

**NORTHWEST UNIVERSITY,
KANO
FACULTY OF COMPUTING
DEPARTMENT OF COMPUTER
SCIENCE**



**LECTURE NOTE
COS 201/CSC2301 COMPUTER PROGRAMMING I (C++)
2024-2025**

COMPILED BY: ABDULAZIZ AMINU & AMINU USMAN JIBRIL

Table of Contents

PART ONE	5
PROGRAM, PROGRAMMER AND PROGRAMMING	2
Program.....	2
Programmer.....	2
Programming.....	3
INTRODUCTION TO PROBLEM SOLVING.....	3
PRINCIPLES OF GOOD PROGRAMMING	4
STRUCTURED PROGRAMMING.....	6
PART TWO	8
PROGRAMMING IN C++.....	9
Comments	12
PART THREE	13
Variables & Data Types.....	14
Identifiers	15
Fundamental data types.....	16
Declaration of variables	17
Scope of variables	19
Initialization of variables	20
Constants.....	21
Literals	21
Integer Numerals.....	21
Floating Point Numbers	22
Character and string literals	23
Boolean literals	25
Defined constants (#define)	25
Declared constants (const)	26
Operators.....	26
Assignment (=)	26
Arithmetic operators (+, -, *, /, %).....	28
Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, =).....	28
Increase and decrease (++ , --).....	29

Relational and equality operators (==, !=, >, <, >=, <=)	30
Logical operators (!, &&,).....	31
&& OPERATOR	31
// OPERATOR	31
Conditional operator (?).....	32
Comma operator (,).....	32
Bitwise Operators (&, , ^, ~, <<, >>).....	33
Explicit type casting operator	33
sizeof()	33
Other operators.....	34
Precedence of operators	34
Basic Input/Output	36
Standard Output (cout).....	36
cin and strings	39
PART FOUR.....	41
Control Structures	42
Conditional structure: if and else	42
The selective structure: switch.....	43
Iteration structures (loops)	45
The while loop	45
The do-while loop	46
The for loop.....	47
Jump statements	49
The break statement	49
The continue statement	49
PART FIVE	50
Functions.....	51
Functions with no type. The use of void.....	55
Arguments passed by value and by reference.....	57
Default values in parameters.....	59
Overloaded functions.....	60
Recursivity.....	62
Declaring functions.....	63
Arrays.....	65

Initializing arrays.	66
Accessing the values of an array.....	66
Multidimensional arrays	68
Arrays as parameters	70

PART ONE



PROGRAM, PROGRAMMER AND PROGRAMMING.

PROGRAM

A program is a structured collection of instructions that a computer follows to perform a specific task or solve a particular problem. These instructions are written in a language that the computer understands called a programming language and they guide the computer through various operations such as input handling, data processing, and output generation. Programs can range from very simple scripts that perform basic calculations to highly complex systems like operating systems, video editing software, web browsers, or mobile apps.

The purpose of a program is to automate tasks, solve problems, or create interactive experiences for users. A program typically involves input (data coming from the user or other systems), processing (where calculations and decisions are made), and output (results displayed or stored). The instructions in a program are executed in sequence, sometimes including loops and conditions that control the flow of operations. Programs must be accurate, efficient, and reliable in order to function effectively. For example, a banking software program must be able to correctly calculate balances, handle user logins, and ensure the security of financial data.

PROGRAMMER

A programmer is an individual who writes, tests, maintains, and improves computer programs. This role requires a strong understanding of both the problem being solved and the tools available through programming languages. Programmers take real-world problems or tasks and break them down into logical steps that can be expressed in code. They design algorithms, which are detailed plans or strategies for solving problems, and then translate these algorithms into a specific programming language such as Python, Java, C++, or JavaScript.

A good programmer possesses not just technical knowledge but also analytical thinking and problem-solving skills. They must be able to think logically, identify errors in code (a process called debugging), and understand how different parts of a system interact. In professional environments, programmers often work in teams and collaborate with designers, testers, and project managers.

Their work is critical to the development of nearly every type of modern technology, including websites, mobile applications, games, artificial intelligence systems, and embedded devices found in cars, appliances, and more.

PROGRAMMING

Programming is the process of creating computer programs. It involves writing code in a programming language to develop instructions that a computer can execute. Programming is not just about writing code. It also includes understanding a problem, designing a logical solution, implementing that solution in code, and then testing and refining the program until it works correctly. This process is fundamental to software development and lies at the core of virtually every digital system and application.

The act of programming typically starts with problem analysis. The programmer first tries to fully understand what the problem is and what outcome is desired. Then, they devise an algorithm; a sequence of logical steps to solve the problem. Once the plan is in place, the programmer writes the code using a specific language. After writing, they test the program to make sure it works as expected and fix any issues that arise (a process known as debugging).

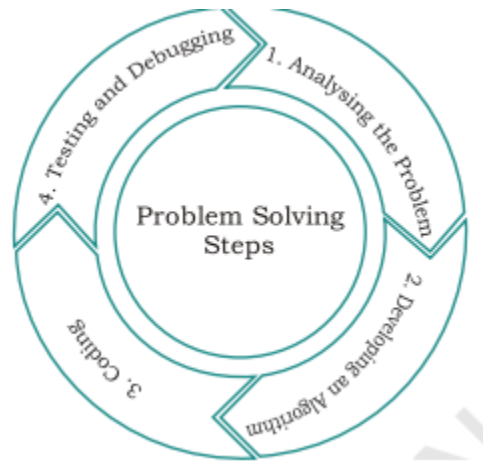
Some common tasks involved in programming include:

- i. Writing algorithms to solve specific problems
- ii. Creating user interfaces for applications
- iii. Interfacing with databases and external devices
- iv. Ensuring programs run efficiently and securely

INTRODUCTION TO PROBLEM SOLVING

In the world of computing, problem-solving refers to the process of identifying an issue, analyzing it, and developing a structured, logical solution that can be implemented using a computer program. It forms the backbone of programming, as writing effective code begins with understanding the problem that the code is supposed to solve. Problem-solving is the art of breaking down complex challenges into smaller, manageable parts and then finding the best way to address each part systematically. When a computer program is created, it is essentially a tool designed to solve a specific problem whether it's calculating numbers, processing data,

managing records, or automating a task. Therefore, before any code is written, it is important for a programmer to fully understand the nature of the problem.



