```

Obviously this is a very simple example since we have only used two small integer values, but consider that your computer can store millions of numbers like these at the same time and conduct sophisticated mathematical operations with them.

Therefore, we can define a variable as a portion of memory to store a determined value.

Each variable needs an identifier that distinguishes it from the others, for example, in the previous code the variable identifiers were `a`, `b` and `result`, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

Identifiers

A valid identifier is a sequence of one or more letters, digits or underline symbols (`_`). The length of an identifier is not limited, although for some compilers only the 32 first characters of an identifier are significant (the rest are not considered).

Neither spaces nor marked letters can be part of an identifier. Only letters, digits and underline characters are valid. In addition, variable identifiers should always begin with a letter. They can also begin with an underline character (`_`), but this is usually reserved for external links. They can never begin with a digit.

Another rule that you have to consider when inventing your own *identifiers* is that they cannot match any **key word** of the C++ language nor your compiler's specific ones since they could be confused with these. For example, the following expressions are always considered key words according to the ANSI-C++ standard and therefore they must not be used as identifiers:

`asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t`

Additionally, alternative representations for some operators do not have to be used as identifiers since they are reserved words under some circumstances:

`and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq`

Your compiler may also include some more specific reserved keywords. For example, many compilers which generate 16 bit code (like some compilers for DOS) also include `far`, `huge` and `near` as key words.

Very important: The C++ language is "case sensitive", 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 variable `RESULT` is not the same as the variable `result` nor the variable `Result`.

Data types

When programming, we store the variables in our computer's memory, but the computer must know what we want to store in them since storing a simple number, a letter or a large number is not going to occupy the same space in memory.

Our computer's memory is organized in bytes. A byte is the minimum amount of memory that we can manage. A byte can store a relatively small amount of data, usually an integer between 0 and 255 or one single character. But in addition, the computer can manipulate more complex data types that come from grouping several *bytes*, such as long numbers or numbers with decimals. Next you have a list of the existing fundamental data types in C++, as well as the range of values that can be represented with each one of them:

DATA TYPES

Name	Bytes*	Description	Range*
char	1	character or integer 8 bits length.	signed: -128 to 127 unsigned: 0 to 255
short	2	integer 16 bits length.	signed: -32768 to 32767 unsigned: 0 to 65535
long	4	integer 32 bits length.	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
int	*	Integer. Its length traditionally depends on the length of the system's Word type , thus in MSDOS it is 16 bits long, whereas in 32 bit systems (like Windows 9x/2000/NT and systems that work under protected mode in x86 systems) it is 32 bits long (4 bytes).	See short, long
float	4	floating point number.	3.4e + / - 38 (7 digits)
double	8	double precision floating point number.	1.7e + / - 308 (15 digits)
long double	10	long double precision floating point number.	1.2e + / - 4932 (19 digits)
bool	1	Boolean value. It can take one of two values: <code>true</code> or <code>false</code> NOTE: this is a type recently added by the ANSI-C++ standard. Not all compilers support it. Consult section bool type for compatibility information.	true or false
wchar_t	2	Wide character. It is designed as a type to store international characters of a two-byte character set. NOTE: this is a type recently added by the ANSI-C++ standard. Not all compilers support it.	wide characters

* Values of columns Bytes and Range may vary depending on your system. The values included here are the most commonly accepted and used by almost all compilers.

In addition to these fundamental data types there also exist the pointers and the **void** parameter type specification, that we will see later.

Declaration of variables

In order to use a variable in C++, we must first declare it specifying which of the data types above we want it to be. The syntax to declare a new variable is to write the data type specifier that we want (like **int**, **short**, **float**...) followed by a valid variable identifier. For example:

```
int a;  
float mynumber;
```

Are 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, variables **a** and **mynumber** can be used within the rest of their scope in the program.

If you need to declare several variables of the same type and you want to save some writing work you can declare all of them in the same line separating the identifiers with commas. For example:

```
int a, b, c;
```

declares three variables (**a**, **b** and **c**) of type **int**, and has exactly the same meaning as if we had written:

```
int a;  
int b;  
int c;
```

Integer data types (**char**, **short**, **long** and **int**) can be signed or unsigned according to the range of numbers that we need to represent. Thus to specify an integer data type we do it by putting the keyword **signed** or **unsigned** before the data type itself. For example:

```
unsigned short NumberOfSons;  
signed int MyAccountBalance;
```

By default, if we do not specify **signed** or **unsigned** it will be assumed that the type is **signed**, therefore in the second declaration we could have written:

```
int MyAccountBalance;
```

with exactly the same meaning and since this is the most usual way, few source codes include the keyword **signed** as part of a compound type name.

The only exception to this rule is the **char** type that exists by itself and it is considered a different type than **signed char** and **unsigned char**.

Finally, **signed** and **unsigned** may also be used as a simple types, meaning the same as **signed int** and **unsigned int** respectively. The following two declarations are equivalent:

```
unsigned MyBirthYear;  
unsigned int MyBirthYear;
```

To see what variable declaration looks like in action in a program, we are going to show the C++ code of the example about your mental memory proposed at the beginning of this section:

```
// operating with variables  
  
#include <iostream.h>
```




```
int main ()
{
    // declaring variables:
    int a, b;
    int result;

    // process:
    a = 5;
    b = 2;
    a = a + 1;
    result = a - b;

    // print out the result:
    cout << result;

    // terminate the program:
    return 0;
}
```

Do not worry if something about the variable declarations looks a bit strange to you. You will see the rest in detail in coming sections.

Initialization of variables

When declaring a local variable, its value is undetermined by default. But you may want a variable to store a concrete value the moment that it is declared. In order to do that, you have to append an equal sign followed by the value wanted to the variable declaration:

type identifier = initial_value ;

For example, if we want to declare an `int` variable called `a` that contains the value `0` at the moment in which it is declared, we could write:

```
int a = 0;
```

Additionally to this way of initializing variables (known as c-like), C++ has added a new way to initialize a variable: by enclosing the initial value between parenthesis ():

type identifier (initial_value) ;

For example:

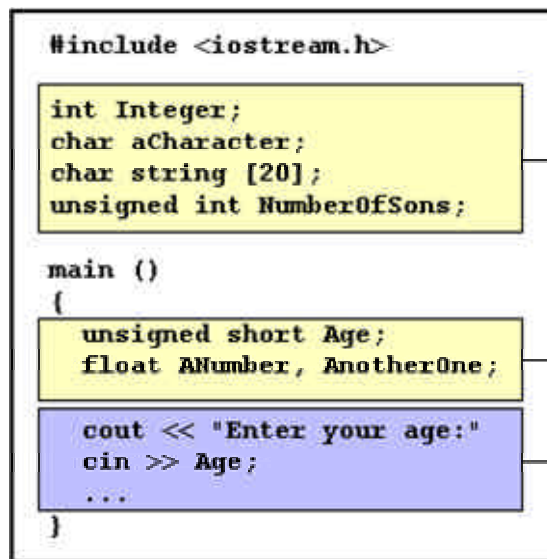
```
int a (0);
```

Both ways are valid and equivalent in C++.

Scope of variables

All the variables that we are going to use must have been previously declared. An important difference between the C and C++ languages, is that in C++ we can declare variables anywhere in the source code, even between two executable sentences, and not only at the beginning of a block of instructions, like happens in C.

Anyway, it is recommended under some circumstances to follow the indications of the C language when declaring variables, since it can be useful when debugging a program to have all the declarations grouped together. Therefore, the traditional C-like way to declare variables is to include their declaration at the beginning of each function (for local variables) or directly in the body of the program outside any function (for global variables).



Global variables can be referred to anywhere in the code, within any function, whenever it is after its declaration.

The scope of the **local variables** is limited to the code level in which they are declared. If they are declared at the beginning of a function (like in `main`) their scope is the whole `main` function. In the example above, this means that if another function existed in

addition to `main()`, the local variables declared in `main` could not be used in the other function and vice versa.

In C++, the scope of a local variable is given by the block in which it is declared (a block is a group of instructions grouped together within curly brackets `{}` signs). If it is declared within a function it will be a variable with function scope, if it is declared in a loop its scope will be only the loop, etc...

In addition to **local** and **global** scopes there exists external scope, that causes a variable to be visible not only in the same source file but in all other files that will be linked together.

Constants: Literals.

A constant is any expression that has a fixed value. They can be divided in Integer Numbers, Floating-Point Numbers, Characters and Strings.

Integer Numbers

```

1776
707
-273

```

they are numerical constants that identify integer decimal numbers. Notice that to express a numerical constant we do not need to write quotes (") nor any special character. There is no doubt that it is a constant: whenever we write `1776` in a program we will be referring to the value `1776`.

In addition to decimal numbers (those that all of us already know) C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we must precede it with a `0` character (zero character). And to express a hexadecimal number we have to precede it with 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 them represent the same number: 75 (seventy five) expressed as a radix-10 number, octal and hexadecimal, respectively.

[Note: You can find more information on hexadecimal and octal representations in the document [Numerical radices](#)]

Floating Point Numbers

They express numbers with decimals and/or exponents. They can include a decimal point, an **e** character (that expresses "by ten at the Xth height", where X is the following integer value) or both.

```
3.14159      // 3.14159
6.02e23      // 6.02 x 1023
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 3 expressed as a floating point numeric literal.

Characters and strings

There also exist non-numerical constants, like:

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

The first two expressions represent single characters, and the following two represent strings of several characters. Notice that to represent a single character we enclose it between single quotes (') and to express a string of more than one character we enclose them between double quotes (").

When writing both single characters and strings of characters in a constant way, it is necessary to put the quotation marks to distinguish them from possible variable identifiers or reserved words. Notice this:

```
x
'x'
```

x refers to variable **x**, whereas **'x'** refers to the character constant **'x'**.

Character constants and string constants have certain peculiarities, like the **escape codes**.

These are special characters that cannot be expressed otherwise in the sourcecode of a program, like *newline* (\n) or *tab* (\t). All of them are preceded by an inverted slash (\). Here you have a list of such escape codes:

\n	newline
\r	carriage return
\t	tabulation
\v	vertical tabulation
\b	backspace
\f	page feed

<code>\a</code>	alert (beep)
<code>\'</code>	single quotes (')
<code>\"</code>	double quotes (")
<code>\?</code>	question (?)
<code>\\</code>	inverted slash (\)

For example:

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

Additionally, you can express any character by its numerical ASCII code by writing an inverted slash bar character (\) followed by the ASCII code expressed as an octal (radix-8) or hexadecimal (radix-16) number. In the first case (octal) the number must immediately follow the inverted slash (for example `\23` or `\40`), in the second case (hexadecimal), you must put an `x` character before the number (for example `\x20` or `\x4A`).

[Consult the document [ASCII Code](#) for more information about this type of escape code].

constants of string of characters can be extended by more than a single code line if each code line ends with an inverted slash (\):

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

You can also concatenate several string constants separating them by one or several blankspaces, tabulators, newline or any other valid blank character:

```
"we form" "a single" "string" "of characters"
```

Defined constants (`#define`)

You can define your own names for constants that you use quite often without having to resort to variables, simply by using the `#define` preprocessor directive. This is its format:

```
#define identifier value
```

For example:

```
#define PI 3.14159265
#define NEWLINE '\n'
#define WIDTH 100
```

they define three new constants. Once they are declared, you are able to use them in the rest of the code as any if they were any other constant, for example:

```
circle = 2 * PI * r;
cout << NEWLINE;
```

In fact the only thing that the compiler does when it finds `#define` directives is to replace literally any occurrence of the them (in the previous example, `PI`, `NEWLINE` or `WIDTH`) by the code to which they have been defined (`3.14159265`, `'\n'` and `100`, respectively). For this reason, `#define` constants are considered *macro constants*.

The `#define` directive is not a code instruction, it is a directive for the preprocessor, therefore it assumes the whole line as the directive and does not require a semicolon (;) at the end of it. If you include a semicolon character (;) at the end, it will also be added when the preprocessor will substitute any occurrence of the defined constant within the body of the program.

declared constants (`const`)

With the `const` prefix you can declare constants with a specific type exactly as you would do with a variable:

```
const int width = 100;
const char tab = '\\t';
const zip = 12440;
```

In case that the type was not specified (as in the last example) the compiler assumes that it is type `int`.

Section 1.3

Operators.



Once we know of the existence of variables and constants we can begin to operate with them. For that purpose, C++ provides the operators, which in this language are a set of keywords and signs that are not part of the alphabet but are available in all keyboards. It is important to know them since they are the basis of the C++ language.

You do not have to memorize all the content of this page, the details are only provided to serve as a later reference in case you need it.

Assignment (=).

The assignment operator serves to assign a value to a variable.

```
a = 5;
```

assigns the integer value **5** to variable **a**. The part at the left of the = operator is known as *lvalue* (left value) and the right one as *rvalue* (right value). *lvalue* must always be a variable whereas the right side can be either a constant, a variable, the result of an operation or any combination of them.

It is necessary to emphasize that the assignment operation always takes place from right to left and never at the inverse.

```
a = b;
```

assigns to variable **a** (*lvalue*) the value that contains variable **b** (*rvalue*) independently of the value that was stored in **a** at that moment. Consider also that we are only assigning the value of **b** to **a** and that a later change of **b** would not affect the new value of **a**.

For example, if we take this code (with the evolution of the variables' content in green color):

```
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
```

will give us the result that the value contained in **a** is **4** and the one contained in **b** is **7**. The final modification of **b** has not affected **a**, although before we have declared **a = b**; (right-to-left rule).

A property that C++ has over other programming languages is that the assignment operation can be used as the *rvalue* (or part of an *rvalue*) for another assignment. For example:

```
a = 2 + (b = 5);
```

is equivalent to:

```
b = 5;
```

```
a = 2 + b;
```

that means: first assign 5 to variable **b** and then assign to **a** the value 2 plus the result of the previous assignment of **b** (that is 5), leaving **a** with a final value of 7. Thus, the following expression is also valid in C++:

```
a = b = c = 5;
```

assigns 5 to the three variables **a**, **b** and **c**.

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

The five arithmetical operations supported by the language are:

+ addition

- subtraction

* multiplication

/ division

% module

Operations of addition, subtraction, multiplication and division should not suppose an understanding challenge for you since they literally correspond with their respective mathematical operators.

The only one that may not be known by you is the **module**, specified with the percentage sign (%). Module is the operation that gives the remainder of a division of two integer values. For example, if we write **a = 11 % 3;**, the variable **a** will contain 2 as the result since 2 is the remainder from dividing 11 between 3.

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

A feature of assignment in C++ that contributes to its fame of sparing language when writing are the compound assignment operators (+=, -=, *= and /= among others), which allow to modify the value of a variable with one of the basic operators:

value += increase; is equivalent to **value = value + increase;**

a -= 5; is equivalent to **a = a - 5;**

a /= b; is equivalent to **a = a / b;**

price *= units + 1; is equivalent to **price = price * (units + 1);**

and the same for all other operations.

Increase and decrease.

Another example of saving language when writing code are the *increase* operator (++) and the *decrease* operator (--). They increase or reduce by 1 the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

```
a++;
```

```
a+=1;
```

```
a=a+1;
```

are all equivalent in its functionality: the three increase by 1 the value of **a**.

Its existence is because in the first C compilers the three previous expressions produced different executable code according to which one was used. Nowadays this type of code optimization is generally done automatically by the compiler.

A characteristic of this operator is that it can be used both as a *prefix* or as a *suffix*. That means it can be written before the variable identifier (**++a**) or after (**a++**). Although in simple expressions like **a++** or **++a** they have exactly the same meaning, in other operations in which the result of the *increase* or *decrease* operation is evaluated as another expression they may have an important difference in their meaning: In case that the increase operator is used as a *prefix* (**++a**) the value is increased before the expression is evaluated and therefore the increased value is considered in the expression; in case that it is used as a *suffix* (**a++**) the value stored in **a** is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the expression. Notice the difference:

Example 1

```
B=3;
A=++B;
// A is 4, B is 4
```

Example 2

```
B=3;
A=B++;
// A is 3, B is 4
```

In Example 1, **B** is increased before its value is copied to **A**. While in Example 2, the value of **B** is copied to **A** and **B** is later increased.

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

In order to evaluate a comparison between two expressions we can use the Relational operators. As specified by the ANSI-C++ standard, the result of a relational operation is a **bool** value that can only be **true** or **false**, according to the result of the comparison.

We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other. Here is a list of the relational operators that can be performed in C++:

== Equal

!= Different

> Greater than

< Less than

>= Greater or equal than

<= Less or equal than

Here you have some examples:

(**7 == 5**) would return **false**.

(**5 > 4**) would return **true**.

(**3 != 2**) would return **true**.

(**6 >= 6**) would return **true**.

(**5 < 5**) would return **false**.

of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that **a=2**, **b=3** and **c=6**,

(**a == 5**) would return **false**.

(**a*b >= c**) would return **true** since (**2*3 >= 6**) is it.

(**b+4 > a*c**) would return **false** since (**3+4 > 2*6**) is it.

((**b=2**) == **a**) would return **true**.

Be aware. Operator **=** (one equal sign) is not the same as operator **==** (two equal signs), the first is an assignation operator (assigns the right side of the expression to the

variable in the left) and the other (==) is a relational operator of equality that compares whether both expressions in the two sides of the operator are equal to each other. Thus, 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 value 2, so the result of the operation is **true**.



In many compilers previous to the publication of the ANSI-C++ standard, as well as in the C language, the relational operations did not return a **bool** value **true** or **false**, rather they returned an **int** as result with a value of 0 in order to represent "false" and a value different from 0 (generally 1) to represent "true". For more information, or if your compiler does not support the bool type, consult the section [bool type](#).

Logic operators (!, &&, ||).

Operator **!** is equivalent to boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to invert the value of it, producing **false** if its operand is **true** and **true** if its operand is **false**. It is like saying that it returns the opposite result of evaluating its operand. For example:

!(5 == 5) returns **false** because the expression at its right (5 == 5) would be **true**.

!(6 <= 4) returns **true** because (6 <= 4) would be **false**.

!true returns **false**.

!false returns **true**.

Logic operators **&&** and **||** are used when evaluating two expressions to obtain a single result. They correspond with boolean logic operations *AND* and *OR* respectively. The result of them depends on the relation between its two operands:

First Operand a	Second Operand b	result a && b	result a b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

For example:

((5 == 5) && (3 > 6)) returns **false** (*true && false*).

((5 == 5) || (3 > 6)) returns **true** (*true || false*).

Conditional operator (?).

The conditional operator evaluates an expression and returns a different value according to the evaluated expression, depending on whether it is *true* or *false*. Its format is:

condition ? result1 : result2

if *condition* is **true** the expression will return *result1*, if not it will return *result2*.

7==5 ? 4 : 3 returns **3** since **7** is not equal to **5**.

7==5+2 ? 4 : 3 returns **4** since **7** is equal to **5+2**.

5>3 ? a : b returns **a**, since **5** is greater than **3**.

a>b ? a : b returns the greater one, **a** or **b**.

Bitwise Operators (&, |, ^, ~, <<, >>).

Bitwise operators modify variables considering the bits that represent the values that they store, that means, their binary representation.

op	asm	Description
----	-----	-------------

&	AND	Logical AND
	OR	Logical OR
^	XOR	Logical exclusive OR
~	NOT	Complement to one (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

For more information about binary numbers and bitwise operations, consult [Boolean logic](#).

Explicit type casting operators

Type casting operators allows you to convert a datum of a given type to another. There are several ways to do this in C++, the most popular one, compatible with the C language is to precede the expression to be converted by the new type enclosed between parenthesis ():

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

The previous code converts the float number **3.14** to an integer value (**3**). Here, the type casting operator was **(int)**. Another way to do the same thing in C++ is using the constructor form: preceding the expression to be converted by the type and enclosing the expression between parenthesis:

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

Both ways of type casting are valid in C++. And additionally ANSI-C++ added new type casting operators more specific for object oriented programming ([Section 5.4, Advanced class type-casting](#)).

sizeof()

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

```
a = sizeof (char);
```

This will return **1** to **a** because **char** is a one byte long type.

The value returned by **sizeof** is a constant, so it is always determined before program execution.

Other operators

Later in the tutorial we will see a few more operators, like the ones referring to pointers or the specifics for object-oriented programming. Each one is treated in its respective section.

Priority of operators

When making complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:

```
a = 5 + 7 % 2
```

we may doubt if it really means:

```
a = 5 + (7 % 2) with result 6, or
```

```
a = (5 + 7) % 2 with result 0
```

The correct answer is the first of the two expressions, with a result of **6**. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference we may already know from mathematics) but for all the operators which can appear in C++. From greatest to lowest priority, the priority order is as follows:

Priority	Operator	Description	Associativity
----------	----------	-------------	---------------

1	::	scope	Left
2	() [] -> . sizeof		Left
3	++ --	increment/decrement	Right
	~	Complement to one (bitwise)	
	!	unary NOT	
	& *	Reference and Dereference (pointers)	
	(type)	Type casting	
	+ -	Unary less sign	
4	* / %	arithmetical operations	Left
5	+ -	arithmetical operations	Left
6	<< >>	bit shifting (bitwise)	Left
7	< <= > >=	Relational operators	Left
8	== !=	Relational operators	Left
9	& ^	Bitwise operators	Left
10	&&	Logic operators	Left
11	?:	Conditional	Right
12	= += -= *= /= %= >>= <<= &= ^= =	Assignment	Right
13	,	Comma, Separator	Left

Associativity defines -in the case that there are several operators of the same priority level- which one must be evaluated first, the rightmost one or the leftmost one.

All these precedence levels for operators can be manipulated or become more legible using parenthesis signs (and), as in this example:

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

might be written as:

```
a = 5 + (7 % 2); or
```

```
a = (5 + 7) % 2;
```

according to the operation that we wanted to perform.

So if you want to write a complicated expression and you are not sure of the precedence levels, always include parenthesis. It will probably also be more legible code.

Section 1.4

Communication through console.



The *console* is the basic interface of computers, normally it is the set composed of the keyboard and the screen. The keyboard is generally the standard *input* device and the screen the standard *output* device.

In the *iostream* C++ library, standard *input* and *output* operations for a program are supported by two data streams: `cin` for input and `cout` for output. Additionally, `cerr` and `clog` have also been implemented - these are two output streams specially designed to show error messages. They can be redirected to the standard output or to a log file.

Therefore `cout` (the standard output stream) is normally directed to the screen and `cin` (the standard input stream) is normally assigned to the keyboard.

By handling these two streams you will be able to interact with the user in your programs since you will be able to show messages on the screen and receive his/her input from the keyboard.