The **steps in problem-solving** typically include:

- i. **Understanding the problem** (What are we solving?)
- ii. **Analyzing the problem** (What do we know? What is required?)
- iii. **Designing an algorithm** (How do we solve it step-by-step?)
- iv. **Implementing the algorithm** (How do we write the solution in code?)
- v. **Testing and debugging** (Does it work? What needs fixing?)
- vi. **Maintaining and refining** (How can we make it better?)

PRINCIPLES OF GOOD PROGRAMMING

1. **Keep it Simple, Stupid (KISS):** The best solutions are often the simplest ones. The principle of simplicity emphasizes that code should do what is required without unnecessary complexity. Keeping your logic straightforward and your functions single-purpose ensures that your program is easier to debug, extend, and optimize in the future.
2. **DRY –Don’t Repeat Yourself:** Repetition in code is not just tedious; it is dangerous. When you copy and paste code, any change in the logic requires changes in every copied instance, increasing the chances of introducing bugs. Instead of repeating logic, abstract common functionality into reusable functions, methods, or classes. This reduces duplication and centralizes control, making updates easier and faster. Following DRY

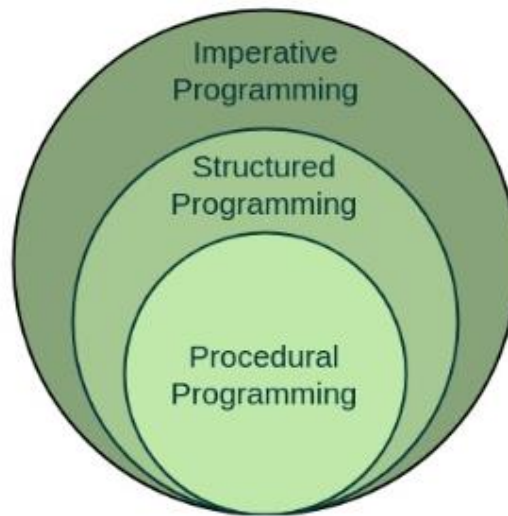
also makes your program shorter and clearer, which aligns perfectly with the principles of simplicity and readability.

3. **Implement the needed functionalities only:** It encourages developers to focus only on what the system currently requires rather than what it might require in the future. The idea is to avoid wasting time and resources on features that may never be used. Using this, we can achieve the following: Reduces Complexity; speed up development process; avoid waste of time.
4. **Think about optimization only when the code is working:** means you should first ensure your code runs and produces the correct results before trying to make it faster or more efficient. Premature optimization often leads to complex, hard-to-read code and wasted effort on parts that may not even impact performance. Once the program is functional and tested, you can then use profiling tools to identify actual bottlenecks and optimize only where necessary. This approach supports simplicity, maintainability, and clarity core principles of good software development.
5. **Modularity:** Modular code is organized into small, independent parts that can be developed, tested, and maintained separately. Each module or function should be responsible for one specific task. This approach improves reusability, as modules can often be used in different parts of the program or in future projects. Modularity also simplifies debugging if something breaks, it is easier to isolate the problem. Moreover, modular code encourages collaboration, allowing multiple developers to work on different parts of the program simultaneously without conflicts. In large-scale projects, modular design is essential for scalability and long-term sustainability.
6. **Scalability:** Scalable code can handle growing amounts of work or data without significant changes to the codebase. This includes structuring the application so that it can support more users, more data, or higher performance demands. Scalable design often involves considerations such as modularity, efficient algorithms, and resource management. Writing scalable software is critical if you expect your application to grow in complexity or usage over time.
7. **Documentation:** Good documentation complements good code. It doesn't replace clarity but enhances understanding, especially for external users or new team members. Documentation should describe the purpose of a project, how to set it up, how to use key

features, and how the major components fit together. Inline comments should explain complex or non-obvious sections of code. API documentation and user manuals should be accurate and kept up to date. Without documentation, even the best code can become a mystery over time.

STRUCTURED PROGRAMMING

It can be defined as a programming approach in which the program is made as a single structure. It means that the code will execute the instruction by instruction one after the other. It doesn't support the possibility of jumping from one instruction to some other with the help of any statement like GOTO, etc. Therefore, the instructions in this approach will be executed in a serial and structured manner. The languages that support Structured programming approach are: C, C++, Java, C#, etc



On the contrary, in the Assembly languages like Microprocessor 8085, etc, the statements do not get executed in a structured manner. It allows jump statements like GOTO. So the program flow might be random. The structured program mainly consists of three types of elements:

- i. **Selection Statements:** This involves using conditional statements (if-else, switch) to make decisions. The program's control flow branches depending on whether a condition is true or false, allowing for different paths of execution.

- ii. **Sequence Statements:** Instructions are executed in a linear order, one after the other. Each step in the sequence logically progresses to the next without causing unexpected side effects.
- iii. **Iteration Statements:** This involves using loops (for, while, do-while) to repeat a block of code until a certain condition is met. This allows for repetitive tasks to be performed efficiently.

The structured program consists of well-structured and separated modules. But the entry and exit in Structured program is a single-time event. It means that the program uses single-entry and single-exit elements. Therefore, a structured program is well maintained, neat and clean program. This is the reason why the Structured Programming approach is well accepted in the programming world.

PART TWO

PROGRAMMING IN C++

C++ is a general-purpose programming language developed by Bjarne Stroustrup in 1979 as an extension of the C language. It combines the speed and simplicity of C with object-oriented features, making it one of the most powerful and widely-used languages in software development. C++ supports procedural, object-oriented, and generic programming, making it a multi-paradigm language.

C++ can be better explained by demonstrating the simplest problem that print welcome to the screen. This program below does more than printing a string but rather explaining the basic structure of a C++ program.

```
1 // Fig. 2.1: fig02_01.cpp
2 // Text-printing program.
3 #include <iostream> // allows program to output data to the screen
4
5 // function main begins program execution
6 int main()
7 {
8     std::cout << "Welcome to C++!\n"; // display message
9
10    return 0; // indicate that program ended successfully
11 } // end function main
```

Welcome to C++!

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

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

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

Hello World!

The first panel shows the source code for our first program. The second one shows the result of the program once compiled and executed.

The program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!" sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

// my first program in C++

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

#include <iostream>

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive `#include <iostream>` tells the preprocessor to include the `iostream` standard file. This specific file (`iostream`) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

using namespace std;

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library, and in fact it will be included in most of the source codes included in these tutorials.

int main ()

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it - the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function.

The word `main` is followed in the code by a pair of parentheses (`()`). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of

expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them. Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

cout << "Hello World!";

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program. cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen).

cout is declared in the iostream standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

return 0;

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by //). There were lines with directives for the compiler's preprocessor (those beginning by #). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into cout), which were all included within the block delimited by the braces ({}) of the main function.

Comments

Comments are parts of the source code disregarded by the compiler. They simply 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:

- Single Line Comment
- Multiple Line Comment

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

The first of them, known as line comment, discards everything from where the pair of slash signs (//) is found up to the end of that same line. The second one, known as block comment, discards everything between the /* characters and the first appearance of the */ characters, with the possibility of including more than one line. We are going to add comments to our second program:

<pre>/* my second program in C++ with more comments */ #include <iostream> using namespace std; int main () { cout << "Hello World! "; // prints Hello World! cout << "I'm a C++ program"; // prints I'm a C++ program return 0; }</pre>	<pre>Hello World! I'm a C++ program</pre>
--	---

PART THREE

VARIABLES & DATA TYPES.

The usefulness of the "Hello World" programs shown in the previous section is quite questionable. We had to write several lines of code, compile them, and then execute the resulting program just to obtain a simple sentence written on the screen as result. It certainly would have been much faster to type the output sentence by ourselves. However, programming is not limited only to printing simple texts on the screen. In order to go a little further on and to become able to write programs that perform useful tasks that really save us work we need to introduce the concept of variable.

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

The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:

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

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

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

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

IDENTIFIERS

A valid identifier is a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline character (_), but in some cases these may be reserved for compiler specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case they can begin with a digit.

Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C++ language nor your compiler's specific ones, which are *reserved keywords*. The standard reserved keywords are:

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

Additionally, alternative representations for some operators cannot be used as identifiers since they are reserved words under some circumstances:

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

Your compiler may also include some additional specific reserved keywords.

Very important: The C++ language is a "case sensitive" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the RESULT variable is not the same as the result variable or the Result variable. These are three different variable identifiers.

FUNDAMENTAL DATA TYPES

When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)

* The values of the columns **Size** and **Range** depend on the system the program is compiled for. The values shown above are those found on most 32-bit systems. But for other systems, the general specification is that int has the natural size suggested by the system architecture (one "word") and the four integer types char, short, int and long must each one be at least as large as the one preceding it, with char being always 1 byte in size.

The same applies to the floating point types float, double and long double, where each one must provide at least as much precision as the preceding one.

DECLARATION OF VARIABLES

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

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

These are two valid declarations of variables. The first one declares a variable of type int with the identifier a. The second one declares a variable of type float with the identifier mynumber. Once declared, the variables a and mynumber can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

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

This declares three variables (a, b and c), all of them of type int, and has exactly the same meaning as:

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

The integer data types char, short, long and int can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier signed or the specifier unsigned before the type name. For example:

```
unsigned    short    int  
NumberOfSisters; signed int  
MyAccountBalance;
```

By default, if we do not specify either signed or unsigned most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

```
int MyAccountBalance;
```

with exactly the same meaning (with or without the keyword `signed`)

An exception to this general rule is the `char` type, which exists by itself and is considered a different fundamental data type from `signed char` and `unsigned char`, thought to store characters. You should use either `signed` or `unsigned` if you intend to store numerical values in a `char`-sized variable.

`short` and `long` can be used alone as type specifiers. In this case, they refer to their respective integer fundamental types: `short` is equivalent to `short int` and `long` is equivalent to `long int`. The following two variable declarations are equivalent:

```
short Year;  
short    int  
Year;
```

Finally, `signed` and `unsigned` may also be used as standalone type specifiers, meaning the same as `signed int` and `unsigned int` respectively. The following two declarations are equivalent:

```
unsigned NextYear;  
unsigned        int  
NextYear;
```

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

```
// operating with variables  
  
#include  
<iostream> using  
namespace std;  
  
int main ()  
{  
    //    declaring  
variables:    int  
a,    b;        int  
result;  
    // process:  
    a    =    5;  
b = 2;    a  
= a + 1;  
result = a  
- b;
```

4

```

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

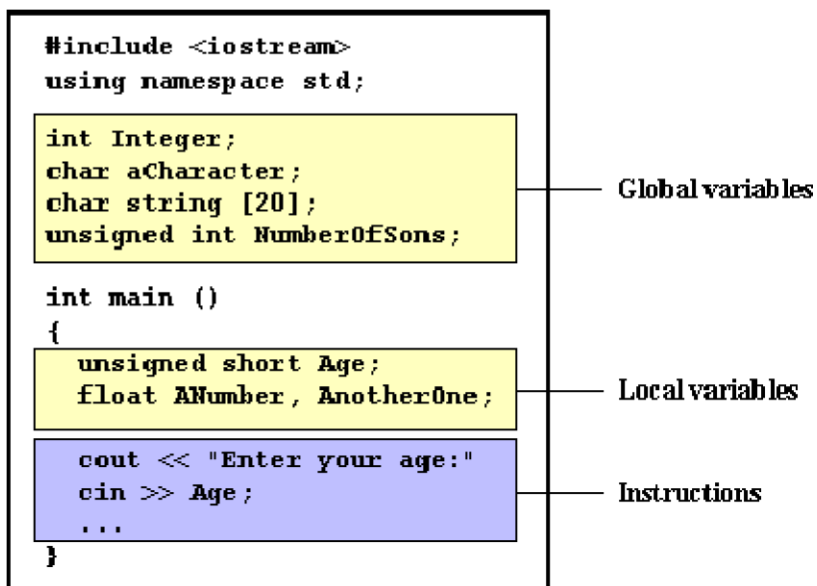
    // terminate the program:
    return 0;
}

```

SCOPE OF VARIABLES

All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function `main` when we declared that `a`, `b`, and `result` were of type `int`.