Output (`cout`)

The `cout` stream is used in conjunction with the overloaded operator `<<` (a pair of "*less than*" signs).

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

The `<<` operator is known as *insertion operator* since it inserts the data that follows it into the stream that precedes it. In the examples above it inserted the constant string `Output sentence`, the numerical constant `120` and the variable `x` into the output stream `cout`. Notice that the first of the two sentences is enclosed between double quotes (") because it is a string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variables. For example, these two sentences are very different:

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

The *insertion operator* (`<<`) may be used more than once in a same sentence:

```
cout << "Hello, " << "I am " << "a C++ sentence";
```

this last sentence would print the message `Hello, I am a C++ sentence` on the screen. The utility of repeating the insertion operator (`<<`) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

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

If we suppose that variable `age` contains the number `24` and the variable `zipcode` contains `90064` the output of the previous sentence would be:

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

It is important to notice that `cout` does not add a line break after its output unless we explicitly indicate it, therefore, the following sentences:

```
cout << "This is a sentence.";
cout << "This is another sentence.";
```

will be shown followed in screen:

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

even if we have written them in two different calls to `cout`. So, in order to perform a line break on output we must explicitly order it by inserting a new-line character, that in C++ can be written as `\n`:

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

produces the following output:

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

Additionally, to add a new-line, you may also use the `endl` manipulator. For example:

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

would print out:

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

The `endl` manipulator has a special behavior when it is used with buffered streams: they are flushed. But anyway `cout` is unbuffered by default.

You may use either the `\n` escape character or the `endl` manipulator in order to specify a line jump to `cout`. Notice the differences of use shown earlier.

Input (`cin`).

Handling the standard input in C++ is done by applying the overloaded operator of *extraction* (`>>`) on the `cin` stream. This must be followed by the variable that will store the data that is going to be read. For example:

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

declares the variable `age` as an `int` and then waits for an input from `cin` (keyboard) in order to store it in this integer variable.

`cin` can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if you request a single character `cin` will not process the input until the user presses RETURN once the character has been introduced.

You must always consider the *type* of the variable that you are using as a container with `cin` extraction. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.

```
// i/o example  
#include <iostream.h>  
  
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.
```

The user of a program may be one of the reasons that provoke errors even in the simplest programs that use `cin` (like the one we have just seen). Since if you request an integer value and the user introduces a name (which is a string of characters), the result may cause your program to misoperate since it is not what we were expecting from the user. So when you use the data input provided by `cin` you will have to trust that the user of your program will be totally cooperative and that he will not introduce his name when an integer value is

requested. Farther ahead, when we will see how to use strings of characters we will see possible solutions for the errors that can be caused by this type of user input.

You can also use `cin` to request more than one datum input from the user:

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

is equivalent to:

```
cin >> a;
```

```
cin >> b;
```

In both cases the user must give two data, one for variable `a` and another for variable `b` that may be separated by any valid blank separator: a space, a tab character or a newline.

Section 2.1

Control Structures.



A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done to perform our program.

With the introduction of control sequences we are going to have to introduce a new concept: the **block of instructions**. A block of instructions is a group of instructions separated by semicolons (;) but grouped in a block delimited by curly bracket signs: { and }.

Most of the control structures that we will see in this section allow a generic **statement** as a parameter, this refers to either a single instruction or a block of instructions, as we want. If we want the statement to be a single instruction we do not need to enclose it between curly-brackets ({}). If we want the statement to be more than a single instruction we must enclose them between curly brackets ({}), forming a block of instructions.

Conditional structure: *if* and *else*

It is used to execute an instruction or block of instructions only if a condition is fulfilled. Its form is:

```
if (condition) statement
```

where *condition* is the expression that is being evaluated. If this condition is **true**, **statement** is executed. If it is false, *statement* is ignored (not executed) and the program continues on the next instruction after the conditional structure.

For example, the following code fragment prints out **x is 100** only if the value stored in variable `x` is indeed 100:

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

If we want more than a single instruction to be executed in case that *condition* is **true** we can specify a *block of instructions* using curly brackets { }:

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

We can additionally specify what we want that happens if the condition is not fulfilled by using the keyword *else*. Its form used in conjunction with *if* is:

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

For example:

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

prints out on the screen **x is 100** if indeed x is worth 100, but if it is not -and only if not- it prints out **x is not 100**.

The *if + else* structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the present value stored in **x** is positive, negative or none of the previous, that is to say, equal to zero.

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

Remember that in case we want more than a single instruction to be executed, we must group them in a *block of instructions* by using curly brackets { }.

Repetitive structures or loops

Loops have as objective to repeat a *statement* a certain number of times or while a condition is fulfilled.

The *while* loop.

Its format is:

```
while (expression) statement
```

and its function is simply to repeat *statement* while *expression* is true.

For example, we are going to make a program to count down using a *while* loop:

```
// custom countdown using while
#include <iostream.h>
int main ()
{
    int n;
    cout << "Enter the starting
number > ";
    cin >> n;
    while (n>0) {
        cout << n << ", ";
        --n;
    }
    cout << "FIRE!";
    return 0;
}
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

When the program starts the user is prompted to insert a starting number for the countdown. Then the *while* loop begins, if the value entered by the user fulfills the condition $n > 0$ (that n be greater than 0), the block of instructions that follows will execute an indefinite number of times while the condition ($n > 0$) remains true.

All the process in the program above can be interpreted according to the following script: beginning in **main**:

- **1.** User assigns a value to n .
- **2.** The while instruction checks if ($n > 0$). At this point there are two possibilities:
 - **true:** execute *statement* (step 3,)
 - **false:** jump *statement*. The program follows in step 5..
- **3.** Execute *statement*:
`cout << n << ", ";`
`--n;`
(prints out n on screen and decreases n by 1).
- **4.** End of block. Return Automatically to step 2.
- **5.** Continue the program after the block: print out **FIRE!** and end of program.

We must consider that the loop has to end at some point, therefore, within the block of instructions (loop's *statement*) we must provide some method that forces *condition* to become false at some moment, otherwise the loop will continue looping forever. In this case we have included `--n;` that causes the *condition* to become **false** after some loop repetitions: when n becomes 0, that is where our countdown ends.

Of course this is such a simple action for our computer that the whole countdown is performed instantly without practical delay between numbers.

The *do-while* loop.

Format:

`do statement while (condition);`

Its functionality is exactly the same as the *while* loop except that *condition* in the *do-while* is evaluated after the execution of *statement* instead of before, granting at least one execution of *statement* even if *condition* is never fulfilled. For example, the following program echoes any number you enter until you enter 0.

```
// number echoer
#include <iostream.h>
int main ()
{
    unsigned long n;
    do {
        cout << "Enter number (0 to
end): ";
        cin >> n;
        cout << "You entered: " << n
<< "\n";
    } while (n != 0);
    return 0;
}
```

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

The *do-while* loop is usually used when the condition that has to determine its end is determined within the loop statement, like in the previous case, where the user input

within the block of instructions is what determines the end of the loop. If you never enter the 0 value in the previous example the loop will never end.

The *for* loop.

Its format is:

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

and its main function is to repeat *statement* while *condition* remains true, like the *while* loop. But in addition, **for** provides places to specify an *initialization* instruction and an *increase* instruction. So this loop is specially designed to perform a repetitive action with a counter.

It works the following way:

1, *initialization* is executed. Generally it is an initial value setting for a counter variable. This is executed only once.

2, *condition* is checked, if it is **true** the loop continues, otherwise the loop finishes and *statement* is skipped.

3, *statement* is executed. As usual, it can be either a single instruction or a block of instructions enclosed within curly brackets { }.

4, finally, whatever is specified in the *increase* field is executed and the loop gets back to step 2.

Here is an example of countdown using a *for* loop.

```
// countdown using a for loop
#include <iostream.h>
int main ()
{
    for (int n=10; n>0; n--) {
        cout << n << ", ";
    }
    cout << "FIRE!";
    return 0;
}
```

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

The *initialization* and *increase* fields are optional. They can be avoided but not the semicolon signs among them. For example we could write: **for** (;*n*<10;) if we want to specify no *initialization* and no *increase*; or **for** (;*n*<10;*n*++) if we want to include an *increase* field but not an *initialization*.

Optionally, using the comma operator (,) we can specify more than one instruction in any of the fields included in a **for** loop, like in *initialization*, for example. The comma operator (,) is an instruction separator, it serves to separate more than one instruction where only one instruction is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

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

This loop will execute 50 times if neither *n* nor *i* are modified within the loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
```

Diagram illustrating the components of the **for** loop:

- Initialization**: *n=0, i=100*
- Condition**: *n!=i*
- Increase**: *n++, i--*

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

Bifurcation of control and jumps.

The *break* instruction.

Using *break* we can leave 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, we are going to stop the count down before it naturally finishes (an engine failure maybe):

```
// break loop example
#include <iostream.h>
int main ()
{
    int n;
    for (n=10; n>0; n--) {
        cout << n << ", ";
        if (n==3)
        {
            cout << "countdown
aborted!";
            break;
        }
    }
    return 0;
}
```

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

The *continue* instruction.

The *continue* instruction causes the program to skip the rest of the loop in the present iteration as if the end of the *statement* block would have been reached, causing it to jump to the following iteration. For example, we are going to skip the number 5 in our countdown:

```
// break loop example
#include <iostream.h>
int main ()
{
    for (int n=10; n>0; n--) {
        if (n==5) continue;
        cout << n << ", ";
    }
    cout << "FIRE!";
    return 0;
}
```

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

The *goto* instruction.

It allows making an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type of nesting limitation.

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

This instruction does not have a concrete utility in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using *goto*:

```
// goto loop example
#include <iostream.h>
int main ()
{
    int n=10;
    loop:
    cout << n << ", ";
    n--;
    if (n>0) goto loop;
    cout << "FIRE!";
    return 0;
}
```

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

The *exit* function.

exit is a function defined in [`cstdlib`](#) (`stdlib.h`) library.

The purpose of *exit* is to terminate the running program with an specific exit code. Its prototype is:

```
void exit (int exit code);
```

The *exit code* is used by some operating systems and may be used by calling programs. By convention, an *exit code* of 0 means that the program finished normally and any other value means an error happened.

The selective Structure: *switch*.

The syntax of the *switch* instruction is a bit peculiar. Its objective is to check several possible constant values for an expression, something similar to what we did at the beginning of this section with the linking of several *if* and *else if* sentences. Its form is the following:

```
switch (expression) {
    case constant1:
        block of instructions 1
        break;
    case constant2:
        block of instructions 2
        break;
    .
    .
    .
    default:
        default block of instructions
}
```

It works in the following way: **switch** evaluates *expression* and checks if it is equivalent to *constant1*, if it is, it executes *block of instructions 1* until it finds the **break** keyword, then the program will jump to the end of the *switch* selective structure.

If *expression* was not equal to *constant1* it will check if *expression* is equivalent to *constant2*. If it is, it will execute *block of instructions 2* until it finds the **break** keyword.

Finally, if the value of *expression* has not matched any of the previously specified constants (you may specify as many **case** sentences as values you want to check), the program will execute the instructions included in the **default:** section, if this one exists, since it is optional.

Both of the following code fragments are equivalent:

switch example

```
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x
unknown";
}
```

if-else equivalent

```
if (x == 1) {
  cout << "x is 1";
}
else if (x == 2) {
  cout << "x is 2";
}
else {
  cout << "value of x unknown";
}
```

I have commented before that the syntax of the **switch** instruction is a bit peculiar. Notice the inclusion of the **break** instructions at the end of each block. This is necessary because if, for example, we did not include it after *block of instructions 1* the program would not jump to the end of the switch selective block (}) and it would continue executing the rest of the blocks of instructions until the first appearance of the **break** instruction or the end of the switch selective block. This makes it unnecessary to include curly brackets { } in each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression evaluated. 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** can only be used to compare an expression with different constants. Therefore we cannot put variables (**case (n*2):**) or ranges (**case (1..3):**) because they are not valid constants.

If you need to check ranges or values that are not constants use a concatenation of **if** and **else if** sentences.

Section 2.2

Functions (I).



Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming in C++ can offer us.

A function is a block of instructions that is executed when it is called from some other point of the program. The following is its format:

type name (argument1, argument2, ...) statement

where:

- **type** is the type of data returned by the function.

- **name** is the name by which it will be possible to call the function.
- **arguments** (as many as wanted can be specified). Each argument consists of a type of data followed by its identifier, like in a variable declaration (for example, `int x`) and which acts within the function like any other variable. They allow passing parameters to the function when it is called. The different parameters are separated by commas.
- **statement** is the function's body. It can be a single instruction or a block of instructions. In the latter case it must be delimited by curly brackets `{}`.

Here you have the first function example:

```
// function example
#include <iostream.h>

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;
    return 0;
}
```

The result is 8

In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution with the **main** function. So we will begin there.

We can see how the **main** function begins by declaring the variable **z** of type **int**. Right after that we see a call to **addition** function. If we pay attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself in the code lines above:

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

The parameters have a clear correspondence. Within the **main** function we called to **addition** passing two values: 5 and 3 that correspond to the **int a** and **int b** parameters declared for the function **addition**.

At the moment at which the function is called from **main**, control is lost by **main** and passed to function **addition**. The value of both parameters passed in the call (5 and 3) are copied to the local variables **int a** and **int b** within the function.

Function **addition** declares a new variable (**int r**), and by means of the expression **r=a+b**, it assigns to **r** the result of **a** plus **b**. Because the passed parameters for **a** and **b** are 5 and 3 respectively, the result is 8.

The following line of code:

```
return (r);
```

finalizes function `addition`, and returns the control back to the function that called it (`main`) following the program from the same point at which it was interrupted by the call to `addition`. But additionally, `return` was called with the content of variable `r` (`return (r);`), which at that moment was 8, so this value is said to be **returned** by the function.

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

↓ 8

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

The value returned by a function is the value given to the function when it is evaluated. Therefore, `z` will store the value returned by `addition (5, 3)`, that is 8. To explain it another way, you can imagine that the call to a function (`addition (5,3)`) is literally replaced by the value it returns (8).

The following line of code in `main` is:

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

that, as you may already suppose, produces the printing of the result on the screen.

```
#include <iostream.h>

int Integer;
char aCharacter;
char string [20];
unsigned int NumberOfSons;

main ()
{
    unsigned short Age;
    float ANumber, AnotherOne;

    cout << "Enter your age:"
    cin >> Age;
    ...
}
```

Scope of variables [re]

You must consider that the scope of variables declared within a function or any other block of instructions is only their own function or their own block of instructions and cannot be used outside of them. For example, in the previous example it had been impossible to use the variables `a`, `b` or `r` directly in function `main` since they were local variables to function `addition`. Also, it had been impossible to use the variable `z` directly within function `addition`, since this was a local variable to the function `main`.

Therefore, the scope of local variables is limited to the same nesting level in which they are declared. Nevertheless you can also declare global variables that are visible from any point of the code, inside and outside any function. In order to declare global variables you must do it outside any function or block of instructions, that means, directly in the body of the program.

Global variables

Local variables

Instructions

And here is another example about functions:

```
// function example
#include <iostream.h>

int subtraction (int a, int b)
{
```

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

```

    int r;
    r=a-b;
    return (r);
}

int main ()
{
    int x=5, y=3, z;
    z = subtraction (7,2);
    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';
    return 0;
}

```

In this case we have created the function **subtraction**. The only thing that this function does is to subtract both passed parameters and to return the result.

Nevertheless, if we examine the function **main** we will see that we have made several calls to function **subtraction**. We have used some different calling methods so that you see other ways or moments when a function can be called.

In order to understand well these examples you must consider once again that a call to a function could be perfectly replaced by its return value. For example the first case (that you should already know because it is the same pattern that we have used in previous examples):

```

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

```

If we replace the function call by its result (that is 5), we would have:

```

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

```

As well as

```

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

```

has the same result as the previous call, but in this case we made the call to **subtraction** directly as a parameter for **cout**. Simply imagine that we had written:

```

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

```

since 5 is the result of **subtraction (7,2)**.

In the case of

```

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

```

The only new thing that we introduced is that the parameters of **subtraction** are variables instead of constants. That is perfectly valid. In this case the values passed to the function **subtraction** are the values of **x** and **y**, that are 5 and 3 respectively, giving 2 as result.

The fourth case is more of the same. Simply note that instead of:

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

we could have put:

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

with exactly the same result. Notice that the semicolon sign (;) goes at the end of the whole expression. It does not necessarily have to go right after the function call. The explanation might be once again that you imagine that a function can be replaced by its result:

```
z = 4 + 2;
```

```
z = 2 + 4;
```

Functions with no types. The use of *void*.

If you remember the syntax of a function declaration:

```
type name ( argument1, argument2 ...) statement
```

you will see that it is obligatory that this declaration begins with a **type**, that is the type of the data that will be returned by the function with the **return** instruction. But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value, moreover, we do not need it to receive any parameters. For these cases, the **void** type was devised in the C language. Take a look at:

```
// void function example
#include <iostream.h>

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

int main ()
{
    dummyfunction ();
    return 0;
}
```

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

Although in C++ it is not necessary to specify **void**, its use is considered suitable to signify that it is a function without parameters or arguments and not something else.

What you must always be aware of is that the format for calling a function includes specifying its name and enclosing the arguments between parenthesis. The non-existence of arguments does not exempt us from the obligation to use parenthesis. For that reason the call to **dummyfunction** is

```
dummyfunction ();
```

This clearly indicates that it is a call to a function and not the name of a variable or anything else.

Section 2.3

Functions (II).




Arguments passed *by value* and *by reference*.

Until now, in all the functions we have seen, the parameters passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were values but never the specified variables themselves. For example, suppose that we called our first function **addition** using the following code :

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

What we did in this case was to call function **addition** passing the values of **x** and **y**, that means 5 and 3 respectively, not the variables themselves.

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



This way, when function **addition** is being called the value of its variables **a** and **b** become 5 and 3 respectively, but any modification of **a** or **b** within the function **addition** will not affect the values of **x** and **y** outside it, because variables **x** and **y** were not passed themselves to the function, only their values.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we have to use *arguments passed by reference*, as in the function **duplicate** of the following example:

```
// passing parameters by reference  
#include <iostream.h>  
  
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

The first thing that should call your attention is that in the declaration of **duplicate** the type of each argument was followed by an *ampersand* sign (&), that serves to specify that the variable has to be passed *by reference* instead of *by value*, as usual.

When passing a variable *by reference* we are passing the variable itself and any modification that we do to that parameter within the function will have effect in the passed variable outside it.


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

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

To express it another way, we have associated **a**, **b** and **c** with the parameters used when calling the function (**x**, **y** and **z**) and any change that we do on **a** within the function will affect the value of **x** outside. Any change that we do on **b** will affect **y**, and the same with **c** and **z**.

That is why our program's output, that shows the values stored in **x**, **y** and **z** after the call to **duplicate**, shows the values of the three variables of **main** doubled.

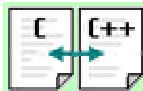
If when declaring the following function:

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

we had declared it thus:

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

that is, without the *ampersand* (&) signs, we would have not passed the variables *by reference*, but their values, and therefore, the output on screen for our program would have been the values of **x**, **y** and **z** without having been modified.



This type of declaration "*by reference*" using the *ampersand* (&) sign is exclusive of C++. In C language we had to use pointers to do something equivalent.

Passing by reference is an effective way to allow a function to return more than one single value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```
// more than one returning value
#include <iostream.h>

void prevnext (int x, int& prev,
int& next)
{
    prev = x-1;
    next = x+1;
}

int main ()
{
    int x=100, y, z;
    prevnext (x, y, z);
    cout << "Previous=" << y << " ,
Next=" << z;
    return 0;
}
```

```
Previous=99, Next=101
```

Default values in arguments.

When declaring a function we can specify a default value for each parameter. This value will be used if that parameter is left blank when calling to the function. To do that we simply have to assign a value to the arguments in the function declaration. If a value for that parameter is

not passed when the function is called, the default value is used, but if a value is specified this default value is stepped on and the passed value is used. For example:

```
// default values in functions
#include <iostream.h>

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

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

```
6
5
```

As we can see in the body of the program there are two calls to the function **divide**. In the first one:

divide (12)

we have only specified one argument, but the function **divide** allows up to two. So the function **divide** has assumed that the second parameter is **2** since that is what we have specified to happen if this parameter is lacking (notice the function declaration, which finishes with **int b=2**). Therefore the result of this function call is **6** ($12/2$).

In the second call:

divide (20,4)

there are two parameters, so the default assignation (**int b=2**) is stepped on by the passed parameter, that is **4**, making the result equal to **5** ($20/4$).

Overloaded functions.

Two different functions can have the same name if the prototype of their arguments are different, that means that you can give the same name to more than one function if they have either a different number of arguments or different types in their arguments. For example,

```
// overloaded function
#include <iostream.h>

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

float divide (float a, float b)
{
    return (a/b);
}

int main ()
```

```
2
2.5
```

```
{
    int x=5,y=2;
    float n=5.0,m=2.0;
    cout << divide (x,y);
    cout << "\n";
    cout << divide (n,m);
    cout << "\n";
    return 0;
}
```

In this case we have defined two functions with the same name, but one of them accepts two arguments of type `int` and the other accepts them of type `float`. The compiler knows which one to call in each case by examining the types when the function is called. If it is called with two `ints` as arguments it calls to the function that has two `int` arguments in the prototype and if it is called with two `floats` it will call to the one which has two `floats` in its prototype.

For simplicity I have included the same code within both functions, but this is not compulsory. You can make two function with the same name but with completely different behaviors.

***inline* functions.**

The *inline* directive can be included before a function declaration to specify that the function must be compiled as code at the same point where it is called. This is equivalent to declaring a macro. Its advantage is only appreciated in very short functions, in which the resulting code from compiling the program may be faster if the overhead of calling a function (stacking of arguments) is avoided.

The format for its declaration is:

```
inline type name ( arguments ... ) { instructions ... }
```

and the call is just like the call to any other function. It is not necessary to include the *inline* keyword before each call, only in the declaration.

Recursivity.

Recursivity is the property that functions have to be called by themselves. It is useful for tasks such as some sorting methods or to calculate the factorial of a number. For example, to obtain the factorial of a number (n) its mathematical formula is:

$$n! = n * (n-1) * (n-2) * (n-3) \dots * 1$$

more concretely, 5! (factorial of 5) would be:

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

and a recursive function to do that could be this:

```
// factorial calculator
#include <iostream.h>

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

```
Type a number: 9
!9 = 362880
```

```
int main ()
{
    long l;
    cout << "Type a number: ";
    cin >> l;
    cout << "!" << l << " = " <<
factorial (l);
    return 0;
}
```

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

This function has a limitation because of the *data type* used in its design (`long`) for more simplicity. In a standard system, the type `long` would not allow storing factorials greater than 12!.

Prototyping functions.

Until now, we have defined the all of the functions before the first appearance of calls to them, that generally was in **main**, leaving the function **main** for the end. If you try to repeat some of the examples of functions described so far, but placing the function **main** before any other function that is called from within it, you will most likely obtain an error. The reason is that to be able to call a function it must have been declared previously (it must be known), like we have done in all our examples.

But there is an alternative way to avoid writing all the code of all functions before they can be used in **main** or in another function. It is by *prototyping functions*. This consists in making a previous shorter, but quite significant, declaration of the complete definition so that the compiler can know the arguments and the return type needed.

Its form is:

```
type name ( argument_type1, argument_type2, ... );
```

It is identical to the header of a function definition, except:

- It does not include a *statement* for the function. That means that it does not include the body with all the instructions that are usually enclose within curly brackets { }.
- It ends with a semicolon sign (;).
- In the argument enumeration it is enough to put the type of each argument. The inclusion of a name for each argument as in the definition of a standard function is optional, although recommended.

For example:

```
// prototyping
#include <iostream.h>

void odd (int a);
```

```
Type a number (0 to exit): 9
Number is odd.
Type a number (0 to exit): 6
Number is even.
```

```

void even (int a);

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

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

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

```

```

Type a number (0 to exit): 1030
Number is even.
Type a number (0 to exit): 0
Number is even.

```

This example is indeed not an example of effectiveness, I am sure that at this point you can already make a program with the same result using only half of the code lines. But this example illustrates how prototyping works. Moreover, in this concrete case the prototyping of - at least- one of the two functions is necessary.

The first things that we see are the prototypes of functions **odd** and **even**:

```

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

```

that allows these functions to be used before they are completely defined, for example, in **main**, which now is located in a more logical place: the beginning of the program's code.

Nevertheless, the specific reason why this program needs at least one of the functions prototyped is because in **odd** there is a call to **even** and in **even** there is a call to **odd**. If none of the two functions had been previously declared, an error would have happened, since either **odd** would not be visible from **even** (because it has not still been declared), or **even** would not be visible from **odd**.

Many programmers recommend that all functions be prototyped. It is also my recommendation, mainly in case that there are many functions or in case that they are very long. Having the prototype of all the functions in the same place can spare us some time when determining how to call it or even ease the creation of a header file.

Section 3.1

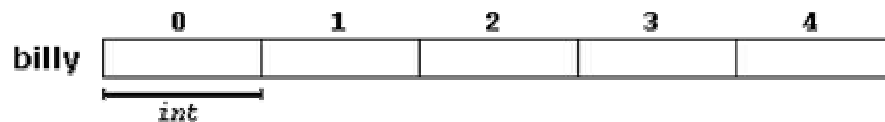
Arrays



Arrays are a series of elements (variables) of the same type placed consecutively in memory that can be individually referenced by adding an index to a unique name.

That means that, for example, we can store 5 values of type `int` without having to declare 5 different variables each with a different identifier. Instead, using an *array* we can store 5 different values of the same type, `int` for example, with a unique identifier.

For example, an array to contain 5 integer values of type `int` called *billy* could be represented this way:



where each blank panel represents an *element* of the array, that in this case are integer values of type `int`. These are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length .

Like any other variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

```
type name [elements];
```

where *type* is a valid object type (`int`, `float`...), *name* is a valid variable *identifier* and the *elements* field, that is enclosed within brackets `[]`, specifies how many of these elements the array contains.

Therefore, to declare *billy* as shown above it is as simple as the following sentence:

```
int billy [5];
```

NOTE: The *elements* field within brackets `[]` when declaring an array must be a constant value, since arrays are blocks of static memory of a given size and the compiler must be able to determine exactly how much memory it must assign to the array before any instruction is considered.

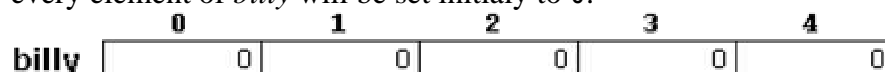
Initializing arrays.

When declaring an array of local scope (within a function), if we do not specify otherwise, it will not be initialized, so its content is undetermined until we store some values in it.

If we declare a global array (outside any function) its content will be initialized with all its elements filled with zeros. Thus, if in the global scope we declare:

```
int billy [5];
```

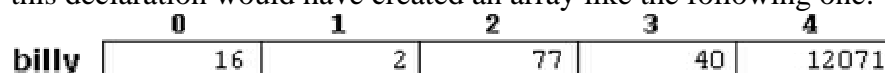
every element of *billy* will be set initially to 0:



But additionally, when we declare an Array, we have the possibility to assign initial values to each one of its elements using curly brackets `{ }`. For example:

```
int billy [5] = { 16, 2, 77, 40, 12071 };
```

this declaration would have created an array like the following one:



The number of elements in the array that we initialized within curly brackets { } must match the length in elements that we declared for the array enclosed within square brackets []. For example, in the example of the *billy* array we have declared that it had 5 elements and in the list of initial values within curly brackets { } we have set 5 different values, one for each element.

Because this can be considered useless repetition, C++ includes the possibility of leaving the brackets empty [] and the size of the Array will be defined by the number of values included between curly brackets { }:

```
int billy [] = { 16, 2, 77, 40, 12071 };
```

Access to the values of an Array.

In any point of the program in which the array is visible we can access individually anyone of its values for reading or modifying as if it was a normal variable. The format is the following:
name[*index*]

Following the previous examples in which *billy* had 5 elements and each of those elements was of type *int*, the name which we can use to refer to each element is the following:

	<code>billy[0]</code>	<code>billy[1]</code>	<code>billy[2]</code>	<code>billy[3]</code>	<code>billy[4]</code>
billy					

For example, to store the value 75 in the third *element* of *billy* a suitable sentence would be:
`billy[2] = 75;`

and, for example, to pass the value of the third element of *billy* to the variable *a*, we could write:

```
a = billy[2];
```

Therefore, for all purposes, the expression `billy[2]` is like any other variable of type *int*.

Notice that the third element of *billy* is specified `billy[2]`, since first is `billy[0]`, the second is `billy[1]`, and therefore, third is `billy[2]`. By this same reason, its last element is `billy[4]`. Since if we wrote `billy[5]`, we would be acceding to the sixth element of *billy* and therefore exceeding the size of the array.

In C++ it is perfectly valid to exceed the valid range of indices for an Array, which can create problems since they do not cause compilation errors but they can cause unexpected results or serious errors during execution. The reason why this is allowed will be seen farther ahead when we begin to use pointers.

At this point it is important to be able to clearly distinguish between the two uses that brackets [] have related to arrays. They perform two differt tasks: one is to set the size of arrays when declaring them; and second is to specify indices for a concrete array element when referring to it. We must simply take care not to confuse these two possible uses of brackets [] with arrays:

```
int billy[5];           // declaration of a new Array (begins with a type
// name)
billy[2] = 75;         // access to an element of the Array.
```

Other valid operations with arrays:

```
billy[0] = a;
billy[a] = 75;
```

```
b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
```

```
// arrays example
#include <iostream.h>

int billy [] = {16, 2, 77, 40,
12071};
int n, result=0;

int main ()
{
    for ( n=0 ; n<5 ; n++ )
    {
        result += billy[n];
    }
    cout << result;
    return 0;
}
```

12206

Multidimensional Arrays

Multidimensional arrays can be described as arrays of arrays. For example, a bidimensional array can be imagined as a bidimensional table of a uniform concrete data *type*.

		0	1	2	3	4
jimmy	0					
	1					
	2					

jimmy represents a bidimensional array of 3 per 5 values of type `int`. The way to declare this array would be:

```
int jimmy [3][5];
```

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

```
jimmy[1][3]
```

		0	1	2	3	4
jimmy	0					
	1					
	2					

↓
jimmy[1][3]

(remember that array indices always begin by 0).

Multidimensional arrays are not limited to two indices (two dimensions). They can contain as many indices as needed, although it is rare to have to represent more than 3 dimensions. Just consider the amount of memory that an array with many indices may need. For example:

```
char century [100][365][24][60][60];
```

assigns a **char** for each second contained in a century, that is more than 3 billion **chars**! This would consume about 3000 *megabytes* of RAM memory if we could declare it.

Multidimensional arrays are nothing more than an abstraction, since we can obtain the same results with a simple array just by putting a factor between its indices:

`int jimmy [3][5];` is equivalent to

`int jimmy [15];` ($3 * 5 = 15$)

with the only difference that the compiler remembers for us the depth of each imaginary dimension. Serve as example these two pieces of code, with exactly the same result, one using bidimensional arrays and the other using only simple arrays:

```
// multidimensional array
#include <iostream.h>

#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main ()
{
    for (n=0;n<HEIGHT;n++)
        for (m=0;m<WIDTH;m++)
        {
            jimmy[n][m]=(n+1)*(m+1);
        }
    return 0;
}
```

```
// pseudo-multidimensional array
#include <iostream.h>

#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main ()
{
    for (n=0;n<HEIGHT;n++)
        for (m=0;m<WIDTH;m++)
        {
            jimmy[n * WIDTH +
m]=(n+1)*(m+1);
        }
    return 0;
}
```

none of the programs above produce any output on the screen, but both assign values to the memory block called `jimmy` in the following way:

jimmy {		0	1	2	3	4
	0	1	2	3	4	5
	1	2	4	6	8	10
	2	3	6	9	12	15

We have used defined constants (`#define`) to simplify possible future modifications of the program, for example, in case that we decided to enlarge the array to a height of **4** instead of **3** it could be done by changing the line:

```
#define HEIGHT 3
```

to

```
#define HEIGHT 4
```

with no need to make any other modifications to the program.

Arrays as parameters

At some moment we may need to pass an array to a function as a parameter. In C++ is not possible to pass by value a complete block of memory as a parameter to a function, even if it is ordered as an array, but it is allowed to pass its address. This has almost the same practical effect and it is a much faster and more efficient operation.

In order to admit arrays as parameters the only thing that we must do when declaring the function is to specify in the argument the base **type** for the array, an identifier and a pair of void brackets `[]`. For example, the following function:

```
void procedure (int arg[])
```

admits a parameter of type "Array of `int`" called `arg`. In order to pass to this function an array declared as:

```
int myarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

Here you have a complete example:

```
// arrays as parameters
#include <iostream.h>

void printarray (int arg[], int
length) {
    for (int n=0; n<length; n++)
        cout << arg[n] << " ";
    cout << "\n";
}

int main ()
{
    int firstarray[] = {5, 10, 15};
    int secondarray[] = {2, 4, 6, 8,
10};
    printarray (firstarray,3);
    printarray (secondarray,5);
    return 0;
}
```

```
5 10 15
2 4 6 8 10
```

As you can see, the first argument (`int arg[]`) admits any array of type `int`, whatever its length is. For that reason we have included a second parameter that tells the function the length of each array that we pass to it as the first parameter. This allows the `for` loop that prints out the array to know the range to check in the passed array.

In a function declaration is also possible to include multidimensional arrays. The format for a tridimensional array is:

```
base_type[][depth][depth]
```

for example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray[][3][4])
```

notice that the first brackets `[]` are void and the following ones are not. This must always be thus because the compiler must be able to determine within the function which is the depth of each additional dimension.

Arrays, both simple or multidimensional, passed as function parameters are a quite common source of errors for less experienced programmers. I recommend the reading of chapter **3.3, Pointers** for a better understanding of how *arrays* operate.

Section 3.2

Strings of Characters.



In all programs seen until now, we have used only numerical variables, used to express numbers exclusively. But in addition to numerical variables there also exist strings of

characters, that allow us to represent successions of characters, like words, sentences, names, texts, et cetera. Until now we have only used them as constants, but we have never considered variables able to contain them.

In C++ there is no specific *elemental* variable type to store strings of characters. In order to fulfill this feature we can use arrays of type **char**, which are successions of **char** elements. Remember that this data type (**char**) is the one used to store a single character, for that reason arrays of them are generally used to make strings of single characters.

For example, the following array (or string of characters):

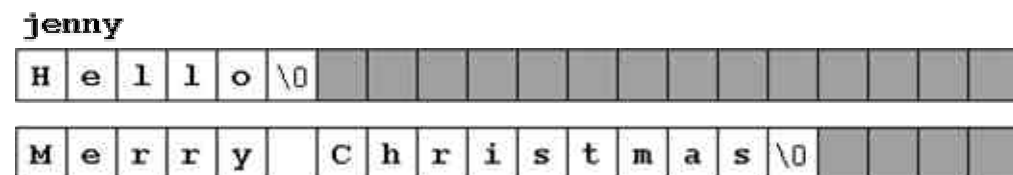
```
char jenny [20];
```

can store a string up to 20 characters long. You may imagine it thus:



This maximum size of 20 characters is not required to always be fully used. For example, **jenny** could store at some moment in a program either the string of characters "Hello" or the string "Merry christmas". Therefore, since the array of characters can store shorter strings than its total length, a convention has been reached to end the valid content of a string with a null character, whose constant can be written 0 or '\0'.

We could represent **jenny** (an array of 20 elements of type **char**) storing the strings of characters "Hello" and "Merry Christmas" in the following way:



Notice how after the valid content a null character ('\0') it is included in order to indicate the end of the string. The panels in gray color represent indeterminate values.

Initialization of strings

Because strings of characters are ordinary arrays they fulfill all their same rules. For example, if we want to initialize a string of characters with predetermined values we can do it just like any other array:

```
char mystring[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

In this case we would have declared a string of characters (array) of 6 elements of type **char** initialized with the characters that compose **Hello** plus a null character '\0'.

Nevertheless, strings of characters have an additional way to initialize their values: using constant strings.

In the expressions we have used in examples in previous chapters constants that represented entire strings of characters have already appeared several times. These are specified enclosed between double quotes ("), for example:

```
"the result is: "
```

is a constant string that we have probably used on some occasion.

Unlike single quotes (') which specify single character constants, double quotes (") are constants that specify a succession of characters. Strings enclosed between double quotes always have a null character ('\0') automatically appended at the end.

Therefore we could initialize the string **mystring** with values by either of these two ways:

```
char mystring [] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char mystring [] = "Hello";
```

In both cases the array or string of characters **mystring** is declared with a size of 6 characters (elements of type **char**): the 5 characters that compose **Hello** plus a final null character ('\0') which specifies the end of the string and that, in the second case, when using double quotes (") it is automatically appended.

Before going further, notice that the assignation of multiple constants like double-quoted constants (") to arrays are only valid when initializing the array, that is, at the moment when declared. Expressions within the code like:

```
mystring = "Hello";
mystring[] = "Hello";
are not valid for arrays, like neither would be:
mystring = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

So remember: We can "assign" a multiple constant to an Array only at the moment of initializing it. The reason will be more comprehensible when you know a bit more about pointers, since then it will be clarified that an array is simply a *constant pointer* pointing to an allocated block of memory. And because of this constantness, the array itself can not be assigned any value, but we can assign values to each of the elements of the array.

The moment of initializing an Array it is a special case, since it is not an assignation, although the same equal sign (=) is used. Anyway, always have the rule previously underlined present.

Assigning values to strings

Since the *lvalue* of an assignation can only be an element of an array and not the entire array, it would be valid to assign a string of characters to an array of **char** using a method like this:

```
mystring[0] = 'H';
mystring[1] = 'e';
mystring[2] = 'l';
mystring[3] = 'l';
mystring[4] = 'o';
mystring[5] = '\0';
```

But as you may think, this does not seem to be a very practical method. Generally for assigning values to an array, and more specifically to a string of characters, a series of functions like **strcpy** are used. **strcpy** (**string copy**) is defined in the **cstring** (**string.h**) library and can be called the following way:

```
strcpy (string1, string2);
```

This does copy the content of *string2* into *string1*. *string2* can be either an array, a pointer, or a constant string, so the following line would be a valid way to assign the constant string **"Hello"** to **mystring**:

```
strcpy (mystring, "Hello");
```

For example:

```
// setting value to string
#include <iostream.h>
#include <string.h>

int main ()
{
    char szMyName [20];
    strcpy (szMyName,"J. Soulie");
    cout << szMyName;
    return 0;
}
```

J. Soulie

Notice that we needed to include **<string.h>** header in order to be able to use function **strcpy**.

Although we can always write a simple function like the following **setstring** with the same operation as cstring's **strcpy**:

```
// setting value to string
#include <iostream.h>

void setstring (char szOut [], char
szIn [])
{
    int n=0;
    do {
        szOut[n] = szIn[n];
    } while (szIn[n++] != '\0');
}

int main ()
{
    char szMyName [20];
    setstring (szMyName,"J. Soulie");
    cout << szMyName;
    return 0;
}
```

J. Soulie

Another frequently used method to assign values to an array is by directly using the input stream (**cin**). In this case the value of the string is assigned by the user during program execution.

When **cin** is used with strings of characters it is usually used with its **getline** method, that can be called following this prototype:

```
cin.getline ( char buffer[], int length, char delimiter = ' \n');
```

where **buffer** is the address of where to store the input (like an array, for example), **length** is the maximum length of the buffer (the size of the array) and **delimiter** is the character used to determine the end of the user input, which by default - if we do not include that parameter - will be the newline character (**'\n'**).

The following example repeats whatever you type on your keyboard. It is quite simple but serves as an example of how you can use **cin.getline** with strings:

```
// cin with strings
#include <iostream.h>

int main ()
{
    char mybuffer [100];
    cout << "What's your name? ";
    cin.getline (mybuffer,100);
    cout << "Hello " << mybuffer <<
    ".\n";
    cout << "Which is your favourite
team? ";
    cin.getline (mybuffer,100);
    cout << "I like " << mybuffer << "
too.\n";
    return 0;
}
```

```
What's your name? Juan
Hello Juan.
Which is your favourite team? Inter
Milan
I like Inter Milan too.
```

Notice how in both calls to `cin.getline` we used the same string identifier (`mybuffer`). What the program does in the second call is simply step on the previous content of `buffer` with the new one that is introduced.

If you remember the section about communication through the console, you will remember that we used the extraction operator (`>>`) to receive data directly from the standard input. This method can also be used instead of `cin.getline` with strings of characters. For example, in our program, when we requested an input from the user we could have written:

```
cin >> mybuffer;
```

this would work, but this method has the following limitations that `cin.getline` has not:

- It can only receive single words (no complete sentences) since this method uses as a delimiter any occurrence of a blank character, including spaces, tabulators, newlines and carriage returns.
- It is not allowed to specify a size for the buffer. That makes your program unstable in case the user input is longer than the array that will host it.

For these reasons it is recommended that whenever you require strings of characters coming from `cin` you use `cin.getline` instead of `cin >>`.

Converting strings to other types

Due to that a string may contain representations of other data types like numbers, it might be useful to translate that content to a variable of a numeric type. For example, a string may contain `"1977"`, but this is a sequence of 5 chars not so easily convertible to a single integer data type. The `cstdlib` (`stdlib.h`) library provides three useful functions for this purpose:

- **atoi:** converts string to `int` type.
- **atol:** converts string to `long` type.
- **atof:** converts string to `float` type.

All of these functions admit one parameter and return a value of the requested type (int, long or float). These functions combined with `getline` method of `cin` are a more reliable way to get the user input when requesting a number than the classic `cin>>` method:

```
// cin and atof functions
#include <iostream.h>
#include <stdlib.h>

int main ()
{
    char mybuffer [100];
    float price;
    int quantity;
    cout << "Enter price: ";
    cin.getline (mybuffer,100);
    price = atof (mybuffer);
    cout << "Enter quantity: ";
    cin.getline (mybuffer,100);
    quantity = atoi (mybuffer);
    cout << "Total price: " <<
price*quantity;
    return 0;
}
```

```
Enter price: 2.75
Enter quantity: 21
Total price: 57.75
```

Functions to manipulate strings

The `cstring` library (`string.h`) defines many functions to perform manipulation operations with C-like strings (like already explained `strcpy`). Here you have a brief look at the most usual:

strcat: `char* strcat (char* dest, const char* src);`

Appends *src* string at the end of *dest* string. Returns *dest*.

strcmp: `int strcmp (const char* string1, const char* string2);`

Compares strings *string1* and *string2*. Returns 0 if both strings are equal.

strcpy: `char* strcpy (char* dest, const char* src);`

Copies the content of *src* to *dest*. Returns *dest*.

strlen: `size_t strlen (const char* string);`

Returns the length of *string*.

NOTE: `char*` is the same as `char[]`

Check the [C++ Reference](#) for extended information about these and other functions of this library.

Section 3.3

Pointers



We have already seen how variables are memory cells that we can access by an identifier. But these variables are stored in concrete places of the computer memory. For our programs, the computer memory is only a succession of 1 *byte* cells (the minimum size for a datum), each one with a unique address.

A good simile for the computer memory can be a street in a city. On a street all houses are consecutively numbered with an unique identifier so if we talk about 27th of Sesame Street we will be able to find that place without trouble, since there must be only one house with that number and, in addition, we know that the house will be between houses 26 and 28.

In the same way in which houses in a street are numbered, the operating system organizes the memory with unique and consecutive numbers, so if we talk about location 1776 in the memory, we know that there is only one location with that address and also that is between addresses 1775 and 1777.

Address (dereference) operator (&).

At the moment in which we declare a variable it must be stored in a concrete location in this succession of cells (the memory). We generally do not decide where the variable is to be placed - fortunately that is something automatically done by the compiler and the operating system at runtime, but once the operating system has assigned an address there are some cases in which we may be interested in knowing where the variable is stored.

This can be done by preceding the variable identifier by an *ampersand sign* (&), which literally means "*address of*". For example:

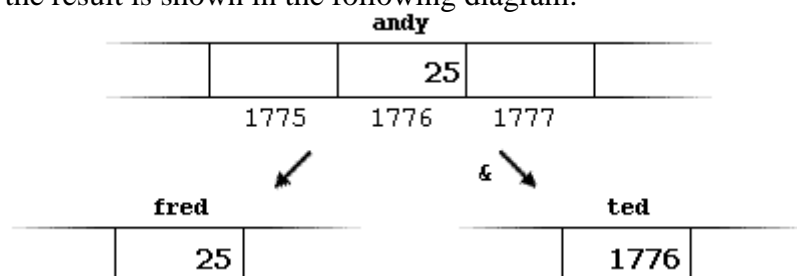
```
ted = &andy;
```

would assign to variable **ted** the address of variable **andy**, since when preceding the name of the variable **andy** with the *ampersand* (&) character we are no longer talking about the content of the variable, but about its address in memory.