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.



Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

The scope of local variables is limited to the block enclosed in braces ({}) where they are declared. For example, if they are declared at the beginning of the body of a function (like in function `main`) their scope is between its declaration point and the end of that function. In the example above, this means that if another function existed in addition to `main`, the local variables declared in `main` could not be accessed from the other function and vice versa.

INITIALIZATION OF VARIABLES

When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++:

The first one, known as *c-like*, is done by appending an equal sign followed by the value to which the variable will be initialized:

```
type identifier = initial_value ;
```

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

```
int a = 0;
```

The other way to initialize variables, known as *constructor initialization*, is done by enclosing the initial value between parentheses (()):

```
type identifier
```

```
(initial_value) ; For
```

example:

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

Both ways of initializing variables are valid and equivalent in C++.


```
// initialization of variables

#include
<iostream> using
namespace std;

int main ()
{
    int a=5;           //
    initial value = 5   int b(2);
    // initial value = 2   int
    result;             // initial
    value undetermined

    a = a + 3;
    result = a -
    b;    cout <<
    result;

    return 0;
}
```

6

CONSTANTS

Constants are expressions with a fixed value.

LITERALS

Literals are used to express particular values within the source code of a program. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
a = 5;
```

the 5 in this piece of code was a literal constant.

Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

INTEGER NUMERALS

```
1776
707
```

```
-273
```

They are numerical constants that identify integer decimal values. Notice that to express a numerical constant we do not have 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 are used to use every day) 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 have to precede it with a 0 (zero character). And in order 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 these represent the same number: 75 (seventy-five) expressed as a base-10 numeral, octal numeral and hexadecimal numeral, respectively.

Literal constants, like variables, are considered to have a specific data type. By default, integer literals are of type `int`. However, we can force them to either be unsigned by appending the `u` character to it, or `long` by appending `l`:

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

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

FLOATING POINT NUMBERS

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

```
3.14159     // 3.14159
6.02e23     // 6.02 x 10^23
1.6e-19     // 1.6 x 10^-19
3.0         // 3.0
```

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small

number) -all of them approximated- and the last one is the number three expressed as a floating-point numeric literal.

The default type for floating point literals is `double`. If you explicitly want to express a `float` or `long double` numerical literal, you can use the `f` or `L` suffixes respectively:

```
3.14159L    // long double
6.02e23f    // float
```

Any of the letters that can be part of a floating-point numerical constant (`e`, `f`, `L`) can be written using either lower or uppercase letters without any difference in their meanings.

CHARACTER AND STRING LITERALS

There also exist non-numerical constants, like:

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

The first two expressions represent single character constants, and the following two represent string literals composed of several characters. Notice that to represent a single character we enclose it between single quotes (`'`) and to express a string (which generally consists of more than one character) we enclose it between double quotes (`"`).

When writing both single character and string literals, it is necessary to put the quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

```
x
'x'
```

`x` alone would refer to a variable whose identifier is `x`, whereas `'x'` (enclosed within single quotation marks) would refer to the character constant `'x'`.

Character and string literals have certain peculiarities, like the escape codes. These are special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (`\n`) or tab (`\t`). All of them are preceded by a backslash (`\`). Here you have a list of some of such escape codes:

<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\t</code>	tab
<code>\v</code>	vertical tab

\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)
\\	backslash (\)

For example:

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

Additionally, you can express any character by its numerical ASCII code by writing a backslash character (\) followed by the ASCII code expressed as an octal (base-8) or hexadecimal (base-16) number. In the first case (octal) the digits must immediately follow the backslash (for example \23 or \40), in the second case (hexadecimal), an x character must be written before the digits themselves (for example \x20 or \x4A).

String literals can extend to more than a single line of code by putting a backslash sign (\) at the end of each unfinished line.

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

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

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

Finally, if we want the string literal to be explicitly made of wide characters (`wchar_t`), instead of narrow characters (`char`), we can precede the constant with the `L` prefix:

```
L"This is a wide character string"
```

Wide characters are used mainly to represent non-English or exotic character sets.

BOOLEAN LITERALS

There are only two valid Boolean values: true and false. These can be expressed in C++ as values of type `bool` by using the Boolean literals `true` and `false`.

DEFINED CONSTANTS (#DEFINE)

You can define your own names for constants that you use very often without having to resort to memory-consuming variables, simply by using the `#define` preprocessor directive. Its format is:

```
#define identifier
```

value For example:

```
#define PI 3.14159
#define NEWLINE '\n'
```

This defines two new constants: `PI` and `NEWLINE`. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

```
//      defined      constants:      calculate 31.4159
circumference

#include <iostream> using
namespace std;

#define PI 3.14159
#define NEWLINE '\n'

int main ()
{
    double r=5.0;           // radius

    double circle;

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

    return 0;
}
```

In fact the only thing that the compiler preprocessor does when it encounters `#define` directives is to literally replace any occurrence of their identifier (in the previous example, these

were `PI` and `NEWLINE`) by the code to which they have been defined (3.14159 and `'\n'` respectively).

The `#define` directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (`;`) at its end. If you append a semicolon character (`;`) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

DECLARED CONSTANTS (CONST)

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

```
const int pathwidth =  
100;      const char  
tabulator = '\t';
```

Here, `pathwidth` and `tabulator` are two typed constants. They are treated just like regular variables except that their values cannot be modified after their definition.

OPERATORS

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.

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

ASSIGNMENT (=)

The assignment operator assigns a value to a variable.