We are going to suppose that **andy** has been placed in the memory address **1776** and that we write the following:

```
andy = 25;  
fred = andy;  
ted = &andy;
```

the result is shown in the following diagram:



We have assigned to **fred** the content of variable **andy** as we have done in many other occasions in previous sections of this tutorial, but to **ted** we have assigned the address in memory where the operating system stores the value of **andy**, that we have imagined was **1776** (it can be any address, I have just invented this one). The reason is that in the allocation of **ted** we have preceded **andy** with an *ampersand* (&) character.

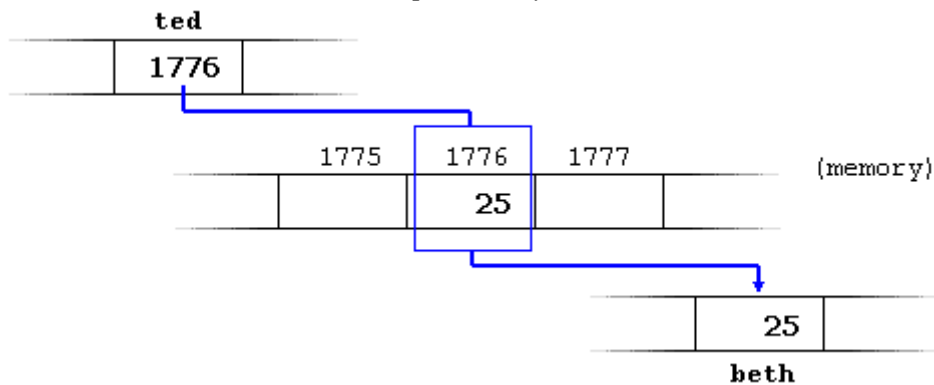
The variable that stores the address of another variable (like **ted** in the previous example) is what we call a **pointer**. In C++ pointers have certain virtues and they are used very often. Farther ahead we will see how this type of variable is declared.

Reference operator (*)

Using a pointer we can directly access the value stored in the variable pointed by it just by preceding the pointer identifier with the reference operator *asterisk* (*), that can be literally translated to "*value pointed by*". Therefore, following with the values of the previous example, if we write:

```
beth = *ted;
```

(that we could read as: "beth equal to value pointed by ted") **beth** would take the value **25**, since **ted** is **1776**, and *the value pointed by 1776* is **25**.



You must clearly differentiate that **ted** stores **1776**, but ***ted** (with an asterisk * before) refers to the value stored in the address **1776**, that is **25**. Notice the difference of including or not including the reference asterisk (I have included an explanatory commentary of how each expression could be read):

```
beth = ted;    // beth equal to ted ( 1776 )  
beth = *ted;   // beth equal to value pointed by ted ( 25 )
```

Operator of address or dereference (&)

It is used as a variable prefix and can be translated as "**address of**", thus: `&variable1` can be read as "*address of variable1*"

Operator of reference (*)

It indicates that what has to be evaluated is the content pointed by the expression considered as an address. It can be translated by "**value pointed by**".

* `mypointer` can be read as "*value pointed by mypointer*".

At this point, and following with the same example initiated above where:

```
andy = 25;  
ted = &andy;
```

you should be able to clearly see that all the following expressions are true:

```
andy == 25  
&andy == 1776  
ted == 1776  
*ted == 25
```

The first expression is quite clear considering that its assignation was **andy=25**; . The second one uses the address (or dereference) operator (&) that returns the address of the variable **andy**, that we imagined to be **1776**. The third one is quite obvious since the second was true and the assignation of **ted** was **ted = &andy**; . The fourth expression uses the reference operator (*) that, as we have just seen, is equivalent to the value contained in the address pointed by **ted**, that is **25**.

So, after all that, you may also infer that while the address pointed by `ted` remains unchanged the following expression will also be true:

```
*ted == andy
```

Declaring variables of type pointer

Due to the ability of a pointer to directly reference the value that it point to, it becomes necessary to specify which data type a pointer points to when declaring it. It is not the same to point to a `char` as it is to point to an `int` or a `float` type.

Therefore, the declaration of pointers follows this form:

```
type * pointer_name;
```

where **type** is the type of data pointed, not the type of the pointer itself. For example:

```
int * number;
```

```
char * character;
```

```
float * greatnumber;
```

they are three declarations of pointers. Each one points to a different data type, but the three are pointers and in fact the three occupy the same amount of space in memory (the size of a pointer depends on the operating system), but the data to which they point do not occupy the same amount of space nor are of the same type, one is `int`, another one is `char` and the other one `float`.

I emphasize that the asterisk (*) that we use when declaring a pointer means only *that it is a pointer*, and should not be confused with the reference operator that we have seen a bit earlier which is also written with an asterisk (*). They are simply two different tasks represented with the same sign.

```
// my first pointer
#include <iostream.h>

int main ()
{
    int value1 = 5, value2 = 15;
    int * mypointer;

    mypointer = &value1;
    *mypointer = 10;
    mypointer = &value2;
    *mypointer = 20;
    cout << "value1==" << value1 << " /
value2==" << value2;
    return 0;
}
```

```
value1==10 / value2==20
```

Notice how the values of `value1` and `value2` have changed indirectly. First we have assigned to `mypointer` the address of `value1` using the deference ampersand sign (&). Then we have assigned 10 to the value pointed by `mypointer`, which is pointing to the address of `value1`, so we have modified `value1` indirectly.

In order that you can see that a pointer may take several different values during the same program we have repeated the process with `value2` and the same pointer.

Here is an example a bit more complicated:

```
// more pointers
#include <iostream.h>

int main ()
{
    int value1 = 5, value2 = 15;
    int *p1, *p2;

    p1 = &value1;      // p1 = address
                        // of value1
    p2 = &value2;      // p2 = address
                        // of value2
    *p1 = 10;          // value pointed
                        // by p1 = 10
    *p2 = *p1;          // value pointed
                        // by p2 = value pointed by p1
    p1 = p2;           // p1 = p2
                        // (value of pointer copied)
    *p1 = 20;          // value pointed
                        // by p1 = 20

    cout << "value1==" << value1 << " /
value2==" << value2;
    return 0;
}
```

```
value1==10 / value2==20
```

I have included as comments on each line how the code can be read: ampersand (&) as "address of" and asterisk (*) as "value pointed by". Notice that there are expressions with pointers `p1` and `p2` with and without the asterisk. The meaning of using or not using a reference asterisk is very different: An asterisk (*) followed by the pointer refers to the place pointed by the pointer, whereas a pointer without an asterisk (*) refers to the value of the pointer itself, that is, the address of where it is pointing.

Another thing that can call your attention is the line:

```
int *p1, *p2;
```

that declares the two pointers of the previous example putting an asterisk (*) for each pointer. The reason is that the type for all the declarations of the same line is `int` (and not `int*`). The explanation is because of the level of precedence of the reference operator asterisk (*) that is the same as the declaration of types, therefore, because they are associative operators from the right, the asterisks are evaluated first than the type. We have talked about this in [section 1.3: Operators](#), although it is enough that you know clearly that -unless you include parenthesis- you will have to put an asterisk (*) before each pointer that you declare.

Pointers and arrays

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, like a pointer is equivalent to the address of the first element that it points to, so in fact they are the same thing. For example, supposing these two declarations:

```
int numbers [20];
int * p;
```

the following allocation would be valid:

```
p = numbers;
```

At this point **p** and **numbers** are equivalent and they have the same properties, the only difference is that we could assign another value to the pointer **p** whereas **numbers** will always point to the first of the 20 integer numbers of type `int` with which it was defined. So, unlike **p**, that is an ordinary *variable pointer*, **numbers** is a *constant pointer* (indeed an array name is a constant pointer). Therefore, although the previous expression was valid, the following allocation is not:

```
numbers = p;
```

because **numbers** is an array (constant pointer), and no values can be assigned to constant identifiers.

Due to the character of *variables* all the expressions that include pointers in the following example are perfectly valid:

```
// more pointers
#include <iostream.h>

int main ()
{
    int numbers[5];
    int * p;
    p = numbers;  *p = 10;
    p++;  *p = 20;
    p = &numbers[2];  *p = 30;
    p = numbers + 3;  *p = 40;
    p = numbers;  *(p+4) = 50;
    for (int n=0; n<5; n++)
        cout << numbers[n] << " , ";
    return 0;
}
```

```
10, 20, 30, 40, 50,
```

In chapter "Arrays" we used bracket signs `[]` several times in order to specify the index of the element of the Array to which we wanted to refer. Well, the bracket signs operator `[]` are known as *offset operators* and they are equivalent to adding the number within brackets to the address of a pointer. For example, both following expressions:

```
a[5] = 0;           // a [offset of 5] = 0
*(a+5) = 0;         // pointed by (a+5) = 0
```

are equivalent and valid either if **a** is a pointer or if it is an array.

Pointer initialization

When declaring pointers we may want to explicitly specify to which variable we want them to point,

```
int number;
int *tommy = &number;
```

this is equivalent to:

```
int number;
int *tommy;
tommy = &number;
```

When a pointer assignation takes place we are always assigning the address where it points to, never the value pointed. You must consider that at the moment of declaring a pointer, the asterisk (*) indicates only that it is a pointer, it in no case indicates the reference operator (*).

Remember, they are two different operators, although they are written with the same sign. Thus, we must take care not to confuse the previous with:

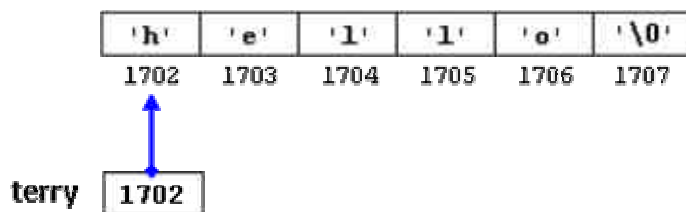
```
int number;  
int *tommy;  
*tommy = &number;
```

that anyway would not have much sense in this case.

As in the case of arrays, the compiler allows the special case that we want to initialize the content at which the pointer points with constants at the same moment as declaring the variable pointer:

```
char * terry = "hello";
```

in this case static storage is reserved for containing "hello" and a pointer to the first **char** of this memory block (that corresponds to 'h') is assigned to **terry**. If we imagine that "hello" is stored at addresses 1702 and following, the previous declaration could be outlined thus:

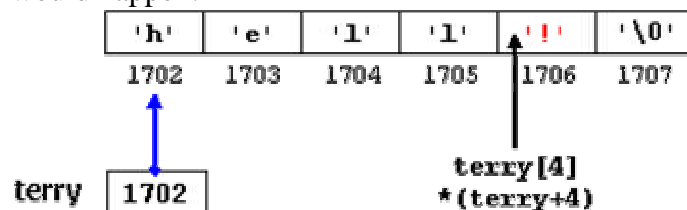


it is important to indicate that **terry** contains the value 1702 and not 'h' nor "hello", although 1702 points to these characters.

The pointer **terry** points to a string of characters and can be used exactly as if it was an Array (remember that an array is just *a constant pointer*). For example, if our temper changed and we wanted to replace the 'o' by a '!' sign in the content pointed by **terry**, we could do it by any of the following two ways:

```
terry[4] = '!';  
*(terry+4) = '!';
```

remember that to write **terry[4]** is just the same as to write ***(terry+4)**, although the most usual expression is the first one. With either of those two expressions something like this would happen:



Arithmetic of pointers

To conduct arithmetical operations on pointers is a little different than to conduct them on other integer data types. To begin with, only addition and subtraction operations are allowed to be conducted, the others make no sense in the world of pointers. But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point.

When we saw the different data types that exist, we saw that some occupy more or less space than others in the memory. For example, in the case of integer numbers, *char* occupies 1 byte, *short* occupies 2 bytes and *long* occupies 4.

Let's suppose that we have 3 pointers:

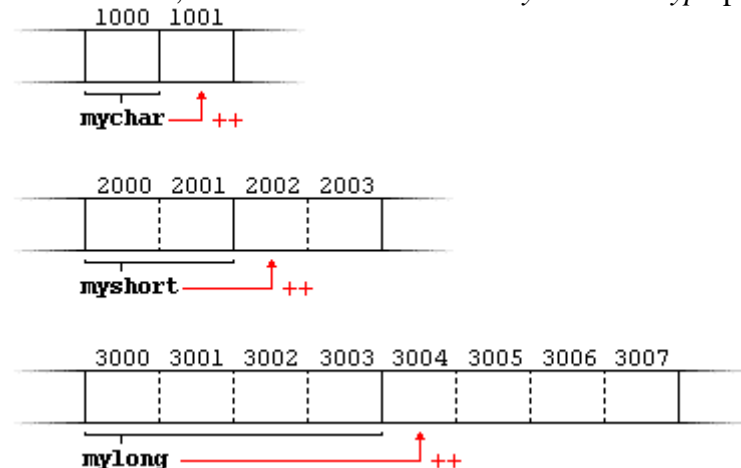
```
char *mychar;  
short *myshort;  
long *mylong;
```

and that we know that they point to memory locations 1000, 2000 and 3000 respectively.

So if we write:

```
mychar++;  
myshort++;  
mylong++;
```

mychar, as you may expect, would contain the value 1001. Nevertheless, *myshort* would contain the value 2002, and *mylong* would contain 3004. The reason is that when adding 1 to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in *bytes* of the *type* pointed is added to the pointer.



This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we write:

```
mychar = mychar + 1;  
myshort = myshort + 1;  
mylong = mylong + 1;
```

It is important to warn you that both increase (`++`) and decrease (`--`) operators have a greater priority than the reference operator asterisk (`*`), therefore the following expressions may lead to confusion:

```
*p++;  
*p++ = *q++;
```

The first one is equivalent to `*(p++)` and what it does is to increase *p* (the address where it points to - not the value that contains).

In the second, because both increase operators (`++`) are after the expressions to be evaluated and not before, first the value of **q* is assigned to **p* and then both *q* and *p* are increased by one. It is equivalent to:

```
*p = *q;  
p++;  
q++;
```

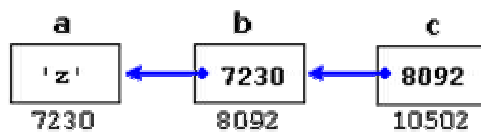
Like always, I recommend you use parenthesis () in order to avoid unexpected results.

Pointers to pointers

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data. In order to do that we only need to add an asterisk (*) for each level of reference:

```
char a;  
char * b;  
char ** c;  
a = 'z';  
b = &a;  
c = &b;
```

this, supposing the randomly chosen memory locations of 7230, 8092 and 10502, could be described thus:



(inside the cells there is the content of the variable; under the cells its location)

The new thing in this example is variable **c**, which we can talk about in three different ways, each one of them would correspond to a different value:

```
c is a variable of type (char **) with a value of 8092  
*c is a variable of type (char*) with a value of 7230  
**c is a variable of type (char) with a value of 'z'
```

void pointers

The type of pointer *void* is a special type of pointer. *void* pointers can point to any data type, from an integer value or a float to a string of characters. Its sole limitation is that the pointed data cannot be referenced directly (we can not use reference asterisk * operator on them), since its length is always undetermined, and for that reason we will always have to resort to *type casting* or assignments to turn our *void* pointer to a pointer of a concrete data type to which we can refer.

One of its utilities may be for passing generic parameters to a function:

```
// integer increaser  
#include <iostream.h>  
  
void increase (void* data, int type)  
{  
    switch (type)  
    {  
        case sizeof(char) :  
        (*(char*)data)++; break;  
        case sizeof(short):  
        (*(short*)data)++; break;  
        case sizeof(long) :  
        (*(long*)data)++; break;  
    }  
}  
  
int main ()
```

6, 10, 13

```

{
    char a = 5;
    short b = 9;
    long c = 12;
    increase (&a,sizeof(a));
    increase (&b,sizeof(b));
    increase (&c,sizeof(c));
    cout << (int) a << ", " << b << ",
" << c;
    return 0;
}

```

sizeof is an operator integrated in the C++ language that returns a constant value with the size in bytes of its parameter, so, for example, **sizeof(char)** is 1, because **char** type is 1 byte long.

Pointers to functions

C++ allows operations with pointers to functions. The greatest use of this is for passing a function as a parameter to another function, since these cannot be passed dereferenced. In order to declare a pointer to a function we must declare it like the prototype of the function except the name of the function is enclosed between parenthesis () and a pointer asterisk (*) is inserted before the name. It might not be a very handsome syntax, but that is how it is done in C++:

```

// pointer to functions
#include <iostream.h>

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

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

int (*minus)(int,int) = subtraction;

int operation (int x, int y, int
(*functocall)(int,int))
{
    int g;
    g = (*functocall)(x,y);
    return (g);
}

int main ()
{
    int m,n;
    m = operation (7, 5, addition);
    n = operation (20, m, minus);
    cout <<n;
    return 0;
}

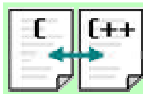
```

8

In the example, **minus** is a global pointer to a function that has two parameters of type **int**, it is immediately assigned to point to the function **subtraction**, all in a single line:
`int (* minus)(int,int) = subtraction;`

Until now, in our programs, we have only had as much memory as we have requested in declarations of variables, arrays and other objects that we included, having the size of all of them fixed before the execution of the program. But, what if we need a variable amount of memory that can only be determined during the program execution (runtime), for example, in case that we need an user input to determine the necessary amount of space?

The answer is *dynamic memory*, for which C++ integrates the operators *new* and *delete*.



Operators *new* and *delete* are exclusive of C++. Farther ahead in this section are shown the C equivalents for these operators.

Operators *new* and *new[]*

In order to request dynamic memory, the operator **new** exists. *new* is followed by a data *type* and optionally the number of elements required within brackets `[]`. It returns a pointer to the beginning of the new block of assigned memory. Its form is:

```
pointer = new type
```

or

```
pointer = new type [elements]
```

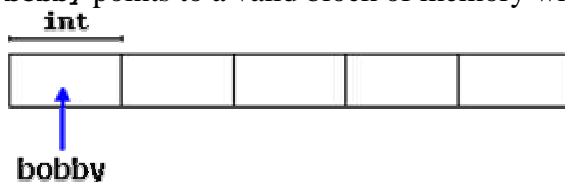
The first expression is used to assign memory to contain one single element of *type*. The second one is used to assign a block (an array) of elements of *type*.

For example:

```
int * bobby;
```

```
bobby = new int [5];
```

in this case, the operating system has assigned space for 5 elements of type `int` in a heap and it has returned a pointer to its beginning that has been assigned to `bobby`. Therefore, now, `bobby` points to a valid block of memory with space for 5 `int` elements.



You could ask what is the difference between declaring a normal array and assigning memory to a pointer as we have just done. The most important one is that the size of an array must be a constant value, which limits its size to what we decide at the moment of designing the program before its execution, whereas the dynamic memory allocation allows assigning memory during the execution of the program using any variable, constant or combination of both as size.

The dynamic memory is generally managed by the operating system, and in multitask interfaces it can be shared between several applications, so there is a possibility that the memory exhausts. If this happens and the operating system cannot assign the memory that we request with the operator **new**, a null pointer will be returned. For that reason it is recommended to always check to see if the returned pointer is null after a call to **new**.

```
int * bobby;
bobby = new int [5];
if (bobby == NULL) {
    // error assigning memory. Take measures.
};
```

Operator *delete*.

Since the necessity of dynamic memory is usually limited to concrete moments within a program, once it is no longer needed it should be freed so that it becomes available for future requests of dynamic memory. The operator **delete** exists for this purpose, whose form is:

```
delete pointer;
```

or

```
delete [] pointer;
```

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for multiple elements (arrays). In most compilers both expressions are equivalent and can be used without distinction, although indeed they are two different operators and so must be considered for operator overloading (we will see that on [section 4.2](#)).

```
// rememb-o-matic
#include <iostream.h>
#include <stdlib.h>

int main ()
{
    char input [100];
    int i,n;
    long * l;
    cout << "How many numbers do you
want to type in? ";
    cin.getline (input,100); i=atoi
(input);
    l= new long[i];
    if (l == NULL) exit (1);
    for (n=0; n<i; n++)
    {
        cout << "Enter number: ";
        cin.getline (input,100);
        l[n]=atol (input);
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
        cout << l[n] << ", ";
    delete[] l;
    return 0;
}
```

```
How many numbers do you want to type
in? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8,
32,
```

This simple example that memorizes numbers does not have a limited amount of numbers that can be introduced, thanks to us requesting to the system to provide as much space as is necessary to store all the numbers that the user wishes to introduce.

NULL is a constant value defined in manyfold C++ libraries specially designed to indicate null pointers. In case that this constant is not defined you can do it yourself by defining it to 0:

```
#define NULL 0
```

It is indifferent to put 0 or NULL when checking pointers, but the use of NULL with pointers is widely extended and it is recommended for greater legibility. The reason is that a pointer is rarely compared or set directly to a numerical literal constant except precisely number 0, and this way this action is symbolically masked.

Dynamic memory in ANSI-C

Operators *new* and *delete* are exclusive of C++ and they are not available in C language. In C language, in order to assign dynamic memory we have to resort to the library `stdlib.h`. We are going to see them, since they are also valid in C++ and they are used in some existing programs.

The function *malloc*

It is the generic function to assign dynamic memory to pointers. Its prototype is:

```
void * malloc (size_t nbytes);
```

where *nbytes* is the number of bytes that we want to be assigned to the pointer. The function returns a pointer of type `void*`, which is the reason why we have to *type cast* the value to the type of the destination pointer, for example:

```
char * ronny;  
ronny = (char *) malloc (10);
```

This assigns to *ronny* a pointer to an usable block of 10 bytes. When we want to assign a block of data of a different type other than `char` (different from 1 byte) we must multiply the number of elements desired by the size of each element. Luckily we have at our disposition the operator *sizeof*, that returns the size of the type of a concrete datum.

```
int * bobby;  
bobby = (int *) malloc (5 * sizeof(int));
```

This piece of code assigns to *bobby* a pointer to a block of 5 integers of type *int*, this size can be equal to 2, 4 or more bytes according to the system where the program is compiled.

The function *calloc*.

calloc is very similar to *malloc* in its operation, its main difference is in its prototype:

```
void * calloc (size_t nelements, size_t size);
```

since it admits 2 parameters instead of one. These two parameters are multiplied to obtain the total size of the memory block to be assigned. Usually the first parameter (*nelements*) is the number of elements and the second one (*size*) serves to specify the size of each element. For example, we could define *bobby* with *calloc* thus:

```
int * bobby;  
bobby = (int *) calloc (5, sizeof(int));
```

Another difference between *malloc* and *calloc* is that *calloc* initializes all its elements to 0.

The function *realloc*.

It changes the size of a block of memory already assigned to a pointer.

```
void * realloc (void * pointer, size_t size);
```

pointer parameter receives a pointer to an already assigned memory block or a null pointer, and *size* specifies the new size that the memory block shall have. The function assigns *size* bytes of memory to the pointer. The function may need to change the location of the memory block so that the new size can fit, in that case the present content of the block is copied to the new one to guarantee that the existing data is not lost. The new pointer is returned by the

function. If it has not been possible to assign the memory block with the new size it returns a null pointer but the *pointer* specified as parameter and its content remains unchanged.

The function *free*.

It releases a block of dynamic memory previously assigned using *malloc*, *calloc* or *realloc*.

```
void free (void * pointer);
```

This function must only be used to release memory assigned with functions *malloc*, *calloc* and *realloc*.

You may obtain more information about these functions in the [C++ reference for cstdlib](#).

Section 3.5

Structures



Data structures.

A data structure is a set of diverse types of data that may have different lengths grouped together under a unique declaration. Its form is the following:

```
struct model_name {  
    type1 element1;  
    type2 element2;  
    type3 element3;  
    .  
    .  
} object_name;
```

where *model_name* is a name for the model of the structure type and the optional parameter *object_name* is a valid identifier (or identifiers) for structure object instantiations. Within curly brackets { } they are the types and their sub-identifiers corresponding to the elements that compose the structure.

If the structure definition includes the parameter *model_name* (optional), that parameter becomes a valid type name equivalent to the structure. For example:

```
struct products {  
    char name [30];  
    float price;  
} ;  
  
products apple;  
products orange, melon;
```

We have first defined the structure model **products** with two fields: **name** and **price**, each of a different type. We have then used the name of the structure type (**products**) to declare three objects of that type: **apple**, **orange** and **melon**.

Once declared, **products** has become a new valid type name like the fundamental ones *int*, *char* or *short* and we are able to declare objects (variables) of that type.

The optional field *object_name* that can go at the end of the structure declaration serves to directly declare objects of the structure type. For example, we can also declare the structure objects **apple**, **orange** and **melon** this way:

```
struct products {  
    char name [30];  
    float price;  
} apple, orange, melon;
```

Moreover, in cases like the last one in which we took advantage of the declaration of the structure model to declare objects of it, the parameter *model_name* (in this case **products**) becomes optional. Although if *model_name* is not included it will not be possible to declare more objects of this same model later.

It is important to clearly differentiate between what is a structure **model**, and what is a structure *object*. Using the terms we used with variables, the *model* is the *type*, and the *object* is the *variable*. We can instantiate many *objects* (variables) from a single *model* (type).

Once we have declared our three objects of a determined structure model (**apple**, **orange** and **melon**) we can operate with the fields that form them. To do that we have to use a point (.) inserted between the object name and the field name. For example, we could operate with any of these elements as if they were standard variables of their respective types:

```
apple.name  
apple.price  
orange.name  
orange.price  
melon.name  
melon.price
```

each one being of its corresponding data type: **apple.name**, **orange.name** and **melon.name** are of type **char[30]**, and **apple.price**, **orange.price** and **melon.price** are of type **float**.

We are going to leave apples, oranges and melons and go with an example about movies:

```
// example about structures  
#include <iostream.h>  
#include <string.h>  
#include <stdlib.h>  
  
struct movies_t {  
    char title [50];  
    int year;  
} mine, yours;  
  
void printmovie (movies_t movie);  
  
int main ()  
{  
    char buffer [50];  
  
    strcpy (mine.title, "2001 A Space  
Odyssey");  
    mine.year = 1968;  
  
    cout << "Enter title: ";  
    cin.getline (yours.title,50);  
    cout << "Enter year: ";
```

```
Enter title: Alien  
Enter year: 1979  
  
My favourite movie is:  
    2001 A Space Odyssey (1968)  
And yours:  
    Alien (1979)
```

```

    cin.getline (buffer,50);
    yours.year = atoi (buffer);

    cout << "My favourite movie is:\n
";
    printmovie (mine);
    cout << "And yours:\n ";
    printmovie (yours);
    return 0;
}

void printmovie (movies_t movie)
{
    cout << movie.title;
    cout << " (" << movie.year <<
    ")\n";
}

```

The example shows how we can use the elements of a structure and the structure itself as normal variables. For example, **yours.year** is a valid variable of type **int**, and **mine.title** is a valid array of 50 *chars*.

Notice that **mine** and **yours** are also treated as valid variables of type **movies_t** when being passed to the function **printmovie()**. Therefore, one of the most important advantages of structures is that we can refer either to their elements individually or to the entire structure as a block.

Structures are a feature used very often to build data bases, specially if we consider the possibility of building arrays of them.

```

// array of structures
#include <iostream.h>
#include <stdlib.h>

#define N_MOVIES 5

struct movies_t {
    char title [50];
    int year;
} films [N_MOVIES];

void printmovie (movies_t movie);

int main ()
{
    char buffer [50];
    int n;
    for (n=0; n<N_MOVIES; n++)
    {
        cout << "Enter title: ";
        cin.getline (films[n].title,50);
        cout << "Enter year: ";
        cin.getline (buffer,50);
        films[n].year = atoi (buffer);
    }
    cout << "\nYou have entered these

```

```

Enter title: Alien
Enter year: 1979
Enter title: Blade Runner
Enter year: 1982
Enter title: Matrix
Enter year: 1999
Enter title: Rear Window
Enter year: 1954
Enter title: Taxi Driver
Enter year: 1975

```

```

You have entered these movies:
Alien (1979)
Blade Runner (1982)
Matrix (1999)
Rear Window (1954)
Taxi Driver (1975)

```

```

movies:\n";
    for (n=0; n<N_MOVIES; n++)
        printmovie (films[n]);
    return 0;
}

void printmovie (movies_t movie)
{
    cout << movie.title;
    cout << " (" << movie.year <<
"\n";
}

```

Pointers to structures

Like any other type, structures can be pointed by pointers. The rules are the same as for any fundamental data type: The pointer must be declared as a pointer to the structure:

```

struct movies_t {
    char title [50];
    int year;
};

```

```

movies_t amovie;
movies_t * pmovie;

```

Here **amovie** is an object of struct type **movies_t** and **pmovie** is a pointer to point to objects of struct type **movies_t**. So, the following, as with fundamental types, would also be valid:

```
pmovie = &amovie;
```

Ok, we will now go with another example, that will serve to introduce a new operator:

```

// pointers to structures
#include <iostream.h>
#include <stdlib.h>

struct movies_t {
    char title [50];
    int year;
};

int main ()
{
    char buffer[50];

    movies_t amovie;
    movies_t * pmovie;
    pmovie = & amovie;

    cout << "Enter title: ";
    cin.getline (pmovie->title,50);
    cout << "Enter year: ";
    cin.getline (buffer,50);
    pmovie->year = atoi (buffer);

    cout << "\nYou have entered:\n";
    cout << pmovie->title;
    cout << " (" << pmovie->year <<
"\n";
}

```

```

Enter title: Matrix
Enter year: 1999

```

```

You have entered:
Matrix (1999)

```

```
    return 0;
}
```

The previous code includes an important introduction: operator `->`. This is a reference operator that is used exclusively with pointers to structures and pointers to classes. It allows us not to have to use parenthesis on each reference to a structure member. In the example we used:

```
pmovie->title
```

that could be translated to:

```
(*pmovie).title
```

both expressions `pmovie->title` and `(*pmovie).title` are valid and mean that we are evaluating the element `title` of the structure pointed by `pmovie`. You must distinguish it clearly from:

```
*pmovie.title
```

that is equivalent to

```
*(pmovie.title)
```

and that would serve to evaluate the value pointed by element `title` of structure `movies`, that in this case (where `title` is not a pointer) it would not make much sense. The following panel summarizes possible combinations of pointers and structures:

Expression	Description	Equivalent
<code>pmovie.title</code>	Element <code>title</code> of structure <code>pmovie</code>	
<code>pmovie->title</code>	Element <code>title</code> of structure <u>pointed by</u> <code>pmovie</code>	<code>(*pmovie).title</code>
<code>*pmovie.title</code>	Value <u>pointed by</u> element <code>title</code> of structure <code>pmovie</code>	<code>*(pmovie.title)</code>

Nesting structures

Structures can also be nested so that a valid element of a structure can also be another structure.

```
struct movies_t {
    char title [50];
    int year;
}
```

```
struct friends_t {
    char name [50];
    char email [50];
    movies_t favourite_movie;
} charlie, maria;
```

```
friends_t * pfriends = &charlie;
```

Therefore, after the previous declaration we could use the following expressions:

```
charlie.name
```

```
maria.favourite_movie.title
```

```
charlie.favourite_movie.year
```

```
pfriends->favourite_movie.year
```

(where, by the way, the last two expressions are equivalent).

The concept of structures that has been discussed in this section is the same as used in C language, nevertheless, in C++, the structure concept has been extended up to the same functionality of a *class* with the peculiarity that all of its elements are considered *public*. But you will have more details about this topic on section [4.1, Classes](#).

Section 3.6

User defined data types



We have already seen a data type that is defined by the user (programmer): the structures. But in addition to these there are other kinds of user defined data types:

Definition of own types (`typedef`).

C++ allows us to define our own types based on other existing data types. In order to do that we shall use keyword **typedef**, whose form is:

```
typedef    existing_type    new_type_name ;
```

where *existing_type* is a C++ fundamental or any other defined type and *new_type_name* is the name that the new type we are going to define will receive. For example:

```
typedef char C;
typedef unsigned int WORD;
typedef char * string_t;
typedef char field [50];
```

In this case we have defined four new data types: **C**, **WORD**, **string_t** and **field** as **char**, **unsigned int**, **char*** and **char[50]** respectively, that we could perfectly use later as valid types:

```
C achar, anotherchar, *ptchar1;
WORD myword;
string_t ptchar2;
field name;
```

`typedef` can be useful to define a type that is repeatedly used within a program and it is possible that we will need to change it in a later version, or if a type you want to use has too long a name and you want it to be shorter.

Unions

Unions allow a portion of memory to be accessed as different data types, since all of them are in fact the same location in memory. Its declaration and use is similar to the one of structures but its functionality is totally different:

```
union model_name {
    type1 element1;
    type2 element2;
    type3 element3;
    .
    .
} object_name;
```

All the elements of the *union* declaration occupy the same space of memory. Its size is the one of the greatest element of the declaration. For example:

```
union mytypes_t {
    char c;
    int i;
    float f;
```



```

    } mytypes;
defines three elements:
mytypes.c
mytypes.i
mytypes.f

```

each one of a different data type. Since all of them are referring to a same location in memory, the modification of one of the elements will affect the value of all of them.

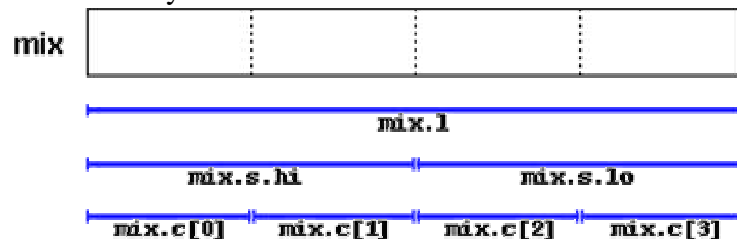
One of the uses a *union* may have is to unite an elementary type with an array or structures of smaller elements. For example,

```

union mix_t{
    long l;
    struct {
        short hi;
        short lo;
    } s;
    char c[4];
} mix;

```

defines three names that allow us to access the same group of 4 *bytes*: **mix.l**, **mix.s** and **mix.c** and which we can use according to how we want to access it, as **long**, **short** or **char** respectively. I have mixed types, arrays and structures in the union so that you can see the different ways that we can access the data:



Anonymous unions

In C++ we have the option that unions be anonymous. If we include a union in a structure without any object name (the one that goes after the curly brackets { }) the union will be anonymous and we will be able to access the elements directly by its name. For example, look at the difference between these two declarations:

union

```

struct {
    char title[50];
    char author[50];
    union {
        float dollars;
        int yens;
    } price;
} book;

```

anonymous union

```

struct {
    char title[50];
    char author[50];
    union {
        float dollars;
        int yens;
    };
} book;

```

The only difference between the two pieces of code is that in the first one we gave a name to the union (**price**) and in the second we did not. The difference is when accessing members **dollars** and **yens** of an object. In the first case it would be:

```

book.price.dollars
book.price.yens

```

whereas in the second it would be:

```
book.dollars  
book.yens
```

Once again I remind you that because it is a union, the fields **dollars** and **yens** occupy the same space in the memory so they cannot be used to store two different values. That means that you can include a price in dollars or yens, but not both.

Enumerations (`enum`)

Enumerations serve to create data types to contain something different that is not limited to either numerical or character constants nor to the constants **true** and **false**. Its form is the following:

```
enum model_name {  
    value1,  
    value2,  
    value3,  
    .  
    .  
}  
object_name;
```

For example, we could create a new type of variable called **color** to store colors with the following declaration:

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

Notice that we do not include any fundamental data type in the declaration. To say it another way, we have created a new data type without it being based on any existing one: the type **color_t**, whose possible values are the colors that we have enclosed within curly brackets {}.

For example, once declared the **colors_t** enumeration in the following expressions will be valid:

```
colors_t mycolor;
```

```
mycolor = blue;
```

```
if (mycolor == green) mycolor = red;
```

In fact our enumerated data type is compiled as an integer and its possible values are any type of integer constant specified. If it is not specified, the integer value equivalent to the first possible value is 0 and the following ones follow a +1 progression. Thus, in our data type **colors_t** that we defined before, **black** would be equivalent to 0, **blue** would be equivalent to 1, **green** to 2 and so on.

If we explicitly specify an integer value for some of the possible values of our enumerated type (for example the first one) the following values will be the increases of this, for example:

```
enum months_t { january=1, february, march, april,  
                may, june, july, august,  
                september, october, november, december} y2k;
```

in this case, variable **y2k** of the enumerated type **months_t** can contain any of the 12 possible values that go from **january** to **december** and that are equivalent to values between 1 and 12, not between 0 and 11 since we have made **january** equal to 1.

Section 4.1

Classes



A class is a logical method to organize data and functions in the same structure. They are declared using keyword **class**, whose functionality is similar to that of the C keyword **struct**, but with the possibility of including functions as members, instead of only data.

Its form is:

```
class class_name {  
    permission_label_1:  
        member1;  
    permission_label_2:  
        member2;  
    ...  
} object_name;
```

where **class_name** is a name for the class (user defined *type*) and the optional field **object_name** is one, or several, valid object identifiers. The body of the declaration can contain **members**, that can be either data or function declarations, and optionally **permission labels**, that can be any of these three keywords: **private:**, **public:** or **protected:**. They make reference to the permission which the following *members* acquire:

- **private** members of a class are accessible only from other members of their same class or from their "*friend*" classes.
- **protected** members are accessible from members of their same class and *friend* classes, and also from members of their *derived* classes.
- Finally, **public** members are accessible from anywhere the class is visible.

If we declare members of a class before including any permission label, the members are considered **private**, since it is the default permission that the members of a class declared with the **class** keyword acquire.

For example:

```
class CRectangle {  
    int x, y;  
    public:  
        void set_values (int,int);  
        int area (void);  
} rect;
```

Declares class **CRectangle** and an object called **rect** of this class (type). This class contains four members: two variables of type **int** (**x** and **y**) in the **private** section (because private is the default permission) and two functions in the **public** section: **set_values()** and **area()**, of which we have only included the prototype.

Notice the difference between class name and object name: In the previous example, **CRectangle** was the class name (i.e., the user-defined type), whereas **rect** was an object of type **CRectangle**. Is the same difference that **int** and **a** have in the following declaration:

```
int a;  
int is the class name (type) and a is the object name (variable).
```

On successive instructions in the body of the program we can refer to any of the public members of the object **rect** as if they were normal functions or variables, just by putting the object's name followed by a point and then the class member (like we did with C **structs**). For example:

```
rect.set_value (3,4);  
myarea = rect.area();
```

but we will not be able to refer to **x** or **y** since they are private members of the class and they could only be referred from other members of that same class. Confused? Here is the complete example of class **CRectangle**:

```
// classes example  
#include <iostream.h>  
  
class CRectangle {  
    int x, y;  
public:  
    void set_values (int,int);  
    int area (void) {return (x*y);}  
};  
  
void CRectangle::set_values (int a,  
int b) {  
    x = a;  
    y = b;  
}  
  
int main () {  
    CRectangle rect;  
    rect.set_values (3,4);  
    cout << "area: " << rect.area();  
}
```

```
area: 12
```

The new thing in this code is the operator **::** of scope included in the definition of **set_values()**. It is used to declare a member of a class outside it. Notice that we have defined the behavior of function **area()** within the definition of the **CRectangle** class - given its extreme simplicity. Whereas **set_values()** has only its prototype declared within the class but its definition is outside. In this outside declaration we must use the operator of scope **::**.

The scope operator (**::**) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if it was directly defined within the class. For example, in the function **set_values()** of the previous code, we have referred to the variables **x** and **y**, that are members of class **CRectangle** and that are only visible inside it and its members (since they are **private**).

The only difference between defining a class member function completely within its class and to include only the prototype, is that in the first case the function will automatically be considered *inline* by the compiler, while in the second it will be a normal (not-inline) class member function.

The reason why we have made **x** and **y** **private** members (remember that if nothing else is said all members of a class defined with keyword *class* have **private** access) it is because we have already defined a function to introduce those values in the object (**set_values()**) and therefore the rest of the program does not have a way to directly access them. Perhaps in a so simple example as this you do not see a great utility protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

One of the greater advantages of a class is that we can declare several different objects from it. For example, following with the previous example of class **CRectangle**, we could have declared the object **rectb** in addition to the object **rect** :

```
// class example
#include <iostream.h>

class CRectangle {
    int x, y;
public:
    void set_values (int,int);
    int area (void) {return (x*y);}
};

void CRectangle::set_values (int a,
int b) {
    x = a;
    y = b;
}

int main () {
    CRectangle rect, rectb;
    rect.set_values (3,4);
    rectb.set_values (5,6);
    cout << "rect area: " <<
rect.area() << endl;
    cout << "rectb area: " <<
rectb.area() << endl;
}
```

```
rect area: 12
rectb area: 30
```

Notice that the call to **rect.area()** does not give the same result as the call to **rectb.area()**. This is because each object of class **CRectangle** has its own variables **x** and **y**, and its own functions **set_value()** and **area()**.

On that is based the concept of ***object*** and ***object-oriented programming***. In that data and functions are properties of the object, instead of the usual view of objects as function parameters in structured programming. In this and the following sections we will discuss advantages of this methodology.

In this concrete case, the class (type of object) to which we were talking about is **CRectangle**, of which there are two instances, or objects: **rect** and **rectb**, each one with its own member variables and member functions.

Constructors and destructors

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become totally operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the function **area()** before having called function **set_values**? Probably an indetermined result since the members **x** and **y** would have never been assigned a value.

In order to avoid that, a class can include a special function: a *constructor*, which can be declared by naming a member function with the same name as the class. This constructor *function* will be called automatically when a new instance of the class is created (when

declaring a new object or allocating an object of that class) and only then. We are going to implement `CRectangle` including a *constructor*:

```
// classes example
#include <iostream.h>

class CRectangle {
    int width, height;
public:
    CRectangle (int,int);
    int area (void) {return
(width*height);}
};

CRectangle::CRectangle (int a, int
b) {
    width = a;
    height = b;
}

int main () {
    CRectangle rect (3,4);
    CRectangle rectb (5,6);
    cout << "rect area: " <<
rect.area() << endl;
    cout << "rectb area: " <<
rectb.area() << endl;
}
```

```
rect area: 12
rectb area: 30
```

As you can see, the result of this example is identical to the previous one. In this case we have only replaced the function `set_values`, that no longer exists, by a class *constructor*. Notice the way in which the parameters are passed to the constructor at the moment at which the instances of the class are created:

```
CRectangle rect (3,4);
CRectangle rectb (5,6);
```

You can also see how neither the prototype nor the later constructor declaration includes a return value, not even `void` type. This must always be thus. A constructor never returns a value nor does the `void` have to be specified, as we have shown in the previous example.

The **Destructor** fulfills the opposite functionality. It is automatically called when an object is released from the memory, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using operator `delete`.

The destructor must have the same name as the class with a tilde (~) as prefix and it must return no value.

The use of destructors is specially suitable when an object assigns dynamic memory during its life and at the moment of being destroyed we want to release the memory that it has used.

```
// example on constructors and
destructors
#include <iostream.h>
```

```
rect area: 12
rectb area: 30
```

```

class CRectangle {
    int *width, *height;
public:
    CRectangle (int,int);
    ~CRectangle ();
    int area (void) {return (*width
* *height);}
};

CRectangle::CRectangle (int a, int
b) {
    width = new int;
    height = new int;
    *width = a;
    *height = b;
}

CRectangle::~~CRectangle () {
    delete width;
    delete height;
}

int main () {
    CRectangle rect (3,4), rectb
(5,6);
    cout << "rect area: " <<
rect.area() << endl;
    cout << "rectb area: " <<
rectb.area() << endl;
    return 0;
}

```

Overloading Constructors

Like any other function, a constructor can also be overloaded with several functions that have the same name but different types or numbers of parameters. Remember that the compiler will execute the one that matches at the moment at which a function with that name is called ([Section 2.3, Functions-II](#)). In this case, at the moment at which a class object is declared.

In fact, in the cases where we declare a class and we do not specify any constructor the compiler automatically assumes two overloaded constructors ("*default constructor*" and "*copy constructor*"). For example, for the class:

```

class CExample {
public:
    int a,b,c;
    void multiply (int n, int m) { a=n; b=m; c=a*b; };
};

```

with no constructors, the compiler automatically assumes that it has the following constructor member functions:

- **Empty constructor**

It is a constructor with no parameters defined as *nop* (empty block of instructions). It does nothing.


```
CExample::CExample () { };
```

- **Copy constructor**

It is a constructor with only one parameter of its same type that assigns to every nonstatic class member variable of the object a copy of the passed object.

- ```
CExample::CExample (const CExample& rv) {
```
- ```
    a=rv.a;  b=rv.b;  c=rv.c;
```
- ```
}
```

It is important to realize that both default constructors: the *empty construction* and the *copy constructor* exist only if no other constructor is explicitly declared. In case that any constructor with any number of parameters is declared, none of these two default constructors will exist. So if you want them to be there, you must define your own ones.