```
a = 5;
```

This statement assigns the integer value 5 to the variable `a`. The part at the left of the assignment operator (`=`) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The *lvalue* has to be a variable whereas the *rvalue* can be either a constant, a variable, the result of an operation or any combination of these. The most important rule when assigning is the *right-to-left* rule: The assignment operation always takes place from right to left, and never the other way:

```
a = b;
```

This statement assigns to variable *a* (the lvalue) the value contained in variable *b* (the rvalue). The value that was stored until this moment in *a* is not considered at all in this operation, and in fact that value is lost.

Consider also that we are only assigning the value of *b* to *a* at the moment of the assignment operation. Therefore a later change of *b* will not affect the new value of *a*.

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

```
// assignment operator

#include <iostream> using namespace
std;

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

    cout << "a:";
    cout << a;
    cout << " b:";
    cout << b;

    return 0;
}
```

a:4 b:7

This code will give us as result that the value contained in *a* is 4 and the one contained in *b* is 7. Notice how *a* was not affected by the final modification of *b*, even though we declared *a = b* earlier (that is because of the *right-toleft 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 operation. 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 (i.e. 5), leaving a with a final value of 7.

The following expression is also valid in C++:

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

It assigns 5 to all the three variables: a, b and c.

ARITHMETIC OPERATORS (+, -, *, /, %)

The five arithmetical operations supported by the C++ language are:

+	addition
-	subtraction
*	multiplication
/	division
%	modulo

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is *modulo*; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

```
a = 11 % 3;
```

the variable a will contain the value 2, since 2 is the remainder from dividing 11 between 3.

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

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

expression	is equivalent to
value += increase;	value = value + increase;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
price *= units + 1;	price = price * (units + 1);

and the same for all other operators. For example:


```
// compound assignment operators

#include <iostream> using
namespace std;

int main ()
{
int a,b=3;
a = b;
a+=2;    // equivalent to a=a+2
cout << a;    return 0;
}
```

5

INCREASE AND DECREASE (++ , --)

Shortening even more some expressions, the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

```
c++;
c+=1;
c=c+1;
```

are all equivalent in its functionality: the three of them increase by one the value of c.

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

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

Example 1	Example 2
<pre>B=3; A=++B; // A contains 4, B contains 4</pre>	<pre>B=3; A=B++; // A contains 3, B contains 4</pre>

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 then B is increased.

RELATIONAL AND EQUALITY OPERATORS (==, !=, >, <, >=, <=)

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.

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

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

Here there are some examples:

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

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

Be careful! The operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (==) is the equality operator that compares whether both expressions in the two sides of it 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 the value 2, so the result of the operation is true.

LOGICAL OPERATORS (!, &&, ||)

The Operator ! is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

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

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

&& OPERATOR

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a || b:

// OPERATOR

a	b	a b
true	true	true
true	false	true
false	true	true
false	false	false

For example:

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

CONDITIONAL OPERATOR (?)

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:

condition ? result1 : result2

If condition is true the expression will return result1, if it is 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 the value of a, since 5 is greater than 3.  
a>b ? a : b // returns whichever is greater, a or b.
```

```
// conditional operator  
  
#include <iostream>  
namespace std;  
  
int main ()  
{  
    int a,b,c;  
    a=2; b=7;  
    c = (a>b) ? a : b;  
  
    cout << c;  
  
    return 0;  
}
```

7

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

COMMA OPERATOR (,)

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

For example, the following code:

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

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

BITWISE OPERATORS (&, |, ^, ~, <<, >>)

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

operator	asm equivalent	description
&	AND	Bitwise AND
	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

EXPLICIT TYPE CASTING OPERATOR

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

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

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

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

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

sizeof()

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

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

This will assign the value 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 these tutorials, 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.

PRECEDENCE OF OPERATORS

When writing 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 a result  
of 6, or  
  
a = (5 + 7) % 2    // with a result of  
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 come from mathematics) but for all the operators which can appear in C++. From greatest to lowest priority, the priority order is as follows:

Level	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	.* ->*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right

8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13		bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15		logical OR	Left-to-right
16	? :	conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^= =	assignment	Right-to-left
18	,	comma	Left-to-right

Grouping defines the precedence order in which operators are evaluated in the case that there are several operators of the same level in an expression.

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

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

might be written either as:

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

or

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

depending on the operation that we want to perform.

So if you want to write complicated expressions and you are not completely sure of the precedence levels, always include parentheses. It will also become a code easier to read.

BASIC INPUT/OUTPUT

Until now, the example programs of previous sections provided very little interaction with the user, if any at all. Using the standard input and output library, we will be able to interact with the user by printing messages on the screen and getting the user's input from the keyboard.

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen or the keyboard. A stream is an object where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know it will accept or provide characters sequentially.

The standard C++ library includes the header file `iostream`, where the standard input and output stream objects are declared.

STANDARD OUTPUT (COUT)

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is `cout`.

`cout` is used in conjunction with the *insertion operator*, which is written as `<<` (two "less than" signs).

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

The `<<` operator inserts the data that follows it into the stream preceding it. In the examples above it inserted the constant string `Output sentence`, the numerical constant `120` and variable `x` into the standard output stream `cout`. Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant 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 variable names. For example, these two sentences have very different results:

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

The insertion operator (`<<`) may be used more than once in a single statement:

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


This last statement would print the message Hello, I am a C++ statement 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 assume the age variable to contain the value 24 and the zipcode variable to contain 90064 the output of the previous statement 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 statements:

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

will be shown on the screen one following the other without any line break between them:

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

even though we had written them in two different insertions into cout. In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line character can be specified as \n (backslash, n):

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

This 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 produces a newline character, exactly as the insertion of `'\n'` does, but it also has an additional behavior when it is used with buffered streams: the buffer is flushed. Anyway, `cout` will be an unbuffered stream in most cases, so you can generally use both the `\n` escape character and the `endl` manipulator in order to specify a new line without any difference in its behavior.

STANDARD INPUT (CIN).

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

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

The first statement declares a variable of type `int` called `age`, and the second one waits for an input from `cin` (the 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, the extraction from `cin` will not process the input until the user presses RETURN after the character has been introduced.

You must always consider the type of the variable that you are using as a container with `cin` extractions. 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> using
namespace std;

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

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

The user of a program may be one of the factors that generate 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 generally 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` extractions you will have to trust that the user of your program will be cooperative and that he/she will not introduce his/her name or something similar when an integer value is requested. A little ahead, when we see the `stringstream` class we will see a possible solution 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 one for variable `b` that may be separated by any valid blank separator: a space, a tab character or a newline.

cin and strings

We can use `cin` to get strings with the extraction operator (`>>`) as we do with fundamental data type variables:

```
cin >> mystring;
```

However, as it has been said, `cin` extraction stops reading as soon as it finds any blank space character, so in this case we will be able to get just one word for each extraction. This behavior

may or may not be what we want; for example if we want to get a sentence from the user, this extraction operation would not be useful.