Of course, you can also overload the class constructor providing different constructors for when you pass parameters between parenthesis and when you do not (empty):

```
// overloading class constructors
#include <iostream.h>

class CRectangle {
 int width, height;
public:
 CRectangle ();
 CRectangle (int,int);
 int area (void) {return
(width*height);}
};

CRectangle::CRectangle () {
 width = 5;
 height = 5;
}

CRectangle::CRectangle (int a, int
b) {
 width = a;
 height = b;
}

int main () {
 CRectangle rect (3,4);
 CRectangle rectb;
 cout << "rect area: " <<
rect.area() << endl;
 cout << "rectb area: " <<
rectb.area() << endl;
}
```

```
rect area: 12
rectb area: 25
```

In this case **rectb** was declared without parameters, so it has been initialized with the *constructor* that has no parameters, which declares both **width** and **height** with a value of 5.

Notice that if we declare a new object and we do not want to pass parameters to it we do not include parentheses ():

```
CRectangle rectb; // right
```

```
CRectangle rectb(); // wrong!
```

## Pointers to classes

It is perfectly valid to create pointers pointing to classes, in order to do that we must simply consider that once declared, the class becomes a valid type, so use the *class name* as the type for the pointer. For example:

```
CRectangle * prect;
```

is a pointer to an object of class **CRectangle**.

As it happens with data structures, to refer directly to a member of an object pointed by a pointer you should use operator `->`. Here is an example with some possible combinations:

```
// pointer to classes example
#include <iostream.h>

class CRectangle {
 int width, height;
public:
 void set_values (int, int);
 int area (void) {return (width *
height);}
};

void CRectangle::set_values (int a,
int b) {
 width = a;
 height = b;
}

int main () {
 CRectangle a, *b, *c;
 CRectangle * d = new
CRectangle[2];
 b= new CRectangle;
 c= &a;
 a.set_values (1,2);
 b->set_values (3,4);
 d->set_values (5,6);
 d[1].set_values (7,8);
 cout << "a area: " << a.area() <<
endl;
 cout << "*b area: " << b->area()
<< endl;
 cout << "*c area: " << c->area()
<< endl;
 cout << "d[0] area: " <<
d[0].area() << endl;
 cout << "d[1] area: " <<
d[1].area() << endl;
 return 0;
}
```

```
a area: 2
*b area: 12
*c area: 2
d[0] area: 30
d[1] area: 56
```

Next you have a summary on how can you read some pointer and class operators (`*`, `&`, `.`, `->`, `[ ]`) that appear in the previous example:

**\*x**      can be read:    pointed by **x**

**&x** can be read: address of **x**  
**x.y** can be read: member **y** of object **x**  
**(\*x).y** can be read: member **y** of object pointed by **x**  
**x->y** can be read: member **y** of object pointed by **x** (equivalent to the previous one)  
**x[0]** can be read: first object pointed by **x**  
**x[1]** can be read: second object pointed by **x**  
**x[n]** can be read: (n+1)<sup>th</sup> object pointed by **x**

Be sure you understand the logic of all of these before going on. If you have doubts, read again this section and/or consult sections "[3.3, Pointers](#)" and "[3.5, Structures](#)".

## Classes defined with keyword **struct**

C++ language has extended the C keyword **struct** to the same functionality of the C++ **class** keyword except that its members are **public** by default instead of being **private**.

Anyway, because both **class** and **struct** have almost the same functionality in C++, **struct** is usually used for data-only structures and **class** for classes that have procedures and member functions.

### Section 4.2

## Overloading operators



C++ incorporates the option to use language standard operators between classes in addition to between fundamental types. For example:

```
int a, b, c;
a = b + c;
```

is perfectly valid, since the different variables of the addition are all fundamental types. Nevertheless, is not so obvious that we can perform the following operation (in fact it is incorrect):

```
struct { char product [50]; float price; } a, b, c;
a = b + c;
```

The assignation of a class (or **struct**) to another one of the same type is allowed (default copy constructor). What would produce an error would be the addition operation, that in principle is not valid between non-fundamental types.

But thanks to the C++ ability to overload operators, we can get to do that. Objects derived from composed types such as the previous one can accept operators which would not be accepted otherwise, and we can even modify the effect of operators that they already admit. Here is a list of all the operators that can be overloaded:

|     |     |    |    |    |    |    |    |    |     |        |    |    |
|-----|-----|----|----|----|----|----|----|----|-----|--------|----|----|
| +   | -   | *  | /  | =  | <  | >  | += | -= | *=  | /=     | << | >> |
| <<= | >>= | == | != | <= | >= | ++ | -- | %  | &   | ^      | !  |    |
| ~   | &=  | ^= | =  | && |    | %= | [] | () | new | delete |    |    |

To overload an operator we only need to write a class member function whose name is **operator** followed by the operator sign that we want to overload, following this prototype:  
*type operator sign (parameters);*

Here you have an example that includes the operator **+**. We are going to sum the bidimensional vectors **a(3,1)** and **b(1,2)**. The addition of two bidimensional vectors is an operation as simple as adding the two *x* coordinates to obtain the resulting *x* coordinate and

adding the two y coordinates to obtain the resulting y. In this case the result will be  $(3+1, 1+2) = (4, 3)$ .

```
// vectors: overloading operators
example
#include <iostream.h>

class CVector {
public:
 int x,y;
 CVector () {} ;
 CVector (int,int);
 CVector operator + (CVector);
};

CVector::CVector (int a, int b) {
 x = a;
 y = b;
}

CVector CVector::operator+ (CVector
param) {
 CVector temp;
 temp.x = x + param.x;
 temp.y = y + param.y;
 return (temp);
}

int main () {
 CVector a (3,1);
 CVector b (1,2);
 CVector c;
 c = a + b;
 cout << c.x << ", " << c.y;
 return 0;
}
```

4,3

If you are baffled seeing **CVector** so many times, consider that some of them make reference to the class name **CVector** and others are functions with that name. Do not confuse them:

```
CVector (int, int); // function name CVector (constructor)
CVector operator+ (CVector); // function operator+ that returns CVector
type
```

The function **operator+** of class **CVector** is the one that is in charge of overloading the arithmetic operator **+**. This one can be called by any of these two ways:

```
c = a + b;
c = a.operator+ (b);
```

Notice also that we have included the empty constructor (without parameters) and we have defined it with a *no-op* block of instructions:

```
CVector () { };
```

this is necessary, since there already exists another constructor,

```
CVector (int, int);
```

so none of the *default constructors* will exist in **CVector** if we do not explicitly declare one as we have done. Otherwise the declaration

```
CVector c;
```

included in **main()** would not be valid.

Anyway, I have to warn you that a *no-op* block is not a recommended implementation for a

constructor, since it does not fulfill the minimum functionality that a constructor should have, which is the initialization of all the variables in the class. In our case this constructor leaves variables **x** and **y** undefined. Therefore, a more advisable declaration would have been something similar to this:

```
CVector () { x=0; y=0; };
```

that for simplicity I have not included in the code.

As well as a class includes by default an empty and a copy constructor, it also includes a default definition for the **assignment operator (=)** between two classes of the same type. This copies the whole content of the non-static data members of the parameter object (the one at the right side of the sign) to the one at the left side. Of course, you can redefine it to any other functionality that you want for this operator, like for example, copy only certain class members.

The overload of operators does not force its operation to bear a relation to the mathematical or usual meaning of the operator, although it is recommended. For example, it is not very logical to use operator **+** to subtract two classes or operator **==** to fill with zeros a class, although it is perfectly possible to do so.

Although the prototype of a function **operator+** can seem obvious since it takes the right side of the operator as the parameter for the function **operator+** of the left side object, other operators are not so clear. Here you have a table with a summary on how the different **operator** functions must be declared (replace @ by the operator in each case):

| Expression | Operator (@)                                           | Function member        | Global function   |
|------------|--------------------------------------------------------|------------------------|-------------------|
| @a         | + - * & ! ~ ++<br>--                                   | A::operator@()         | operator@(A)      |
| a@         | ++ --                                                  | A::operator@(int)      | operator@(A, int) |
| a@b        | + - * / % ^ &<br>  < > == != <=<br>>= << >> &&   <br>, | A::operator@(B)        | operator@(A, B)   |
| a@b        | = += -= *= /=<br>%= ^= &=  =<br><<= >>= [ ]            | A::operator@(B)        | -                 |
| a(b, c...) | ()                                                     | A::operator()(B, C...) | -                 |
| a->b       | ->                                                     | A::operator->()        | -                 |

\* where **a** is an object of class **A**, **b** is an object of class **B** and **c** is an object of class **C**.

You can see in this panel that there are two ways to overload some class operators: as *member function* and as *global function*. Its use is indistinct, nevertheless I remind you that functions that are not members of a class cannot access the **private** or **protected** members of the class unless the global function is *friend* of the class (friend is explained later).

## The keyword **this**

The keyword **this** represents within a class the address in memory of the object of that class that is being executed. It is a pointer whose value is always the address of the object.

It can be used to check if a parameter passed to a member function of an object is the object itself. For example,

```
// this
#include <iostream.h>

class CDummy {
public:
 int isitme (CDummy& param);
};

int CDummy::isitme (CDummy& param)
{
 if (¶m == this) return 1;
 else return 0;
}

int main () {
 CDummy a;
 CDummy* b = &a;
 if (b->isitme(a))
 cout << "yes, &a is b";
 return 0;
}
```

yes, &a is b

It is also frequently used in **operator=** member functions that return objects by reference (avoiding the use of temporary objects). Following with the vector's examples seen before we could have written an **operator=** function like this:

```
CVector& CVector::operator= (const CVector& param)
{
 x=param.x;
 y=param.y;
 return *this;
}
```

In fact this is a probable default code generated for the class if we include no **operator=** member function.

## Static members

A class can contain static members, either data or functions.

Static data members of a class are also known as "class variables", because their content does not depend on any object. There is only one unique value for all the objects of that same class.

For example, it may be used for a variable within a class that can contain the number of objects of that class that have been declared, as in the following example:

```
// static members in classes
#include <iostream.h>

class CDummy {
public:
 static int n;
```

7  
6

```

 CDummy () { n++; };
 ~CDummy () { n--; };
};

int CDummy::n=0;

int main () {
 CDummy a;
 CDummy b[5];
 CDummy * c = new CDummy;
 cout << a.n << endl;
 delete c;
 cout << CDummy::n << endl;
 return 0;
}

```

In fact, static members have the same properties as global variables but they enjoy class scope. For that reason, and to avoid that they may be declared several times, according to ANSI-C++ standard, we can only include the prototype (declaration) in the class declaration but not the definition (initialization). In order to initialize a static data-member we must include a formal definition outside the class, in the global scope, as in the previous example.

Because it is a unique variable for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

```

cout << a.n;
cout << CDummy::n;

```

These two calls included in the previous example are referring to the same variable: the static variable `n` within class `CDummy`.

Once again, I remind you that in fact it is a global variable. The only difference is its name outside the class.

Just as we may include static data within a class, we can also include static functions. They represent the same: they are global functions that are called as if they were object members of a given class. They can only refer to static data, in no case to nonstatic members of the class, as well as they do not allow the use of the keyword `this`, since it makes reference to an object pointer and these functions in fact are not members of any object but direct members of the class.

### Section 4.3

## Relationships between classes



## Friend functions (`friend` keyword)

In the previous section we have seen that there were three levels of internal protection for the different members of a class: **public**, **protected** and **private**. In the case of members *protected* and *private*, these could not be accessed from outside the same class at which they are declared. Nevertheless, this rule can be transgressed with the use of the *friend* keyword

in a class, so we can allow an external function to gain access to the **protected** and **private** members of a class.

In order to allow an external function to have access to the **private** and **protected** members of a class we have to declare the prototype of the external function that will gain access preceded by the keyword **friend** within the class declaration that shares its members. In the following example we declare the friend function **duplicate**:

```
// friend functions
#include <iostream.h>

class CRectangle {
 int width, height;
public:
 void set_values (int, int);
 int area (void) {return (width *
height);}
 friend CRectangle duplicate
(CRectangle);
};

void CRectangle::set_values (int a,
int b) {
 width = a;
 height = b;
}

CRectangle duplicate (CRectangle
rectparam)
{
 CRectangle rectres;
 rectres.width = rectparam.width*2;
 rectres.height =
rectparam.height*2;
 return (rectres);
}

int main () {
 CRectangle rect, rectb;
 rect.set_values (2,3);
 rectb = duplicate (rect);
 cout << rectb.area();
}
```

24

From within the **duplicate** function, that is a friend of **CRectangle**, we have been able to access the members **width** and **height** of different objects of type **CRectangle**. Notice that neither in the declaration of **duplicate()** nor in its later use in **main()** have we considered **duplicate** as a member of class **CRectangle**. It isn't.

The friend functions can serve, for example, to conduct operations between two different classes. Generally the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to make the process. Such as in the previous example, it would have been shorter to integrate **duplicate()** within the class **CRectangle**.

## Friend classes (**friend**)



Just as we have the possibility to define a friend function, we can also define a class as friend of another one, allowing that the second one access to the **protected** and **private** members of the first one.

```
// friend class
#include <iostream.h>

class CSquare;

class CRectangle {
 int width, height;
public:
 int area (void)
 {return (width * height);}
 void convert (CSquare a);
};

class CSquare {
 private:
 int side;
 public:
 void set_side (int a)
 {side=a;}
 friend class CRectangle;
};

void CRectangle::convert (CSquare a)
{
 width = a.side;
 height = a.side;
}

int main () {
 CSquare sqr;
 CRectangle rect;
 sqr.set_side(4);
 rect.convert(sqr);
 cout << rect.area();
 return 0;
}
```

16

In this example we have declared **CRectangle** as a friend of **CSquare** so that **CRectangle** can access the **protected** and **private** members of **CSquare**, more concretely **CSquare::side**, that defines the square side width.

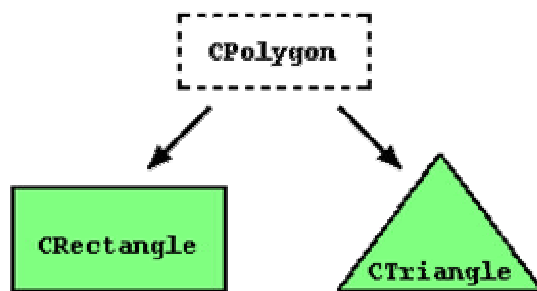
You may also see something new in the first instruction of the program, that is the empty prototype of class **CSquare**. This is necessary because within the declaration of **CRectangle** we refer to **CSquare** (as a parameter in **convert()**). The definition of **CSquare** is included later, so if we did not include a previous definition for **CSquare** this class would not be visible from within the definition of **CRectangle**.

Consider that friendships are not corresponded if we do not explicitly specify it. In our **CSquare** example **CRectangle** is considered as a class friend, but **CRectangle** does not do the proper thing with **CSquare**, so **CRectangle** can access to the **protected** and **private** members of **CSquare** but not the reverse way. Although nothing prevents us from declaring **CSquare** as a friend of **CRectangle**.

## Inheritance between classes

An important feature of classes is inheritance. This allows us to create an object derived from another one, so that it may include some of the other's members plus its own. For example, we are going to suppose that we want to declare a series of classes that describe polygons like our **CRectangle**, or **CTriangle**. They have certain common features, such as both can be described by means of only two sides: height and base.

This could be represented in the world of classes with a class **CPolygon** from which we would derive the two referred ones, **CRectangle** and **CTriangle**.



The class **CPolygon** would contain members that are common for all polygons. In our case: **width** and **height**. And **CRectangle** and **CTriangle** would be its derived classes.

Classes derived from others inherit all the visible members of the base class. That means that if a base class includes a member **A** and we derive it to another class with another member called **B**, the derived class will contain both **A** and **B**.

In order to derive a class from another, we must use the operator **:** (colon) in the declaration of the derived class in the following way:

```
class derived_class_name: public base_class_name;
```

where *derived\_class\_name* is the name of the *derived* class and *base\_class\_name* is the name of the class on which it is based. **public** may be replaced by any of the other access specifiers **protected** or **private**, and describes the access for the inherited members, as we will see right after this example:

```
// derived classes
#include <iostream.h>

class CPolygon {
protected:
 int width, height;
public:
 void set_values (int a, int b)
 { width=a; height=b;}
};

class CRectangle: public CPolygon {
public:
 int area (void)
 { return (width * height); }
};

class CTriangle: public CPolygon {
public:
```

```
20
10
```

```

 int area (void)
 { return (width * height / 2);
 }
};

int main () {
 CRectangle rect;
 CTriangle trgl;
 rect.set_values (4,5);
 trgl.set_values (4,5);
 cout << rect.area() << endl;
 cout << trgl.area() << endl;
 return 0;
}

```

As you may see, objects of classes **CRectangle** and **CTriangle** each contain members of **CPolygon**, that are: **width**, **height** and **set\_values()**.

The **protected** specifier is similar to **private**, its only difference occurs when deriving classes. When we derive a class, **protected** members of the base class can be used by other members of the derived class, nevertheless **private** member cannot. Since we wanted **width** and **height** to have the ability to be manipulated by members of the derived classes **CRectangle** and **CTriangle** and not only by members of **CPolygon**, we have used **protected** access instead of **private**.

We can summarize the different access types according to whom can access them in the following way:

| Access                     | public | protected | private |
|----------------------------|--------|-----------|---------|
| members of the same class  | yes    | yes       | yes     |
| members of derived classes | yes    | yes       | no      |
| not-members                | yes    | no        | no      |

where "*not-members*" represent any reference from outside the class, such as from **main()**, from another class or from any function, either global or local.

In our example, the members inherited by **CRectangle** and **CTriangle** follow with the same access permission as in the base class **CPolygon**:

```

CPolygon::width // protected access
CRectangle::width // protected access

CPolygon::set_values() // public access
CRectangle::set_values() // public access

```

This is because we have derived a class from the other as **public**, remember:

```
class CRectangle: public CPolygon;
```

this **public** keyword represents the minimum level of protection that the inherited members of the base class (**CPolygon**) must acquire in the new class (**CRectangle**). The minimum access level for the inherited members can be changed by specifying **protected** or **private** instead of **public**. For example, **daughter** is a class derived from **mother** that we defined thus:

```
class daughter: protected mother;
```

this would establish **protected** as the minimum access level for the members of **daughter** that it inherited from **mother**. That is, all members that were **public** in **mother** would become **protected** in **daughter**, that would be the minimum level at which they can be inherited. Of course, this would not restrict that **daughter** could have its own **public** members. The minimum level would only be established for the inherited members of **mother**.

The most common use of an inheritance level different from **public** is **private** that serves to completely encapsulate the base class, since, in that case, nobody except its own class will be able to access the members of the base class from which it is derived. Anyway, in most cases classes are derived as **public**.

If no access level is explicitly written **private** is assumed for classes created with the **class** keyword and **public** for those created with **struct**.

## What is inherited from the base class?

In principle every member of a base class is inherited by a derived class except:

- **Constructor and destructor**
- **operator=( ) member**
- **friends**

Although the constructor and destructor of the base class are not inherited, the default constructor (i.e. constructor with no parameters) and the destructor of the base class are always called when a new object of a derived class is created or destroyed.

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:

```
derived_class_name (parameters) : base_class_name (parameters) {}
```

For example:

```
// constructors and derivated
classes
#include <iostream.h>

class mother {
public:
 mother ()
 { cout << "mother: no
parameters\n"; }
 mother (int a)
 { cout << "mother: int
parameter\n"; }
};

class daughter : public mother {
public:
 daughter (int a)
 { cout << "daughter: int
parameter\n\n"; }
};
```

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

```

class son : public mother {
public:
 son (int a) : mother (a)
 { cout << "son: int
parameter\n\n"; }
};

int main () {
 daughter cynthia (1);
 son daniel(1);

 return 0;
}

```

Observe the difference between which mother's constructor is called when a new **daughter** object is created and which when it is a **son** object. The difference is because the constructor declaration of **daughter** and **son**:

```

daughter (int a) // nothing specified: call default constructor
son (int a) : mother (a) // constructor specified: call this one

```

## Multiple inheritance

In C++ it is perfectly possible that a class inherits fields and methods from more than one class simply by separating the different base classes with commas in the declaration of the derived class. For example, if we had a specific class to print on screen (**COutput**) and we wanted that our classes **CRectangle** and **CTriangle** also inherit its members in addition to those of **CPolygon** we could write:

```

class CRectangle: public CPolygon, public COutput {
class CTriangle: public CPolygon, public COutput {

```

here is the complete example:

```

// multiple inheritance
#include <iostream.h>

class CPolygon {
protected:
 int width, height;
public:
 void set_values (int a, int b)
 { width=a; height=b;}
};

class COutput {
public:
 void output (int i);
};

void COutput::output (int i) {
 cout << i << endl;
}

class CRectangle: public CPolygon,
public COutput {
public:
 int area (void)
 { return (width * height); }
}

```

20  
10

```

};

class CTriangle: public CPolygon,
public COutput {
public:
 int area (void)
 { return (width * height / 2); }
};

int main () {
 CRectangle rect;
 CTriangle trgl;
 rect.set_values (4,5);
 trgl.set_values (4,5);
 rect.output (rect.area());
 trgl.output (trgl.area());
 return 0;
}

```

## Section 4.4

# Polymorphism



For a suitable understanding of this section you should clearly know how to use **pointers** and **inheritance between classes**. I recommend that if some of these expressions seem strange to you, you review the indicated sections:

```

int a::b(c) {}; // Classes (Section 4.1)
a->b // pointers and objects (Section 4.2)
class a: public b; // Relationships between classes (Section 4.3)

```

## Pointers to base class

One of the greater advantages of deriving classes is that **a pointer to a derived class is type-compatible with a pointer to its base class**. This section is fully dedicated to taking advantage of this powerful C++ feature. For example, we are going to rewrite our program about the rectangle and the triangle of the previous section considering this property:

```

// pointers to base class
#include <iostream.h>

class CPolygon {
protected:
 int width, height;
public:
 void set_values (int a, int b)
 { width=a; height=b; }
};

class CRectangle: public CPolygon {
public:
 int area (void)
 { return (width * height); }
};

```

20  
10

```

class CTriangle: public CPolygon {
public:
 int area (void)
 { return (width * height / 2);
 }
};

int main () {
 CRectangle rect;
 CTriangle trgl;
 CPolygon * ppoly1 = ▭
 CPolygon * ppoly2 = &trgl;
 ppoly1->set_values (4,5);
 ppoly2->set_values (4,5);
 cout << rect.area() << endl;
 cout << trgl.area() << endl;
 return 0;
}

```

The function **main** creates two pointers that point to objects of class **CPolygon**, that are **\*ppoly1** and **\*ppoly2**. These are assigned to the addresses of **rect** and **trgl**, and because they are objects of classes derived from **CPolygon** they are valid assignments.

The only limitation of using **\*ppoly1** and **\*ppoly2** instead of **rect** and **trgl** is that both **\*ppoly1** and **\*ppoly2** are of type **CPolygon\*** and therefore we can only refer to the members that **CRectangle** and **CTriangle** inherit from **CPolygon**. For that reason when calling the **area()** members we have not been able to use the pointers **\*ppoly1** and **\*ppoly2**.

To make it possible for the pointers to class **CPolygon** to admit **area()** as a valid member, this should also have been declared in the base class and not only in its derived ones. (see the following section).

## Virtual members

In order to declare an element of a class which we are going to redefine in derived classes we must precede it with the keyword **virtual** so that the use of pointers to objects of that class can be suitable.

Take a look at the following example:

```

// virtual members
#include <iostream.h>

class CPolygon {
protected:
 int width, height;
public:
 void set_values (int a, int b)
 { width=a; height=b; }
 virtual int area (void)
 { return (0); }
};

class CRectangle: public CPolygon {
public:

```

```

20
10
0

```

```

 int area (void)
 { return (width * height); }
 };

class CTriangle: public CPolygon {
public:
 int area (void)
 { return (width * height / 2); }
};

int main () {
 CRectangle rect;
 CTriangle trgl;
 CPolygon poly;
 CPolygon * ppoly1 = ▭
 CPolygon * ppoly2 = &trgl;
 CPolygon * ppoly3 = &poly;
 ppoly1->set_values (4,5);
 ppoly2->set_values (4,5);
 ppoly3->set_values (4,5);
 cout << ppoly1->area() << endl;
 cout << ppoly2->area() << endl;
 cout << ppoly3->area() << endl;
 return 0;
}

```

Now the three classes (**CPolygon**, **CRectangle** and **CTriangle**) have the same members: **width**, **height**, **set\_values()** and **area()**.

**area()** has been defined as **virtual** because it is later redefined in derived classes. You can verify if you want that if you remove this word (**virtual**) from the code and then you execute the program the result will be **0** for the three polygons instead of **20,10,0**. That is because instead of calling the corresponding **area()** function for each object (**CRectangle::area()**, **CTriangle::area()** and **CPolygon::area()**, respectively), **CPolygon::area()** will be called for all of them since the calls are via a pointer to **CPolygon**.

Therefore what the word **virtual** does is to allow that a member of a derived class with the same name as one in the base class be suitably called when a pointer to it is used, as in the above example.

Note that in spite of its virtuality we have also been able to declare an object of type **CPolygon** and to call its **area()** function, that always returns **0** as the result.

## Abstract base classes

Basic abstract classes are something very similar to the class **CPolygon** of our previous example. The only difference is that in our previous example we have defined a valid **area()** function for objects that were of class **CPolygon** (like object **poly**), whereas in an *abstract base class* we could have simply left without defining this function by appending **=0** (equal to zero) to the function declaration.

The class **CPolygon** could have been thus:

```
// abstract class CPolygon
```



```

class CPolygon {
protected:
 int width, height;
public:
 void set_values (int a, int b)
 { width=a; height=b; }
 virtual int area (void) =0;
};

```

Notice how we have appended **=0** to **virtual int area (void)** instead of specifying an implementation for the function. This type of function is called a *pure virtual function*, and all classes that contain a *pure virtual function* are considered *abstract base classes*.

The greatest difference of an abstract base class is that instances (objects) of it cannot be created, but we can create pointers to them. Therefore a declaration like:

**CPolygon poly;**

would be incorrect for the abstract base class declared above. Nevertheless the pointers:

```

CPolygon * ppoly1;
CPolygon * ppoly2

```

are perfectly valid. This is because the *pure virtual function* that it includes is not defined and it is impossible to create an object if it does not have all its members defined. Nevertheless a pointer that points to an object of a derived class where this function has been defined is perfectly valid.

Here you have the complete example:

```

// virtual members
#include <iostream.h>

class CPolygon {
protected:
 int width, height;
public:
 void set_values (int a, int b)
 { width=a; height=b; }
 virtual int area (void) =0;
};

class CRectangle: public CPolygon {
public:
 int area (void)
 { return (width * height); }
};

class CTriangle: public CPolygon {
public:
 int area (void)
 { return (width * height / 2); }
};

int main () {
 CRectangle rect;
 CTriangle trgl;
 CPolygon * ppoly1 = ▭
 CPolygon * ppoly2 = &trgl;
}

```

20  
10

```

 ppoly1->set_values (4,5);
 ppoly2->set_values (4,5);
 cout << ppoly1->area() << endl;
 cout << ppoly2->area() << endl;
 return 0;
}

```

If you review the program you will notice that we can refer to objects of different classes using a unique type of pointer (**CPolygon\***). This can be tremendously useful. Imagine, now we can create a function member of **CPolygon** that is able to print on screen the result of the **area()** function independently of what the derived classes are.

```

// virtual members
#include <iostream.h>

class CPolygon {
protected:
 int width, height;
public:
 void set_values (int a, int b)
 { width=a; height=b; }
 virtual int area (void) =0;
 void printarea (void)
 { cout << this->area() <<
endl; }
};

class CRectangle: public CPolygon {
public:
 int area (void)
 { return (width * height); }
};

class CTriangle: public CPolygon {
public:
 int area (void)
 { return (width * height / 2); }
};

int main () {
 CRectangle rect;
 CTriangle trgl;
 CPolygon * ppoly1 = ▭
 CPolygon * ppoly2 = &trgl;
 ppoly1->set_values (4,5);
 ppoly2->set_values (4,5);
 ppoly1->printarea();
 ppoly2->printarea();
 return 0;
}

```

```

20
10

```

Remember that **this** represents a pointer to the object whose code is being executed.

Abstract classes and virtual members grant to C++ the polymorphic characteristics that make object-oriented programming such a useful instrument. Of course we have seen the simplest way to use these features, but imagine these features applied to arrays of objects or objects assigned through dynamic memory.



Templates are a new feature introduced by ANSI-C++ standard. If you use a C++ compiler that is not adapted to this standard it is possible that you cannot use them.

## Function templates

Templates allow to create generic functions that admit any data type as parameters and return a value without having to overload the function with all the possible data types. Until certain point they fulfill the functionality of a macro. Its prototype is any of the two following ones:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

the only difference between both prototypes is the use of keyword **class** or **typename**, its use is indistinct since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class GenericType>
GenericType GetMax (GenericType a, GenericType b) {
 return (a>b?a:b);
}
```

As the first line specifies, we have created a template for a generic data type that we have called **GenericType**. Therefore in the function that follows, **GenericType** becomes a valid data type and it is used as the type for its two parameters **a** and **b** and as the return type for the function **GetMax**.

**GenericType** still does not represent any concrete data type; when the function **GetMax** will be called we will be able to call it with any valid data type. This data type will serve as a *pattern* and will replace **GenericType** in the function. The way to call a template class with a type pattern is the following:

```
function <pattern> (parameters);
```

Thus, for example, to call **GetMax** and to compare two integer values of type **int** we can write:

```
int x,y;
GetMax <int> (x,y);
```

so **GetMax** will be called as if each appearance of **GenericType** was replaced by an **int** expression.

Here is the complete example:

```
// function template
#include <iostream.h>

template <class T>
T GetMax (T a, T b) {
 T result;
```

```
6
10
```

```

 result = (a>b)? a : b;
 return (result);
}

int main () {
 int i=5, j=6, k;
 long l=10, m=5, n;
 k=GetMax<int>(i,j);
 n=GetMax<long>(l,m);
 cout << k << endl;
 cout << n << endl;
 return 0;
}

```

(In this case we have called the generic type **T** instead of **GenericType** because it is shorter and in addition is one of the most usual identifiers used for templates, although it is valid to use any valid identifier).

In the example above we used the same function **GetMax()** with arguments of type **int** and **long** having written a single implementation of the function. That is to say, we have written a function template and called it with two different patterns.

As you can see, within our **GetMax()** template function the type **T** can be used to declare new objects:

**T result;**

**result** is an object of type **T**, like **a** and **b**, that is to say, of the type that we enclose between angle-brackets **<>** when calling our template function.

In this concrete case where the generic **T** type is used as a parameter for function **GetMax** the compiler can find out automatically which data type is passed to it without having to specify it with patterns **<int>** or **<long>**. So we could have written:

```

int i,j;
GetMax (i,j);

```

since both **i** and **j** are of type **int** the compiler would assume automatically that the wished function is for type **int**. This implicit method is more usual and would produce the same result:

```

// function template II
#include <iostream.h>

template <class T>
T GetMax (T a, T b) {
 return (a>b?a:b);
}

int main () {
 int i=5, j=6, k;
 long l=10, m=5, n;
 k=GetMax(i,j);
 n=GetMax(l,m);
 cout << k << endl;
 cout << n << endl;
 return 0;
}

```

```

6
10

```

Notice how in this case, within function `main()` we called our template function `GetMax()` without explicitly specifying the type between angle-brackets `<>`. The compiler automatically determines what type is needed on each call.

Because our template function includes only one data type (`class T`) and both arguments it admits are both of that same type, we cannot call our template function with two objects of different types as parameters:

```
int i;
long l;
k = GetMax (i,l);
```

This would be incorrect, since our function waits for two arguments of the same type (or class).

We can also make template-functions that admit more than one generic class or data type. For example:

```
template <class T, class U>
T GetMin (T a, U b) {
 return (a<b?a:b);
}
```

In this case, our template function `GetMin()` admits two parameters of different types and returns an object of the same type as the first parameter (`T`) that is passed. For example, after that declaration we could call the function by writing:

```
int i,j;
long l;
i = GetMin<int,long> (j,l);
```

or simply

```
i = GetMin (j,l);
```

even though `j` and `l` are of different types.

## Class templates

We also have the possibility to write class templates, so that a class can have members based on generic types that do not need to be defined at the moment of creating the class or whose members use these generic types. For example:

```
template <class T>
class pair {
 T values [2];
public:
 pair (T first, T second)
 {
 values[0]=first; values[1]=second;
 }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type `int` with the values `115` and `36` we would write:

```
pair<int> myobject (115, 36);
```

this same class would also serve to create an object to store any other type:

```
pair<float> myfloats (3.0, 2.18);
```

The only member function has been defined *inline* within the class declaration. If we define a function member outside the declaration we must always precede the definition with the prefix `template <... >`.

```
// class templates
#include <iostream.h>

template <class T>
class pair {
 T value1, value2;
public:
 pair (T first, T second)
 {value1=first; value2=second;}
 T getmax ();
};

template <class T>
T pair<T>::getmax ()
{
 T retval;
 retval = value1>value2? value1 :
value2;
 return retval;
}

int main () {
 pair <int> myobject (100, 75);
 cout << myobject.getmax();
 return 0;
}
```

100

notice how the definition of member function **getmax** begins:

```
template <class T>
T pair<T>::getmax ()
```

All **T**s that appear are necessary because whenever you declare member functions you have to follow a format similar to this (the second **T** makes reference to the type returned by the function, so this may vary).

## Template specialization

A template specialization allows a template to make specific implementations when the pattern is of a determined type. For example, suppose that our class template **pair** included a function to return the result of the module operation between the objects contained in it, but we only want it to work when the contained type is **int**. For the rest of the types we want this function to return **0**. This can be done the following way:

```
// Template specialization
#include <iostream.h>

template <class T>
class pair {
 T value1, value2;
public:
 pair (T first, T second)
 {value1=first; value2=second;}
 T module () {return 0;}
};

template <>
class pair <int> {
 int value1, value2;
public:
```

25  
0

```

 pair (int first, int second)
 {value1=first; value2=second;}
 int module ();
};

template <>
int pair<int>::module() {
 return value1%value2;
}

int main () {
 pair <int> myints (100,75);
 pair <float> myfloats
(100.0,75.0);
 cout << myints.module() << '\n';
 cout << myfloats.module() << '\n';
 return 0;
}

```

As you can see in the code the specialization is defined this way:

```
template <> class class_name <type>
```

The specialization is part of a template, for that reason we must begin the declaration with **template <>**. And indeed because it is a specialization for a concrete type, the generic type cannot be used in it and the first angle-brackets <> must appear empty. After the class name we must include the type that is being specialized enclosed between angle-brackets <>.

When we specialize a type of a template we must also define all the members equating them to the specialization (if one pays attention, in the example above we have had to include its own constructor, although it is identical to the one in the generic template). The reason is that no member is "inherited" from the generic template to the specialized one.

## Parameter values for templates

Besides the template arguments preceded by the **class** or **typename** keywords that represent a type, function templates and class templates can include other parameters that are not types whenever they are also constant values, like for example values of fundamental types. As an example see this class template that serves to store arrays:

```

// array template
#include <iostream.h>

template <class T, int N>
class array {
 T memblock [N];
public:
 void setmember (int x, T value);
 T getmember (int x);
};

template <class T, int N>
array<T,N>::setmember (int x, T
value) {
 memblock[x]=value;
}

```

```

100
3.1416

```

```

template <class T, int N>
T array<T,N>::getmember (int x) {
 return memblock[x];
}

int main () {
 array <int,5> myints;
 array <float,5> myfloats;
 myints.setmember (0,100);
 myfloats.setmember (3,3.1416);
 cout << myints.getmember(0) <<
'\n';
 cout << myfloats.getmember(3) <<
'\n';
 return 0;
}

```

It is also possible to set default values for any template parameter just as it is done with function parameters.

Some possible template examples seen above:

```

template <class T> // The most usual: one class parameter.
template <class T, class U> // Two class parameters.
template <class T, int N> // A class and an integer.
template <class T = char> // With a default value.
template <int Tfunc (int)> // A function as parameter.

```

## Templates and multiple-file projects

From the point of view of the compiler, templates are not normal functions or classes. They are compiled on demand, meaning that the code of a template function is not compiled until an instantiation is required. At that moment, when an instantiation is required, the compiler generates a function specifically for that type from the template.

When projects grow it is usual to split the code of a program in different source files. In these cases, generally the interface and implementation are separated. Taking a library of functions as example, the interface generally consists of the prototypes of all the functions that can be called. These are generally declared in a "header file" with `.h` extension, and the implementation (the definition of these functions) is in an independent file of `c++` code.

The macro-like functionality of templates, forces a restriction for multi-file projects: the implementation (definition) of a template class or function must be in the same file as the declaration. That means we cannot separate the interface in a separate header file and we must include both interface and implementation in any file that uses the templates.

Going back to the library of functions, if we wanted to make a library of function templates, instead of creating a header file (`.h`) we should create a "template file" with both the interface and implementation of the function templates (there is no convention on the extension for this type of file other than there be no extension at all or to keep the `.h`). The inclusion more than once of the same template file with both declarations and definitions in a project doesn't generate linkage errors, since they are compiled on demand and compilers that allow templates should be prepared to not generate duplicate code in these cases.



Namespaces allow us to group a set of global classes, objects and/or functions under a name. To say it another way, they serve to split the global scope in sub-scopes known as *namespaces*.

The form to use *namespaces* is:

```
namespace identifier
{
 namespace-body
}
```

Where *identifier* is any valid identifier and *namespace-body* is the set of classes, objects and functions that are included within the *namespace*. For example:

```
namespace general
{
 int a, b;
}
```

In this case, **a** and **b** are normal variables integrated within the **general** *namespace*. In order to access these variables from outside the namespace we have to use the scope operator `::`. For example, to access the previous variables we would have to put:

```
general::a
general::b
```

The functionality of *namespaces* is specially useful in case there is a possibility that a global object or function has the same name as another one, causing a redefinition error. For example:

```
// namespaces
#include <iostream.h>

namespace first
{
 int var = 5;
}

namespace second
{
 double var = 3.1416;
}

int main () {
 cout << first::var << endl;
 cout << second::var << endl;
 return 0;
}
```

```
5
3.1416
```

In this case two global variables with the **var** name exist, one defined within *namespace first* and another one in *second*. No redefinition errors thanks to *namespaces*.

## using namespace

The **using** directive followed by **namespace** serves to associate the present nesting level with a certain *namespace* so that the objects and functions of that *namespace* can be accessible directly as if they were defined in the global scope. Its utilization follows this prototype:

**using namespace identifier;**

Thus, for example:

```
// using namespace example
#include <iostream.h>

namespace first
{
 int var = 5;
}

namespace second
{
 double var = 3.1416;
}

int main () {
 using namespace second;
 cout << var << endl;
 cout << (var*2) << endl;
 return 0;
}
```

```
3.1416
6.2832
```

In this case we have been able to use **var** without having to precede it with any scope operator.

You have to consider that the sentence **using namespace** has validity only in the block in which it is declared (understanding as a block the group of instructions within key brackets **{}**) or in all the code if it is used in the global scope. For example, if we had intention to first use the objects of a *namespace* and then those of another one we could do something similar to:

```
// using namespace example
#include <iostream.h>

namespace first
{
 int var = 5;
}

namespace second
{
 double var = 3.1416;
}

int main () {
 {
 using namespace first;
 cout << var << endl;
 }
 {
 using namespace second;
 cout << var << endl;
 }
 return 0;
}
```

```
5
3.1416
```

## alias definition

We have the possibility to define alternative names for *namespaces* that already exist. The form to do it is:

```
namespace new_name = current_name ;
```

## Namespace std

One of the best examples that we can find about *namespaces* is the standard C++ library itself. As defined in the ANSI C++ standard, all the classes, objects and functions of the standard C++ library are defined within *namespace std*.

You may have noticed that we have ignored this rule all through this tutorial. I've decided to do so since this rule is almost as recent as the ANSI standard itself (1997) and many older compilers do not comply with this rule.

Almost all compilers, even those complying with ANSI standard, allow the use of the traditional header files (like `iostream.h`, `stdlib.h`, etc), the ones we have used throughout this tutorial. Nevertheless, the ANSI standard has completely redesigned these libraries taking advantage of the templates feature and following the rule to declare all the functions and variables under the namespace `std`.

The standard has specified new names for these "header" files, basically using the same name for C++ specific files, but without the ending `.h`. For example, `iostream.h` becomes `iostream`.

If we use the ANSI-C++ compliant include files we have to bear in mind that all the functions, classes and objects will be declared under the `std` namespace. For example:

```
// ANSI-C++ compliant hello world
#include <iostream>

int main () {
 std::cout << "Hello world in ANSI-
C++\n";
 return 0;
}
```

Hello world in ANSI-C++

Although it is more usual to use `using namespace` and save us to have to use the scope operator `::` before all the references to standard objects:

```
// ANSI-C++ compliant hello world
(II)
#include <iostream>
using namespace std;

int main () {
 cout << "Hello world in ANSI-
C++\n";
 return 0;
}
```

Hello world in ANSI-C++

The name for the C files has also suffered some changes. You can find more information on the new names for the standard header files in the document [Standard header files](#).

The use of the ANSI-compliant way to include the standard libraries, apart for the ANSI-compliance itself, is highly recommended for STL users.

## Section 5.3

# Exception handling



Exception handling explained in this section is a new feature introduced by ANSI-C++ standard. If you use a C++ compiler that is not adapted to this standard it is possible that you cannot use this feature.

During the development of a program, there may be some cases where we do not have the certainty that a piece of the code is going to work right, either because it accesses resources that do not exist or because it gets out of an expected range, etc...

These types of anomalous situations are included in what we consider exceptions and C++ has recently incorporated three new operators to help us handle these situations: **try**, **throw** and **catch**.

Their form of use is the following:

```
try {
 // code to be tried
 throw exception;
}
catch (type exception)
{
 // code to be executed in case of exception
}
```

And its operation:

- The code within the **try** block is executed normally. In case that an exception takes place, this code must use the **throw** keyword and a parameter to throw an exception. The type of the parameter details the exception and can be of any valid type.
- If an exception has taken place, that is to say, if it has executed a **throw** instruction within the **try** block, the **catch** block is executed receiving as parameter the exception passed by **throw**.

For example:

```
// exceptions
#include <iostream.h>

int main () {
 char myarray[10];
 try
 {
 for (int n=0; n<=10; n++)
 {
 if (n>9) throw "Out of range";
 myarray[n]='z';
 }
 }
}
```

Exception: Out of range

```

 }
}
catch (char * str)
{
 cout << "Exception: " << str <<
endl;
}
return 0;
}

```

In this example, if within the `n` loop, `n` gets to be more than 9 an exception is thrown, since `myarray[n]` would in that case point to a non-trustworthy memory address. When `throw` is executed, the `try` block finalizes right away and every object created within the `try` block is destroyed. After that, the control is passed to the corresponding `catch` block (that is only executed in these cases). Finally the program continues right after the `catch` block, in this case: `return 0;`.

The syntax used by `throw` is similar to that of `return`: Only the parameter does not need to be enclosed between parenthesis.

The `catch` block must go right after the `try` block without including any code line between them. The parameter that `catch` accepts can be of any valid type. Even more, `catch` can be overloaded so that it can accept different types as parameters. In that case the `catch` block executed is the one that matches the type of the exception sent (the parameter of `throw`):

```

// exceptions: multiple catch blocks
#include <iostream.h>

int main () {
 try
 {
 char * mystring;
 mystring = new char [10];
 if (mystring == NULL) throw
"Allocation failure";
 for (int n=0; n<=100; n++)
 {
 if (n>9) throw n;
 mystring[n]='z';
 }
 }
 catch (int i)
 {
 cout << "Exception: ";
 cout << "index " << i << " is
out of range" << endl;
 }
 catch (char * str)
 {
 cout << "Exception: " << str <<
endl;
 }
 return 0;
}

```

**Exception: index 10 is out of range**

In this case there is a possibility that at least two different exceptions could happen:

1. That the required block of 10 characters cannot be assigned (something rare, but possible): in this case an exception is thrown that will be caught by `catch (char *str)`.
2. That the maximum index for `mystring` is exceeded: in this case the exception thrown will be caught by `catch (int i)`, since the parameter is an integer number.

We can also define a `catch` block that captures all the exceptions independently of the type used in the call to `throw`. For that we have to write three points instead of the parameter type and name accepted by `catch`:

```
try {
 // code here
}
catch (...) {
 cout << "Exception occurred";
}
```

It is also possible to nest `try-catch` blocks within more external `try` blocks. In these cases, we have the possibility that an internal `catch` block forwards the exception received to the external level, for that the expression `throw;` with no arguments is used. For example:

```
try {
 try {
 // code here
 }
 catch (int n) {
 throw;
 }
}
catch (...) {
 cout << "Exception occurred";
}
```

## Exceptions not caught

If an exception is not caught by any `catch` statement because there is no catch statement with a matching type, the special function `terminate` will be called.