In order to get entire lines, we can use the function `getline`, which is the more recommendable way to get user input with `cin`:

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

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

```
What's your name? Juan SouliÃÂÂ
Hello Juan SouliÃÂÂ.
What is your favorite team? The
Isotopes
I like The Isotopes too!
```

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

PART FOUR

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 by our program, when and under which circumstances.

With the introduction of control structures we are going to have to introduce a new concept: the *compoundstatement* or *block*. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: { }:

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

Most of the control structures that we will see in this section require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces ({ }). But in the case that we want the statement to be a compound statement it must be enclosed between braces ({ }), forming a block.

Conditional structure: if and else

The `if` keyword is used to execute a statement or block 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 right after this conditional structure.

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

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

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces { }:

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

We can additionally specify what we want to happen 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 on the screen `x is 100` if indeed `x` has a value of 100, but if it has 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 value currently stored in `x` is positive, negative or none of them (i.e. 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 that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces `{ }`.

The selective structure: `switch`.

The syntax of the `switch` statement 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 concatenation of several `if` and `else if` instructions. Its form is the following:

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

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).

Both of the following code fragments have the same behavior:

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

The switch statement is a bit peculiar within the C++ language because it uses labels instead of blocks. This forces us to put break statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the switch selective block or a break statement is reached.

For example, if we did not include a break statement after the first group for case one, the program will not automatically jump to the end of the switch selective block and it would continue executing the rest of statements until it reaches either a break instruction or the end of the switch selective block. This makes unnecessary to include braces { } surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being 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 against constants. Therefore we cannot put variables as labels (for example `case n:` where `n` is a variable) or ranges (`case (1..3):`) because they are not valid C++ constants.

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

ITERATION STRUCTURES (LOOPS)

Loops have as purpose 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 functionality is simply to repeat statement while the condition set in expression is true. For example, we are going to make a program to countdown using a while-loop:

```
// custom countdown using while

#include <iostream> using
namespace std;

int main ()
{   int n;
    cout << "Enter the starting number > ";
    cin >> n;

    while (n>0) {
        cout << n << ", ";
        --n;
    }
    cout << "FIRE!\n";
    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` is greater than zero) the block that follows the condition will be executed and repeated while the condition (`n>0`) remains being true.

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

1. User assigns a value to `n`
2. The while condition is checked (`n>0`). At this point there are two possibilities:
 - * condition is true: statement is executed (to step 3)
 - * condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:

```
cout << n << ", ";  
--n;
```

(prints the value of `n` on the screen and decreases `n` by 1)
4. End of block. Return automatically to step 2
5. Continue the program right after the block: print FIRE! and end program.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. In this case we have included `--n`; that decreases the value of the variable that is being evaluated in the condition (`n`) by one - this will eventually make the condition (`n>0`) to become false after a certain number of loop iterations: to be more specific, when `n` becomes 0, that is where our while-loop and our countdown end.

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

The do-while loop

Its format is:

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

Its functionality is exactly the same as the while loop, except that `condition` in the do-while loop 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 example program echoes any number you enter until you enter 0.

```
// number echoer  
  
#include <iostream> using  
namespace std;  
  
int main ()  
{  
    unsigned long n;  
    do {  
        cout << "Enter number (0 to end): ";  
        cin >> n;
```

```
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
```

```
    cout << "You entered: " << n << "\n";  
}while (n != 0);  
return 0;  
}
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value 0 in the previous example you can be prompted for more numbers forever.

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, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in 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 ends and `statement` is skipped (not executed).
3. `statement` is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
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>
using namespace std;
int main ()
{
    for (int n=10; n>0; n--){
        cout << n << ", ";
    }
    cout << "FIRE!\n";
    return 0;
}
```

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

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: `for (;n<10;)` if we wanted to specify no initialization and no increase; or `for (;n<10;n++)` if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop, like in initialization, for example. The comma operator (,) is an expression separator, it serves to separate more than one expression where only one 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 for 50 times if neither `n` or `i` are modified within the loop:

The diagram shows a for loop: `for (n=0, i=100 ; n!=i ; n++, i--)`. Arrows point from labels to parts of the loop: 'Initialization' points to `n=0, i=100`; 'Condition' points to `n!=i`; and 'Increase' points to `n++, i--`.

`n` starts with a value of 0, and `i` with 100, the condition is `n!=i` (that `n` is 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.

JUMP STATEMENTS

The break statement

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 its natural end (maybe because of an engine check failure?):

```
// break loop example

#include <iostream>
using namespace std;

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 STATEMENT

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

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

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

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

PART FIVE

FUNCTIONS

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

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

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

- `type` is the data type specifier of the data returned by the function.
- `name` is the identifier by which it will be possible to call the function.
- `parameters` (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: `int x`) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- `statements` is the function's body. It is a block of statements surrounded by braces `{ }`.

Here you have the first function example:

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

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

int main ()
{
    int z;
    z = addition (5,3);
    cout << "The result is " << z;
    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 by 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 a function called `addition`. Paying 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 some code lines above:

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

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

The parameters and arguments 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 function `addition`.

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

Function `addition` declares another local variable (`int r`), and by means of the expression `r=a+b`, it assigns to `r` the result of `a` plus `b`. Because the actual parameters passed 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 in the first place (in this case, `main`). At this moment the program follows its regular course from the same point at which it was interrupted by the call to `addition`. But additionally, because the `return` statement in function `addition` specified a value: the content of variable `r` (`return (r);`), which at that moment had a value of 8. This value becomes the value of evaluating the function call.

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

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable `z` will be set to 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).

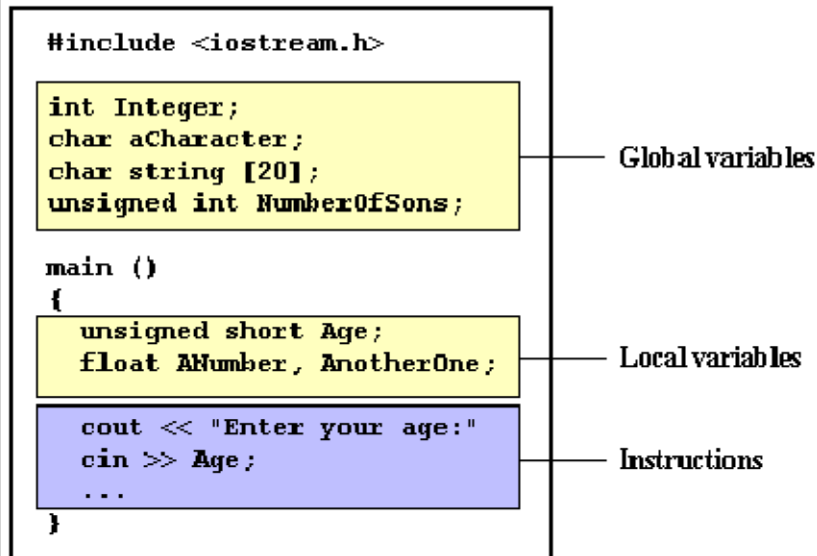
The following line of code in `main` is:

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


That, as you may already expect, produces the printing of the result on the screen.

Scope of variables

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



Therefore, the scope of local variables is limited to the same block level in which they are declared. Nevertheless, we also have the possibility to declare global variables; These are visible from any point of the code, inside and outside all functions. In order to declare global variables you simply have to declare the variable outside any function or block; that means, directly in the body of the program.

And here is another example about functions:

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

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

int main ()
{
    int x=5, y=3, z;
    z = subtraction(7,2);
    cout << "The first result is " << z << '\n';
    cout << "The second result is " << subtraction (7,2)
    << '\n';
    cout << "The third result is " << subtraction (x,y) <<
```

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

```
'\n';
z= 4 + subtraction (x,y);
    cout << "The fourth result is " << z << '\n';
return 0;
}
```

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

Nevertheless, if we examine 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 fully understand these examples you must consider once again that a call to a function could be replaced by the value that the function call itself is going to return. 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 the value it returns (i.e., 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 an insertion parameter for cout. Simply consider that the result is the same as if we had written:

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

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

In the case of:

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

The 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 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 written:

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

with exactly the same result. I have switched places so you can see that the semicolon sign (;) goes at the end of the whole statement. 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 returned value:

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

Functions with no type. The use of void.

If you remember the syntax of a function

declaration: type name (argument1,

argument2 ...) statement

you will see that the declaration begins with a type, that is the type of the function itself (i.e., the type of the datum that will be returned by the function with the return statement). 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. In this case we should use the void type specifier for the function. This is a special specifier that indicates absence of type.

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

void printmessage ()
{
    cout << "I'm a function!";
}
int main ()
{
    Printmessage();
    return 0;
}
```

I'm a function!

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

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

Although it is optional to specify `void` in the parameter list. In C++, a parameter list can simply be left blank if we want a function with no parameters.

What you must always remember is that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to `printmessage` is:

```
printmessage ();
```

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement. The following call would have been incorrect:

```
printmessage;
```


ARGUMENTS PASSED BY VALUE AND BY REFERENCE.

Until now, in all the functions we have seen, the arguments 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 copies of their values but never the 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 to function `addition` passing the values of `x` and `y`, i.e. 5 and 3 respectively, but not the variables `x` and `y` themselves.

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



This way, when the function `addition` is called, the value of its local variables `a` and `b` become 5 and 3 respectively, but any modification to either `a` or `b` within the function `addition` will not have any effect in the values of `x` and `y` outside it, because variables `x` and `y` were not themselves passed to the function, but only copies of their values at the moment the function was called.

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 can use arguments passed by reference, as in the function `duplicate` of the following example:

```
// passing parameters by reference  
#include  
<iostream> using  
namespace std;  
  
void duplicate (int& a, int& b,  
int& c)  
{  
a*=2;  
b*=2;  
c*=2;  
}  
  
int main ()  
{  
    int x=1, y=3, z=7;  
    duplicate (x, y, z);  
    cout << "x=" << x << ", y=" << y
```

x=2, y=6, z=14

```
<< ", z=" << z;
return 0;
}
```

The first thing that should call your attention is that in the declaration of `duplicate` the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.

```
void duplicate (int& a,int& b,int& c)
               ↑      ↑      ↑
               x      y      z
duplicate (  x  ,   y  ,   z  );
```

To explain it in another way, we associate `a`, `b` and `c` with the arguments passed on the function call (`x`, `y` and `z`) and any change that we do on `a` within the function will affect the value of `x` outside it. 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 all the three variables of `main` doubled.

If when declaring the following function:

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

we had declared it this way:

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

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of `x`, `y` and `z` without having been modified.

Passing by reference is also an effective way to allow a function to return more than one 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> using namespace std;

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 PARAMETERS.

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for 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 ignored and the passed value is used instead. For example:

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

int divide (int a, int b=2)
{
    int
    r;
    r=a/b;
    return
    (r);
}
int main ()
{
    cout      <<
divide      (12);
    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 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 was not passed (notice the function declaration, which finishes with `int b=2`, not just `int b`). 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 value for b (`int b=2`) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 ($20/4$).

Overloaded functions.

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:


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

int operate (int a, int b)
{
    return (a*b);
}
float operate (float a, float b)
{
    return (a/b);
}
int main ()
{
    int x=5,y=2;
    float n=5.0,m=2.0;
    cout << operate (x,y);
    cout << "\n";
    cout << operate (n,m);
    cout << "\n";
    return 0;
}
```

```
10
2.5
```

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

In the first call to `operate` the two arguments passed are of type `int`, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type `float`, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to `operate` depends on the type of the arguments passed because the function has been *overloaded*.

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

The `inline` specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

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

and the call is just like the call to any other function. You do not have to include the `inline` keyword when calling the function, only in its declaration.

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

Recursivity.

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number ($n!$) the mathematical formula would be: $n! = n * (n-1) * (n-2) * (n-3) \dots * 1$

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

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

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

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

long factorial (long a)
{
    if (a > 1)
        return (a * factorial (a-
1)); else
        return (1);
}
int main ()
{
    long number;
    cout << "Please type a
number: "; cin >> number;
    cout << number << "! = " << factorial
(number); return 0;
}
```

```
Please      type      a
number:      9      9!      =
362880
```

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

This function has a limitation because of the data type we used in its design (`long`) for more simplicity. The results given will not be valid for values much greater than $10!$ or $15!$, depending on the system you compile it.