This function is generally defined so that it terminates the current process immediately showing an "Abnormal termination" error message. Its format is:

```
void terminate();
```

## Standard exceptions

Some functions of the standard C++ language library send exceptions that can be captured if we include them within a `try` block. These exceptions are sent with a class derived from `std::exception` as type. This class (`std::exception`) is defined in the C++ standard header file `<exception>` and serves as a pattern for the standard hierarchy of exceptions:

```
exception
├── bad_alloc (thrown by new)
├── bad_cast (thrown by dynamic_cast when fails with a referenced type)
├── bad_exception (thrown when an exception doesn't match any catch)
├── bad_typeid (thrown by typeid)
```

```

└─logic_error
└─domain_error
└─invalid_argument
└─length_error
└─out_of_range
└─runtime_error
└─overflow_error
└─range_error
└─underflow_error
or
ios_base::failure (thrown by ios::clear)

```

Because this is a class hierarchy, if you include a **catch** block to capture any of the exceptions of this hierarchy using the argument by reference (i.e. adding an ampersand & after the type) you will also capture all the derived ones (rules of inheritance in C++).

The following example catches an exception of type **bad\_typeid** (derived from **exception**) that is generated when requesting information about the type pointed by a null pointer:

```

// standard exceptions

#include <iostream.h>
#include <exception>
#include <typeinfo>

class A {virtual f() {} };

int main () {
 try {
 A * a = NULL;
 typeid (*a);
 }
 catch (std::exception& e)
 {
 cout << "Exception: " <<
e.what();
 }
 return 0;
}

```

Exception: Attempted typeid of NULL pointer

You can use the classes of standard hierarchy of exceptions to throw your exceptions or derive new classes from them.

## Section 5.4

# Advanced Class Type-casting



Until now, in order to type-cast a simple object to another we have used the traditional type casting operator. For example, to cast a floating point number of type **double** to an integer of type **int** we have used:

```
int i;
double d;
i = (int) d;
or also
i = int (d);
```

This is quite good for basic types that have standard defined conversions, however this operators can also be indiscriminately applied on classes and pointers to classes. So, it is perfectly valid to write things like:

```
// class type-casting
#include <iostream.h>

class CDummy {
 int i;
};

class CAddition {
 int x,y;
public:
 CAddition (int a, int b) {
x=a; y=b; }
 int result() { return x+y;}
};

int main () {
 CDummy d;
 CAddition * padd;
 padd = (CAddition*) &d;
 cout << padd->result();
 return 0;
}
```

Although the previous program is syntactically correct in C++ (in fact it will compile with no warnings on most compilers) it is code with not much sense since we use function **result**, that is a member of **CAddition**, without having declared an object of that class: **padd** is not an object, it is only a pointer which we have assigned the address of a non related object. When accessing its **result** member it will produce a run-time error or, at best, just an unexpected result.

In order to control these types of conversions between classes, ANSI-C++ standard has defined four new casting operators: **reinterpret\_cast**, **static\_cast**, **dynamic\_cast** and **const\_cast**. All of them have the same format when used:

```
reinterpret_cast <new_type> (expression)
dynamic_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)
```

Where *new\_type* is the destination type to which *expression* has to be casted. To make an easily understandable parallelism with traditional type-casting operators these expression mean:

```
(new_type) expression
new_type (expression)
```

but with their own special characteristics.

## **reinterpret\_cast**



**reinterpret\_cast** casts a pointer to any other type of pointer. It also allows casting from a pointer to an integer type and vice versa.

This operator can cast pointers between non-related classes. The operation results in a simple binary copy of the value from one pointer to the other. The content pointed to does not pass any kind of check nor transformation between types.

In the case that the copy is performed from a pointer to an integer, the interpretation of its content is system dependent and therefore any implementation is non-portable. A pointer casted to an integer large enough to fully contain it can be casted back to a valid pointer.

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

**reinterpret\_cast** treats all pointers exactly as traditional type-casting operators do.

## static\_cast

**static\_cast** performs any casting that can be implicitly performed as well as the inverse cast (even if this is not allowed implicitly).

Applied to pointers to classes, that is to say that it allows to cast a pointer of a derived class to its base class (this is a valid conversion that can be implicitly performed) and it can also perform the inverse: cast a base class to its derived class.

In this last case the base class that is being casted is not checked to determine whether this is a complete class of the destination type or not.

```
class Base {};
class Derived: public Base {};
Base * a = new Base;
Derived * b = static_cast<Derived*>(a);
```

**static\_cast**, aside from manipulating pointers to classes, can also be used to perform conversions explicitly defined in classes, as well as to perform standard conversions between fundamental types:

```
double d=3.14159265;
int i = static_cast<int>(d);
```

## dynamic\_cast

**dynamic\_cast** is exclusively used with pointers and references to objects. It allows any type-casting that can be implicitly performed as well as the inverse one when used with polymorphic classes, however, unlike **static\_cast**, **dynamic\_cast** checks, in this last case, if the operation is valid. That is to say, it checks if the casting is going to return a valid complete object of the requested type.

Checking is performed during run-time execution. If the pointer being casted is not a pointer to a valid complete object of the requested type, the value returned is a **NULL** pointer.

```
class Base { virtual dummy(){}; };
class Derived : public Base { };
```

```
Base* b1 = new Derived;
Base* b2 = new Base;
Derived* d1 = dynamic_cast<Derived*>(b1); // succeeds
Derived* d2 = dynamic_cast<Derived*>(b2); // fails: returns NULL
```

If the type-casting is performed to a reference type and this casting is not possible an *exception* of type **bad\_cast** is thrown:

```
class Base { virtual dummy(){}; };
class Derived : public Base { };

Base* b1 = new Derived;
Base* b2 = new Base;
Derived d1 = dynamic_cast<Derived*>(b1); // succeeds
Derived d2 = dynamic_cast<Derived*>(b2); // fails: exception thrown
```

## const\_cast

This type of casting manipulates the *const* attribute of the passed object, either to be set or removed:

```
class C {};
const C * a = new C;
C * b = const_cast<C*> (a);
```

Neither of the other three new **cast** operators can modify the constness of an object.

## typeid

ANSI-C++ also defines a new operator called **typeid** that allows checking the type of an expression:

**typeid** (*expression*)

this operator returns a reference to a constant object of type **type\_info** that is defined in the standard header file **<typeinfo>**. This returned value can be compared with another using operators **==** and **!=** or can serve to obtain a string of characters representing the data type or class name by using its **name()** method.

```
// typeid, typeinfo
#include <iostream.h>
#include <typeinfo>

class CDummy { };

int main () {
 CDummy* a,b;
 if (typeid(a) != typeid(b))
 {
 cout << "a and b are of
different types:\n";
 cout << "a is: " <<
typeid(a).name() << '\n';
 cout << "b is: " <<
typeid(b).name() << '\n';
 }
 return 0;
}
```

```
a and b are of different types:
a is: class CDummy *
b is: class CDummy
```

## Preprocessor directives

Preprocessor directives are orders that we include within the code of our programs that are not instructions for the program itself but for the preprocessor. The preprocessor is executed automatically by the compiler when we compile a program in C++ and is in charge of making the first verifications and digestions of the program's code.

All these directives must be specified in a single line of code and they do not have to include an ending semicolon ;.

### #define

At the beginning of this tutorial we have already spoken about a preprocessor directive:

**#define**, that serves to generate what we called *defined constants* or *macros* and whose form is the following:

```
#define name value
```

Its function is to define a macro called *name* that whenever it is found in some point of the code is replaced by *value*. For example:

```
#define MAX_WIDTH 100
char str1[MAX_WIDTH];
char str2[MAX_WIDTH];
```

It defines two strings to store up to 100 characters.

**#define** can also be used to generate macro functions:

```
#define getmax(a,b) a>b?a:b
int x=5, y;
y = getmax(x,2);
```

after the execution of this code **y** would contain 5.

### #undef

**#undef** fulfills the inverse functionality of **#define**. It eliminates from the list of defined constants the one that has the name passed as a parameter to **#undef**:

```
#define MAX_WIDTH 100
char str1[MAX_WIDTH];
#undef MAX_WIDTH
#define MAX_WIDTH 200
char str2[MAX_WIDTH];
```

### #ifdef, #ifndef, #if, #endif, #else and #elif

These directives allow to discard part of the code of a program if a certain condition is not fulfilled.

**#ifdef** allows that a section of a program is compiled only if the *defined constant* that is specified as the parameter has been defined, independently of its value. Its operation is:

```
#ifndef name
// code here
#endif
```

For example:

```
#ifdef MAX_WIDTH
char str[MAX_WIDTH];
#endif
```

In this case, the line `char str[MAX_WIDTH];` is only considered by the compiler if the *defined constant* `MAX_WIDTH` has been previously defined, independently of its value. If it has not been defined, that line will not be included in the program.

`#ifndef` serves for the opposite: the code between the `#ifndef` directive and the `#endif` directive is only compiled if the constant name that is specified has not been defined previously. For example:

```
#ifndef MAX_WIDTH
#define MAX_WIDTH 100
#endif
char str[MAX_WIDTH];
```

In this case, if when arriving at this piece of code the *defined constant* `MAX_WIDTH` has not yet been defined it would be defined with a value of 100. If it already existed it would maintain the value that it had (because the `#define` statement won't be executed).

The `#if`, `#else` and `#elif` (*elif* = *else if*) directives serve so that the portion of code that follows is compiled only if the specified condition is met. The condition can only serve to evaluate constant expressions. For example:

```
#if MAX_WIDTH>200
#undef MAX_WIDTH
#define MAX_WIDTH 200

#elif MAX_WIDTH<50
#undef MAX_WIDTH
#define MAX_WIDTH 50

#else
#undef MAX_WIDTH
#define MAX_WIDTH 100
#endif
```

```
char str[MAX_WIDTH];
```

Notice how the structure of chained directives `#if`, `#elif` and `#else` finishes with `#endif`.

## #line

When we compile a program and errors happen during the compiling process, the compiler shows the error that happened preceded by the name of the file and the line within the file where it has taken place.

The `#line` directive allows us to control both things, the line numbers within the code files as well as the file name that we want to appear when an error takes place. Its form is the following one:

```
#line number "filename"
```

Where *number* is the new line number that will be assigned to the next code line. The line number of successive lines will be increased one by one from this.

*filename* is an optional parameter that serves to replace the file name that will be shown in case of error from this directive until another one changes it again or the end of the file is reached. For example:

```
#line 1 "assigning variable"
int a?;
```

This code will generate an error that will be shown as error in file "assigning variable", line 1.

## #error

This directive aborts the compilation process when it is found returning the error that is specified as the parameter:

```
#ifndef __cplusplus
#error A C++ compiler is required
#endif
```

This example aborts the compilation process if the *defined constant* `__cplusplus` is not defined.

## #include

This directive has also been used assiduously in other sections of this tutorial. When the preprocessor finds an `#include` directive it replaces it by the whole content of the specified file. There are two ways to specify a file to be included:

```
#include "file"
#include <file>
```

The only difference between both expressions is the directories in which the compiler is going to look for the file. In the first case where the file is specified between quotes, the file is looked for in the same directory that includes the file containing the directive. In case that it is not there, the compiler looks for the file in the default directories where it is configured to look for the standard header files.

If the file name is enclosed between angle-brackets `<>` the file is looked for directly where the compiler is configured to look for the standard header files.

## #pragma

This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with `#pragma`.

### Section 6.1

## Input/Output with files



C++ has support both for input and output with files through the following classes:

- **ofstream**: File class for writing operations (derived from `ostream`)
- **ifstream**: File class for reading operations (derived from `istream`)
- **fstream**: File class for both reading and writing operations (derived from `iostream`)

## Open a file

The first operation generally done on an object of one of these classes is to associate it to a real file, that is to say, to open a file. The open file is represented within the program by a stream object (an instantiation of one of these classes) and any input or output performed on this stream object will be applied to the physical file.

In order to open a file with a stream object we use its member function `open()`:

```
void open (const char * filename, openmode mode);
```

where *filename* is a string of characters representing the name of the file to be opened and *mode* is a combination of the following flags:

|                          |                                             |
|--------------------------|---------------------------------------------|
| <code>ios::in</code>     | Open file for reading                       |
| <code>ios::out</code>    | Open file for writing                       |
| <code>ios::ate</code>    | Initial position: end of file               |
| <code>ios::app</code>    | Every output is appended at the end of file |
| <code>ios::trunc</code>  | If the file already existed it is erased    |
| <code>ios::binary</code> | Binary mode                                 |

These flags can be combined using bitwise operator OR: `|`. For example, if we want to open the file "example.bin" in binary mode to add data we could do it by the following call to function-member `open`:

```
ofstream file;
file.open ("example.bin", ios::out | ios::app | ios::binary);
```

All of the member functions `open` of classes **ofstream**, **ifstream** and **fstream** include a default mode when opening files that varies from one to the other:

| class           | default <i>mode</i> to parameter   |
|-----------------|------------------------------------|
| <b>ofstream</b> | <code>ios::out   ios::trunc</code> |
| <b>ifstream</b> | <code>ios::in</code>               |
| <b>fstream</b>  | <code>ios::in   ios::out</code>    |

The default value is only applied if the function is called without specifying a *mode* parameter. If the function is called with any value in that parameter the default mode is stepped on, not combined.

Since the first task that is performed on an object of classes **ofstream**, **ifstream** and **fstream** is frequently to open a file, these three classes include a constructor that directly calls the `open` member function and has the same parameters as this. This way, we could also have declared the previous object and conducted the same opening operation just by writing:

```
ofstream file ("example.bin", ios::out | ios::app | ios::binary);
```

Both forms to open a file are valid.

You can check if a file has been correctly opened by calling the member function `is_open()`:

```
bool is_open();
```

that returns a **bool** type value indicating **true** in case that indeed the object has been correctly associated with an open file or **false** otherwise.

## Closing a file

When reading, writing or consulting operations on a file are complete we must close it so that it becomes available again. In order to do that we shall call the member function **close()**, that is in charge of flushing the buffers and closing the file. Its form is quite simple:

```
void close ();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function **close**.

## Text mode files

Classes **ofstream**, **ifstream** and **fstream** are derived from **ostream**, **istream** and **iostream** respectively. That's why *fstream* objects can use the members of these parent classes to access data.

Generally, when using text files we shall use the same members of these classes that we used in communication with the console (**cin** and **cout**). As in the following example, where we use the overloaded insertion operator **<<**:

```
// writing on a text file
#include <fstream.h>

int main () {
 ofstream examplefile
("example.txt");
 if (examplefile.is_open())
 {
 examplefile << "This is a
line.\n";
 examplefile << "This is another
line.\n";
 examplefile.close();
 }
 return 0;
}
```

file **example.txt**

```
This is a line.
This is another line.
```

Data input from file can also be performed in the same way that we did with **cin**:

```
// reading a text file
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

int main () {
 char buffer[256];
 ifstream examplefile
```

```
This is a line.
This is another line.
```

```

("example.txt");
 if (! examplefile.is_open())
 { cout << "Error opening file";
 exit (1); }

 while (! examplefile.eof())
 {
 examplefile.getline
(buffer,100);
 cout << buffer << endl;
 }
 return 0;
}

```

This last example reads a text file and prints out its content on the screen. Notice how we have used a new member function, called `eof` that `ifstream` inherits from class `ios` and that returns `true` in case that the end of the file has been reached.

## Verification of state flags

In addition to `eof()`, other member functions exist to verify the state of the stream (all of them return a `bool` value):

**bad()**

Returns `true` if a failure occurs in a reading or writing operation. For example in case we try to write to a file that is not open for writing or if the device where we try to write has no space left.

**fail()**

Returns `true` in the same cases as `bad()` plus in case that a format error happens, as trying to read an integer number and an alphabetical character is received.

**eof()**

Returns `true` if a file opened for reading has reached the end.

**good()**

It is the most generic: returns `false` in the same cases in which calling any of the previous functions would return `true`.

In order to reset the state flags checked by the previous member functions you can use member function `clear()`, with no parameters.

## *get* and *put* stream pointers

All i/o streams objects have, at least, one stream pointer:

- `ifstream`, like `istream`, has a pointer known as *get pointer* that points to the next element to be read.
- `ofstream`, like `ostream`, has a pointer *put pointer* that points to the location where the next element has to be written.
- Finally `fstream`, like `iostream`, inherits both: *get* and *put*

These stream pointers that point to the reading or writing locations within a stream can be read and/or manipulated using the following member functions:

`tellg()` and `tellp()`



These two member functions admit no parameters and return a value of type `pos_type` (according ANSI-C++ standard) that is an integer data type representing the current position of *get* stream pointer (in case of `tellg`) or *put* stream pointer (in case of `tellp`).

**`seekg()` and `seekp()`**

This pair of functions serve respectively to change the position of stream pointers *get* and *put*. Both functions are overloaded with two different prototypes:

```
seekg (pos_type position);
```

```
seekp (pos_type position);
```

Using this prototype the stream pointer is changed to an absolute position from the beginning of the file. The type required is the same as that returned by functions `tellg` and `tellp`.

```
seekg (off_type offset, seekdir direction);
```

```
seekp (off_type offset, seekdir direction);
```

Using this prototype, an offset from a concrete point determined by parameter *direction* can be specified. It can be:

|                              |                                                                  |
|------------------------------|------------------------------------------------------------------|
| <b><code>ios::beg</code></b> | offset specified from the beginning of the stream                |
| <b><code>ios::cur</code></b> | offset specified from the current position of the stream pointer |
| <b><code>ios::end</code></b> | offset specified from the end of the stream                      |

The values of both stream pointers *get* and *put* are counted in different ways for text files than for binary files, since in text mode files some modifications to the appearance of some special characters can occur. For that reason it is advisable to use only the first prototype of **`seekg`** and **`seekp`** with files opened in text mode and always use non-modified values returned by `tellg` or `tellp`. With binary files, you can freely use all the implementations for these functions. They should not have any unexpected behavior.

The following example uses the member functions just seen to obtain the size of a binary file:

```
// obtaining file size
#include <iostream.h>
#include <fstream.h>

const char * filename =
"example.txt";

int main () {
 long l,m;
 ifstream file (filename,
ios::in|ios::binary);
 l = file.tellg();
 file.seekg (0, ios::end);
 m = file.tellg();
 file.close();
 cout << "size of " << filename;
 cout << " is " << (m-l) << "
bytes.\n";
 return 0;
}
```

**size of example.txt is 40 bytes.**

## Binary files

In binary files inputting and outputting data with operators like << and >> and functions like `getline`, does not make too much sense, although they are perfectly valid.

File streams include two member functions specially designed for input and output of data sequentially: **write** and **read**. The first one (**write**) is a member function of `ostream`, also inherited by `ofstream`. And **read** is member function of `istream` and it is inherited by `ifstream`. Objects of class `fstream` have both. Their prototypes are:

```
write (char * buffer, streamsize size);
read (char * buffer, streamsize size);
```

Where *buffer* is the address of a memory block where the read data are stored or from where the data to be written are taken. The *size* parameter is an integer value that specifies the number of characters to be read/written from/to the *buffer*.

```
// reading binary file
#include <iostream.h>
#include <fstream.h>

const char * filename =
"example.txt";

int main () {
 char * buffer;
 long size;
 ifstream file (filename,
ios::in|ios::binary|ios::ate);
 size = file.tellg();
 file.seekg (0, ios::beg);
 buffer = new char [size];
 file.read (buffer, size);
 file.close();

 cout << "the complete file is in a
buffer";

 delete[] buffer;
 return 0;
}
```

the complete file is in a buffer

## Buffers and Synchronization

When we operate with file streams, these are associated to a *buffer* of type `streambuf`. This *buffer* is a memory block that acts as an intermediary between the stream and the physical file. For example, with an out stream, each time the member function `put` (write a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in the *buffer* for that stream.

When the buffer is flushed, all data that it contains is written to the physic media (if it is an out stream) or simply erased (if it is an in stream). This process is called synchronization and it takes place under any of the following circumstances:

- **When the file is closed:** before closing a file all buffers that have not yet been completely written or read are synchronized.
- **When the *buffer* is full:** *Buffers* have a certain size. When the *buffer* is full it is automatically synchronized.

- **Explicitly with manipulators:** When certain manipulators are used on streams a synchronization takes place. These manipulators are: `flush` and `endl`.
- **Explicitly with function `sync()`:** Calling member function `sync()` (no parameters) causes an immediate synchronization. This function returns an `int` value equal to `-1` if the stream has no associated *buffer* or in case of failure.

The C++ Language Tutorial

## The Author



Hi,

My name is Juan Soulié, I'm a self-taught programmer born in 1977 in the Mediterranean island of Mallorca belonging to the Kingdom of Spain.

Before bumping into C and subsequently C++ I used to program in Pascal and when I was a child with Basic in my Spectrum and MSX-16kB machines. I also have some respect for unix shell scripting and nowadays Perl. Right now I'm specialized in Windows' API Programming and internet-related technologies.

The aim that pushed me to write this tutorial was to share with everyone interested in learning this amazing and versatile programming language what I've learnt here and there throughout the years using what seems to me simple and understandable explanations and avoiding useless (or slightly useful) theory. I've tried to explain what you can do with C++ instead on emphasizing what you should do (I'm not saying that that is not important, it is simply not covered in this tutorial).

Finally, I want to thank these people that have found some typos in previous versions of the tutorial: Mike H, Proto, Anderson Fabiano, Alex Hoover, Jose Castaneda , nameless person, Scott A. Fanjoy, Mr. Venom, Weilan W Wu, Vern Hamberg, Brian Agbay, Thomas Texier, Cory Wheeler, Jay, Hugo Lavalle, Joshua Smith, Jaime Tenorio, sassi, Bruce Bertrand, Nikolai Shevchuk, Devrim Ersanli, Guillermo, Luke Kurach, Nick Malden, Hans Verbrugge, mikeg, Chouputra, Anna Grishkan, Patrick Seafeld, Fede, Samuel Schultz, Mitchell Markin, and some others whose names were not disclosed in their messages.

There are probably some other errors to find. If you find one please mail me to <[jsoulie@cplusplus.com](mailto:jsoulie@cplusplus.com)>. Please notice that I don't know all about everything related to C++ and that I am not a volunteer programming consultant, so if you have a particular programming question you will probably get a better result posting your question in a programming forum, mailing list or newsgroup rather than sending it to me.

Regards,  
Juan Soulié