Declaring functions.

Until now, we have defined all of the functions before the first appearance of calls to them in the source code. These calls were generally in function `main` which we have always left at the end of the source code. If you try to repeat some of the examples of functions described so far, but placing the function `main` before any of the other functions that were called from within it, you will most likely obtain compiling errors. The reason is that to be able to call a function it must have been declared in some earlier point of the code, like we have done in all our examples.

But there is an alternative way to avoid writing the whole code of a function before it can be used in `main` or in some other function. This can be achieved by declaring just a prototype of the function before it is used, instead of the entire definition. This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters.

Its form is:

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

It is identical to a function definition, except that it does not include the body of the function itself (i.e., the function statements that in normal definitions are enclosed in braces `{ }`) and instead of that we end the prototype declaration with a mandatory semicolon `;`.

The parameter enumeration does not need to include the identifiers, but only the type specifiers. The inclusion of a name for each parameter as in the function definition is optional in the prototype declaration. For example, we can declare a function called `protofunction` with two `int` parameters with any of the following declarations:

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

Anyway, including a name for each variable makes the prototype more legible.

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

void odd (int a); 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): 9
Number is odd.
Type a number (0 to exit): 6
Number is even.
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 efficiency. I am sure that at this point you can already make a program with the same result, but using only half of the code lines that have been used in this example. Anyway this example illustrates how prototyping works. Moreover, in this concrete example the prototyping of at least one of the two functions is necessary in order to compile the code without errors.

The first things that we see are the declaration of functions `odd` and `even`:

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

This allows these functions to be used before they are defined, for example, in `main`, which now is located where some people find it to be a more logical place for the start of a program: the beginning of the source code.

Anyway, the reason why this program needs at least one of the functions to be declared before it is defined 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, a compilation error would happen, since either `odd` would not be visible from `even` (because it has still not been declared), or `even` would not be visible from `odd` (for the same reason).

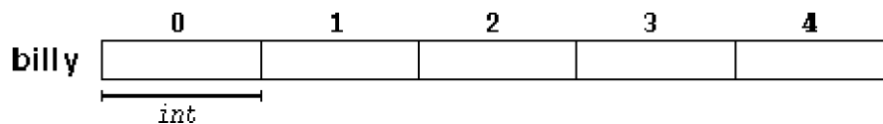
Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by declaring all functions prototypes at the beginning of a program.

ARRAYS

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, we can store 5 values of type `int` in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, 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 like this:



where each blank panel represents an element of the array, that in this case are integer values of type `int`. These elements are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length. Like a regular 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 type (like `int`, `float`...), `name` is a valid identifier and the `elements` field (which is always enclosed in square brackets `[]`), specifies how many of these elements the array has to contain.

Therefore, in order to declare an array called `billy` as the one shown in the above diagram it is as simple as:

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

NOTE: The `elements` field within brackets `[]` which represents the number of elements the array is going to hold, must be a constant value, since arrays are blocks of non-dynamic memory whose size must be determined before execution. In order to create arrays with a variable length dynamic memory is needed, which is explained later in these tutorials.

INITIALIZING ARRAYS.

When declaring a regular array of local scope (within a function, for example), if we do not specify otherwise, its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros.

In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces { }. For example:

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

This declaration would have created an array like this:

	0	1	2	3	4
billy	16	2	77	40	12071

The amount of values between braces { } must not be larger than the number of elements that we declare for the array between square brackets []. For example, in the example of array `billy` we have declared that it has 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element.

When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty []. In this case, the compiler will assume a size for the array that matches the number of values included between braces { }:

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

After this declaration, array `billy` would be 5 ints long, since we have provided 5 initialization values.

ACCESSING THE VALUES OF AN ARRAY.

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as:

`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`, we could write the following statement:

```
billy[2] = 75;
```

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

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

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

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

In C++ it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause compilation errors but can cause runtime errors. The reason why this is allowed will be seen further 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 different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements. Do not confuse these two possible uses of brackets `[]` with arrays.

```
int billy[5];           // declaration of
a new array billy[2] = 75; //
access to an element of the array.
```

If you read carefully, you will see that a type specifier always precedes a variable or array declaration, while it never precedes an access.

Some 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>
using namespace std;

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

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

Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a bidimensional table made of elements, all of them of a same uniform data type.

		0	1	2	3	4
jimmy	0					
	1					
	2					

jimmy represents a bidimensional array of 3 per 5 elements of type `int`. The way to declare this array in C++ 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 zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. But be careful! The amount of memory needed for an array rapidly increases with each dimension. For example:

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

declares an array with a `char` element for each second in a century, that is more than 3 billion chars. So this declaration would consume more than 3 gigabytes of memory!

Multidimensional arrays are just an abstraction for programmers, 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 with multidimensional arrays the compiler remembers the depth of each imaginary dimension for us. Take as example these two pieces of code, with both exactly the same result. One uses a bidimensional array and the other one uses a simple array:

multidimensional array	pseudo-multidimensional array
<pre>#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; }</pre>	<pre>#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; }</pre>

None of the two source codes above produce any output on the screen, but both assign values to the memory block called `jimmy` in the following way:

		0	1	2	3	4
jimmy {	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 simply 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++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address. In practice this has almost the same effect and it is a much faster and more efficient operation.

In order to accept arrays as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets `[]`. For example, the following function:

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

accepts 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> using namespace std;

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 parameter (`int arg[]`) accepts any array whose elements are of type `int`, whatever its length. For that reason we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter. This allows the `for` loop that prints out the array to know the range to iterate in the passed array without going out of range.

In a function declaration it is also possible to include multidimensional arrays. The format for a tridimensional array parameter 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 left blank while the following ones are not. This is so because the compiler must be able to determine within the function which is the depth of each additional dimension.